A Java-based introduction

Quantum Computing for Developers

Johan Vos

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Quantum Computing for Developers
A Java-based introduction
Version 9

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP for *Quantum Computing for Developers: A Java-based introduction*.

With this book, I want to introduce the potential and relevance of Quantum Computing to Java Developers. This book is targeting both beginning developer as well as very experienced Java developer -- and anything in between.

Most developers already heard something about Quantum Computing, but to many it seems very abstract, futuristic or mysterious. In this book, I try to explain why Quantum Computing will be very relevant to most Java developers, and why you better start learning about it today, and not wait for the first powerful quantum computers to arrive.

Quantum Computing is expected to have a big impact in many IT areas, including encryption, communication, security, scientific research, optimization, databases,... .

In the book, we show how Java developers can use their existing skills (Java development) and still leverage the benefits of Quantum Computing. You don't need a degree in physics in order to use Quantum Computing.

We explain the core concepts of Quantum Computing through the eyes of a developer, and we show a number of Java samples that leverage the benefits of Quantum Computers. These Java samples run on an open-source Quantum Computer Simulator we refer to, and with minor modifications they can run on real Quantum hardware.

I realize that the subject of Quantum Computers is very new to Java developers. When talking about Quantum Computing on conferences, I highly appreciate the feedback from the audience. The more I hear from developers, the better we can make this book. Therefore, I recommend you to use the book's forum to ask question or provide comments.

If you have any questions, comments, or suggestions, please share them in Manning's liveBook's Discussion Forum for my book.

—Johan Vos

brief contents

1 Evolution/Revolution/Hype?

This chapter covers:

- Setting the expectations for Quantum Computing
- Explaining what kinds of problems are suited for Quantum Computers
- Different options for Java Developers to work with Quantum Computing

The amount of books, articles and blog posts about Quantum Computing is increasing. Even if you read only very basic information about Quantum Computing, it is clear that this is not just an incremental enhancement of classical computing. The core concepts of Quantum Computing are fundamentally different, but also its application area is very different. In some areas, Quantum computers are expected to be able to address problems that classical computers are unable to.

Furthermore, since Quantum Computing is based on quantum physics, there is often some mystery associated with it. Quantum physics is not the simplest part of physics, and some aspects of quantum physics are extremely difficult to understand.

All combined, Quantum Computing is often pictured as some mysterious new way of working with data, that will change the world drastically. The latter is true, at least based on what we know at this moment. Many analysts believe it will take between 5 and 10 years before real useful Quantum Computing is possible, and most believe the impact will be huge.

In this book, we try to stay close to reality. We want to explain to existing and new Java developers how they can leverage Quantum Computing in their existing and new applications. As we will show, Quantum Computing has indeed a huge impact on a number of important issues in the IT industry. We will also explain why it is important to prepare for the arrival of real quantum computers, and how you can do that, using Java and your favourite toolset (i.e. your IDE and build tools). While it is true that real quantum hardware is not yet available on a

wide scale, developers should realise that building software leveraging quantum computing takes time as well. Thanks to quantum simulators and early prototypes, there is nothing that prevents developers to start working on exploring quantum computing in their projects today. This increases the chances that their software is ready by the day the hardware is available.

1.1 Expectation Management

NOTE **take-aways:**

- Don't assume Quantum Computing (QC) will fix everything
- QC is fundamentally different from classical computing
- QC is mainly suitable for complex problems
- QC and classical computers will have to work together
- the hardware is very complex, and not in our scope
- although the hardware is not yet crystalized, we can already work on software

The **potential** impact of Quantum Computing is huge. Researchers are still trying to estimate the impact, but at least in theory, there might be very large consequences for the IT industry, security, healthcare and scientific research and thus for mankind in general. Because of this large impact, a Quantum Computer is often incorrectly pictured as "a huge classical computer". This is not true, and in order to be able to see the relevance of quantum computing, one must understand why Quantum Computing is so fundamentally different from classical computing.

It has to be stressed that there are still many roadblocks that need to be addressed before the big ambitions can be realised.

The potential success of Quantum Computing depends on a number of factors that can be put in two categories:

- Hardware: new and complex hardware is needed
- Software: in order to leverage the capabilities offered by quantum hardware, dedicated software needs to be developed

1.1.1 Hardware

There are a number of uncertainties that prevent wide-scale usage of Quantum Computing at this moment. Adding to those uncertainties, it should be stressed that Quantum Computers will not fix every single problem.

The hardware needed for Quantum Computing is by no means ready for mass production. Creating Quantum hardware, in the form of a Quantum Computer or a Quantum co-processor, is

extremely challenging.

The core principles of Quantum Computing, which we will explain in this book, are based on the core principles of quantum mechanics. In quantum mechanics, the fundamental particles of nature are studied. It is generally considered to be one of the most difficult aspects of physics, and it is still in an evolving phase. Some of the brightest physicists, including Albert Einstein, Max Planck and Ludwig Boltzmann have been worked on the theory of quantum mechanics. One of the major problems in the research of quantum mechanics is that it is often extremely hard to check whether the theory matches with the reality. It is no less than amazing that theories were created predicting the existence of some particles that were not yet observed. Observing the smallest elements of nature, and their behavior requires very special hardware.

It is already difficult to investigate and manipulate quantum effects in closed lab environments. Leveraging those quantum effects in a controllable way in real-world situations is an even bigger challenge.

Most of the experimental quantum computers that exist today are based on the principles of superconducting, and operate at a very low temperature (e.g. 10 milli Kelvin, or close to -273 degrees Celcius). This has some practical restrictions that are not encountered with classical computers, operating at room temperature.

In this book, we make abstraction of the hardware. Clearly, the hardware problem isn't solved, and it is generally expected to take "a number of years" before hardware is available that can be leveraged to solve problems that are currently impossible to solve with classical computing. At the time of this writing, a number of early quantum computer prototypes already exist. IBM has a 5 qubit quantum computer that is available for public usage through a cloud interface, and quantum computer with more qubits in the research labs and for clients. Google has a quantum processor containing 72 qubits, named Bristlecone. Specialised companies like D-Wave and Rigetti have quantum computing prototypes as well. It has to be mentioned that it is not trivial to compare different quantum computers. At first sight, the number of qubits may sound the most important criterium, but it can be misleading. One of the major difficulties when building quantum computers, is to keep the quantum states as long as possible. The slightest disturbance can destroy the quantum states, and therefore quantum computers are subject to errors that need to be corrected.

As we will discuss later, there is no reason for software developers to wait until the hardware is ready before they can start thinking about software algorithms that should eventually run on Quantum hardware. The principles of Quantum computing are understood, and can be simulated via Quantum Computer simulators. It is expected that quantum software written for quantum computer simulators will also work on real quantum computers, provided that the core quantum concepts are similar.

1.1.2 Software

While there are a number of areas where Quantum Computing could, in theory, lead to a huge break-through, it is generally agreed that Quantum Computers will not replace classical computers.

There is a growing consensus where Quantum computers, or Quantum processors, can take over some tasks from classical computers, but they won't replace classical computers.

The problems that can be solved using Quantum Computing do not differ from problems that today are tackled using classical computers. However, since Quantum Computing uses a completely different underlying approach, the problems can be handled in a completely different way, and for a set of problems a dramatic increase in performance can be achieved using Quantum Computing. As a consequence, Quantum computers should be able to solve problems that today are not practically solvable because there are not enough computing resources to solve them now.

SIDEBAR **A few words on time complexity**

The complexity of algorithms is often expressed as the time complexity. In general, algorithms will take longer to complete when the amount of input data becomes bigger.

Let us assume that there are *n* items of input data. If each item requires a fixed amount of steps, the total time for the algorithm to complete is linear with *n*, the number of input items. In this case, the algoritm is said to take linear time.

Many algorithm are more complex than this. When the number of input itemsgrows, the total amount of required steps may grow with e.g. the square of n,n^2 , or even with the third power of *n*, n^k for a fixed value of *k*.In this case, the algorithm is said to take polynomial time. In this case, the algorithm is said to take polynomial time.

Some algorithms are even harder to solve when the number of input items grows.If no known algorithm is known that can solve a problem in polynomial time, we say the algorithm takes non-polynomial time. Algorithms are said to be exponential time if they require exponentially more steps when *n* increases. For example, if the amount of required steps is 2^{12n^2} , the problem is said to be of exponential complexity.

It turns out that quantum computers will be most helpful for tackling problems that can not be solved by classical computers in polynomial time, but that can be solved by a quantum computer in polynomial time.

A very common example is integer factorization, which is a very common operation in

encryption. The basic idea in integer factorization is to decompose a number into prime numbers that, when multiplied together, yield the original number. For example, **`15 = 3 x 5`**. While this is easy to do without a computer, you can imagine a computer is helpful when the numbers become bigger, e.g. **`146963 = 281 x 523`**.

The larger the number we want to factor, the longer it will take to find the solution. This is the basis of many security algorithms. They leverage the idea that it is close to impossible to factor a number consisting of e.g. 1024 bits. It can be shown that the time required to solve this problem is in the order of

EQUATION 1.1

$$e\sqrt{(64/9)b(\log b)^2}$$

where *b* is the number of bits in the original number. The *e* at the beginning of this equation is the important part. In short, it means that by making b larger, the time required to factor the number becomes exponentially larger. The diagram in Figure 1.1 shows the time it takes to factor a number with *b* bits.

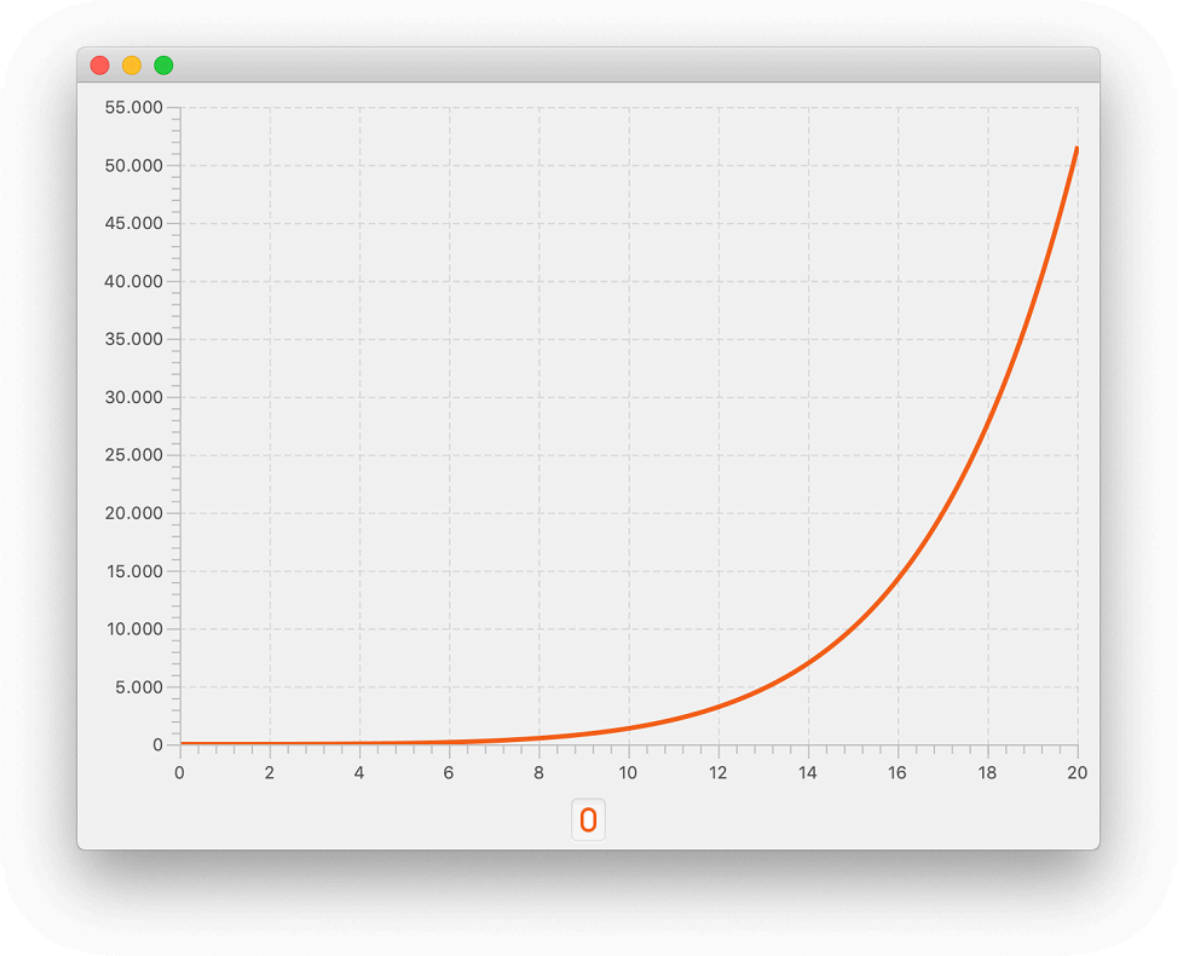

Figure 1.1 Time grows exponential with number of bits

Note that the absolute time is not relevant. Even if the fastest existing computers are used, adding a single bit makes a huge difference.

This problem is said to be non-polynomial, as there is no known classical algorithm that can solve the problem in polynomial time. Hence, by increasing the number of bits, it will be almost impossible for classical computers to find a solution to this problem.

However, this same problem can be handled by a quantum algorithm in polynomial time. As we will show in Chapter 4, using Shor's algorithm, the time to solve this problem using a Quantum Computer is in the order of *\b^3*.

To show what that means, we overlay the required time using a quantum algorithm on a quantum computer over the required time using a classical algorithm on a classical computer. This is illustrated in Figure 1.2.

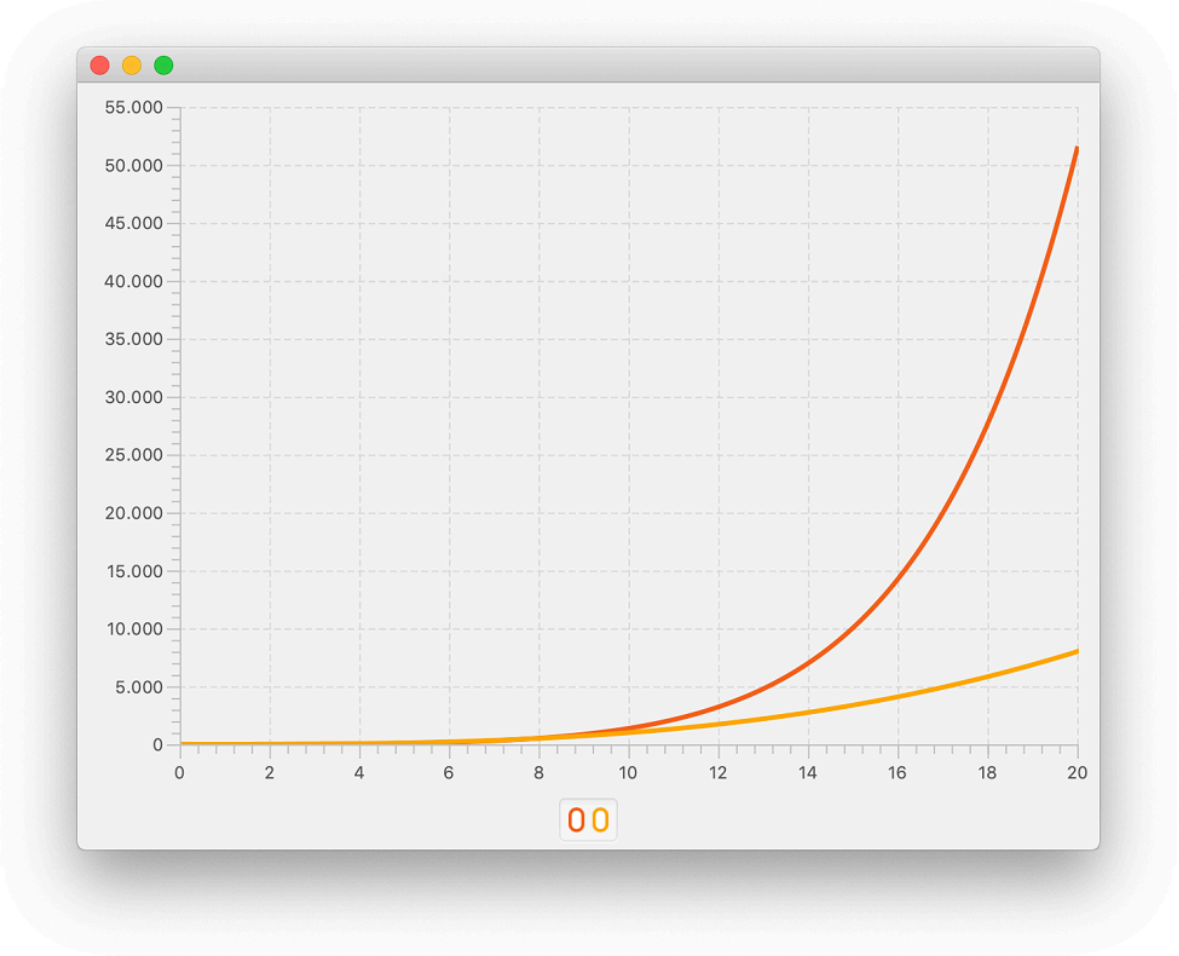

Figure 1.2 Polynomial time versus exponential time

Starting from a number of bits, the quantum computer will be much faster than the classical computer. Moreover, the larger the amount of bits, the larger the difference. This is because the required time for solving the problem on a classical computer increases exponentially when the amount of bits is growing, where the same increase of bits will "only" cause a polynomial increase for the quantum algorithm.

These kinds of problems, that are said to be polynomial in Quantum, are the ones that makes mose sense for Quantum Computers to deal with.

1.1.3 Algorithms

Shor's algorithms is a great example of a computational problem that is hard to solve on a classical computer (non-polynomial in time) and relative easy on a quantum computer (polynomial in time). Where does the difference come from? As we will discuss in Chapter 4, Shor's algorithm transforms the problem of integer factorisation into the problem of finding the periodicity of a function, i.e. find the value p for which the function evaluation `f(x+p) = f(x)` for all possible values of x. This problem is still very hard to solve on a classical computer, but it is relative easy to fix on a Quantum Computer.

Most algorithms that are known today to be very suitable for quantum computers are based on the same principle: transform the original problem into a problem space that is easy to solve using Quantum Computers.

The classic approach is shown in Figure 1.3. The best known algorithm is applied to the problem, and the result is obtained.

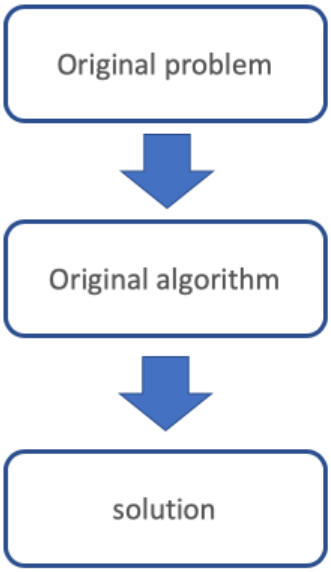

Figure 1.3 Typical approach solving a problem on a classical computer

If we can somehow transform the original problem to a different problem that can easier be handled by a quantum computer, we can expect a performance improvement. This is shown in Figure 1.4.

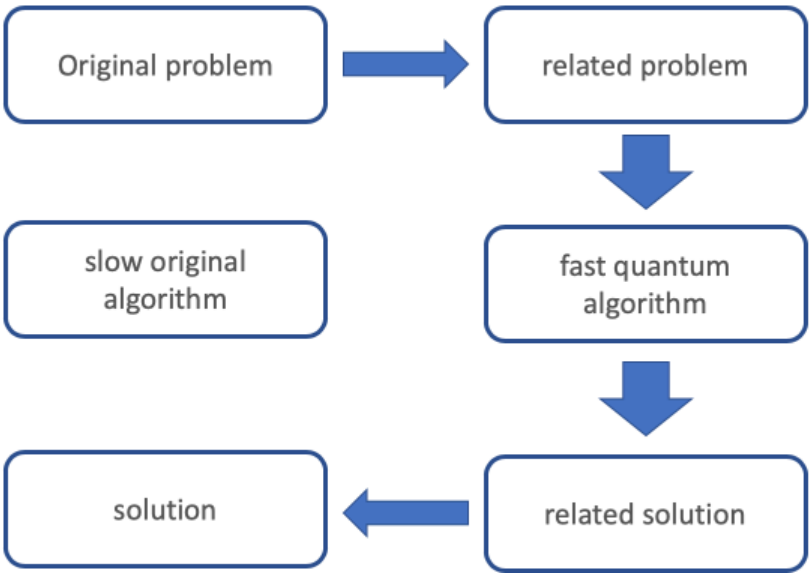

Figure 1.4 Transforming a problem to an area where quantum computers can make a big difference

Note that we have to take into account the cost of transforming the original problem to a different problem, and vice versa for the final result. However, when talking about real compution-intensive algorithms, this cost should be neglectible.

> **NOTE** When you see a Quantum algorithm being explained, you may wonder why it seems to take a detour from the original problem. Quantum computers are capable of solving particular problems very fast, so moving an original problem to one of those particular problems allows for a much faster algorithm, using quantum computing.

Coming up with those algorithms often involves a very deep mathematical background. Typically, developers will not create new quantum algorithms for applications that will benefit from quantum computers, but they will use existing algorithms. However, developers who know the basics about quantum algorithms, why they are faster, and how to use them, will have an advantage.

1.1.4 Why start with quantum computing today?

Programmers sometimes wonder why they should start learning about quantum computing, when real, usable quantum computers are still years away. Developers have to realise though that writing software that involves quantum computing is different from writing classical software. While it is expected that there will be libraries that make it convenient for developers to leverage quantum computers, those libraries have to be written, and even then it will require skills and knowledge to be able to use the best tools for a particular project.

Any developer working on a project that requires encryption or secure communication benefits from learning about quantum computing. Some existing classical encryption algorithms will become insecure when quantum computers are available. It would be a bad idea to wait for the first time a quantum computer breaks encryption before hardening the encryption software. At the contrary, you want to be prepared before the hardware is available. Since quantum computing is really disruptive, it can be expected that most developers need more time learning quantum computing than they typically need when using a new library.

While we do not want to scare people with doom scenario's, it is important to understand that there is no need for a wide installed base of quantum computers before existing encryption techniques can be compromised. Cyber attacks do not require a large amount of computers, and can be carried out from any place.

SIDEBAR There is a reasonable chance that a number of existing communication protocols and encryption techniques will become vulnerable once quantum computers become more powerful. It is important for developers to understand what kind of software might be vulnerable, and how to address this. This is not something that can be done overnight, hence it is recommended to start looking into this sooner rather than later.

The software examples we will discuss in this book are very basic applications. They illustrate the core principles of quantum computing, and they make it clear what kind of problems can really benefit from quantum computing. But the gap between basic algorithms and fully functional software is large. Hence, while it will take years before the hardware is ready, developers have to understand that it will probably also take a long time before they have optimised their software projects so that they leverage quantum computing as much as possible, where applicable.

In the mid of the previous century, when the first digital computers were built, software languages needed to be created as well. The difference with today is that we can now use classical computers to simulate quantum computers. We can work on software for quantum computers, without having access to a quantum computer.

This is a very important benefit, and it stresses the importance of quantum simulators. Developers starting today looking into quantum computing using simulators will have a huge advantage on other developers when the quantum hardware becomes more widely available.

1.2 The disruptive parts of Quantum Computing, getting closer to nature

One of the main application areas of quantum computing is everything related to physics. For a long time, scientists have been trying to understand the core concepts of modern physics by simulating the concepts on classical computers. However, since the most granluar particles of nature do not follow classic laws, it is complex to simulate them on classical computers. Using exactly those quantum particles and their laws as the cornerstones of quantum computers makes it much easier to tackle those problems.

1.2.1 Evolutions in classical computers

Over the past decades, computers have become more powerful. Improvements in performance are often realized because of

- an increase in the memory of the computer
- an increase in the performance of the processor
- an increase in the number of processors in a computer

These improvements typically lead into incremental, linear benefits.

The potential performance gains that are expected to be realised using Quantum computers have nothing to do with these improvements.

A Quantum computer is not a classical computer with smaller chips, more memory, or faster communication.

Instead, Quantum Computing starts with a completely different fundamental concept, which is a qubit. We will discuss the qubit in detail in Chapter XX but since it is a crucial concept, we introduce it here.

1.2.2 Revolution in quantum computers

In a classical computer, a bit is smallest piece of information, and it can be either 0 or 1. Different operations are possible on those bits, and bits can be altered or combined. At any moment though, all bits in a computer are in a clear state: 0 or 1. The physical analogy of a classical bit is related to current. A "0" state corresponds with no current, and a "1" state corresponds with current.

All existing classical software development is based on the manipulations of those bits. Using combinations of bits, and applying gate operations of bits is the essence of classical software development. We will discuss this in more detail in Chapter 3.

In Quantum Computing, the fundamental concept is a qubit. Similar to a classical bit, a qubit can hold the values 0 and 1. But the disruptive difference is that the value of a qubit can be a combination of the values 0 and 1. When people first hear about this, they are often confused. It sounds artificial to have the qubit, the elementary component of Quantum Computing to be more complex than the elementary component of classical computing, the bit. It turns out, however, that a qubit is closer to the fundamental concepts of nature than the classical bit.

1.2.3 Quantum Physics

As its name implies, the foundation for Quantum Computing comes from quantum physics. In quantum physics, the smallest particles, their behavior and their interactions are investigated. It turns out that some of those particles have properties with interesting characteristics. For example, an electron has a property called spin, which can take two values: up and down. The interesting thing is that the spin of an electron can, at a given moment, be in a so called superposition of these two values. This is a hard-to-understand physical phenomenon, and it comes down to the easier-to-understand mathematical formula where the spin can be a linear combination of the up value and the down value — with some restrictions that we talk about in chapter XX.

The spin of an electron is just one sample of a physical phenomenon that allows for a property to

be in more than one state at the same moment.

In Quantum Computing, the qubit is realised by this physical phenomenon. As a consequence, the qubit is extremely close to the reality of quantum physics. The physical realisation of a qubit is a real-world concept. Therefore, Quantum Computing is often said to be very close to how nature works.

One of the goals in Quantum Computing is to take advantage of physical phenomenons that happen at the scale of the smallest particles. Hence, Quantum Computing is more "natural" and although it seems much more complex than classical computing at first sight, it can be argued that it is at the contrary much simpler, as it requires less artificial constructs.

Understanding quantum phenomenons is one thing, being able to manipulate them is another. It took lots of time and resources to be able to prove that quantum phenomenons really exist. In order to allow computational representations on qubits, one must be able to manipulate the elementary parts. While this is what is typically done in large scientific research centers, it is still very hard to do this in a typical computer environment.

1.3 Hybrid Computing

We already mentioned that Quantum Computers can be excellent when dealing with specific problems, but not for all kinds of problems. Therefore, the best results can probably be achieved using a new form of hybrid computing, where a quantum system solves part of the problem, where a classical computer is solving the other parts of the problem.

Actually, this approach is not entirely new. A very similar pattern is already being used in most modern computer systems, where the Central Processing Unit (CPU) is accompanied by a Graphics Processing Unit (GPU). GPU’s are good in some particular tasks (e.g. doing vector operations that are needed in graphical applications, or in deep learning applications), but not in all tasks.

Many modern UI frameworks, including JavaFX, leverage the availability of both CPU’s and GPU’s, and optimize the tasks they have to perform by delegating parts of the work to the CPU and other parts to the GPU, as shown in Figure 1.5.

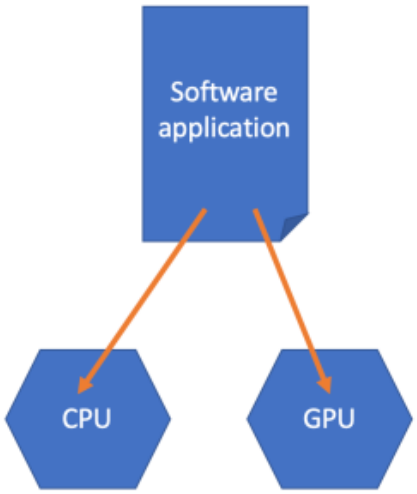

Figure 1.5 CPU and GPU sharing work

The idea of using different co-processors for different tasks can be extended to Quantum Computing. In the ideal scenario, a software application delegates some tasks to a CPU, other tasks to a GPU and other tasks to a Quantum Processing Unit (QPU), as shown in Figure 1.6.

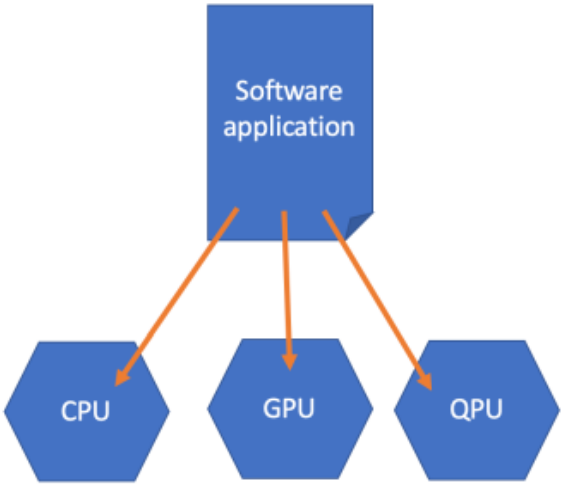

Figure 1.6 CPU, GPU and QPU sharing work

The best results can be achieved when the best tools are used for a specific job. In this case, it means that the software application should use the GPU for e.g. vector computations, the QPU for algorithms that are slow on classical systems but fast on quantum systems, and the CPU for everything that doesn't benefit from either the GPU or the QPU.

If every end-application has to judge what parts should be delegated to which processor, the job of a software developer would be extremely difficult. We expect though that frameworks and libraries will provide help here, and abstract this problem away from the end-developer.

If you are using the JavaFX API's to create user interfaces in Java, you don't have to worry about what parts are executed on the GPU, and what parts are executed on the CPU. The internal implementations of the JavaFX API already do that for you. The JavaFX framework detects the

information about the GPU, and will delegate work to it. While it is still possible for developers to directly access either the CPU or the GPU, this is typically something high-level languages as Java shield away.

In the picture above, we oversimplified the QPU. Where a GPU easily fits in modern servers, desktop systems, but also in mobile and embedded devices, providing a Quantum Processor might be more tricky, due to the specific requirements for quantum effects to be manipulated in a controlled, noise-free environment.

It is very well possible that, at least initially, most of the real quantum computing resources will be available via specific cloud servers, instead of via co-processors on embedded chips.

The principles stay the same though, since the end-software application can benefit from libraries splitting the complex tasks, and delegate some tasks to a quantum system that is accessible via a cloud service as shown in Figure 1.7.

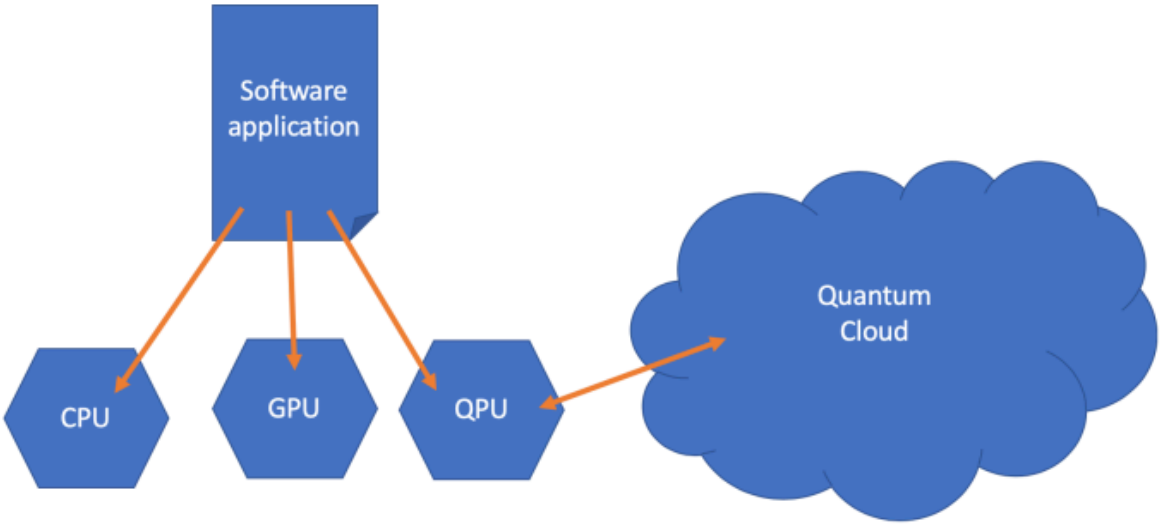

Figure 1.7 Quantum calculations relayed to cloud

1.4 Abstracting software for Quantum Computers

Although real Quantum Computers already exist as we mentioned before, they are by no means ready for mass production. While the achievements in the past years for creating hardware for Quantum Computing are huge, there is still lots of uncertainty about the implementation of a real, useful Quantum Computer or Quantum processor.

However, this should not be a reason to not start working on the software. We learned a lot from classical hardware, and from the software that is built on top of it. The high-level programming languages that have been created in the past decades allow software developers to create applications in a convenient way, such that they do not have to worry about, or even understand, the underlying hardware. Java, being a high-level programming language, is particularly good in

making abstraction of the underlying low-level software and hardware. Ultimately, when a Java application is executed, some very low-level, hardware-specific instructions are executed. Depending on the hardware being used, specific machine instructions, for different processors with different architectures are used.

Hardware for classical computers is still evolving. Software is evolving as well. Most of the changes in the Java language, however, are not related to hardware changes. The decoupling of hardware and software evolutions allows for much faster innovation. There are a number of areas, though, where improvements in hardware ultimately lead to more specific evolutions in software but for most developers, hardware and software can be decoupled from each other. Figure 1.8 shows how a Java application ultimately results in operations on hardware, but different abstraction layers shield the real hardware (and the evolutions in the hardwar) from the end application

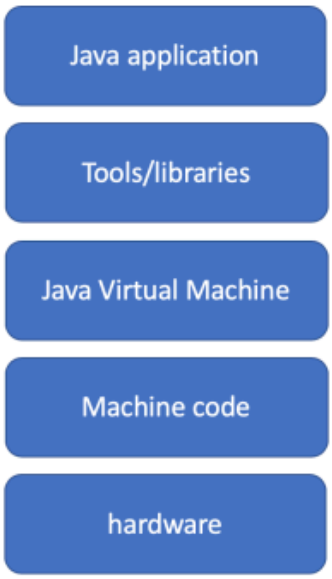

Figure 1.8 Classic software stack

For a large part, software for Quantum Computing can be decoupled from the hardware evolutions. While the hardware implementation "details" are far from clear, the general principles are becoming very clear. We discuss those principles in Chapters 2 - 5. Software development can be based on those general principles. Similar to how a classical software developer doesn't have to worry about how transistors (a low-level building block for classical computers) are combined on a single chip, a developer of quantum software does not have to think about the physical representation of a qubit (one of the low level building blocks in quantum computing). As long as the quantum software conforms with and exploits the general principles, it will be usable on real quantum computers or quantum processors when they become available.

A major benefit while developing software for Quantum Computers, is the availability of classical computers. The behavior of quantum hardware can be simulated via classical software.

This is a huge advantage, since it implies that quantum software can be tested today, using a quantum simulator written in classical software, on a classical computer. Obviously there are major differences between a quantum computer simulator and a real hardware quantum computer. Almost by definition, a typical quantum algorithm will execute much faster on a quantum computer than on a quantum simulator. But from a functional point, the results should be the same.

Apart from real quantum computers and quantum computer simulators, cloud services should be taken into account. By delegating the work to a cloud service, an application doesn't even know if it is running on a simulator, or on a real quantum computer. The cloud provider can update its service from a simulator to a real quantum computer. The results should be obtained much faster when a real quantum computer is used, but they should not be different from when a simulator is used.

These options can be combined in Figure 1.9

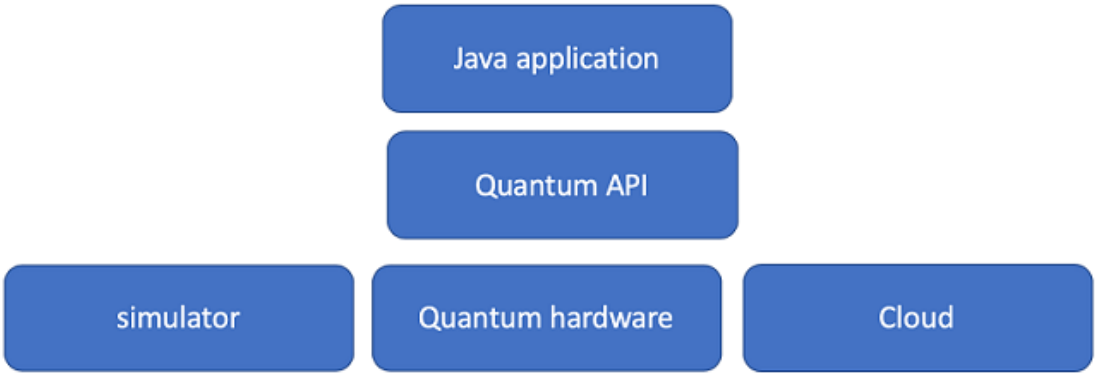

Figure 1.9 Stack for Java applications using Quantum API's

In this picture, we show that Java applications can leverage libraries that provide Quantum API's. The implementation of these library can do the work on a real quantum computer, use a quantum computer simulator, or delegate the work to the cloud. For the end application, the results should be similar.

As we already discussed, Quantum algorithms are particular useful when dealing with problems that require exponential scaling when dealt by with classical computers. One of the typical examples for this is integer factorization. A Quantum Computer will be capable of decomposing large integers into their prime factors (at least, it will provide a part of the algorithm), something that is not possible today even with all computing power in the world combined. As a consequence, a quantum computer simulator written in classical software is also not able to factor those large numbers.

The same Quantum Algorithm is of course also capable of factoring small integers. Quantum simulators can thus be used to factor small integers. The Quantum Algorithm can be created, tested and optimized using small numbers on a quantum simulator. Whenever the hardware

becomes ready for it, that same algorithm can then be used to factor numbers on real hardware (a 5 qubit system has already factored 21).

When the quantum hardware improves (more qubits are added, or less errors occur), the algorithm will allow larger numbers to be factorised.

In summary, the principles of Quantum Computers can be mimicked in software simulators running on classical computers. Developers can take advantage of this, and run their quantum experiments on those simulators. Throughout this book, we use an open-source Quantum Simulator written in Java that works both locally on your laptop/desktop, as well as in cloud environments. The developer doesn't have to worry about where the code is being executed. Also, when the hardware topology changes in the future (e.g. a quantum co-processor is added), the end-application doesn't have to be modified. The library will be updated, but the top-level API's should not be affected to this.

We explain some of the Quantum Computing principles by looking at the source code of the algorithms in the library. While this is not stricly needed to write applications leveraging Quantum Computing, it will give the reader more insight in how and when quantum algorithms might lead to a real advantage.

In this chapter, you learned:

- Quantum computing is not just an upgrade of classical computing
- Quantum computing leverages the real core concepts of physics, and is therefore more "real" than classical computing
- It may take many years before hardware is powerfull enough to gain the real benefits of quantum computing
- Quantum computers are expected to generate a huge speedup in the execution of some algorithms that are practically impossible to solve in the classic way, but they won't replace classical computers since they are only good at particular (but important) tasks
- Software development at a high level should not worry about the low level quantum details
- Software developers should be aware of the fact that moving some parts of an algorithm to a different area might lead to huge improvements.

2 Hello World, Quantum Computing

This chapter covers

- an introduction to Strange, a quantum computing library in Java
- a simple demo of the high-level API of Strange
- a very basic sample of the low-level API of Strange
- a basic visualisation of quantum circuit
- references to concepts that will be explained later in the book

In this chapter, you will be introduced to Strange, an Open Source Quantum Computing project including a Quantum simulator, and a library that exposes a Java API that you can use in regular Java applications.

Throughout the book, we will dicuss concepts of Quantum Computing, and their relevance to Java developers. We will show how Java developers can benefit from these concepts.

Strange contains a pure Java implementation of the required quantum concepts. When discussing the concepts, we point the interested reader to the relevant code implementation of the concept in Strange. This is part of a low-level API.

Most Java developers will not have to deal with low-level quantum concepts at all. However, they might benefit from algorithms that take advantage of these concepts. For this group, Strange provide a set of high-level algorithms that can be used in regular Java applications. These algorithms are what we call the high-level Java API.

2.1 Introducing Strange

Figure 2.1 shows a high-level overview of the components of Strange.

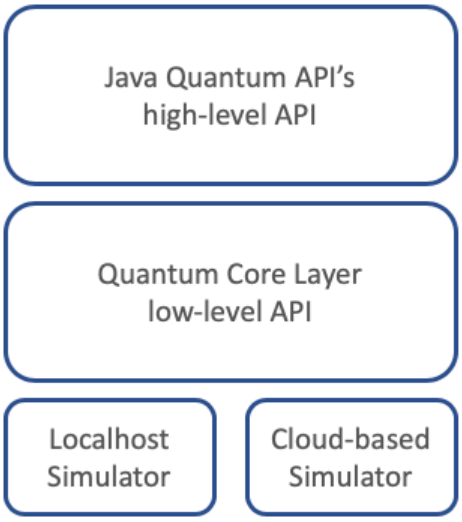

Figure 2.1 High-level overview of the Strange architecture.

The Java Quantum API provides an implementation for a number of typical quantum algorithms. These are the high-level algorithms that can be used by Java developers in their regular Java applications. No knowledge about quantum computing is required in order to use the algorithms.

The Quantum Core Layer contains the low-level API which provides deeper access to the real quantum aspects. The high-level API does not contain a concept specific to quantum computing, but its implementation leverages the low-level Quantum Core Layer. Where the high-level API shields the user from the quantum concepts, the low-level API at the contrary exposes those concepts to the user.

The high-level API provides developers with a ready-to-use interface to quantum algorithms. By using it, you can benefit from the gains realised by quantum computing. However, if you want to be able to create your own algorithms, or modify existing algorithms, the low-level API is the starting point.

2.2 Running a first demo with Strange

This book comes with a repository containing a number of samples that leverage Strange. The requirements and instructions for running those samples are explained in Appendix A. The first demo sample is located in the `hellostrange` folder in the `ch02` directory.

We use the gradle build tool for building and running the samples, but users familiar with maven will be able to easily run the samples with maven.

We do recommend you run the samples using your favourite IDE (IntelliJ, Eclipse or NetBeans). The instructions on how to run Java applications are different for each IDE. Therefore, in this book, we use the gradle build system from the command line.

Using the provided gradle (or maven) scripts implicitly makes sure all required code

dependencies are downloaded. The code is compiled, and executed, as illustrated in Figure 2.2

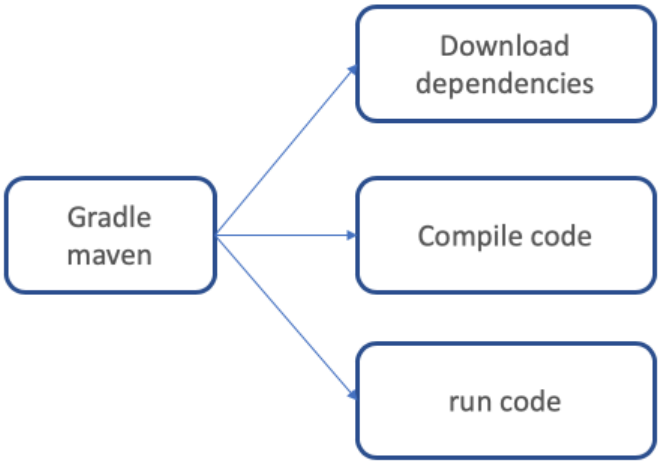

Figure 2.2 Using gradle or maven to run java applications

The result of

```
./gradlew run
```

on linux and macos or

```
gradlew.bat run
```

on Windows will result in the following output:

```
> Task :run
Using high-level Strange API to generate random bits
--------------------------------------------------
Generate one random bit, which can be 0 or 1. Result = 1
Generated 10000 random bits, 4961 of them were 0, and 5039 were 1.

BUILD SUCCESSFUL in 3s
```

Congratulations! You just executed a program that involves Quantum Computing.

2.2.1 Inspecting the code for HelloStrange

In order to understand the output of the HelloStrange demo application, it is recommended to have a look at the source code for the application. Before we investigate the Java code, we have a look at the `build.gradle` file that is in the root directory of the sample. The `build.gradle` file contains the instructions that allow `gradle` to compile the Java classes, download and install dependencies, and run the application.

Typically, you shouldn't worry about the structure of the `build.gradle` file, unless you plan to create applications or projects yourself. In that case, you can find great resoures about using Gradle online.

For clarity, the `build.gradle` file is shown in Listing 2.1:

Listing 2.1 build.gradle file for HelloStrange sample

```
plugins {                                                          ❶
    id 'application'
    id 'org.javamodularity.moduleplugin' version '1.2.1'
}

repositories {                                                     ❷
    mavenCentral();
}

dependencies {                                                     ❸
    compile 'com.gluonhq:strange:0.0.5'
}

mainClassName = 'com.gluonhq.javaqc.ch02.hellostrange.Main'        ❹
```

❶ declare what plugins gradle should use Gradle is a build system that allows third parties to provide plugins in order to make it easier to build and deploy applications. The demo application is an application, and therefore uses the `application` plugin. Strange is using Java 11 and the modularity concepts that have been introduced in Java 9. Our demo applications don't require knowledge about the modular system in Java though. However, in order for the build tools to be able to leverage the modularity, we also declare the use of the `javamodularity` plugin.

❷ declare where to download dependencies . Since our demo application is using a Java library, Gradle needs to know where to find this library in order to use it for compiling and running the demo application. The Strange library is uploaded to the `mavenCentral` repository, hence we declare that in the `repositories` section.

❸ declare the dependencies. The HelloStrange demo application uses the Strange library. In the `dependencies` section of the build.gradle, we declare that we need version `0.0.3` of the Strange library, which is defined by the combination of a package name `com.gluonhq` and an artifact name `strange`. The `compile` keyword tells Gradle that this library is needed to compile the application, and by default it will then also use this library to run the application.

❹ declare the main class that should be executed when running the demo. Finally, we need to tell Gradle where it can find the main entry point to our application. In this case, the project has only a single Java source file with a `main` method, hence this is the entry point.

The `build.gradle` file is interesting to developers and code maintainers who are working on project development, deployment, testing and distribution.

The Java source files in a project are very relevant to all developers. Gradle requires by default that Java source files are placed in a folder `src/main/java`, followed by the packagename and the name of the Java source file. In the case of the HelloStrange application, the single source file is thus located in `src/main/java/com/gluonhq/javaqc/ch02/hellostrange/Main.java`.

Before we show the code, we will briefly explain what we want to achieve. In this first sample,

we will invoke a method on the high-level Strange API. This method is called *randomBit()* and it generates a classic bit which is either *0* or *1*. We will discuss the *randomBit()* method call shortly. Apart from this call, all Java code used in the sample only uses the standard API's that are part of the JDK. The flow for the sample is shown in Figure 2.3

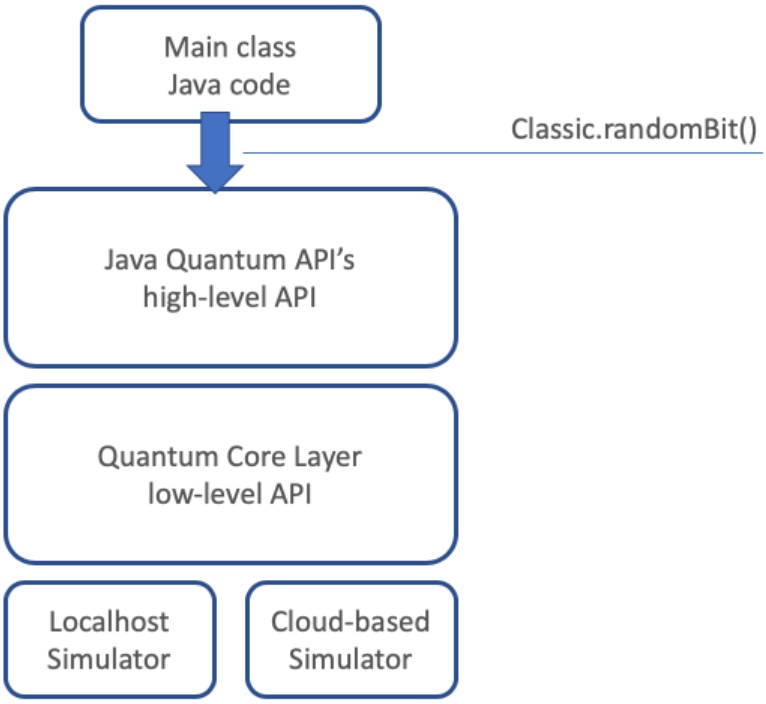

Figure 2.3 High-level overview of the first Java sample

From this flow, it can be seen that the Java class we create depends on the high-level Strange API. We don't have to worry about how it is implemented in the lower layers of Strange.

The complete source code for the application is shown in Listing 2.2. We will analyse this source code right away.

Listing 2.2 Main.java file for HelloStrange sample

```
package com.gluonhq.javaqc.ch02.hellostrange;

import com.gluonhq.strange.algorithm.Classic;

public class Main {

    public static void main (String[] args) {
        System.out.println("Using high-level Strange API to generate random bits");
        System.out.println("-----------------------------------------------------");
        int randomBit = Classic.randomBit();                    ❶
        System.out.println("Generate one random bit, which can be 0 or 1. Result = "+randomBit);
        int cntZero = 0;
        int cntOne = 0;
        for (int i = 0; i < 10000; i++) {                       ❷
            if (Classic.randomBit() > 0) {
                cntOne ++;
            } else {
                cntZero ++;
            }
        }
        System.out.println("Generated 10000 random bits, "+cntZero+" of them were 0,
                           and "+cntOne+" were 1.");
    }
}
```

❶ We call the Strange high-level API to generate one random bit

❷ We generate 10000 random bits

This Java code follows the basic Java conventions, which we assume you are familiar with. For this sample, we briefly mention the typical concepts in a Java application.

The Java code in this source file belongs to the package `com.gluonhq.javaqc.ch02.hellostrange` which is declared at the top of the file.

We rely on functionality provided by the Strange library, and we import the Java class that provides the functionality we need:

```
import com.gluonhq.strange.algorithm.Classic
```

We will have a deeper look at this `Classic` class later. For now, we simply assume it provides the functionality we need.

The name of this Java class is `Main`, as it has to match the name of the file.

In Java, entrypoints in files need to be declared with a method `public static void main(String[] args)`. Build tools like Gradle will invoke this method when asked to execute an application.

When the `main` method is invoked, it will first print some information:

```
System.out.println("Using high-level Strange API to generate random bits");
```

```
System.out.println("--------------------------------------------------");
```

In the next line of code, we call a method on the `Classic` class, that is part of the Strange library that we imported. The method we call is the `Classic.randomBit()` method, and it returns a Java integer that either holds the value `0` or the value `1`.

After the statement

```
int randomBit = Classic.randomBit();
```

the value of randomBit is thus `0` or `1`.

NOTE The classname *Classic* indicates that Strange offers this class for classic invocations. Code calling this class is not expected to contain any quantum-specific implementations. However, the implementation of the *Classic* class itself contains quantum implementations. Therefore, the implementation of *Classic.randomBit()* is not simply returning a default Java random bit, but it is using a quantum circuit to do so --- as we will show later in this chapter.

In the next line, this value is printed. Note that when you execute the application, there is 50% chance you will see a `0` printed, and 50% chance that you will see the `1` printed.

The `Classic.randomBit()` is a Java method that under the hood leverages quantum principles. We will discuss the implementation later. For now, we assume that there is an equal change for this method to return `0` and `1`.

In order to demonstrate this, the next part in the Java source code will call this `Classic.randomBit()` 10,000 times, and it will keep track on how many times a `0` is returned and how many times a `1` is returned.

Two variables are introduced, for keeping track of this occurences:

```
int cntZero = 0;
int cntOne = 0;
```

Clearly, `cntZero` will hold the number of times the returned value is `0` where `cntOne` holds the count for the calls that return `1`.

We then create a loop which inner code calls the `randomBit()` method and increments the appropriate variable. This is done in this code snippet:

```
for (int i = 0; i < 10000; i++) {
    if (Classic.randomBit() > 0) {
        cntOne ++;
    } else {
        cntZero ++;
```

```
    }
}
```

Finally, the results are printed. Since the random values are truly random, the final results will very likely be different every time you run the application. The sum of the `cntOne` and `cntZero` values will always be 10,000 and it is expected that the `cntZero` and `cntOne` values both are in the neighbourhood of `5000`.

2.2.2 Java API's versus implementations

If you are familiar with Java development, the code we have shown and used so far will be very familiar. No specific knowledge on quantum physics or quantum computing has been required. We only used the `Classic.randomBit()` method call, which is a method call similar to all other Java method calls that you see in Java applications. Under the hood, however, the `Classic.randomBit()` call is using either a Quantum Simulator or a real Quantum Computer. The Java developer is not confronted with the implementation though, as one of the great things about Java is that the implementation is typically hidden for developers, who program their applications using API's. In this case, `Classic.randomBit()` is an API that is called by the developer.

Although Java developers don't need to know the details about the underlying implementations, it often helps to have at least some insight in those details. This is not only the case for algorithms on Quantum Computing, it is applicable to many fields. While documentation (e.g. JavaDoc) is typically very helpful for general cases, it might help to understand some of the details if you want to keep track of performance, for example. In the case of Quantum Computing, it is recommended for Java developers to at least have some basic knowledge about the underlying implementation of the quantum API's, as this provides useful information that can be used to judge wether a quantum algorithm is applicable or not for a specific usecase, and what the performance impact will be.

Also, without this basic knowledge, users might worry about the initial performance of some of the algorithms. Indeed, if a quantum algorithm is executed on a Quantum Simulator, the performance will probably be worse than if a classic algorithm was used. However, if the quantum algorithm is well-written and if the problem is applicable for quantum speedup, the performance will dramatically improve once real quantum hardware is used.

2.3 Obtaining and installing the Strange code

As explained in the previous section, developers typically don't need to understand the implementation details of an algorithm. However, in this book we explain the basic concepts of quantum computing by showing code snippets of quantum algorithms. By having a look at the implementation of some algorithms, developers learn more about the concepts of quantum computing, and they will be more knowledgable about the areas where quantum computing can make a big difference.

The Strange library we use throughout this book is written in Java. This not only allows Java developers to use the quantum API's in their own applications, it also enables them to have a deeper look in the implementations, and maybe modify or extend them when needed.

If you are using a partiular IDE (e.g. NetBeans, IntelliJ or Eclipse), you should have no problems opening the library and reading the files.

2.3.1 Downloading the code

Similar to the samples and demoes used in this book, the code for the Strange library can be downloaded from github as well. The following command will provide you with a local copy of the Strange library:

```
git clone https://github.com/gluonhq/strange.git
```

Note: if you want to use the Strange library in your application, you don't need to download the source code. Binary releases of Strange are uploaded to Maven Central, and build tools like Gradle and Maven will retrieve them from this uploaded location.

If for some reason you want to make modifications to Strange, and test them locally, you can easily compile the whole project. Similar to our demo application in the previous section, Strange uses the Gradle build system to create the library.

The following gradle command can be used to build the library:

```
./gradlew build
```

The result of this operation will be a local copy of the Strange library, that you can then use in local applications. Before you can use your own library, you need to take into account two things:

- the `build.gradle` file contains a `version` key. You can change that to whatever you want, but you have to be sure to use the same version in the `dependencies` section of your application.
- your application now needs to include `mavenLocal()` in the list of its repositories.

2.3.2 A first look into the library

You can open the code in your IDE, or you can manually browse through the different files. As an example, you can open the `Classic` file that we referred to in the HelloStrange application that we discussed in the previous section. The source code for the `Classic` class is in the `Classic.java` file which is in the `src/main/java/com/gluonhq/strange/algorithm` folder under the directory where you cloned the git repository.

We will discuss this file in detail in Chapter 5, but we already show a snippet that shows the link between the `Classic.randomBit()` call from the previous section to the implementation using a Quantum Computer or a Quantum Simulator:

```
public static int randomBit() {
    Program program = new Program(1);
    Step s0 = new Step();
    s0.addGate(new Hadamard(0));
    program.addStep(s0);
    QuantumExecutionEnvironment qee = new SimpleQuantumExecutionEnvironment();
    Result result = qee.runProgram(program);
    Qubit[] qubits = result.getQubits();
    int answer = qubits[0].measure();
    return answer;
}
```

This snippet shows that the random bit returned by the `randomBit()` method is not simply generated by a classic random function, but it involves steps specific to quantum computing. Again, Java developers typically don't need to know much about the implementation, but by looking at it, you can learn a lot about quantum computing.

2.4 Next steps

Now that you downloaded the Strange library and ran your first Java application leveraging quantum computing, it is time to learn more about the basic concepts of Quantum Computing. If you want to look into more code first, you're welcome to browse through the different files in the Strange library. However, it is recommended to first read about the basic concepts. Whenever we introduce a concept, we will point to some code in Strange where the concept is applied.

In this chapter, you learned

- the basic concepts of a Quantum Simulator
- how to call the high-level API of this Quantum Simulator
- to run a very basic application that is leveraging the high-level API
- some basic information on how the high-level API interacts with the low-level API.

3

Qubits and Quantum Gates, the basic units in Quantum Computing

This chapter covers:

- we will introduce the important concept of a qubit and compare this with the more familiar concept of the (classical) bit
- we introduce 2 notations for qubits
- we discuss how quantum gates allow to perform operations on qubits
- we show a very simple gate, and use the Strange UI applicationto visualise the effect of this gate.

When creating typical applications using classic computers, most developers don't think about the transistors and the operations at the lowest level that ultimately allow applications to execute on hardware. Classic hardware is commodity in a sense that most developers take it for granted and don't think about it. The details about how it works are not relevant to allmost all applications that are being developed. High-level programming languages shield developers from the low-level (assembly) code, and standards in chip design make it even less relevant for developers to understand the physical working of the hardware in a computer.

This used to be different. In the early days of classical computing, there were no high-level programming languages, and developers were working closer on the "bare metal". Once the hardware for classic computers became more mainstream and standardized, focus moved to higher-level programming languages.

It can be expected that Quantum Computing will follow a similar path. In the future, no knowledge about the basic concepts of quantum computing will be required for a developer who leverages quantum computing. Similar to the situation in classic computers, higher-level languages and intermediate layers will shield developers from the implementation details in the hardware.

Today, if developers want to leverage quantum computing, it definitely helps if they have at least some basic understanding of the underlying principles that allow quantum computing.

In this chapter, we introduce those basic concepts. We discuss qubits and quantum gates, and we briefly touch the link to the physical world that allows their implementation. By no means this chapters is an introduction into quantum mechanics. The interested reader is referred to the specialized literature (links!).

3.1 Classic bit versus Qubit

"Suppose you work for a bank, and you need to make sure the account number and balances for each customer are stored, and can be retrieved. Somehow, developers need to work with this information. How can you represent this information on a classic computer?"

Before computers can work with information (numbers, text, images, videos,…), the information needs to be represented in a way computers understand it.

The classic `bit` is one of the most common low-level structures that is understood and used by most developers. A bit contains the most granular information in classic computing, and it has a value of eiter `0` or `1`.

Figure 3.1 A single bit can be 0 or 1

The bit allows other structures, for example the `byte` which is an ordered sequence of 8 bits.

At any moment in the execution of a classic algorithm, each bit is in a very specific state: it is `0` or it is `1`. As a consequence, a byte is, at any given moment, in a very specific state as well. Each of the 8 bits in a byte is either `0` or `1`,

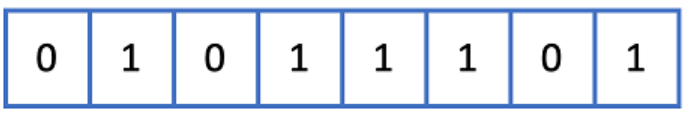

Figure 3.2 A single byte contains a sequence of 0 and 1

The size of the memory of a computer is expressed as the number of bits that can be accessed by the processor. The amount of memory is one of the main contributors to the quality and performance of computers. The more memory a computer has, the more data it can hold.

The core idea of a bit, the fact that its value at a given moment is either '0' or '1' is also one of its limitations.

In quantum computing, the equivalent of the bit is the qubit. Similar to a bit, a qubit can hold the values '0' and '1'. But contrary to a bit, a qubit can also hold values that are "combinations" of the '0' and the '1' state. When this is the case, the qubit is in a so-called **superposition** state. While this may sound counterintuitive at first, it is actually exactly what is happening in nature, with a number of the most granular particles, and it is directly linked to the core ideas of quantum mechanics. The fact that this superposition state occurs in nature with very granular particles is a good indication that building quantum computers is very realistic. Classic computers ignore those quantum effects, and therefore the classic hardware can not be made smaller and smaller indefinitely without hitting the boundaries where quantum effects come into the picture.

When a qubit is measured, it will return '0' or '1' and not something in between. The relation between the superposition state of the qubit and the actual value when it is measured is explained in the next chapter. Very roughly, the superposition relates to how likely it is that a given qubit, when measured, will hold the value '0' or the value '1'.

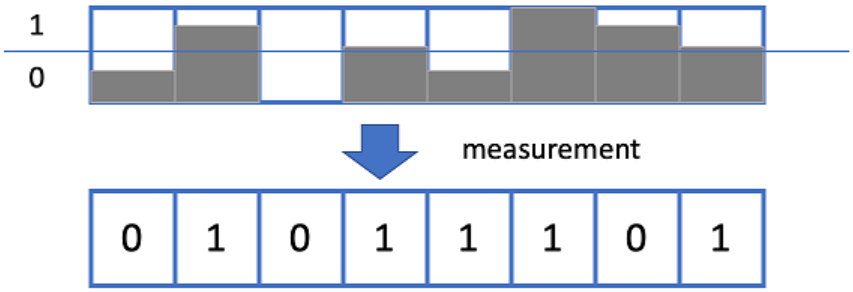

Figure 3.3 When measured, qubits fall back to either '0' or '1'.

We will discuss the idea of "superpositions" of the '0' and '1' state in the next chapter. For now, the most important part is that as a consequence, a number of qubits can contain more information than the same number of classical bits since a single qubit contains more complex information than simply '0' or '1'. This is important for problems or algorithms that theoretically require exponentially more bits for linear increasingly complexity.

3.2 Qubit notations

Although we didn't discuss superposition in detail yet, the previous paragraphs means that it is not (always) possible to identify the state of a qubit with a single *0* or a *1*.

There are different notations for a Qubit. Depending on the use case (e.g. showing the state of a circuit, explaining how gates work), one notation may be preferred over another notation. We will now cover two different notations, the **Dirac** notation and the **vector** notation. We will only

cover the simple cases in this chapter. Once we discussed superposition in the next chapter, we will come back to these notations and extend them. For now, we only consider the basis states of a Qubit, which represent the values *0* and *1*.

3.2.1 One qubit

SIDEBAR **Linear algebra**

At this point, we will sometimes use concepts and notations that are taken from linear algebra. If you want to get more background on these concepts, you can first read Appendix B to get a short introduction on the linear algebra we use in this book.

For the simple case where a Qubit is in one of its basic states, the vector representation of a single qubit is very straightforward. We represent the qubit as a vector with 2 elements. If the qubit holds the value *0*, the first element in the vector is *1*, and the other is *0*, as shown in Equation 3.1

EQUATION **3.1**

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

The Dirac notation of this qubit is as follows:

EQUATION **3.2**

$|0\rangle$

Since both notations are interchangable, we can also write

EQUATION **3.3**

$$|0> = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Similar, if the qubit holds the value *1*, we can represent it in a vector where the first element is *0*

and the second element is *1*. The Dirac representation of this single qubit is |1>, hence the representations can be written as

EQUATION 3.4

$$|1>=\begin{bmatrix}0\\1\end{bmatrix}$$

3.2.2 Multiple qubits

In a system with more than one qubit, the state of the qubits in the Dirac notation is achieved by concatting the individual qubits. For example, 2 qubits, each holding the value of *0* can be described by

EQUATION 3.5

|0>|0>

This is often abbreviated as follows:

EQUATION 3.6

|00>

The vector notation of a multiple qubit system requires some vector operations. The resulting vector, representing the multiple qubit system, is obtained by the tensor multiplication of the vectors of each qubit. Tensor multiplication is explained in Appendix B. Although it helps providing more insight, you do not need to know *how* those vectors are obtained.

EQUATION 3.7

$$|00>=\begin{bmatrix}1\\0\end{bmatrix}\otimes\begin{bmatrix}1\\0\end{bmatrix}=\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$$

In a system where the first qubit is *1* and the second qubit is *0*, the notation of the qubits is as

follows:

EQUATION 3.8

$$|10>= \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$$

SIDEBAR

From binary to decimal

Classic computers work with bits which are '0' or '1' but combined they can represent more complex information. A decimal number (e.g. '14') can be described by a number of bits. When bits are put in a sequential order, they can considered to be indicators as follows:

EQUATION 3.9

.

$$0\ 1\ 0\ 1 = 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

$$= 0 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 5$$

SIDEBAR

Hence, each bit in the sequence indicates whether or not a corresponding power of 2 should be added to the decimal number. When a bit is '1', the corresponding power of 2 is added, when the bit is '0' it is not added. The most right bit of a sequence is said to have an index of '0'. The bit left of it has index '1' and so on. In general, a bit with index 'i' corresponds with the value of '2^i'.

There is another handy relation between the Dirac notation and the vector notation. If we would consider the qubits as bits, the bits in the Dirac notation would equal an integer value, e.g.

EQUATION 3.10

.

(1) |00> = 0

(2) |01> = 1

(3) |10> = 2

(4) |11> = 3

The only element which has the value of *1* in the corresponding vector notation occurs at the position indicated by this integer value, assuming we start to count from position 0. Indeed, as shown above, |10> corresponds to a vector whose third element equals *1*.

Hence, if we read the bits in the Dirac notation as a decimal number, say *n*, the corresponding vector will be a vector with all zeroes, expect for the element at position *n* (starting from 0), which will be '1'.

If we add another qubit to the system, we need to add another tensor multiplication. For example, a 3-qubit system where the first qubit is *1*, the second qubit = *0* and the third qubit is *1* can be represented as follows:

EQUATION 3.11

$$|101> = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\\0\\0\end{bmatrix} = \begin{bmatrix}0\\0\\0\\0\\0\\1\\0\\0\end{bmatrix}$$

Note that the above mentioned relation still holds: |101> is the digital representation of the integer *5*, and if we start counting the first row in the vector as row *0*, the element at row *5* equals *1*.

The size of the resulting vector quickly grows when the number of bits increases. In general, for *n* bits, the resulting vector contains 2^n elements.

One may wonder why we make it so complex. Why do we need a vector with 8 elements when we just use 3 qubits, and only one out of those 8 elements is 1? The answer will be given in the next chapter. So far, we only discussed qubits in a basic state. Once we talk about qubits in a superposition state, it will become very useful and even required to represent qubits in this way.

SIDEBAR **Physical representations of a qubit**

Although it should not influence the behavior of an application, it is interesting to have a rough idea on how bits or qubits are created and maintained in the real, physical world. It is important to realize that there are different options for the physical realisations of bits and qubits, and that developers are abstracted away from those physical realisations. For example, a bit stored in the main memory of a computer can be realised by an electrical pulse that keeps the bit "on". When a bit is stored on a hard disk, a different technique is used, for example leveraging magnetic properties.

The general principle for storing qubits is similar to the principle for storing bits: we leverage phenomenons that are encountered in nature, and apply them to our goal. The electrical pulse that can be used to keep a bit "on" (giving it the value of 1) is a classic example of this. Quantum phenomenons that describe a two-state system can be used to represent qubits. In such a system, the state is not simply 0 or 1, but it can be in a more complex "superposition" of 0 and 1. We will discuss superposition in chapter 4. The important thing to understand is that there are physical phenomenons that exactly represent the behavior of a qubit. This is not a coincidence of course, and the statement could very well be reversed into "qubits behave exactly similar to some phenomenons encountered in quantum mechanics."

There are different physical phenomenons that lead to a quantum two-state system. The wikipedia en.wikipedia.org/wiki/Quantum_computing#Developments lists those options. At this moment, most of the efforts for creating and manipulating qubits are based on superconducting electronic circuits. The physical superconducting qubits that can be created in a superconducting environment can have different characteristics, so there are still different possible implementations of qubits leveraging superconducting circuits. Most important to developers, though, is that this context allows to create qubits that can be in a superposition until they are measured—which is what we will describe in Chapter 4.

3.3 Gates: Manipulating and measuring qubits

Being able to represent and store data is fine, but in computing, we need to be able to manipulate data. Forms need to be processed, interest rates need to be applied, colors need to change,… and so on; all kinds of operations are possible on data. In a high-level software language like Java, there are a huge amount of libraries that somehow manipulate input data. At the lowest level, all these operations come down to a sequence of simple manipulations of the bits in the computer systems. Those low-level operations are achieved using gates. It can be shown that with a limited number of gates, all possible scenario's can be achieved.

Gates are typically represented using simple pictures. A very simple classical gate is the NOT gate, also known as the inverter.

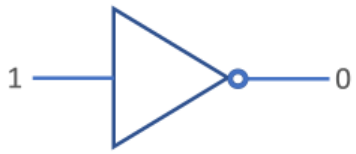

Figure 3.4 Representation of the NOT gate

This gate has one input bit, and one output bit. The output bit of the gate is the inverse of the input bit. If the input is "0", the output will be "1". If the input is "1", the output will be "0".

The behavior of gates is often explained via simple tables where the possible combinations of input bits are listed, and the resulting output is listed in the last column. The following table shows the behavior of the NOT gate:

Table 3.1 Behavior of the NOT gate

input	output
A	NOT A
0	1
1	0

When the input of the gate is '0', the output is '1'. When the input of the gate is '1', the output is '0'.

The NOT gate involves a single bit only, but other gates involve more bits. The XOR gate, for example, takes the input of 2 bits, and outputs a value that is '1' in case exactly one of the 2 input bits is 1 and the other is '0'.

Figure 3.5 Representation of the XOR gate

The following table shows the behavior of the XOR gate:

Table 3.2 Behavior of the XOR gate

input		output
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Quantum gates have similar characteristics as classical gates, but there are also important differences. Similar to classical gates, quantum gates operate on the core concept, in this case on the qubits. They can alter the value of qubits. One of the important differences between classical gates and quantum gates though is that quantum gates should be reversible. That is, it should always be possible to apply another gate and go back to the state of the system before the first gate was applied. This restriction is not in place with classic gates. For example, the XOR gate is not reversible. If the result of an XOR gate is '1', it is impossible to know whether the first bit was '0', or whether it was the second bit.

Because of the need for gate operations to be reversible, a quantum system needs different gates than a classical system. Therefore, low-level quantum applications require a different approach then low-level classical applications.

3.4 A very first [quantum] gate: the Pauli-X gate

Suppose you work for a bank. You managed to create a system that stores data (account numbers and balances). Now you are asked to modify balances, e.g. apply an interest to a balance. That means you need to manipulate data. How will you do this?

One of the core ideas of software development is to write functionality that manipulates data, e.g. "add one EUR to all balances". This requires the ability to modify data, and this is what happens at a huge scale in classic computers.

If we want Quantum computers to execute your algorithm, those computers should be able to manipulate data. This is what, at a low level, is done by quantum gates.

A first example of a quantum gate is the Pauli-X gate.

Figure 3.6 Symbol of the Pauli-X gate

This gate 'inverts' the value of a qubit. When we delve into superposition in the next chapter, we will come back to this example. For now, we only take the special cases into account, where a qubit is in the '0' or in the '1' state. The Pauli-X gate, will flip the value of '0' into '1' and vice versa, as shown in the following picture.

This process is reversible. If the value of a qubit, after a Pauli-X gate has been applied, is '1', we know it had the value of '0' before the gate was applied. If on the other hand the end value is '0', we know the original value was '1'. Hence, the principle of reversible gates holds so far. By applying a second Pauli-X gate after applying the first Pauli-X gate, the original state of the system is restored, as explained in 3.7.

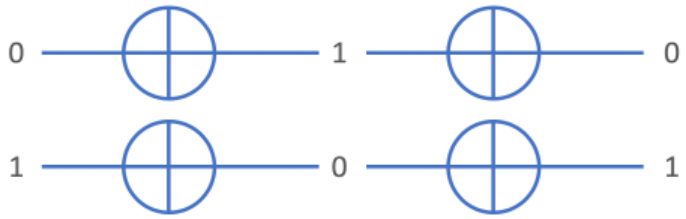

Figure 3.7 Two Pauli-X gates restore the system

3.5 Playing with Qubits in Strange

We haven’t discussed superposition and entanglement yet, and our introduction to qubits was only very basic. At this point, however, we create a very simple application using Strange to see the Pauli-X gate in action.

In Chapter 2, we used the high-level API of Strange to create an application that uses a quantum algorithm return random values. In the following demo, we will use the low-level API of Strange and directly work with Qubits and Gates.

The code in Listing 3.1 will create a single Qubit (which has the initial value of '0'), apply the Pauli-X gate, and measure the resulting value.

TIP	The source code for this demo can be found in the ch03/paulix directory in the sample repository. See Appendix A for more information on how to obtain the sample code.

Listing 3.1 Java application using aPauli-X gate.

```
public static void main(String[] args) {
    QuantumExecutionEnvironment simulator =
        new SimpleQuantumExecutionEnvironment();          ❶
    Program program = new Program(1);                     ❷
    Step step = new Step();
    step.addGate(new X(0));
    program.addStep(step);
    Result result = simulator.runProgram(program);        ❸
    Qubit[] qubits = result.getQubits();
    Qubit zero = qubits[0];
    int value = zero.measure();
    System.out.println("Value = "+value);                 ❹
}
```

❶ an Environment is created for declaring and executing a quantum application.

❷ a Program is defined.

❸ after the Program is defined, it can be executed on the Environment and a Result can be obtained.

❹ the Result can be processed and returned to the user.

Now, let's run the code!

```
./gradlew run
```

As could be expected, the output of the program is as follows

```
Value = 1
```

In this code, we introduce a number of concepts encountered in Strange. We talk about an execution environment, a program consisting of steps, and some results. Note that those concepts are typically used in all kinds of Quantum Computing simulators and editors.

3.5.1 QuantumExecutionEnvironment

The physical location and conditions of where and how a Quantum application is executed are not relevant to the developer and the options are still evolving. There are already cloud services offering real Quantum infrastructure (e.g. IBM, Rigetti), but it also possible to assume a Quantum co-processor will be able to execute Quantum applications. Today, most Quantum applications are executed on Quantum Simulators, which can run either locally or in a cloud enviroment.

In summary, this means that there might be a number of completely different execution environments that are capable of executing Quantum applications.

Strange abstracts the differences in execution environments, and provide an interface `QuantumExecutionEnvironment` in the `com.gluonhq.strange` package which provides the

API for Quantum applications to interact with the execution environment. Strange contains a number of implementations of this `QuantumExecutionEnvironment` but the most important thing is that Quantum applications written with Strange can run on all current and future implementations without being modified.

The simplest execution environment is using a built-in simulator, and it is instantiated using

```
QuantumExecutionEnvironment simulator = new
SimpleQuantumExecutionEnvironment();
```

The `SimpleQuantumExecutionEnvironment` which is in the `com.gluonhq.strange.local` package provides a quantum simulator which executes quantum operations using classical software. Clearly, it is slower than real hardware, and since Quantum simulators are memory hungry when dealing with large numbers of qubits, it is not recommended to be used with lots of qubits.

For the demos in this book, the `SimpleQuantumExecutionEnvironment` is more than good enough. We will talk about other execution environments in Chapter XX.

3.5.2 Program

If you want to create a Quantum application in Strange, you have to create a new instance of `Program`. The `Program` class is in the `com.gluonhq.strange` package, and it provides an entry point to quantum applications you want to write.

The `Program` constructor requires a single integer parameter, defining the number of qubits you will use in this application.

In the case of our simple application, we will only use a single qubit, which explains the

```
Program program = new Program(1)
```

line.

3.5.3 Steps and Gates

SIDEBAR **What is a Quantum Program**

A `Program` is composed by one or more steps operating on the qubit.

Each step is defined by an instance of `Step`. The `Step` class is in the `com.gluonhq.strange` package as well, and has a zero-argument constructor. Inside a step, you define which gates are used.

In our sample, we have a single step that is created by

```
Step step = new Step()
```

and that is further defined by adding a gate. The gate we use here is the Pauli-X gate, which is defined by the `X` class in the `com.gluonhq.strange.gate` package. The constructor of the Pauli-X gate requires one integer to be passed, which is the index of the qubit the gate is acting on. In this case, since we have a single qubit only, the index is `0`.

Creating this gate, and adding it to the `Step` instance we just created is thus done via

```
step.addGate(new X(0));
```

In a single step, each qubit may be affected by not more than one gate. A gate may act on more than one qubit, but two gates in the same step can not act on the same qubit. For example, the following code snippet is wrong, as we add two gates to the same step, and both gates operate on the same qubit (with index '0');

```
step.addGate(new X(0));
step.addGate(new H(0));
```

Note that we introduced another gate here, the Hadamard gate represented by the `H` class. We will cover this gate in the next chapter, and only used it here to show that it is not allowed to have two gates operating on the same qubit in a single step.

At this point, the single execution step in our program is ready. We have to instruct the `Program` instance that our `Step` instance should be added to the program, which is done by

```
program.addStep(step);
```

3.5.4 Results

We briefly mentioned in this chapter that a Qubit can be in a so-called superpositions, but once it is measured, it will either hold the value '0' or the value '1'. Therefore, it is impossible to have intermediate results in Quantum applications. Quantum simulators that are not using real phyical qubits, do not have this restriction though, so for debugging purposes intermediate values can be used and can be useful, as we will demonstrate later.

When a Quantum application or a `Program` has been executed, a result can be obtained. Strange defines the `Result` class in the `com.gluonhq.strange` package, and instances of it are created by the execution environment. The result is returned when the `runProgram()` method is called on the `QuantumExecutionEnvironment`.

```
Result result = simulator.runProgram(program);
```

The resulting instance of the `Result` class contains information about the final state of the quantum system. We will talk about this in more detail in the next chapters. For now, we are

only interested in the status of the single qubit that is in our system.

The `Result` class contains a method to retrieve the qubits:

```
Qubit[] qubits = result.getQubits();
```

Since we only have one qubit in the system, it can be obtained as follows:

```
Qubit zero = qubits[0];
```

We can now ask for the value of this qubit after the program has been executed:

```
int value = zero.measure();
```

Finally, we print the value using simple Java commands:

```
System.out.println("Value = "+value);
```

Initially, qubits are in the '0' state. Our simple application sends the qubit through a Pauli-X gate, and then measures the new value, which is always equal to '1'.

3.6 Visualisation of Quantum circuits

The code in the snippet above is not hard to understand and easy to follow, but it represents a very simple Quantum circuit with only a single qubit and a single gate being involved.

Once the applications become more complex, it might be difficult to read the code and have a clear understanding what is happening. Many quantum simulators or applications that allow to generate quantum applications therefore come with a visualisation tool.

The Strange library has a companion library called StrangeFX which allows to render programs in an intuitive way. StrangeFX is written in Java as well, and it uses JavaFX, the standard Java UI Platform, for rendering.

The example in this chapter named 'paulixui' shows this library in action.

Once you have a Program, it is very easy to visualise it. You just have to modify the `build.gradle` file and add a dependency to StrangeFX. The build.gradle now looks as follows:

```
plugins {
    id 'application'
    id 'org.openjfx.javafxplugin' version '0.0.6'
}

repositories {
    jcenter();
}

dependencies {
    compile 'com.gluonhq:strange:0.0.6'
```

```
    compile 'com.gluonhq:strangefx:0.0.1'
}

javafx {
    modules = [ 'javafx.controls' ]
}

mainClassName = 'com.gluonhq.javaqc.ch03.paulixui.Main'
```

Note that we added the

```
id 'org.openjfx.javafxplugin' version '0.0.6'
```

line to the plugins. This plugin will make sure all the code required for running JavaFX application can be used. Further, we have to add the depdendency to StrangeFX to the list of dependencies:

```
compile 'com.gluonhq:strangefx:0.0.1'
```

Finally, since our application will now use the JavaFX Controls module, we have to tell the Java system to load this module:

```
javafx {
    modules = [ 'javafx.controls' ]
}
```

The code required for rendering a program is very simple. StangeFX contains a class `com.gluonhq.strangefx.render.Renderer` that has a static method

```
Renderer.renderProgram(Program program);
```

This method will analyse the program, and create a visual representation of the program where each qubit is represented on a line. The initial state, with all qubits in the |0> state, is on the left. Going to the right, quantum gates are pictured when they are encountered. At the end of the line, the probability of this gate being measured with `1` is shown.

Hence, if we want to render the circuit we composed before, we have to modify the end of our application as follows:

```
    int value = zero.measure();
    System.out.println("Value = "+value);
    Renderer.renderProgram(program);
}
```

Running this program results in the following user interface being rendered:

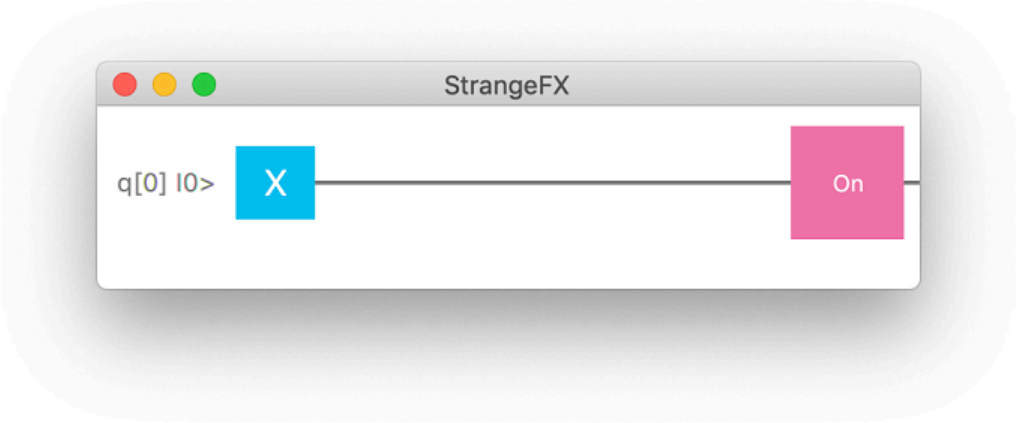

Figure 3.8 UI representation of a qingle qubit with a Pauli X gate

In this diagram, the visual components refer to the different components of the application, as explained in Figure 3.1.

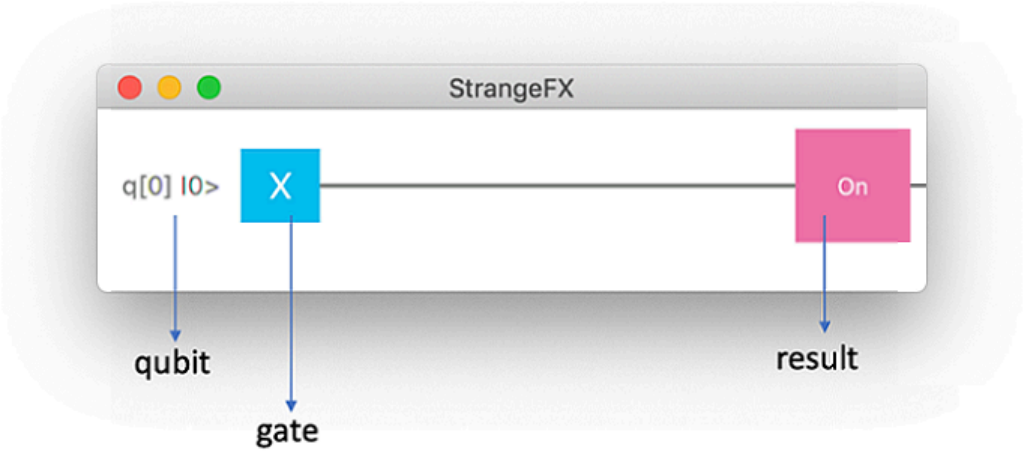

Figure 3.9 Explaining the different components in the StrangeFX screenshot.

3.7 What did we learn?

In this chapter, we introduced the most fundamental concepts of quantum computing: qubits (or quantum bits) and quantum gates. We showed similarities and differences between those concepts and their counterparts in classical computing. We introduced two different notations for qubits. We didn't really touch the key reasons of why qubits are so powerful, and that is what we will do in the next chapters.

We created a simple application that introduced us to some of the core aspects of working with Quantum simulators and related software.

4 Superposition

This chapter covers:

- we introduce the term "superposition"
- we explain why superposition allows for much more (*exponentially more*) data in quantum system to be processed
- we explain that processing is done via quantum gates
- we show that quantum gates can be represented by matrix operations
- we introduce the Hadamard gate, which brings a qubit in a superposition state.
- we show code that applies the Hadamard gate, and measure the resulting value

In the previous chapter, we briefly mentioned superposition. It is one of the most fundamental concepts of Quantum Computing, and it is one of the reasons why Quantum Computers are expected to be able to run some applications much faster than classical computers.

In this chapter you will learn what superposition is, and how it is relevant in creating quantum algorithms. We will talk about a specific gate that brings a qubit in a superposition state, and we show a simple but very relevant sample that demonstrates superposition.

We try to keep the physical explanations to a minimal. The scientific work behind the physics is mind boggling, but it requires different skills and is less relevant to software development. Keep in mind that even for the most knowledgable persons Quantum Computing and its concepts are very difficult to grasp, so do not worry when the physical concepts behind superposition are not clear. What matters to the developer is how to use these concepts and write more suitable applications.

The flow of the chapter is explained in Figure 4.1.

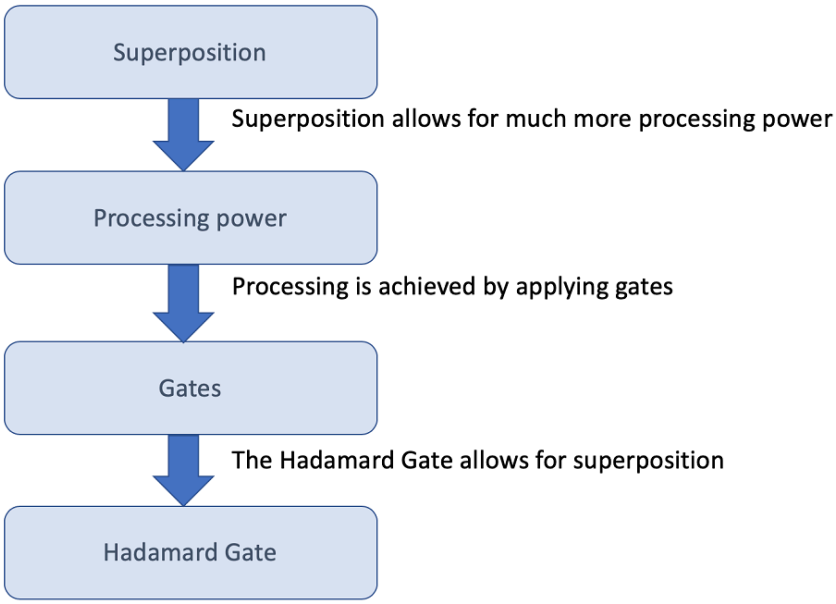

Figure 4.1 From superposition to the Hadamard gate.

4.1 What is superposition

A qubit can be in different states. We mentioned before that a qubit can hold the value '0', the value '1', but also some sort of a combination of the value '0' and '1'. There are some important restrictions on what combination are allowed though, and we will discuss these now.

We said before that when a qubit is measured, it will always return the value `0` or the value `1`. But that doesn't say everything about what is happening **before** we measure it.

In order to understand this, we'll make a short detour to the world of quantum mechanics. Remember that what software developers call a "qubit" is backed by some real-world phenomenon. The software behavior and properties of a qubit therefore have to correspond somehow with the behavior and properties of the real-world phenomena.

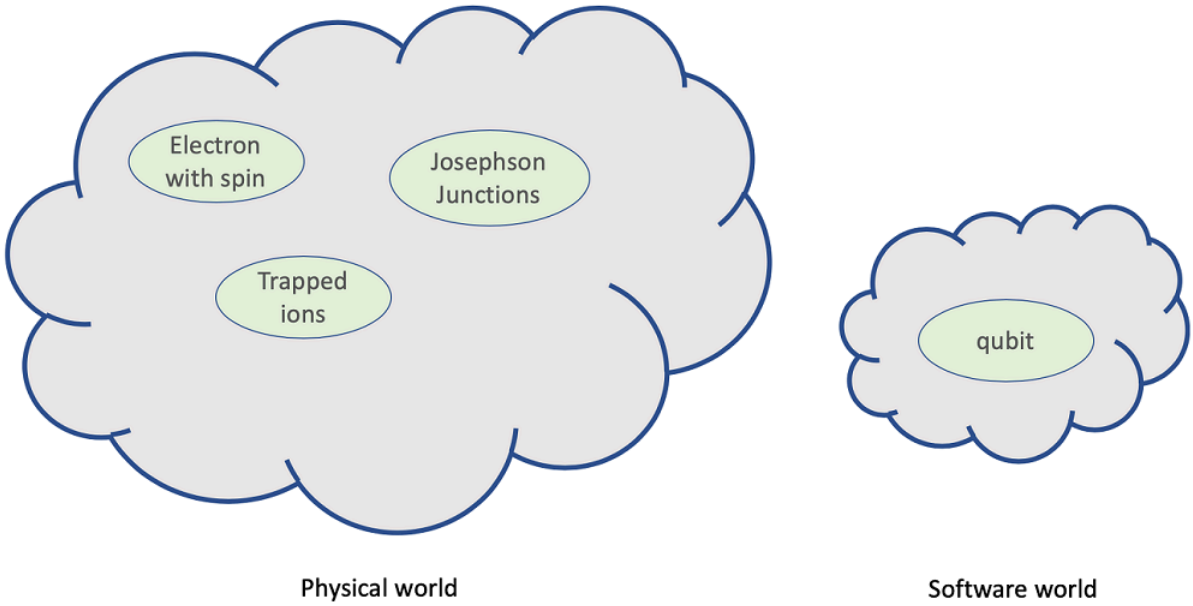

Figure 4.2 The characteristics of physical particles match the characteristics software qubits.

In quantum mechanics, some particles have interesting properties. An electron, for example, has a property called "spin". When measured, this property can have two states: 'up' and 'down'. Note the similarity with bits so far, who can be '1' (corresponding to 'up') or '0' (corresponding to 'down'). This is shown in Figure 4.3.

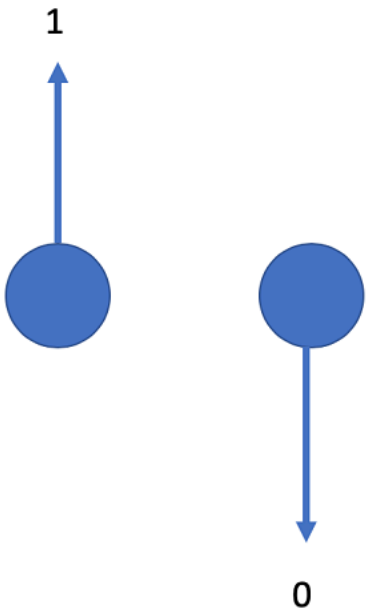

Figure 4.3 An electron spin that is up (left) and another one that is down (right), corresponding to 1 and 0 respectively.

The quantum theory, however, describes that the spin of an electron can also be in a so-called superposition of the 'up' and 'down' states --- which we call the basis states. Symbolically, this can be represented by Figure 4.4.

Figure 4.4 An electron spin that is in a superposition of up and down

Again, when it is measured, it always falls down to one of those two basis states.

There are a number of misconceptions about superposition state.

1. Being in a superposition does not mean that the electron is both in the `0` (spin down) state or the `1` (spin up) state. Actually, the theory of superposition does not say in what state it is, it rather describes the probability of states when it was going to be measured at that particular point.
2. Being in a superposition does not mean that the spin of the electron is in either the `up` or the `down` state and that we simply don't know yet. One of the fundamental (and weird) things of quantum computing is that a system is influenced when it is measured. It is only when measuring the spin that it takes a "decision" to be in the `up` or the `down` state.

From the previous chapter, you remember that a qubit that holds the value *0* is described in the Dirac notation as `|0>` . In case the corresponding physical element is an electron, you can say that this is similar to the electron having a 'spin down' property. Similar, when the qubit holds the value *1*, this is described in the Diract notation as `|1>` which can correspond to the real-world situtation where an electron has a 'spin down' property

> **NOTE** Most existing prototypes for quantum computers are not using electrons as qubit representations. However, the spin up/down property of an electron is often easier to understand than the more complex phenomena used by most of the quantum computers (e.g. Josephson junctions). Since we try to make abstraction of the physical background as much as possible, we prefer to make the analogy by using a more simple physical representation.

As we explained, an electron can be in a spin up or a spin down state, but its spin can also be in a superposition of the up and down states. As a consequence, a qubit can be in a "superposition" as well.

We now give the qubit a name, similar to how we give variables or parameters in classical programs names. Greek symbols are often used for this, and in order to be consistent with most

of the literature, we will use those symbols as well. Hence, a qubit named ψ (pronounced 'psai') that holds the value *0* (corresponding to an electron with spin down) is desribed as follows:

EQUATION 4.1

$$|\psi> = |0>$$

Similar, when the qubit named ψ holds the value 1 (corresponding to an electron with spin up), you can describe it as

EQUATION 4.2

$$|\psi> = |1>$$

The interesting thing is the description of a qubit in a superposition state, corresponding to an electron with a spin that is in a superposition of up and down. When a qubit is in a superposition state, this state can be described as a linear combination of the basis states:

EQUATION 4.3

$$|\psi> = \alpha|0> + \beta|1>$$

The equation tells you that the state of the qubit is a linear combination of the basis state $|0>$ and the basis state $|1>$ with α and β being numbers related to probabilities as we will explain shortly.

This equation highlights one of the fundamental differences between classical computing and quantum computing. While the "simple" cases of a qubit holding the value *0* or the value *1* can also be reproduced with classical variables, the combination of both values is impossible for a classical computer, as explained in Figure 4.5.

Quantum computer	Classical computer
$\|\psi> = \|0>$	boolean a = false
$\|\psi> = \|1>$	boolean a = true
$\|\psi> = \alpha\|0> + \beta\|1>$	boolean a = ????

Figure 4.5 Variable assignments in quantum computers versus classical computers.

You can also write Equation 4.3 in vector notation. Leveraging the definitions of the Dirac notations for $|0>$ and $|1>$ you can rewrite equation 4.3 as follows:

EQUATION 4.4

$$|\psi >= \alpha \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

The Dirac notation and the vector notation refer to the same principle: the considered qubit is in a superposition of the $|0>$ state and the $|1>$ state.

There are a number of ways that try to explain what this equation pysically means. At its core, the equation means that the electron is in such a state that, when it would be measured at that moment, there is a probablity of α^2 that we will measure `0` and a probability of β^2 that we will measure `1`.

Since we will measure either `0` or `1`, there is an additional restriction on the values of α and β. The sum of the probabilities should be 1 (since you measure something).

Hence,

EQUATION 4.5

$$\alpha^2 + \beta^2 = 1$$

As we said a number of times before, understanding quantum mechanics is very hard. Fortunately, as a developer, you only have to take into account the equations, and leave out the physical interpretation.

What we described above for the spin of electrons also applies for other properties of other elementary particles. When we talk about a `Qubit`, its underlying physical implementation leverages the behavior of these properties. As a developer, you are shielded away from the physical behavior. Hence, when we talk about a Qubit in a superposition, the developer does not need to know anything about the physical representation of this qubit.

At this point, one of the most common questions asked by developers is the following:

"*Great, a qubit can be in a superposition of 0 and 1, but when we measure it, is is still either 0 or 1. What's the difference with a classical computer then?*"

This is a very reasonable question, and we'll give the answer in the next sections.

> **NOTE** Quantum computing is sometimes linked to working with probabilities instead of working with certainties. A classical bit is either 0 or 1 and can always be measured. As you just learned, in quantum computing the state of a system is rather described by probabilities. This requires a different way of thinking.

4.2 The state of a quantum system as a probability vector

So far, we mainly talked about qubits, and the values that those qubits hold. After introducing superposition, you now know that a single qubit can be in a combination of 2 base states during processing, and it will fall back to one of the base states (either *0* or *1*) when measured. In classical computing, the value of parameters is the most important concept in processing. When talking about quantum computers, however, those values are not uniquely defined during processing — due to superposition. Therefore, it is often more convenient to talk about *probabilities* instead of values of qubits. This is what we explain in this section, and one of the consequences is the processing power of quantum computers.

In the previous chapter, you learned that the state of a quantum system can be represented by a vector. For a quantum system with 1 qubit, a vector with 2 elements describes the probalities for the value of that single qubit, when it would be measured. A quantum system with 2 qubits can be represented by a vector with 4 elements, and in general a quantum system with n qubits is represented by a vector with 2^n elements. Figure 4.6 explains this principle.

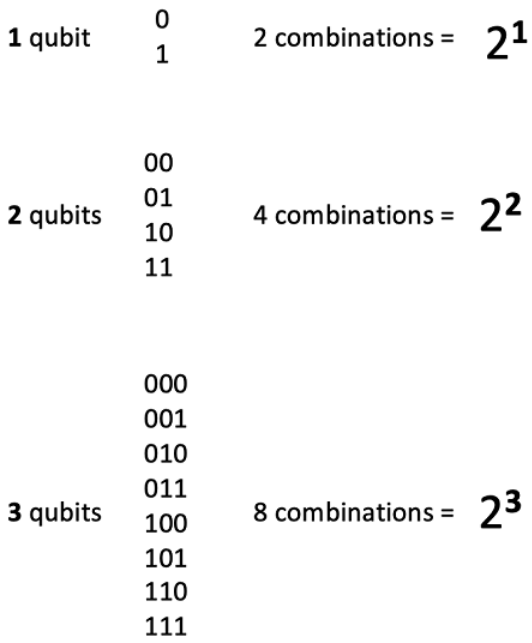

Figure 4.6 With a growing number of qubits, an exponentially growing number of combinations is possible.

The probability vectors we showed in Chapter 3 representing the state of a quantum system with all qubits in a base state were very simple: all elements are 0, expect for one element. That element defines the state of the system and it correspond to a clear value for each qubit: either 0 or 1.

You now learned that a qubit can be in a superposition, in which its state is a linear combination of the 0 value and the 1 value.

For a single qubit system, the state can be described in Dirac Notation and in vector notation as follows:

EQUATION 4.6

$$|\psi >= \alpha|0 > +\beta|1 >= \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

A system with two qubits can be described as follows:

EQUATION 4.7

$$|\psi_0\psi_1 >= \begin{bmatrix} \alpha_0 \\ \beta_0 \end{bmatrix} \otimes \begin{bmatrix} \alpha_0 \\ \beta_0 \end{bmatrix} = \begin{bmatrix} \alpha_0\alpha_1 \\ \alpha_0\beta_1 \\ \beta_0\alpha_1 \\ \beta_0\beta_1 \end{bmatrix}$$

This equation shows that a system with two qubits can be described by a (probability) vector with 4 values. Two qubits can hold 4 values simultaneously. Of course, once measured, only 1 value for each qubit remains. But all the computations in a quantum algorithm operate on the 4 values.

By extension, a system with *n* qubits corresponds to a vector with 2^n elements. This is an indication why quantum computers are expected to help with *exponential* problems: with an increasing number of qubits, a quantum system can work with an exponentially increasing number of values.

In Figure 4.1 we show a system with 6 qubits.

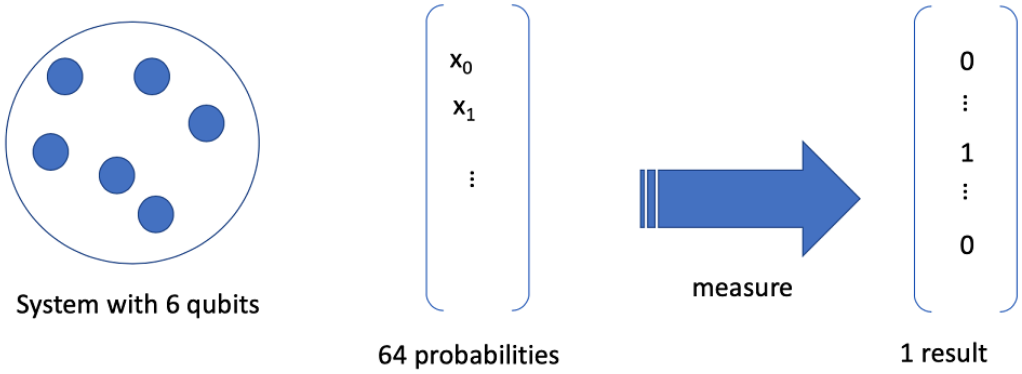

Figure 4.7 Quantum system with 6 qubits.

In this picture, it is shown that 6 qubits correspond to a vector with 64 (which is 2^6) elements. Once measured, one of these elements holds the value 1, and all other elements have the value 0. From the index of the element that holds the value 1, the individual values for the 6 qubits can be calculated. Hence, you started with 6 values (each either 0 or 1) and you ended with 6 values (each either 0 or 1).

At first sight, there might be no clear value due to superposition. You can hold an exponential number of elements in the system, but once you measure it, it seems you are back in the classical state where each bit has exactly one well defined value.

SIDEBAR

The real value lies in the fact that a quantum system can do processing while the qubits are in a superposition state. Hence, the operations defined by the quantum algorithms do not manipulate just 6 bits, they manipulate 64 probability values. One step in a quantum algorithm on a quantum computer with 6 qubits is therefore modifying 64 values. Adding one qubit doubles the processing capabilities of the quantum computer. This explains the term "exponential" that is often used together with quantum computing: adding n qubits adds processing power proportional to 2^n, where the n is in the exponent of this equation.

Let's compare a classical computer with 6 bits to a quantum computer with 6 qubits. Both computers have 1 value, consisting of 6 bits as input to an algorithm, and after measuring the output of the algorithm, they will both read a value of 6 bits again. Both computers can process 64 possible combinations as an input value. The key difference is that a quantum computer can process those 64 combinations **at the same time**.

This is shown in Figure 4.2

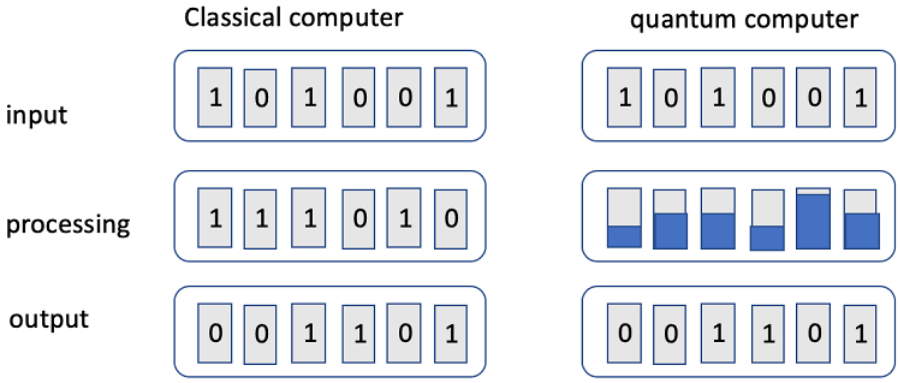

Figure 4.8 Comparing a classical computer and a quantum computer with 6 bits input and output

We can show this with some Java code. First, let's assume you use a classical computer with a single bit, and you apply a function to that bit. You will use the *boolean* type as this is the Java primitive type that can hold 2 values: *false* and *true*, corresponding to *0* and *1*.

```
   boolean input;
   boolean output;

   output = someFunction(input);
```

where *someFunction* is a Java function with the following signature:

```
public boolean someFunction(boolean v) {
    boolean answer;
    ... // do some processing                    ❶
    return answer;
}
```

❶ the real processing is done here.

If you have to apply *someFunction* to all possible values of *input*, you have to invoke the function twice:

```
   boolean[] input = new boolean[2];
   boolean[] output = new boolean[2];
   input[0] = false;
   input[1] = true;

   for (int i = 0; i < 2; i++) {
       output[i] = someFunction(input[i]);
   }
```

You will now do the same for a Quantum Computer with some pseudo Java code.

> **NOTE** For this sample, you will not use real Java code, since we will make a number of simplications. The code below is pseudo-code for a number of reasons. There are more differences between classical algorithms and quantum algorithms than only the concept of superposition, as we will see in the next chapters. One of the other important differences that we will explain in the next chapter is that qubits don't operate in an isolated way. An operation on one qubit may affect another, seemingly unrelated qubit. As a consequence, when doing operations on qubits, the whole system (all qubits) need to be taken into consideration.

You create an instance of a Qubit, and bring it into superposition with a fictive *superposition* method. Later in this chapter we explain *how* a qubit can be brought in a superposition.

```
Qubit qubit = new Qubit();
qubit.superposition();
```

The *someFunction* now has to work with a qubit, hence you define it as follows:

```
public Qubit someFunction(Qubit v) {
    Qubit answer;
    ... // do some processing
    return answer;
}
```

So far, this looks very similar to the classical case. However, you can now evaluate the case where the qubit is *0* and the case where the qubit holds the value *1* with a single function evaluation by applying the function to the qubit in superposition:

```
Qubit qubit = new Qubit();
qubit.superposition();
qubit = someFunction(qubit);
```

The key element here is that the function *someFunction* takes a qubit as input, and has a qubit as output. If the input qubit is in a superposition state, the function operates on both states. Similarly, if you have a function that has 2 qubits as input, it can operate on the 4 different combinations of the qubit states. In general, a function operating on n qubits can operate on 2^n possible states. This explains why the probability vectors are often used when talking about quantum computing, as those vectors have 2^n elements, describing the probablity to find the qubits in a specific state.

Now that you know why quantum computers allow for exponentially scaling, you need to find out how to benefit from this, as the exponential power only applies during processing — and not during measurement.

The trick when writing quantum algorithms is to come up with those operations that, when applied, will lead to a measurement that tells more about the solution of a problem.

This is shown in Figure 4.3.

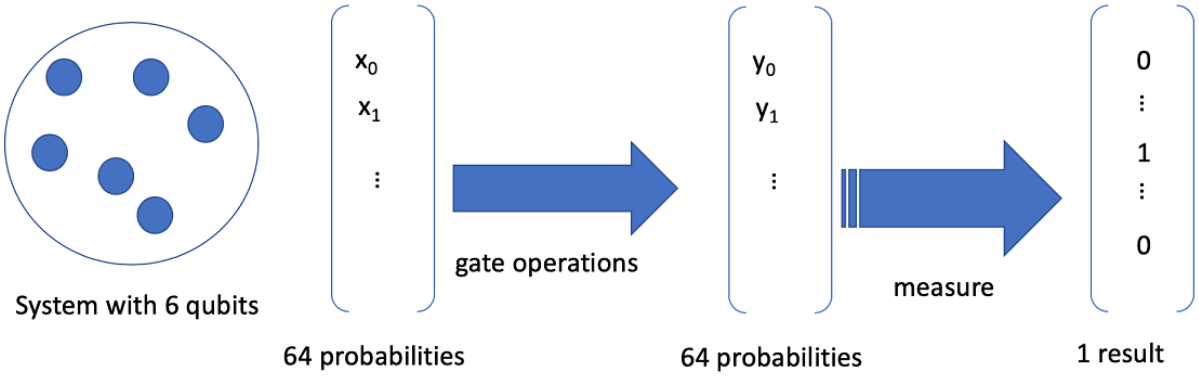

Figure 4.9 Gates applied to a quantum system with 6 qubits.

In this picture, we showed that before the quantum system is measured, processing is done by applying quantum gates. Those quantum gate operations modify the state of the probability vector.

> **NOTE** an analogy to this is the following: suppose that someone gives you 1000 numbers, and tells you that one of those number is a prime number. You have to find the index of the prime number. Imagine you could manipulate all those numbers simultaneously, and process them in a way that all numbers become 0, except for the prime number which becomes 1. A single measurement then reveals the position of the prime number. While there is no easy quantum algorithm for this, the analogy shows that there is a benefit in being able to process a large number of values, even if the end result is a single value only.

We talked a number of times about operations on quantum systems. This brings us closer to software, as we ultimately want to use software to manipulate the state of a quantum system.

Before we turn our attention to software, we will explain how quantum gates manipulate qubits and the probability vector.

4.3 Introducing matrix gate operations

We try to keep the mathematical parts in this book to a minimum. However, in order to understand the core concepts of Quantum Gates, it helps to have a basic understanding of linear algebra and matrix operations. In this section, we briefly describe the required background.

We will explain this using the most simple, yet useful quantum gate: the Pauli-X gate. After we discuss how the Pauli-X gate corresponds to a matrix operation, we will generalize the concept to

all gates.

In the beginning of this chapter, we showed the flow (see Figure 4.1). In this section, we will add some details on the `Gates` block in that picture, as explained in the following detail:

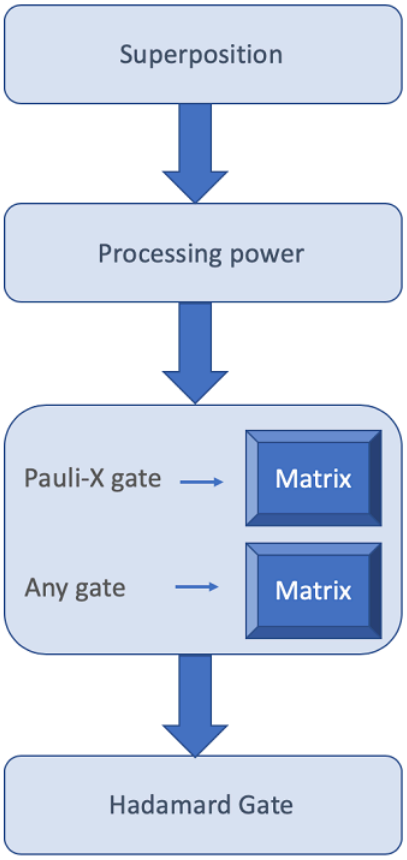

Figure 4.10 Detailing the gate concept from the flow: first discuss the Pauli-X gate, then make it more general.

These steps are needed in order to get at the final part of this chapter: the discussion of the Hadamard gate. The Pauli-X gate we will discuss now is an easy to understand gate, and you will get a clear understanding of how gates work after this section.

In the previous paragraph, we explained that we can represent the state of a system with n qubits with a vector containing 2^n elements. We mentioned that a quantum computer can do processing on this vector, and in this section we explain what we mean by this. The fact that a quantum computer can operate on a combination of states at the same moment is a great opportunity for performance, but it comes with some complexity: instead of thinking about individual qubits, you need to think about *probabilities* for the *combination* of qubits.

4.3.1 The Pauli-X gate as a matrix

In the previous chapter, we described the Pauli-X gate. We mentioned the Pauli-X gate has similarities with the classical NOT gate, and we used a simple table to explain the behavior of the NOT gate. For clarity, we repeat that behavior table here again:

Table 4.1 Behavior of the NOT gate

input	output
A	NOT A
0	1
1	0

This table would also make sense when talking about the Pauli-X gate, but it only takes into account the basis states, where the input is either '0' or '1'. As we mentioned earlier in this chapter, the general state of a Qubit can be a linear combination of the basis states. The state is not simple '0' or '1', but a combination of probabilities: the probability that if you measure the qubit you will measure '0' and the probability that you will measure '1'. In this case, a simple table is no longer sufficient to describe the behavior of a gate. You would need a table with an infinite number of rows as shown in the following table --- taking into account that there are already infinite values between 99% and 100%.

Table 4.2 Behavior of the NOT gate on a qubit, shown as a table.

input	output
A	NOT A
100% chance on 0, 0% change on 1	0% chance on 0, 100% change on 1
99% chance on 0, 1% change on 1	1% chance on 0, 99% change on 1
98% chance on 0, 2% change on 1	2% chance on 0, 98% change on 1
...	...
0% chance on 0, 100% change on 1	100% chance on 0, 0% change on 1

In Quantum computing, a typical way to describe gates is by using matrix operations.

4.3.2 Applying the Pauli-X gate to a qubit in superposition

The state of a quantum system with qubits can always be represented by a vector. When gates act upon qubits, the values in the vector change. In linear algebra, this can be achieved by representing the gate with a matrix and multiplying the matrix with the qubit vector in order to obtain the new state of the qubit vector.

We will now show that the Pauli-X gate can be represented by the following matrix:

EQUATION 4.8

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Let's start with something simple. First, suppose that the qubit originally holds the value '0'. You already learned in the previous chapter that after applying a Pauli-X gate to this qubit, the qubit will hold the value '1'.

In the Dirac notation, the qubit is originally written as $|0\rangle$. In vector representation, this corresponds to

EQUATION 4.9

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Applying a gate to a qubit corresponds to multiplying the gate matrix and the qubit vector.

EQUATION 4.10

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

In this equation, a matrix-vector multiplication is introduced. Note that a vector is a special matrix, as it has exactly one column. The result of the matrix-vector multiplication is a new vector. When multiplying a matrix and a vector, there is a strong requirement that the number of columns in the matrix equals the number of rows in the vector. In this case, there are 2 columns in the matrix, and 2 rows in the vector, so that matches. Second, the resulting vector will have the same number of rows as the original matrix. The Pauli-X matrix has 2 rows, and the resulting vector has 2 rows as well.

The values in the resulting vector are calculated as follows: the element at position *i* in the resulting vector is the sum of the multiplications of all elements at row *i* of the matrix with the

corresponding element in the original vector. Concrete, the first element in the vector is obtained by

Hence, the Pauli-X gate applied to a qubit in state |0> (the original vector in the equation) results in the qubit having state |1> (the result after applying the matrix multiplication). This is indeed what we learned in the previous chapter.

Second, suppose the qubit orinially holds the value 1, and thus is represented with the Dirac notation |1>, or by the vector

EQUATION 4.11

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

In this case, multiplying the Paul-X gate matrix with the qubit vector goes as follows:

EQUATION 4.12

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

The result is the vector representation for a qubit with a value of 0, or in Dirac notation |0>.

The 2 use cases we calculated above show that for the "simple" cases where the qubit either holds the value 0 or 1, the matrix we created in Equation 4.8 indeed corresponds with what we expect from the Pauli-X gate.

But those are 2 "edge" cases, and we want to know what happens when a qubit is in a superposition state. In this case, the state of the qubit is written as

EQUATION 4.13

$$\psi = \alpha|0> + \beta|1>$$

or in vector notation:

EQUATION 4.14

$$\psi = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

You will now apply a Pauli-X gate to this qubit, by multiplying the matrix from Equation 4.8 with this vector:

EQUATION 4.15

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}$$

As you can see from this equation, the Pauli-X gate in general swaps the probabilities of finding 0 and finding 1 when the qubit is measured. In the extreme case that the qubit is either 0 or either 1 before the Pauli-X gate is applied, the gate will simply invert the value.

4.3.3 A matrix that works for all gates

In the previous section, we showed how the Pauli-X gate operating on a single qubit can be described by a multiplication of the Pauli-X gate matrix with the probability vector of the qubit.

In this section, you will learn how this principle of matrix multiplications work for any gate. Throughout this book, we will introduce new gates, and it helps understanding the general principle on how gates relate to matrices in general.

The general action of applying a gate to a single qubit, as shown in Figure 4.4

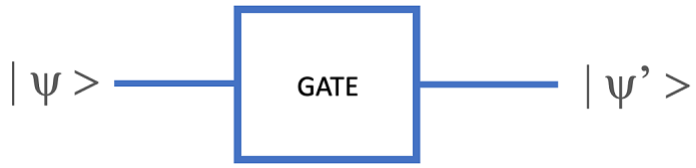

Figure 4.11 Applying a gate to a qubit

is the equivalent of the following matrix multiplication:

EQUATION 4.16

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix} \rightarrow \begin{pmatrix} 00 a_0 1 \\ 10 a_1 1 \end{pmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha' \\ \beta' \end{bmatrix}$$

Originally, the qubit is in the state $|\psi>$ which can also be written as

EQUATION 4.17

$$|\psi >= \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

The gate in Figure 4.4 corresponds to the matrix

EQUATION 4.18

$$\begin{pmatrix} 00 a_{01} \\ 10 a_{11} \end{pmatrix}$$

Applying the gate to the qubit corresponds to multiplying the matrix with the qubit probability vector:

EQUATION 4.19

$$\begin{pmatrix} 00 a_{01} \\ 10 a_{11} \end{pmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

The multiplication of the gate matrix with the qubit state results in a new vector describing the qubit state.

EQUATION 4.20

$$\begin{pmatrix} 00 & a_{01} \\ 10 & a_{11} \end{pmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 00\alpha + a_{01}\beta \\ 10\alpha + a_{11}\beta \end{bmatrix}$$

After applying the gate to the qubit thus brings the qubit in the following state:

EQUATION 4.21

$$\begin{bmatrix} 00\alpha + a_{01}\,\beta \\ 10\alpha + a_{11}\,\beta \end{bmatrix} = \begin{bmatrix} \alpha' \\ \beta' \end{bmatrix}$$

Now that you learned how gates correspond to a matrix operation, it is time to talk about a gate that is essential to the subject of this chapter: superposition.

4.4 The Hadamard Gate, the gate to superposition

In order to bring a particle into a superposition state, some very high-skilled phyiscs need to be applied. Fortunately, as a developer, bringing a qubit into a superposition state simply requires applying a specific gate to that qubit.

Figure 4.5 shows the gate that brings a qubit that is originally in the 0 state into a superposition state. This gate is called the Hadamard Gate.

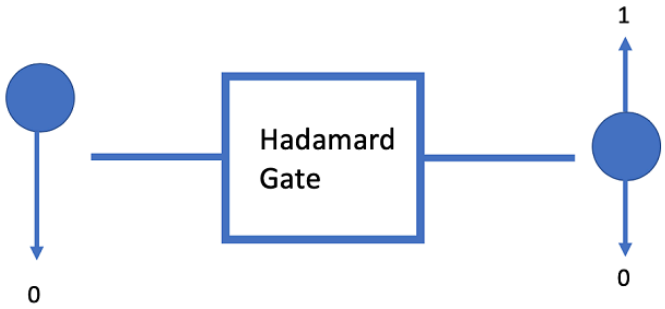

Figure 4.12 Hadamard gate brings a qubit in superposition

The Hadamard Gate is one of the most fundamental concepts in quantum computing. After applying a Hadamard Gate to a qubit that holds the value *0*, there is 50% chance that the qubit will be measured as *0*, and there is 50% chance that the qubit, when measured, will hold the value *1*.

NOTE We mentioned this before, but it can't be repeated enough: the wording *when measured* is extremely important in the previous explanation. As long as the qubit is not measured, it can stay in a superposition. Other gates can be applied, and the probabilities will change. Only when the qubit is measured, it will have a value of either *0* or *1*.

Similar to the Pauli-X gate, the Hadamard gate acts on a single qubit, and can be represented by a 2 x 2 matrix as well.

The Hadamard Gate is defined as follows:

EQUATION 4.22

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$$

We want to find out what happens when we apply this gate on a qubit that is in the $|0>$ state. This can be inspected by multiplying the gate matrix to the qubit vector:

EQUATION 4.23

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{bmatrix}1\\ 0\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\ 1\end{bmatrix}$$

This equation shows that, after applying the Hadamard gate to a qubit that is in the $|0>$ state, the qubit enters a new state where the probability of measuring 0 is

EQUATION 4.24

$$\left(\frac{1}{\sqrt{2}}\right)^2 = \frac{1}{2}$$

And the probability of measuring 1 is also

EQUATION 4.25

$$\left(\frac{1}{\sqrt{2}}\right)^2 = \frac{1}{2}$$

In conclusion, applying the Hadamard gate to a qubit that is in state |0> brings the qubit in a superposition state where the probability of measuring 0 is equal to the probability of measuring 1.

What would happen if you apply the Hadamard gate to a qubit that is in state |1> ?

The vector representation of a qubit in that state is given by

EQUATION 4.26

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Hence, applying a Hadamard gate to this qubit means multiplying the Hadamard matrix with the above vector:

EQUATION 4.27

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

If you would measure the qubit at this point, the chance of measuring 0 would be

EQUATION 4.28

$$\left(\frac{1}{\sqrt{2}}\right)^2 = \frac{1}{2}$$

and the chance of measuring 1 would be

EQUATION 4.29

$$\left(\frac{-1}{\sqrt{2}}\right)^2 = \frac{1}{2}$$

Hence, in both cases (qubit |0> or qubit |1>) applying a Hadamard Gate gives an equal chance for the qubit to be `0` or `1'` when measured.

4.5 Java code using the Hadamard gate

You are now going to use the Hadamard gate to create a random number generator. This is already a useful application, as random numbers are very useful in cryptography.

You learned the theory about the Hadamard gate, now it is time to use it in quantum applications.

Similar to how you created a quantum application with a Pauli-X gate in the previous chapter, you will now create a simple quantum application with a Hadamard gate. The code for this sample can be found in our samples, under `ch04/hadamard`. This sample contains 2 parts. In the first part, you only run the application once. The relevant code for this part is shown below:

Listing 4.1 first code snippet using a Hadamard gate.

```
public static void singleExecution(String[] args) {
    QuantumExecutionEnvironment simulator = new SimpleQuantumExecutionEnvironment();
    Program program = new Program(1);
    Step step = new Step();                                    ❶
    step.addGate(new Hadamard(0));                             ❷
    program.addStep(step);
    Result result = simulator.runProgram(program);             ❸
    Qubit[] qubits = result.getQubits();
    Qubit zero = qubits[0];
    int value = zero.measure();                                ❹
    System.out.println("Value = "+value);
}
```

❶ At this point, the environment is ready and you can add gates.

❷ A Hadamard gate is added to the qubit.

❸ The quantum program is executed.

❹ The qubit is measured, and will have a value of '0' or '1'.

Note the similarity between this sample and the Pauli-X sample from the previous chapter. This

time, we skip the detailed explanation about the steps that are similar to the Pauli-X sample.

You create the `QuantumExecutionEnvironment` which will run your program. Next you create a `Program` instance that will deal with a single qubit, and you create a `Step` instance.

Instead of adding the Pauli-X gate to that step, you now add the Hadamard gate to the step:

```
step.addGate(new Hadamard(0));
```

This will apply a Hadamard gate to the qubit. By default, qubits are originally in the |0> state. You learned in this chapter that after applying a Hadamard gate to a qubit that is in that state, there is 50% chance the qubit, when measured will be `0` and 50% chance that it will be `1`.

The remainder of the code snippet is again similar to the Pauli-X sample: you add the step to the program, and run the program on the simulator.

Finally, you measure the qubit, and print the value.

If you run this program once, you will either see

```
Value = 0
```

or

```
Value = 1
```

After the sample code prints out the measured value, it visualises the quantum circuit using StrangeFX. This is done using the following line of code:

```
Renderer.renderProgram(program);
```

As a result of this action, a window is shown containing the quantum circuit, as can be seen in Figure 4.13

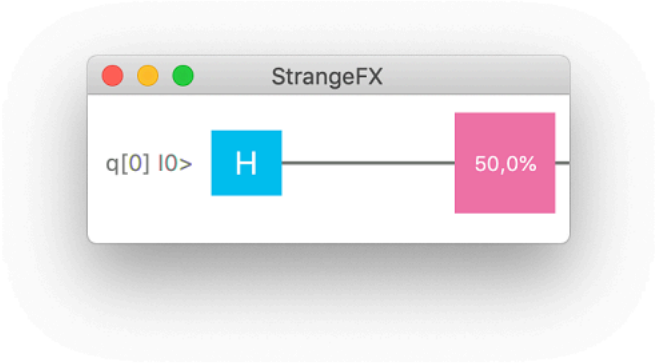

Figure 4.13 Rendering a quantum circuit with one qubit, and one gate: the Hadamard gate.

As can be seen from this picture, the resulting qubit has a 50% probability of being measured as

1.

The second part of the sample invokes the `manyExecution` function, which is very similar to the `singleExecution` discussed above, but this time you run the program 1000 times. The `QuantumExecutionEnvironment` and the `Program` have to be created only once. After the program has been created, the following loop is added to the application:

Listing 4.2 doing multiple runs of the Hadamard snippet

```
int cntZero = 0;
int cntOne = 0;
for (int i = 0; i < 1000; i++) {A                    ❶
    Result result = simulator.runProgram(program);   ❷
    Qubit[] qubits = result.getQubits();
    Qubit zero = qubits[0];
    int value = zero.measure();                      ❸
    if (value == 0) cntZero++;                       ❹
    if (value == 1) cntOne++;
}
```

❶ You run the following loop 1000 times

❷ You run the quantum program

❸ You measure the qubit

❹ Based on the measured valued ('0' or '1') you increment one counter or the other.

From this snippet, it can be seen that the `runProgram` method is called 1000 times on the simulator. Each time, you measure the resulting qubit. If the qubit holds the value `0`, the `cntZero` counter is incremented. If the qubit holds the value `1`, the `cntOne` counter is incremented. After applying this loop, the results are printed:

```
System.out.println("Applied Hadamard circuit 1000 times, got "
                + cntZero + " times 0 and " + cntOne + " times 1.");
```

The result of this application therefore shows something similar to

```
================================================
1000 runs of a Quantum Circuit with Hadamard Gate
Applied Hadamard circuit 1000 times, got 510 times 0 and 490 times 1.
================================================
```

What you created here, is a random number generator using the low-level Quantum API's. The single qubit in the program is brought into a superposition, and then measured. When running this program on the Quantum Simulator, or by extension on any classical computer that *simulates* quantum behavior, the randomness is still somehow deterministic, as you use classic algorithms to generate a random number. Typically, simulators work with probability vectors, and when a measurement is required, a random number is used to pick one of the probabilities — taking into account, of course, the value of the probabilities.

On real quantum hardware, this is different. Nature itself will pick one value when we measure the qubit. This process is truly random (at least, this is what most quantum physicists currently assume). While this simple application seems a complex way to generate a random number, it has real value. It shows how one can generate a truly random number using quantum hardware. Random numbers are extremely important in a number of areas including encryption.

4.6 Summary

In this chapter

- you learned the idea behind superposition, the important concept that indicates why quantum computing is interesting for dealing with algorithms that show exponential complexity.
- you learned about the different notations for the state of a quantum computer.
- you made the link between applying gates on qubits and multiplying the probability vector with a matrix
- you managed to bring a qubit in a superposition state by applying the Hadamard gate.
- you hacked a very simple algorithm using Strange that shows the Hadamard gate in action.

5 Entanglement

This chapter covers

- the analogy between flipping a coin and getting a random number
- how flipping a number of coins comes is related to the mathematical concept of a probability vector
- the physical concept of "quantum entanglement"
- quantum entanglement to create random numbers that are connected with each other
- a game that allows you to learn more about how the concepts of superposition and entanglement can be leveraged in Java applications.

In the previous chapter, we introduced and explained the concept of superposition. This concept does not exist in classical computing, and it is one of the reasons why quantum computing is fundamentally different from classical computing. Nevertheless, we managed to describe superposition in such a way that a Java programmer can leverage it in his own code. In this chapter, we will introduce quantum entanglement, a concept that is also not encountered in classical computing and that makes quantum computing really powerful. Again, we will show how you can simulate quantum entanglement and deal with it using Java code.

5.1 Predicting heads or tails

Have you ever been at a magician show where the magician is able to predict a property that seems to be random? A spectator can choose a card from a deck, and the magician tells which card it is without seeing it. Or the spectator can toss a coin, hides the result, and the magician is capable of telling whether the coin landed heads or tails.

In this chapter, you will learn to write code that does something similar to this example. However, there is no magic involved here. You will only use the programatic consequences of quantum physics.

We will use the spinning coin that can land in either heads or tails analogy throughout this chapter. First, you will write classic code that simulates 2 spinning coins, and measure the result. Next, you will write a quantum algorithm that achieves the same, leveraging the superposition principle explained in the previous chapter. Finally, you will use a new gate and *entangle* the 2 coins. Although the measured values are still random, measuring one coin tells you the value of the other coin.

This is shown in Figure 5.1.

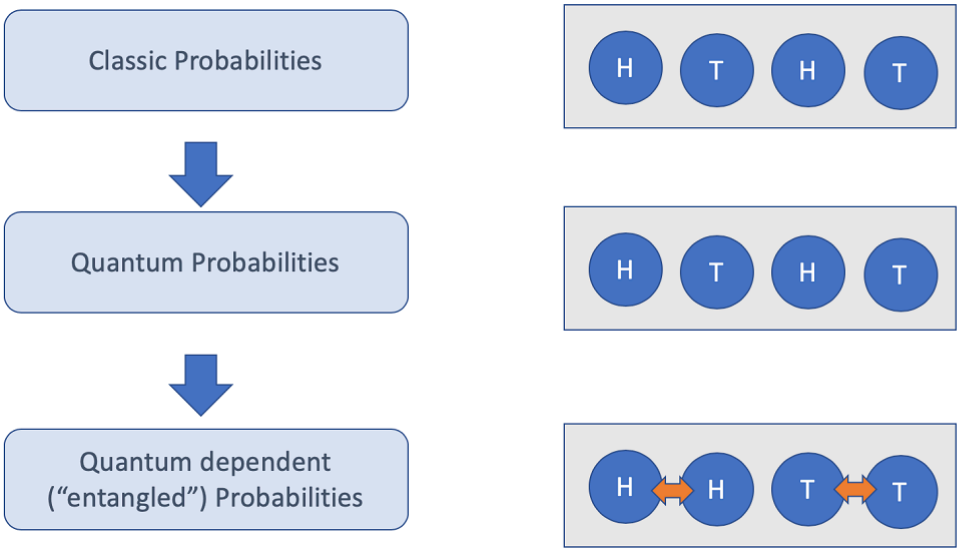

Figure 5.1 Using heads-tails coins throughout this chapter.

5.2 Independent probabilities, the classic way

Suppose you have 2 coins, coin A and coin B. Each coin can be head or tails. You spin them, and while spinning, you move them apart from each other, into different rooms. You then wait for the coins to stop spinning, and you see if they are heads or tails. What will be the result? We can't tell that with certainty, but we can say something about the probabilities. There is 50% chance that coin A will be heads, and 50% chance that coin A will be tails. Similarly, there is 50% chance that coin B will be heads, and 50% chance that coin B will be tails. If we link the outcome *heads* with the value `0` and the outcome *tails* with the value `1`, there is 50% chance a coin will be measured as `0` and 50% chance a coint will be measured as `1`, as shown in Figure 5.2.

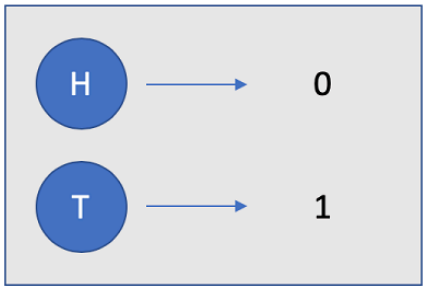

Figure 5.2 Heads = 0, Tails = 1

In total, there are 4 possible combinations that we can measure:

- coin A can be heads (`0`) and coin B can be heads (`0`). We denote this as `00` in binary representation which is `0` in decimal representations
- coin A can be heads (`0`) and coin B can be tails (`1`). We denote this as `01` in binary representation which is `1` in decimal representations
- coin A can be tails (`1`) and coin B can be heads (`0`). We denote this as `10` in binary representation which is `2` in decimal representations
- coin A can be tails (`1`) and coin B can be tails (`1`). We denote this as `11` in binary representation which is `3` in decimal representations

As we stated before, we often talk about "probabilities" when dealing with quantum computing. In this case, we have 4 possible outcomes, and each outcome has a specific probability. Hence, the probabilities can be stored in an array where the decimal representation is the index in that array. This array is also called the *probability vector*.

The conversion between heads/tails, binary digits and decimal numbers is shown in Figure 5.3.

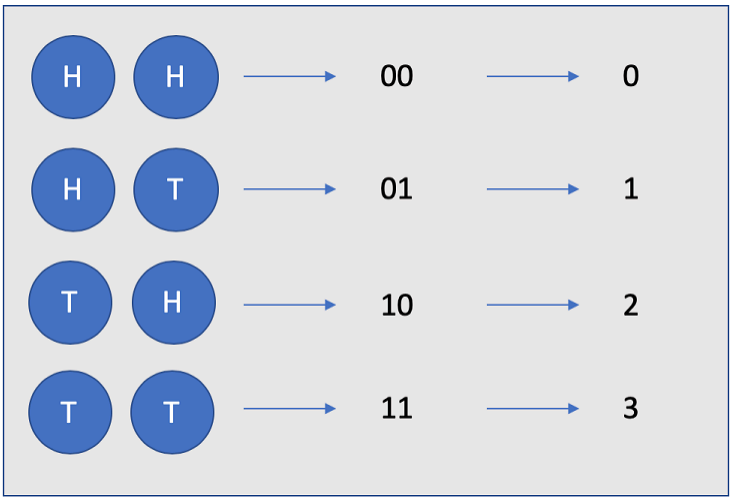

Figure 5.3 Different combinations of heads and tails

If the coins are totally fair, each of these combinations has an equal chance to occur. Hence, since the total probability needs to be 100%, each combination has a 25% chance to be measured. In this case, the probability vector is written as follows:

EQUATION 5.1

$$p = \begin{bmatrix} 25\% \\ 25\% \\ 25\% \\ 25\% \end{bmatrix}$$

As a consequence, if we make 1000 different, independent measurements, we expect every combination to be measured more or less 250 times.

We don't need a quantum computer to test this, we can do this with classical software, so let's write it. The *classiccoin* sample contains a class *TwoCoins* which does the bulk of the calculations.

Listing 5.1 Classic application for 2 coins

```
private static boolean randomBit() {                    ❶
    boolean answer = new Random().nextBoolean();
    return answer;
}

public static int[] calculate(int count) {              ❷
    int results[] = new int[4];
    for (int i = 0; i < count; i++) {
        boolean coinA = randomBit();                    ❸
        boolean coinB = randomBit();
        if (!coinA && !coinB) results[0]++;             ❹
        if (!coinA && coinB) results[1]++;
        if (coinA && !coinB) results[2]++;
        if (coinA && coinB) results[3]++;
    }
    return results;                                     ❺
}
```

❶ In this function, a random boolean is created and returned.

❷ This function calculates the probality vector for 2 coins that can be heads or tails.

❸ First, create 2 random bits, which can be either true or false, independent from each other.

❹ Based on the values of the 2 bits, one element in the probablilty vector is incremented.

❺ The probability vector is returned to the caller of the function.

The `randomBit()` function in this snippet returns a random boolean. The `Math.random()` Java

function is used to generate a random number between `0` and `1`. There is 50% chance that this number if smaller than `0.5` in which case the random boolean will be '0', and 50% chance that the random number will be bigger than 0.5, in which case the random boolean returns '1'.

The `calculate(int count)` function takes an integer as input, which defines how many times the experiment needs to be done. It returns an array of 4 integers, with each value containing the number of cases the the experiment led to a specific outcome. In each experiment, 2 random booleans (named `coinA` and `coinB`) are obtained using the `randomBit()` function. Based on the coversion scheme shown in <<ch5:hhtt>, one of the counters is incremented. For example, if `coinA` is `true` and `coinB` is `false`, which means we have a `Tails-Head` outcome, equivalent to a `10` outcome, the counter at index `2` will be incremented.

The `main` method for this application is as follows:

```
public static void main(String[] args) {
    int results[] = TwoCoins.calculate(count);                  ❶
    System.out.println("We did "+count+" experiments.");        ❷
    System.out.println("0 0 occured "+results[0]+" times.");
    System.out.println("0 1 occured "+results[1]+" times.");
    System.out.println("1 0 occured "+results[2]+" times.");
    System.out.println("1 1 occured "+results[3]+" times.");
    Platform.startup(() -> showResults(results));               ❸
}
```

❶ First, the `calculate` function is invoked, returning the array with occurences of the different possible outcomes.

❷ The different outcomes are printed.

❸ The different outcomes are shown in a graph.

Note: the code for showing the results in a graph is using JavaFX, but the details are outside of the scope of this book.

If you run this application, you'll see a more or less evenly spread distribution:

```
We did 1000 experiments.
0 0 occured 244 times.
0 1 occured 246 times.
1 0 occured 272 times.
1 1 occured 238 times.
```

The application shows a chart as well with this distribution, as shown in Figure~5.4

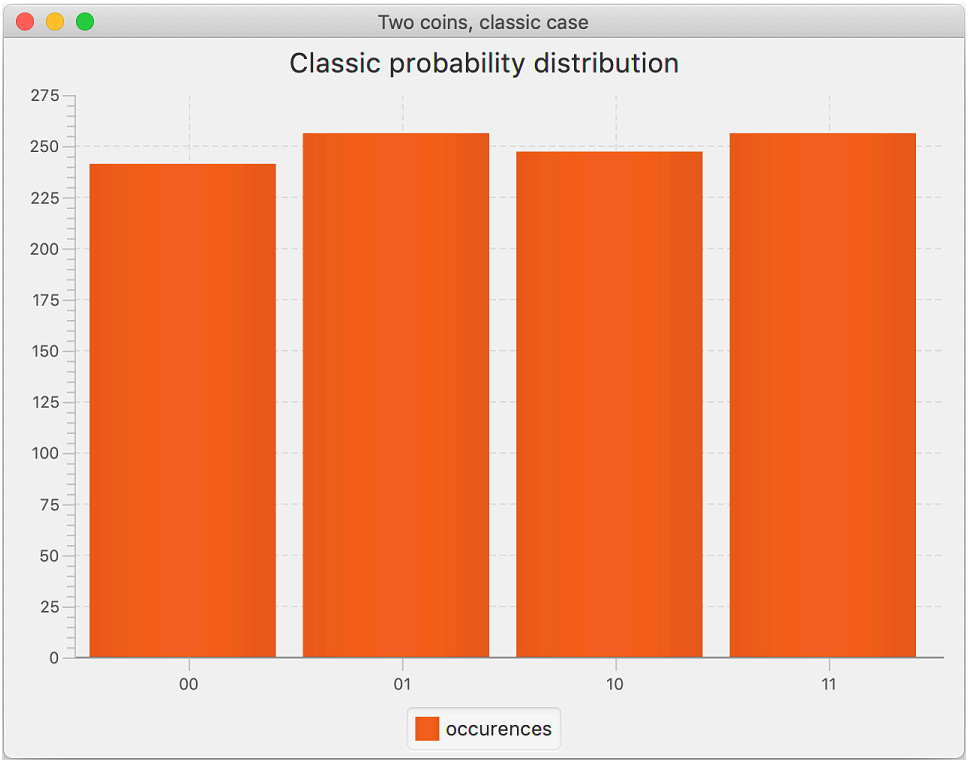

Figure 5.4 Distribution of probabilities

If you run this application multiple times, the individual probabilities will be different, but in general, all probabilities are equally possible, so therefore the numbers are in the same range.

5.3 Independent probabilities, the Quantum way

So far, there is nothing really exciting about our experiment. We just showed we can simulate the random values of the coins using a classic algorithm on a classic computer. We will now move to quantum computers, and do something similar. In the previous chapter, we created a random number generator using quantum gates. More specifically, we learned that the Hadamard gate brings a qubit in a superposition state. When measured, the qubit will fall down into one of its basis states, and we measure either a value of *0* or a value of *1*.

If we extend our system from the previous chapter with another qubit, and apply a Hadamard gate to that qubit as well, we can simulate the two coins from the previous (classic) sample using qubits. The circuit for this is shown in Figure 5.5

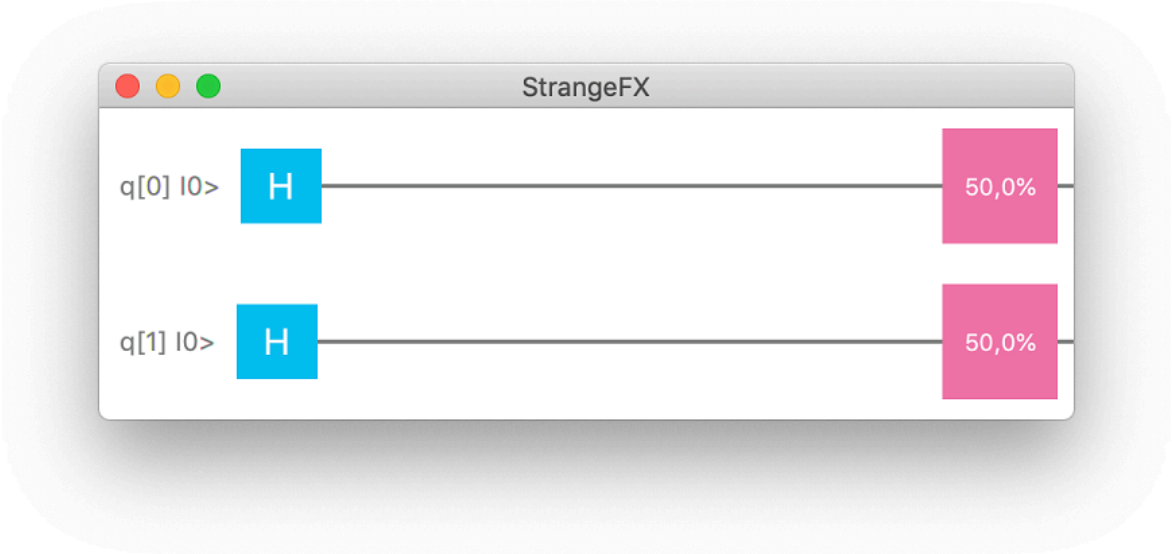

Figure 5.5 Quantum circuit with 2 qubits

You will now write the code for generating this circuit, and measuring the results. The code for this sample is in the `ch05/quantumcoin` directory of the sample repository and shown below as well:

Listing 5.2 Quantum application for 2 coins

```
private static final int COUNT = 1000;                          ❶

public static void main(String[] args) {
    int results[] = new int[4];                                 ❷
    QuantumExecutionEnvironment simulator = new
            SimpleQuantumExecutionEnvironment();                ❸
    Program program = new Program(2);
    Step step1 = new Step();
    step1.addGate(new Hadamard(0));
    step1.addGate(new Hadamard(1));
    program.addStep(step1);
    for (int i = 0; i < COUNT; i++) {                           ❹
        Result result = simulator.runProgram(program);
        Qubit[] qubits = result.getQubits();
        Qubit zero = qubits[0];
        Qubit one = qubits[1];
        boolean coinA = zero.measure() == 1;
        boolean coinB = one.measure() == 1;
        if (!coinA && !coinB) results[0]++;                     ❺
        if (!coinA && coinB) results[1]++;
        if (coinA && !coinB) results[2]++;
        if (coinA && coinB) results[3]++;
    }
    System.out.println("We did "+COUNT+" experiments.");
    System.out.println("[AB]: 0 0 occured "+results[0]+" times.");
    System.out.println("[AB]: 0 1 occured "+results[1]+" times.");
    System.out.println("[AB]: 1 0 occured "+results[2]+" times.");
    System.out.println("[AB]: 1 1 occured "+results[3]+" times.");

    Renderer.renderProgram(program);
    Renderer.showProbabilities(program, 1000);

}
```

❶ You will do 1000 experiments

❷ The `results` array contains the occurences for the different possible outcomes

❸ A quantumExecutionEnvironment is created, and the Program is constructed

❹ The program is executed 1000 times, and the results are measured

❺ Depending on the outcome, one of the counters is incremented.

If we run this code, we'll see a rather similar distribution to the classical case. The program prints the following output:

```
We did 1000 experiments.
[AB]: 0 0 occured 268 times.
[AB]: 0 1 occured 260 times.
[AB]: 1 0 occured 216 times.
[AB]: 1 1 occured 256 times.
```

Also, the program visualizes this output. The image in Figure 5.6 shows this distribution.

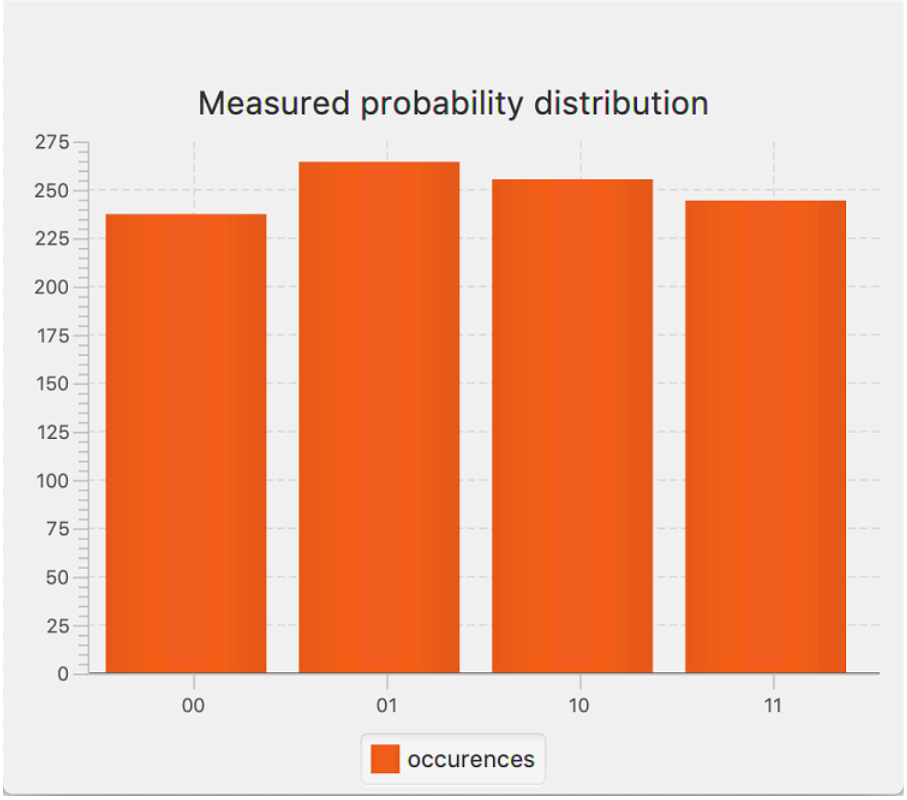

Figure 5.6 Probability distribution for 2 quantum coins

All outcomes have a 25% probability. For example, there is a 25% chance an experiment will result in probability index *0*, corresponding to a measurement of *00*. All other outcomes have 25% chance as well.

This outcome can be expected if you analyse the code, or look at the circuit in Figure 5.5. In the

code, you create a Quantum Program that involves 2 qubits by using the appropriate constructor:

```
Program program = new Program(2);
```

The program contains a single step only. In this single step, you assign a Hadamard Gate to each individual qubit:

```
Step step1 = new Step();
step1.addGate(new Hadamard(0));
step1.addGate(new Hadamard(1));
```

The step is added to the program via

```
program.addStep(step1);
```

Hence, the program you created contains of a single step, which contains a Hadamard gate on each individual qubit. Intuitively, it is clear that there is no connection between the qubits, and both qubits will be randomly *0* or *1* when measured, independent from each other. Again, this looks similar to what we showed in the previous section with classical bits.

5.4 The physical concept of entanglement

With the algorithm in the previous section, we showed that we can use a quantum algorithm to achieve the same as a classic algorithm. But we promised we would go beyond the classic capabilities.

In this section, we make a detour to the physical world, to explain the physical concept of quantum entanglement. After this, we go back to the software world, and we will show how we represent this phenomenon.

With qubits, it is possible to do something that is impossible to achieve with classic bits. With the *physical representation* of qubits, it is possible to achieve something that is impossible to achieve with the *physical representation* of bits, and we can leverage this in our software representation.

In classic software, 2 different bits do not influence each other. Clearly, we can copy the value of one bit into another bit, but then we explicitly assign a value to the second bit.

The quantum phenonemon that we will leverage now is called quantum entanglement. This is one of the weirdest physical phenomenons, and some of the brightest physisists in history had strong discussions about this. There are still ongoing discussions about what quantum entanglement really is, and how it fits with other physical concepts.

SIDEBAR As the famous physisist Richard Feynman said: "If you think you understand quantum mechanics, you don't understand quantum mechanics." Fortunately, you don't need to understand quantum mechanics if you want to leverage it. While it is interesting to think about the quantum mechanical concepts behind quantum computing, you do not need to understand them before you can program quantum computers. Similarly, you can be a great developer in classical computing without understanding how a transistor works.

We said before that the physical representation of a qubit has a property (let's call it **spin**) that can be in any of two states, but also in a superposition of those two states. We can create many qubits, and bring them in a superposition state. This is actually the underlying physical approach in the previous section. If we send our application to a real quantum computer, the corresponding phyical flow will be that two qubits are each brought in a superposition state, and then both are measured. This is shown in Figure 5.7 shows this distribution.

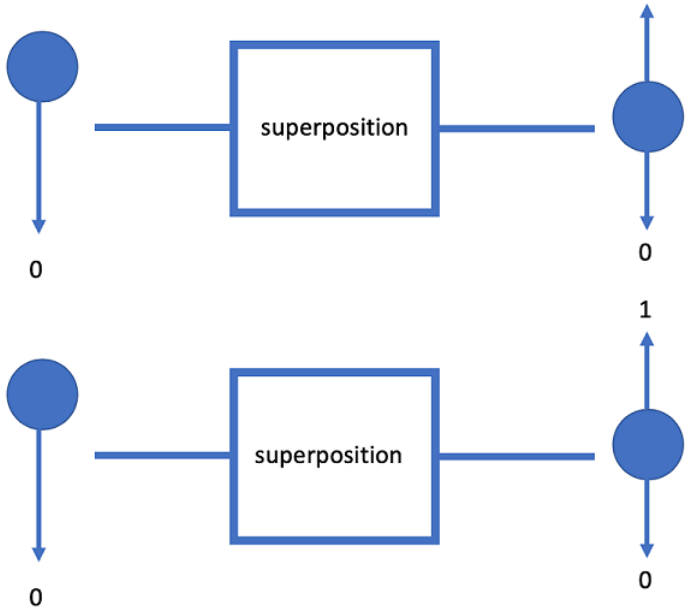

Figure 5.7 2 particles, each brought in a superposition state

If one of the particles is now measured, there is 50% chance we will measure spin up, and 50% chance we will measure spin down. If we then measure the other particle, there is again 50% chance this will be in spin up, and 50% chance it will be in spin down. The result of the second measurement is independent of the result of the first measurement.

This is shown in Figure 5.8

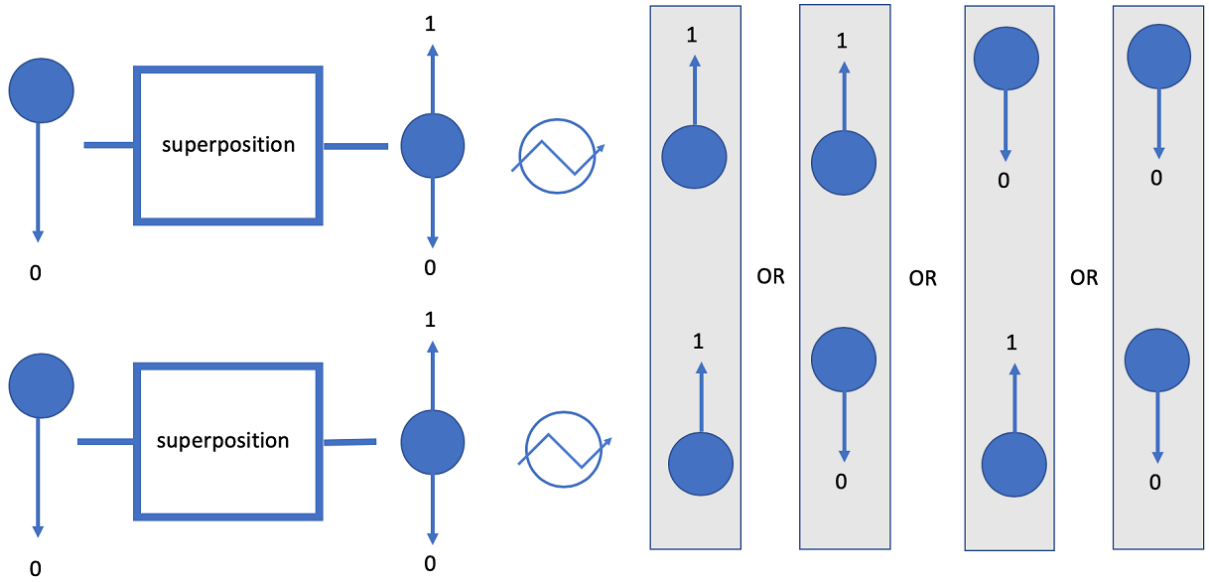

Figure 5.8 Measurement on 2 particles in superposition

As you can see, there are 4 possible outcomes from the measurement. This corresponds to what we showed in the previous section: when measuring the states of 2 qubits that are independent from each other in a superposition, there are 4 possible outcomes, each with a probability of 25%.

It is already fantastic that this superposition state can be realised, but entanglement goes one step further. Entangled particles, or entangled qubits, share their state. They might appear indepent particles in a superposition, but as soon as one of them is measured, the outcome of a measurement on the other one is fixed as well. There are a number of ways to create 2 entanglement particles, and we will safely stay away from discussing this creation process. Schematically, though, this can be represented as in Figure 5.9.

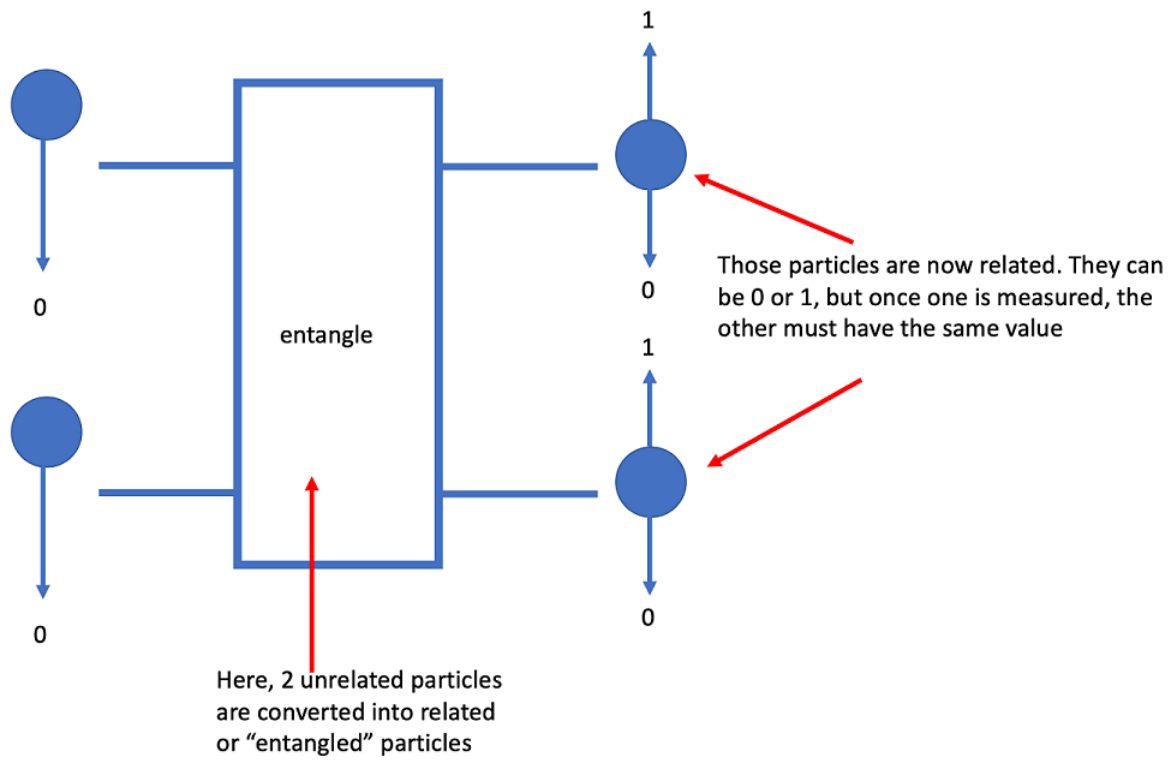

Figure 5.9 2 entangled particles being created.

In this scheme, after the physical entanglement operation, both particles are in a superposition and so far, it looks exactly the same as in the previous case. It turns out, however, that when one particle of an entangled pair is observed and its state is measured, the state of the other particle is determined as well. Depending on what entanglement technique is used, the state of the second particle will be the same or the opposite of the state of the first particle. For simplicity, we will assume that the entanglement technique that is used will generate two particles that when measured have the same state.

When the first particle is measured, there is 50% chance we measure spin up, and 50% chance we measure spin down. So far, this is exactly the same as in the previous case (with 2 particles in superposition). When the second particle is measured, there is also 50% chance we measure spin up, and 50% chance we measure spin down. However, and this is the crucial difference with the previous case, the results are not independent anymore. When the first particle results in a spin up measurement, there is 100% chance the second particle also results in a spin up measurement. Vice versa, when the first particle results in a spin down measurement, there is 100% chance the second particle also results in a spin down measurement.

This is shown in Figure 5.10

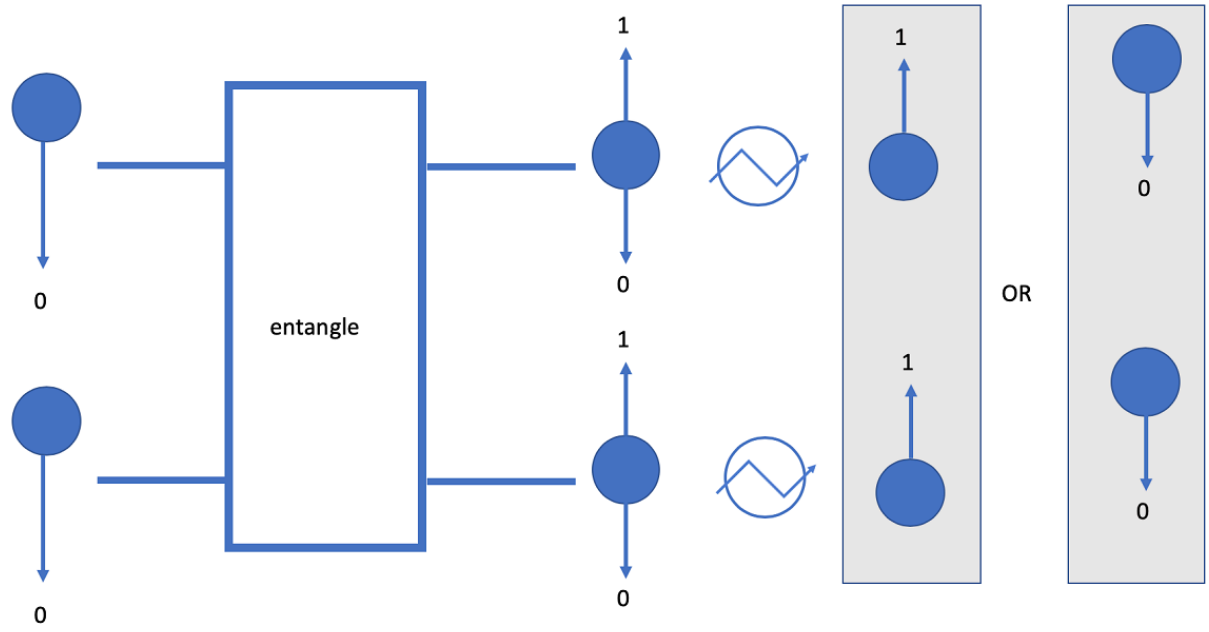

Figure 5.10 Measurement on 2 entangled particles

In this case, there are only 2 possible outcomes, instead of the four in the previous case. The only possible outcomes are {up, up} or {down, down}. Each of these outcomes has a probability of 50%. It is impossible to have an outcome of {up, down}, or {down, up}.

Hence, when we measure the individual particles, it seems they produce a random value. While that is true, the measurements are 100% dependent on each other.

SIDEBAR This is something we can use to address the goal posted in the beginning of this chapter: we want to "predict" the outcome of a coin spin, which seems to be completely random. Indeed, the coin can land in heads or tails, but the result will be the same as the result of our entangled coin. So while looking at our coin, we know the end result of the other coin without looking at it. Again, note that this is just an analogy, the quantum entanglement we are talking about here works with sub-atomic particles, and can not be extrapolated to large objects as coins!

NOTE When two particles are entangled, that does not mean they are both in an up state or a down state and that we simply don't know yet. They can really be in a superposition, until one of them is measured. This has very important consequences, as we will see in the next chapters.

After this detour to the physical world, we will now move back to software. We need to find a way to represent quantum entanglement using gates, so that we can create programs.

5.5 A Gate representation for Quantum Entanglement

In the previous chapter, you learned that the physical concept of superposition can be leveraged in quantum computing by the Hadamard gate. In this section, you will see how the concept of quantum entanglement can be leveraged in quantum computing as well, by using a combination of 2 gates.

5.5.1 Converting to probabiliy vectors

The end result of the entanglement of 2 qubits is that, when measured, the qubits are either both in spin up, or both in spin down. We will write this information using a probability vector, as we did before. Remember that a qubit with spin up corresponds to a '1' and a spin down corresponds to a '0' value. Hence, the only possible combinations are '00' (both qubits in spin down) or '11' (both qubits in spin up).

The probability vector we need to create thus contains:

- 00 (index 0) : 50% chance both qubits have spin down
- 01 (index 1) : 0% change one qubit has spin down and the other spin up.
- 10 (index 2) : 0% change one qubit has spin up and the other spin down.
- 11 (index 3) : 50% change both qubits have spin up

This corresponds to the following matrix:

EQUATION 5.2

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\1\end{bmatrix}$$

Note the square root of 2 in this equation. The reason for this is that a probability correspond to the square of the value at a specific position.

Indeed, the probability of measuring `00` (index 0) is the square of the value at index 0:

EQUATION 5.3

$$\left(\frac{1}{\sqrt{2}}1\right)^2 = 0.5$$

which corresponds to 50%. Similarly, the probability of measuring `11` (index 3) is the square of the value at index 0, which leads to 50% as well.

The probability of measuring `01` (index 1) is the square of the value at index 1:

EQUATION 5.4

$$\left(\frac{1}{\sqrt{2}}0\right)^2 = 0$$

which corresponds to 0%.

5.5.2 CNot gate

We now need to find a combination of gates that leads to the probability vector we just created. It turns out that this can easily be achieved, but we need a new gate for this, the CNot gate.

The CNot gate operates on 2 qubits, and is symbolically depicted as in Figure 5.11.

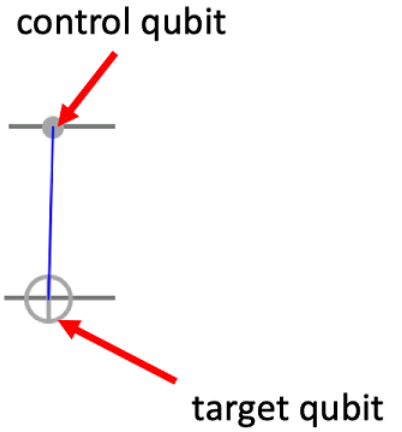

Figure 5.11 Schematic representation of a CNot gate

The 2 qubits involved in the CNot gate are often called the control qubit (the upper one) and the target qubit (the lower one). The behavior of the CNot gate is as follows:

- In case the control qubit is |0, nothing happens.
- In case the control qubit is |1, the target qubit will be flipped: if the target qubit was |0 it will be flipped to |1 and if the target qubit was |1 it will be flipped to |0.

We can verify this with a simple application using Strange. The sample called "cnot" shows a CNot gate in action in 4 different cases. The CNot gate acts on 2 qubits, and we will check the outcome for the 4 different edge cases, where the qubits are either in the |0 or the |1 state.

The main method of the sample will invoke those 4 cases as follows:

```
public static void main(String[] args) {
    run00();
    run01();
    run10();
    run11();
}
```

The `run00()` method will apply the CNot gate to 2 qubits that are both in the |0 state. The `run01()` method does the same for the case where the first qubit is in the |0 state and the second is in the |1 state. Similarly, the `run10()` method applies a CNot gate do a set of qubits where the first is in the |1 state and the second in the |0 state. Finally, the `run11()` method applies the CNot gate to 2 qubits that are both in the |1 state.

In the first case, where we apply the `run00()` method, we will have both qubits in the |0 state before we apply the CNot gate. Since the control qubit (the first qubit) is |0, we don't expect anything to change in the outcome.

The visual result of the first sample is show in Figure 5.12.

Figure 5.12 CNot gate applied to |00>

As expected, the outcome of this circuit always has the qubits in the 'Off' value, which means that when measured, we will always measure '0'.

Let's have a look at the code that leads to this output. By now, most of the statements in the code should look familiar.

```
QuantumExecutionEnvironment simulator =
        new SimpleQuantumExecutionEnvironment();        ❶
Program program = new Program(2);                       ❷
Step step1 = new Step();
step1.addGate(new Cnot(0,1));                           ❸
program.addStep(step1);
Result result = simulator.runProgram(program);
Qubit[] qubits = result.getQubits();                    ❹
Qubit q0 = qubits[0];
Qubit q1 = qubits[1];
int v0 = q0.measure();
int v1 = q1.measure();
System.out.println("v0 = "+v0+" and v1 = "+v1);
Renderer.renderProgram(program);                        ❺
```

❶ We create a new environment to run our quantum program

❷ A quantum program working on 2 qubits is created.

❸ A CNot gate is added to the first (and only) step in this program. Since the CNot gate operates on 2 qubits, we need to specify on which qubits it operates. The CNot constructor therefore takes two arguments: the control qubit (in this case the first one, with index `0`) and the target qubit (in this case the second one, with index `1`).

❹ The program is executed, and the results are measured.

❺ The program and the outcome is rendered.

The other 3 cases require an additional step: before applying the CNot gate, at least one of the qubits needs to be brought in the |1 state. As you learned in Chapter 3, this can be done by

applying the Pauli X gate. For example, the following snippet shows how to create the program for applying the case where the control qubit is |0 and the target qubit initially is |1:

```
Program program = new Program(2);
Step step1 = new Step();                    ❶
step1.addGate(new X(1));                    ❷
program.addStep(step1);
Step step2 = new Step();
step2.addGate(new Cnot(0,1));               ❸
program.addStep(step2);
```

❶ A first Step is created

❷ A Pauli-X gate is applied to the target qubit (with index 1) and added to this step.

❸ A second step, this time with the CNot gate is created and added to the program.

The visual output for this circuit is shown in Figure 5.13

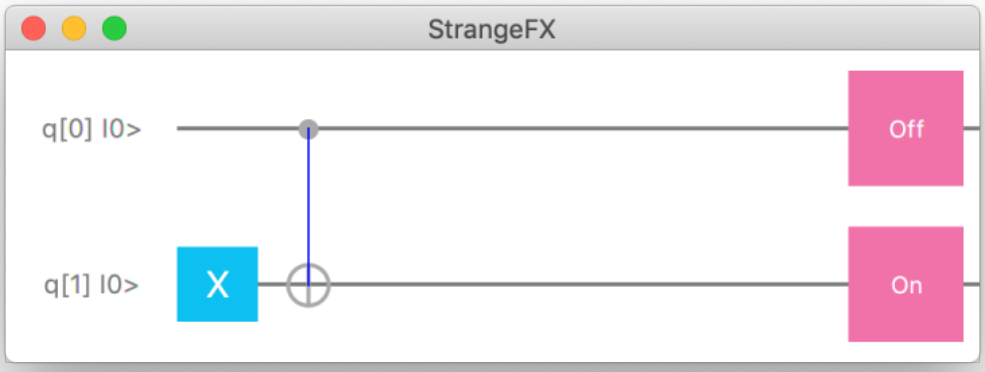

Figure 5.13 CNot gate applied to |01>

The code for the cases where the input state is |10> and |11> is very similar, and you can find it in the sample.

If you run the sample, you will see 4 different visuals, with the outcome of the program applied to the 4 different input states. The first two visuals has already been shown, the other 2 are displayed below.

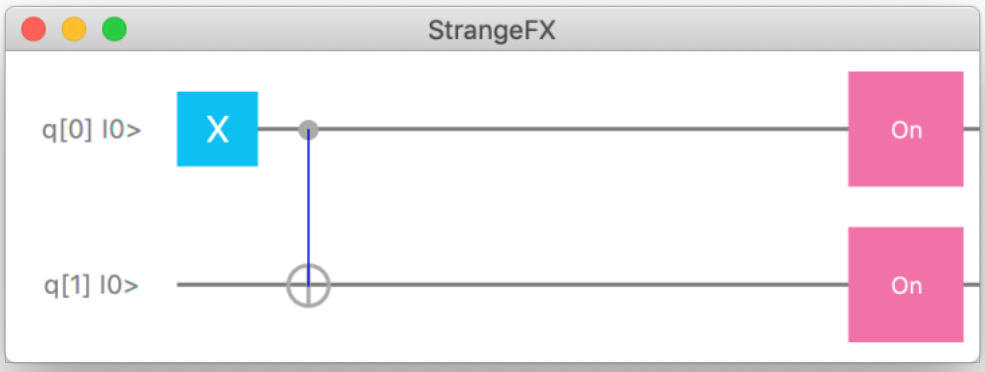

Figure 5.14 CNot gate applied to |10>

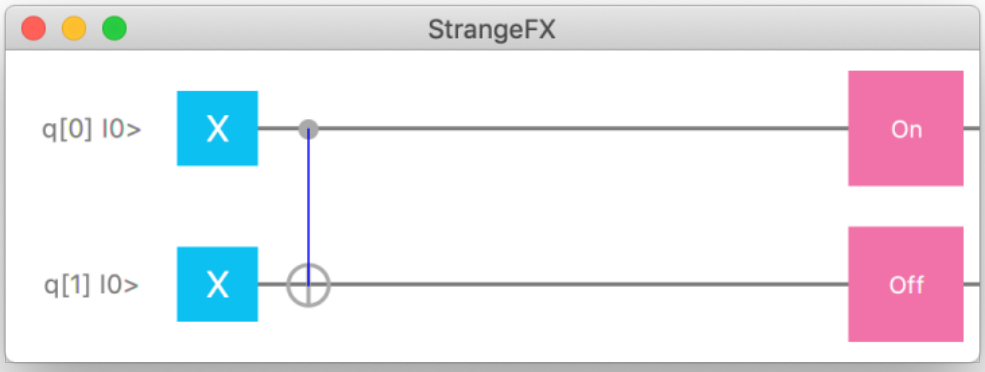

Figure 5.15 CNot gate applied to |11>

In summary, we can create the following table showing how the CNot gate alters (or keeps) the value of the control qubit and the target qubit:

Table 5.1 Behavior of the CNOT gate

control qubit	target qubit	apply CNOT	q0	q1
0	0		0	0
0	1		0	1
1	0		1	1
1	1		1	0

5.6 Creating a Bell state: dependent probabilities

The four samples in the previous section where special cases, where the input to the CNot gate is either |0 or |1. But what should we expect when the control qubit is in a superposition state, as shown in Figure 5.16?

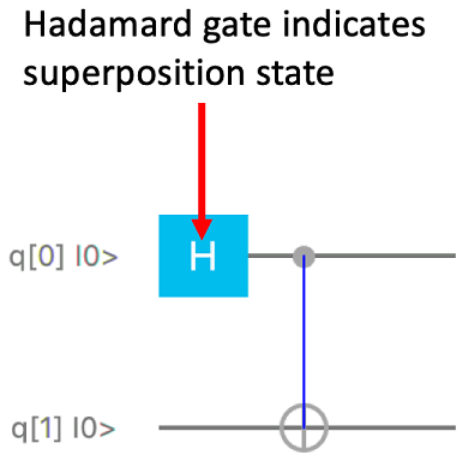

Figure 5.16 CNot gate applied to a control qubit in superposition

Remember from the previous chapter that you can use a Hadamard Gate to bring a qubit in a superposition state.

The code for creating this circuit can be found in the sample named `bellstate`

```
public static void main(String[] args) {
    QuantumExecutionEnvironment simulator = new SimpleQuantumExecutionEnvironment();
    Program program = new Program(2);
    Step step1 = new Step();
    step1.addGate(new Hadamard(0));
    program.addStep(step1);
    Step step2 = new Step();
    step2.addGate(new Cnot(0,1));
    program.addStep(step2);
    Result result = simulator.runProgram(program);
    Qubit[] qubits = result.getQubits();
    Qubit q0 = qubits[0];
    Qubit q1 = qubits[1];
    int v0 = q0.measure();
    int v1 = q1.measure();

    Renderer.renderProgram(program);
    Renderer.showProbabilities(program, 1000);
}
```

If you run this application, you will see either one of the following lines as the output:

```
Result of H-CNot combination: q0 = 0, q1 = 0
```

or

```
Result of H-CNot combination: q0 = 1, q1 = 1
```

No matter how many times you run this application, you will notice that the output is always one of those two outcomes. You will never see

```
Result of H-CNot combination: q0 = 0, q1 = 1
```

or

```
Result of H-CNot combination: q0 = 1, q1 = 0
```

When you run the application, apart from the text output, you will also see the circuit output and the probability distribution.

From the circuit output, shown in Figure 5.17 you can tell that there is 50% chance that qubit 0 will be measured as `0` and 50% chance that it will be measured as `1`.

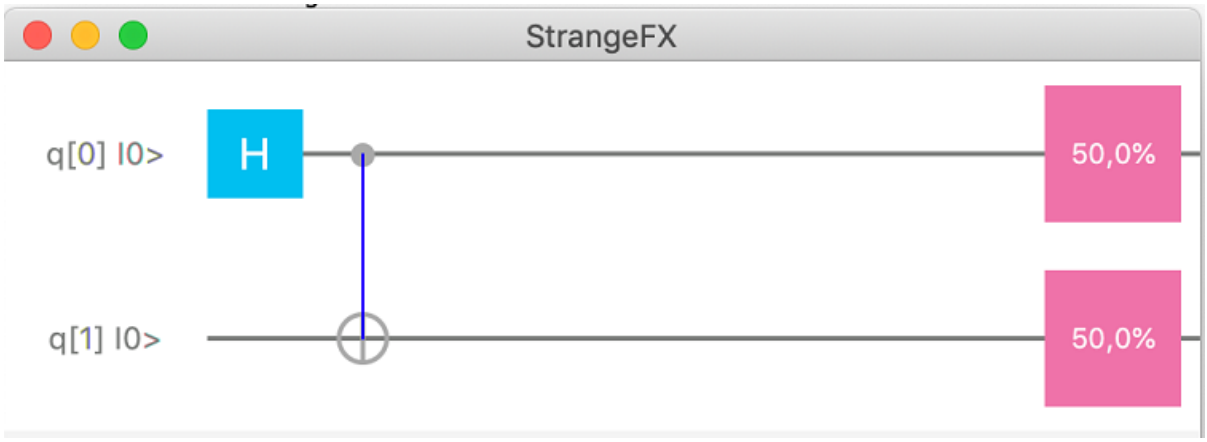

Figure 5.17 Result of a CNot gate applied to a control qubit in superposition

Similarly, that output shows that there is 50% change that qubit 1 will be measured as `0` and 50% chance it will be measured as `1`. This matches your observations, when you run the example many times. Both the first qubit and the second qubit can be measured as `0` and `1`. However, this output does not show the additional restriction that we observe, namely the fact that the *combinations* are limited. From our observations from the text output, it shows that if the first qubit is measured as `0`, the second qubit is measured as `0` as well. And when the first qubit is measured as `1`, the second qubit is measured as `1` as well.

This is shown in the probability distribution, which is rendered in Figure 5.18.

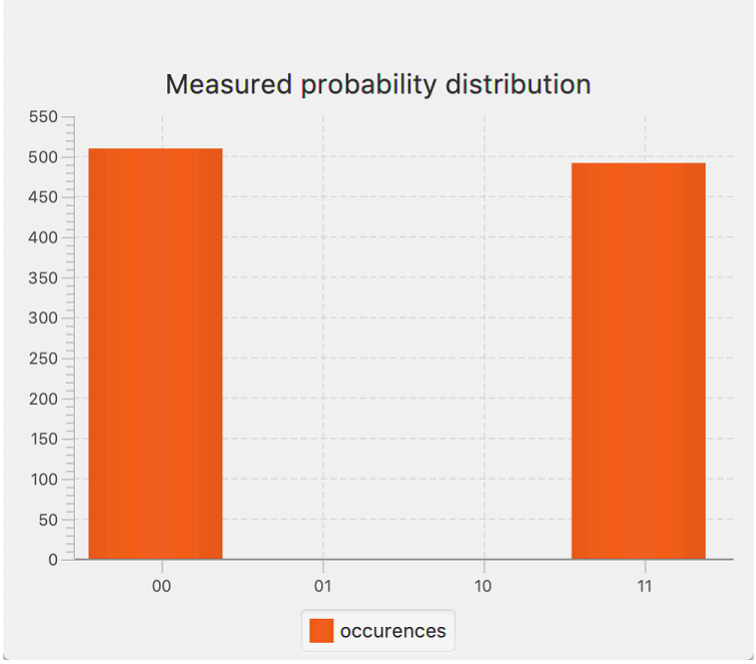

Figure 5.18 Probability distribution of a CNot gate applied to a control qubit in superposition

This is an interesting result. It seems that applying the CNot operator on a pair of qubits where the control qubit is in a superposition and the target qubit is in the |0 state always results in an entangled pair. The result we see here is exactly the state that we described before for an entangled pair. This result is also called a Bell state.

SIDEBAR When measuring individual elements of an entangled pair, the values you measure seem to be entirely random. While that is true, however, there is a full match between the "random" values of bit elements of the entangled pair.

Hence, by using a combination of a Hadamard Gate and a CNot Gate, we can "create" quantum entanglement. The word "create" is not entirely correct, as we didn't physically create entanglement, but the circuit we created results in the same probabilities as the state of two entangled qubits. This means we found a programmatic way to represent the quantum entanglement behavior.

In the next chapters, we will extensively use this behavior.

5.7 Mary had a little qubit

Now that you learned the basic concepts of quantum computing, you can start using them in applications. To get you started, we created a simple game based on the "Mary had a little lambda" application that was created by Stephen Chin in order to demonstrate the usage of Streams and Lambda's in Java. You can read more about this game at www.oracle.com/technical-resources/articles/java/rich-client-lambdas.html.

We modified the game so that the lambs that are managed by Mary are actually qubits. The code for this game is available in the repository, in the `ch05` directory under the name `maryqubit`. You can start it by entering

```
mvn clean javafx:run
```

This will show the start screen from Figure 5.19.

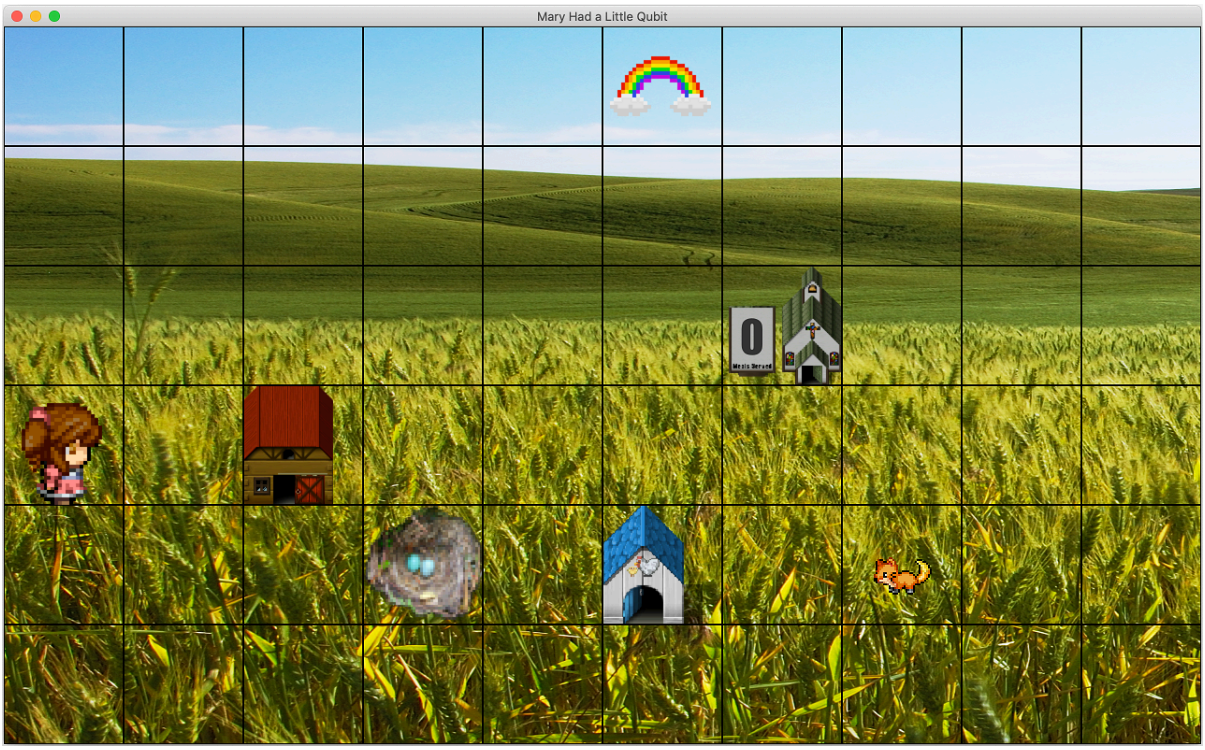

Figure 5.19 Start screen for Mary had a little Qubit

You see Mary in a landscape with a number of elements. Some of these elements corresponds to quantum gates, and when a lamb visits the quantum gate while it is active, that gate is applied to the lamb-qubit.

On the top of the screen, the corresponding circuit is shown.

There are many things to be discovered in this game, and the interested reader is encouraged to browse through the source code of the game. In a later chapter, we will come back to specific

parts of this game.

One particular interesting exercise is the following: Walk with Marh through the gates, and create a 3 qubit circuit that shows a Bell State and a third qubit with a Hadamard gate. The result of this is shown in Figure 5.20.

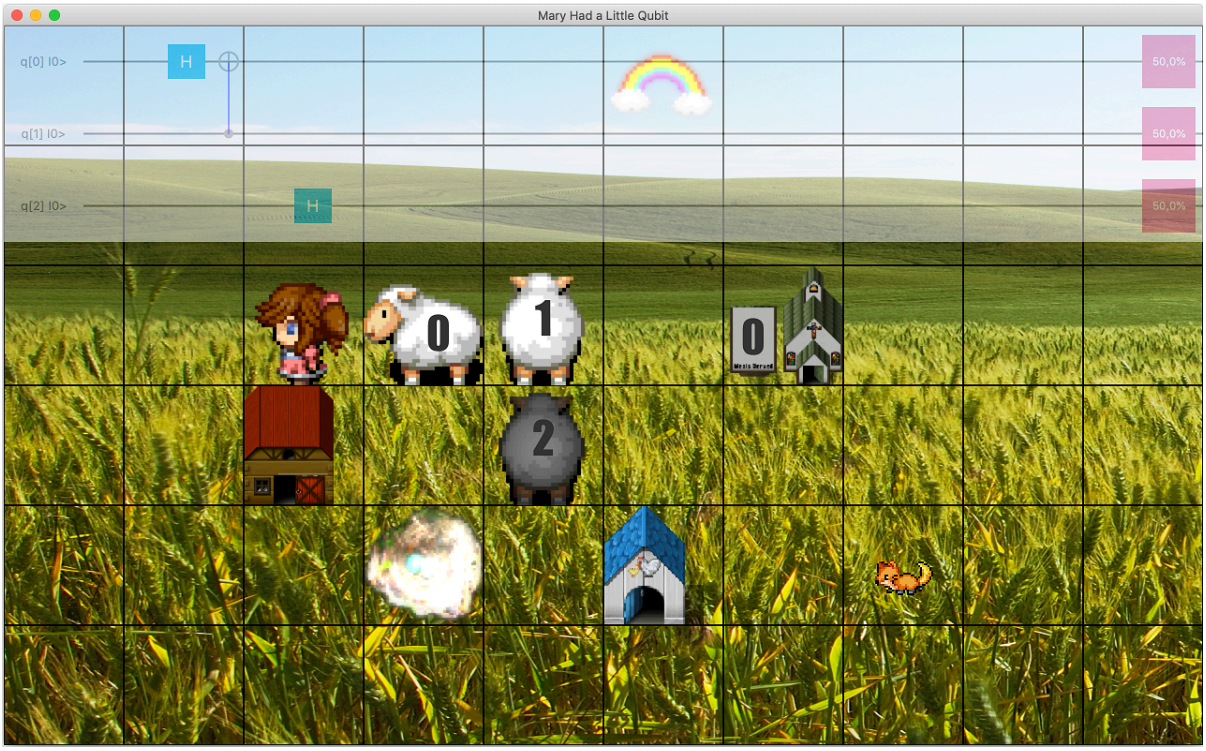

Figure 5.20 Bell state with a Hadamard gate applied to a third qubit

If you look at the code for this game, you will see how you can combine the Strange simulator, the StrangeFX visualisation and your own application. The `StrangeBridge` class is where everything comes together.

5.8 Summary

In this chapter,

- you created a random coin simulator that creates 2 coins with random values using classical software
- you did the same using quantum software
- you modified the quantum generator so that it generates an entangled pair or random values.

6 Quantum Networking, the basics

This chapter covers:

- how quantum computers and quantum networks are related
- the challenges for creating a quantum network
- a teleportation algorithm that sends a qubit from one part of the system to another.
- a quantum repeater, allowing to send qubits over a long distance

6.1 Quantum computing versus quantum networking.

So far, we talked about quantum **computing**. Computing is indeed a very important part in the software world, but most applications developed by todays software developers do not work in isolation. At the contrary, applications typically contain different modules that may or may not be located on the same server. They talk to external components, e.g. over REST interfaces. They read and write information from and to data storage systems. In general, software is typically very distributed. One of the key elements to get a complete software application working is a reliable, predictable **network** of computers.

6.1.1 From classical networks to quantum networks

A typical setup of a classical application that combines different modules over a network is shown in Figure 6.1.

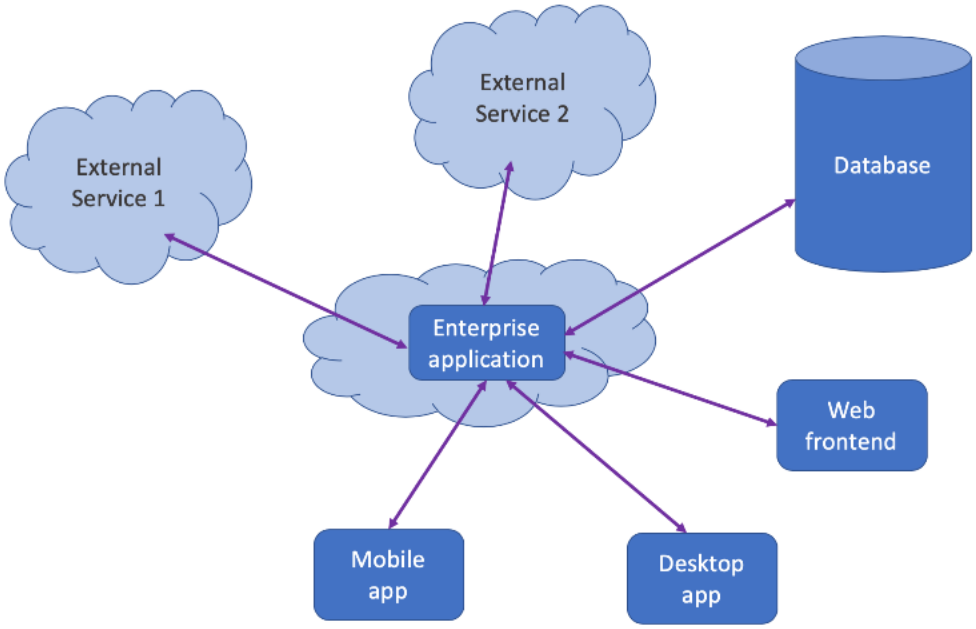

Figure 6.1 classic application using modules in a network

Classical computing heavily relies on a classical network. Similarly, quantum computing can benefit from quantum networks as we will learn in this chapter.

In the previous chapters, the focus was on a quantum computer. You learned how a quantum computer manages a set of qubits and applies gates to those qubits. You created small programs, dealing with a number of qubits and gates. All qubits and all gates were local to the program: although we did not make specific assumptions, it is reasonable to assume that the qubits are located inside a single quantum computer, and that the gates are inside this quantum computer as well, which is shown in Figure 6.2

Figure 6.2 A single quantum computer

Everything that is needed to execute the quantum programs we created in the previous chapters can be contained inside a single quantum computer.

A similar observation applies to our quantum simulator Strange. So far, all applications we created are executed in a `SimpleQuantumExecutionEnvironment`.

It is expected that most useful quantum applications running on a quantum computer require a large number of qubits. However, as you will experience in this chapter, there are also quantum applications that can work with a network of quantum computers that have a small number of

qubits. An example of such a quantum network is shown in Figure 6.3

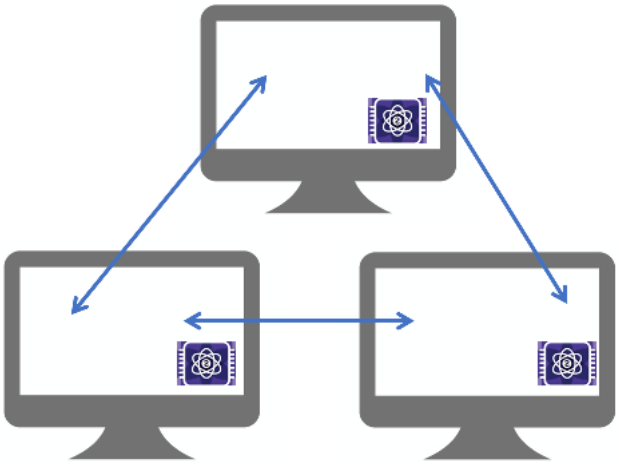

Figure 6.3 A quantum network combining 3 small quantum computers

In this Figure, we show 3 small quantum computers that are connected to each other using a quantum network. Actually, the "small quantum computers" could be real classical computers, with some quantum capabilities --- e.g. the capability to measure or manipulate a single or a few qubits.

The quantum network allows the quantum computers to exchange qubits. As a consequence, a qubit from one computer may be transfered to another computer. This sounds very similar to the classic case where a computer sends a bit to another computer.

6.1.2 Topology of a quantum network

The simplest form of quantum networking is the direct connection between two quantum computers, as shown in Figure 6.4.

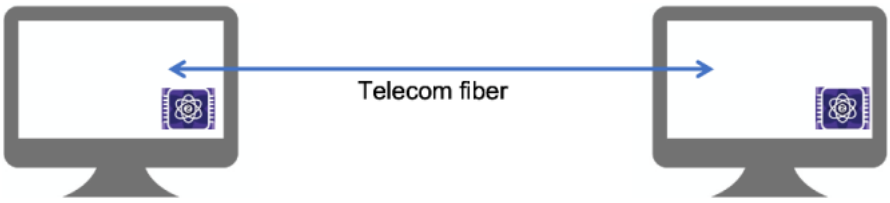

Figure 6.4 A quantum network combining 2 small quantum computers

With some limitations that we discuss later in this chapter, existing fibre optical connections can be used to transfer qubits from one quantum computer to another one.

A more valuable quantum network contains more than two quantum computers. Similar to classical networks, opportunities often grow with the number of connected computers. This can spread the load of heavy computation between different computing instances, or it can connect system with different input to each other. A possible setup is shown in Figure 6.5.

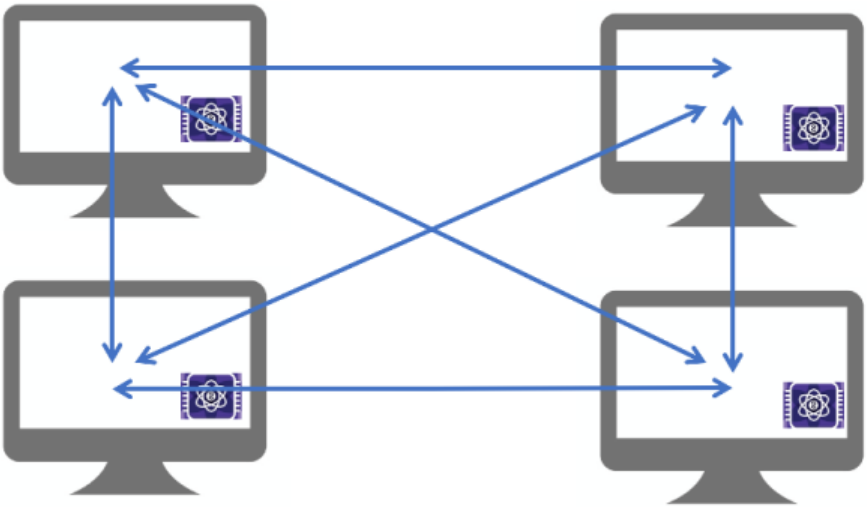

Figure 6.5 A quantum network combining a larger number of small quantum computers

Such a network topology would be harder to realise as it requires direct connections between the different computers. A typical networking approach is to use switches that direct traffic to the right computer, which is shown in Figure 6.6.

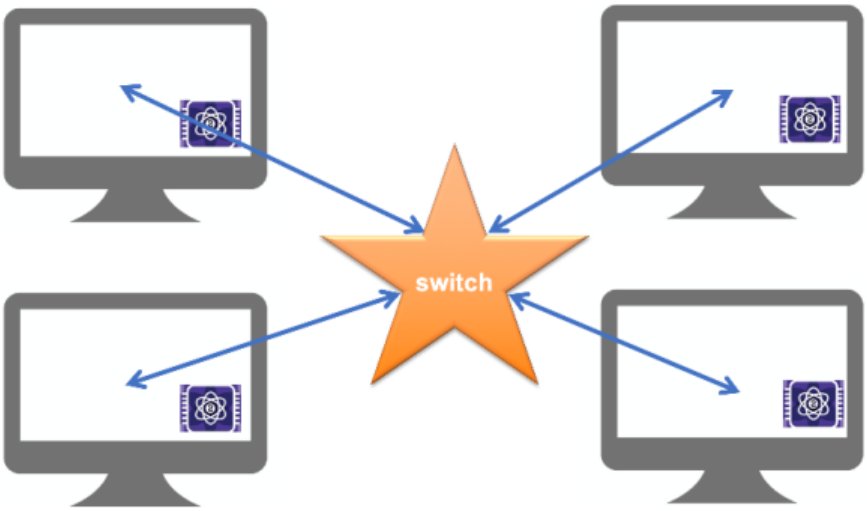

Figure 6.6 A quantum network using a switch to combine a number of of small quantum computers

While the network topology for a quantum network might look similar to the network topology for a classical network, there are important differences that on one hand make it much harder to realise a quantum network, while on the other hand open new opportunities.

TIP	One of the most intriguing aspects that you can think about for a moment already is this: in the previous chapter, you wrote code that shows that 2 qubits in a system can be entangled with each other. The outcome of a quantum program depends very much on this entanglement. What happens when one of those 2 qubits is sent to another quantum computer, where it becomes part of another quantum program?

6.2 Obstacles for quantum networking.

Before we talk about the benefits of quantum networking, we want to temper the expectations. In this section, you will learn that the typical approach for sending bits over a classical network does not easily apply to quantum networks. In the next section, you will see that this problem can be "solved", and moreover, it leads to new huge opportunities, including secure communication.

6.2.1 Classical networking in Java

In the typical case of classic networking, information is *transfered* from one computer to another computer.

Let’s have a look at how this would happen at the level of a Java Application.

NOTE	In a typical Java application, developers use libraries on top of the low level networking API’s that are part of the Java Platform. These libraries typically provide convenience methods for easily transfering data from one computer to another, using protocols like XML or JSON. The low level networking API’s that we use in our samples help to understand how information is really transfered from one computer to another. A typical stack is shown in Figure 6.7

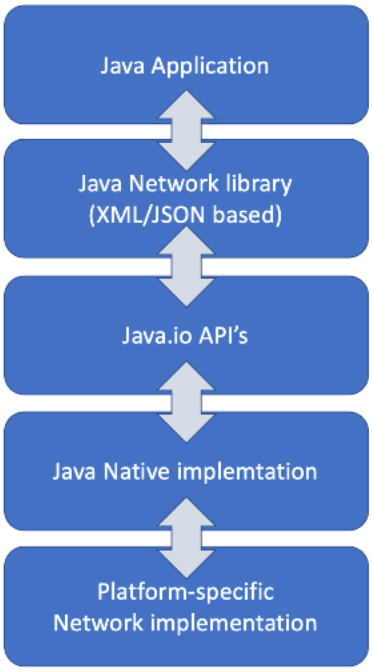

Figure 6.7 A typical approach where a Java application uses libraries for networking.

Developers typically leverage high-level network libraries, that shields them from low-level code. In most enterprise applications, Java developers hardly ever manually create an instance of the `java.net.Socket` class, but the libraries they use make plenty use of this and related classes. Similarly, it can be expected that many developers writing applications leveraging quantum computing do not directly deal with the low-level classes that enable quantum communication. High-level libraries will hide the low-level complexity and allow developers to take advantage of quantum communication without having to manually code for it.

The blueprint for such a quantum network stack is not yet finalized. A number of groups and standardisation organisations are discussing a quantum version of the classical stack, shown in 6.7. One of the interesting initiatives is the Quantum Internet Alliance (see quantum-internet.team) . The Figure in 6.8 shows a proposal for a quantum network stack, created by Axel Dahlberg from QuTech (qutech.nl) , Delft University of Technology. More information about this stack can be found in the paper at arxiv.org/pdf/1903.09778.pdf

Application	
Transport	Qubit transmission
Network	Long distance entanglement
Link	Robust entanglement generation
Physical	Attempt entanglement generation

Figure 6.8 A proposal for a quantum network stack, by QuTech.

In order to understand the challenges and opportunities associated with quantum networks, it helps to see how they compare with classical networks.

In the sample in the `ch06/classic` directory, a `Main` java file demonstrates how networking at a low level is done in Java.

The relevant code for this is shown belog :

Listing 6.1 Classic network application for sending a byte

```
static final int PORT = 9753;                                          ❶

public static void main(String[] args)
                throws InterruptedException {
    startReceiver();                                                   ❷
    startSender();                                                     ❸
}

static void startSender() {
    Thread t = new Thread() {
        @Override public void run() {
            try {
                byte b = 0x8;
                System.err.println("[Sender] Create a connection to port "+PORT);
                Socket socket = new Socket("localhost", PORT);         ❹
                OutputStream outputStream = socket.getOutputStream();
                System.err.println("[Sender] Write a byte: "+b);
                outputStream.write(b);                                 ❺
                outputStream.close();
                System.err.println("[Sender] Wrote a byte: "+b);       ❻
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    };
    t.start();
}

static void startReceiver() throws InterruptedException {
    final CountDownLatch latch = new CountDownLatch(1);
    Thread thread = new Thread() {
        @Override public void run() {
            try {
                System.err.println("[Receiver] Starting to listen for incoming data
                                                at port "+PORT);
                ServerSocket serverSocket = new ServerSocket(PORT);    ❼
                latch.countDown();
                Socket s = serverSocket.accept();                      ❽
                InputStream inputStream = s.getInputStream();
                int read = inputStream.read();                         ❾
                System.err.println("[Receiver] Got a byte "+read);      ❿
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    };
    thread.start();
    latch.await();
}
```

❶ The PORT number you define here is shared between the thread that sends data and the thread that reads data.

❷ The code for receiving bytes is performed on one thread

❸ The code for sending bytes is performed on a second thread

4. The sender thread opens a low-level Java network socket
5. The sender writes a specific byte over that socket to the receiver
6. The value of the *transfered* byte can still be used (e.g. printed) by the sender
7. The receiver opens a low-level Java network serversocket, listening for incoming requests
8. When a connection is discovered at the serversocket, a direct socket connection is created between the sender and the receiver
9. The receiver reads a byte from this connection.
10. The receiver prints the value of the byte he received.

The output of this application looks as follows:

```
[Receiver] Starting to listen for incoming data at port 9753
[Sender] Create a connection to port 9753
[Sender] Write a byte: 8
[Sender] Wrote a byte: 8
[Receiver] Got a byte 8
```

Note that after we sent the byte to the other "computer", we still have its value locally. What happens internally is that the `byte` that we declared in Java, points to some memory in our computer. When the byte is *transfered* to the other computer, it is not removed from memory. Rather, the low-level network drivers of the operating system read the *value* of the byte at the specific memory location, and they send a copy of that value to the other computer.

That sounds very trivial — and indeed, in the case of classic computing it is trivial — but this is not trivial when we talk about quantum computers.

To explain the challenges at a high level, have a look at the schematic representation of the Java program you just created, shown in Figure 6.9.

NOTE The program above created 2 threads on the same computer, where the picture shows 2 different computers. The latter is of course more realistic, but in network demonstrations it is common to use communication between 2 threads on the same computer that communicate over a socket. The endpoint of a socket is defined by a combination of the `hostname` and the `portnumber`. The `hostname` corresponds to the physical address of the computer, where the `portnumber` corresponds to an internal *port* of the computer. For our purposes, it is enough to specify a port number only, and keep the communication inside one computer.

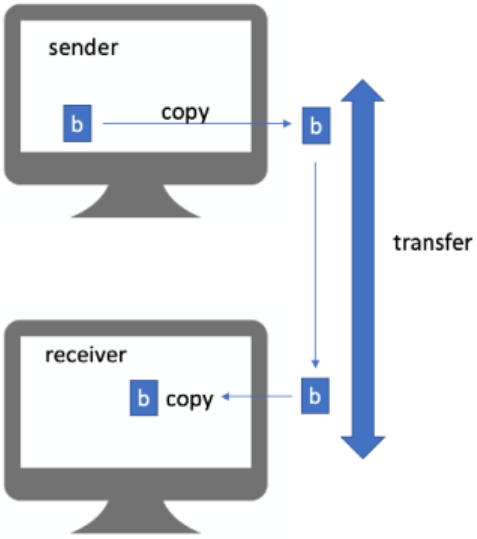

Figure 6.9 Transfering a byte from the sender to the receiver.

This picture shows two major steps in classical networking:

1. the byte we want to tranfser is *copied*
2. the byte is *transfered* over a network to the other computer (where it is copied again).

The bad news is that there are issues with both of those steps in quantum computing:

1. qubits can not be *copied*
2. qubits can not easily be *transfered* over long distances

In the next sections, we will discuss these issues in more details. After that, we will explain how these issues can be solved, and turned into opportunities.

6.2.2 No cloning theorem

One of the core concepts of quantum computing that we didn't discuss yet, is the *no cloning theorem*. This states that it is impossible to make an exact copy of a qubit. In classical computing, you can inspect the value of a bit, and create a new bit with the exact same value. By doing so, you don't alter the value of the original bit.

The following java code demonstrates how we can copy the value of a Boolean object into another Boolean object in Java. Note that this can be done much simpler in Java, but we want to mimick the clone-behaviour so that we can try to apply that to a Qubit object as well.

```
static Boolean makeCopy(Boolean source) {                    ❶
    Boolean target;
    if (source == true) {
        target = Boolean.valueOf(true);
    } else {
        target = Boolean.valueOf(false);
    }
    return target;
}
```

```
  public static void main(String[] args) {
    Boolean trueSource = Boolean.TRUE;
    Boolean trueCopy = makeCopy(trueSource);                              ❷
    System.err.println("Source: "+trueSource+" and copy : "+trueCopy);

    Boolean falseSource = Boolean.FALSE;
    Boolean falseCopy = makeCopy(falseSource);                            ❸
    System.err.println("Source: "+falseSource+" and copy : "+falseCopy);
  }
```

❶ the makeCopy method takes a Boolean object as a parameter, and return a new Boolean which holds the same value as the passed source parameter

❷ you make a copy of a Boolean containing `true`, and print both the original and the copy

❸ you make a copy of a Boolean containing `false`, and print both the original and the copy

In this code snippet, which you can find in `ch06/classiccopy` in the sample repository, we have a method `makeCopy` which takes a `Boolean source` as an argument, and which returns a new `Boolean` instance that has the same value as the `source` instance. When the `source` instance holds the value `true`, the returned instance will also hold the value `true`. In case the `source` instance holds the value `false`, the returned instance will hold the value `false` as well.

The returned `Boolean` object is a new and independent object. After applying the `makeCopy` method, the values of the `source` and the returned instance are exactly the same. By repeating this procedure, we can make exact copies or a series of bits,

It is tempting to do this in quantum computing as well. Remember from Equation XREF diracqubit that we can write the state of a qubit as a linear combination of the |0 state and the |1> state:

EQUATION 6.1

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

If you look at the source code for the `Qubit` class in the Strange simulator under the package `com.gluonhq.strange`, you notice this class contains the following fields

```
private Complex alpha;
private Complex beta;
```

These fields contain the information about the qubit, needed for the simulator to perform operations. We could easily add a constructor to the Qubit class that does the following:

```
public Qubit(Qubit src) {
    this.alpha = src.alpha;
    this.beta = src.beta;
}
```

Doing so, we have a Java approach for copying the information from one qubit into another qubit. Our quantum applications could leverage it, and we can write code where we happily copy qubits all over the place.

However, it would be impossible to implement this on a real quantum computer. Hence, your application could copy qubits in the Strange simulator, but it would not work on a real quantum computer. Therefore, there is no "copy constructor" for a Qubit in Strange.

SIDEBAR **It's all about probabilities**

The real *value* of a Qubit is not the value that you measure. What makes a qubit so powerful, is that it actually holds a *probability* to measure '0' or '1'. Simply measuring a qubit is not enough to reconstruct this probability.

In quantum computing, if you want to inspect the value of a qubit in a superposition, you have to measure it. And as you learned before, measuring a qubit destroys its superposition state, and it falls back into one of the basis states. Hence, by doing so, not only you destroy the original qubit, but you also don't have enough information to create a new qubit with the same value. Let's show that with an example.

Suppose you have a qubit that has 25% chance to be measured as 0, and 75% chance to be measured as 1. The state of this qubit can be written as follows:

EQUATION **6.2**

$$|\psi\rangle = 1/2\,|0\rangle + \sqrt{3}/2\,|1\rangle$$

Remember that if we want to obtain the possibility of measuring `0`, we need to take the square of α . In this case, α equals $1/2$ hence the square of α is $1/4$ or 25%. Similar, the possibility of measuring `1` is obtained as the square of $\sqrt{3}/2$ which is $3/4$ or 75%.

Next, suppose that you measure the qubit, and you measure the value `0`. At that point, you still don't know if that qubit was actually holding a value of 0, or if it was in a superposition with 25% chance on a 0 measurement, or 95% chance on measuring `0`, or any other state — apart from the state where α equals 0. In order to get an accurate idea about the original state of the qubit, you would need an infinite number of measurements on that qubit. However, the laws of quantum physics determine that you get only a single shot. After one measurement, the qubit is no longer in a superposition state, and the information is lost.

In summary, you are not able to reconstruct the *probability* and that is the number that matters when talking about qubits.

> **NOTE** The no cloning theorem is directly related to quantum physics. Any software approach we could easily circumvent this --- but then, your appliacation would only run on the simulator, and not on real hardware when that is available.

The no cloning aspect of qubits makes a number of things very challenging:

- sending a qubit from one place to another can not be done by taking a copy and transmit that
- the concepts of network switches and signal repeaters require that the bits being read, perhaps amplified, and put on a different wire again. If we can't copy a qubit, how can we deal with this networking requirements?

On the other hand, the cloning aspect also leads to interesting opportunities: it is impossible to evedrop on a quantum communication channel without being notified. When an attacker wants to intercept a qubit that is being sent from A to B, he has to measure the qubit. As a consequence, the information kept in the original qubit is gone, and the receiver knows there was an issue. We will discuss this opportunity in a next chapter, but we first have a few issues to solve.

In section 6.4, you will learn how to bypass the problem created by the no cloning theorem. You will create a quantum circuit that allows to send the information contained in one qubit to another qubit.

6.2.3 Physical limitations on transfering qubits

Let's start with some good news: qubits can be transfered via a number of existing physical communication channels. For example, qubits can be represented by photons that can be transfered via existing optical fiber or via satellite connections. This is very interesting, as it means that the investment of telecom companies in classical physical communication infrastructure can largely be reused for quantum communication.

The bad news is that it is very difficult to preserve the state of the qubits over a long distance. The longer a photon travels over an optical fiber cable, the more likely errors will occur. At present, the maximum distance that can be covered is in the range of 100 km. This is large enough to be practical, but not large enough to allow for long-distance connections without additional solutions.

If we want to send qubits over long distances using existing optical fiber, we somehow need to be able to connect the different segments and have the qubits travelling from the end of one segment to the beginning of the next segment, as shown in Figure 6.10.

Figure 6.10 Sending a qubit over a long distance, covering multiple segments.

At first sight, this problem may look very similar to existing situation with classical communication: when using a signal to transmit data over a physical connection, the signal becomes weaker (the signal-to-noise ratio decreases) while traveling. At some point, the signal needs to be amplified to increase the signal-to-noise ratio again. This is done by so-called repeaters.

The problem with quantum communication is that we can not simply use a traditional repeater, as that will somehow measure the qubit and "amplify" it. But by measuring the qubit, the information it carries is gone. Hence, we need a different kind of repeater: a quantum repeater. The current technologies already allow to create such quantum repeaters, and it can be expected that these components, which are crucial for a real long-distance quantum network, will become available over the next years. In section 6.5, you will write a software solution that allows a quantum repeater to be created.

6.3 Pauli-Z gate and Measurement

Before we can work on the quantum algorithm that allows to overcome the obstacles described in the previous section, we will to introduce a new gate, and we have to spend a bit more time on what a measurement means in terms of a quantum program. We will use the new gate and the measurement block in the upcoming algorithm.

6.3.1 Pauli-Z gate

In section 3.4, we introduced the Pauli-X gate, which is often called the *Quantum NOT* gate. In section 4.3, we explained how this gate can be represented by the matrix

EQUATION 6.3

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

A variant to this gate is the Pauli-Z gate, which is represented by the following matrix:

EQUATION 6.4

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

If you want to know what this gate does when applied to a qubit, we have to multiply this matrix with the probability vector of the qubit.

The considered qubit is represented as

EQUATION 6.5

$$\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

and this corresponds with the following probability vector:

EQUATION 6.6

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

After the Pauli-Z gate is applied to this qubit, the probability vector is calculated as follows:

EQUATION 6.7

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ -\beta \end{bmatrix}$$

Hence, the qubit is now in the state

EQUATION 6.8

$$\psi\rangle = \alpha\,|0\rangle - \beta\,|1\rangle$$

When only this gate is applied to a qubit, the probability of measuring either `0` or `1` is not altered. Don't let the minus sign in front of β confuse you: the probability of measuring `1` is the square of

$-\beta$ which is still β^2.

The physical relevance of this gate is beyond the scope of this book. It is important though to realise that there are physical ways to realise this gate, and it corresponds to real quantum physics behavior, hence we can leverage it in software.

The symbol for this `Z` gate is shown in Figure 6.11

Figure 6.11 Strange symbol for the Pauli-Z gate

Similar to how the Controlled-Not gate, which is introduced in section 5.5.2, conditionally applies the Pauli-X gate when a control qubit would be measured as `1`, the Controlled-`Z` gate, or short Cz, is a two-qubit gate that applies a Pauli-`Z` gate when the control qubit measures as `1`.

The symbol for this Controlled-`Z` gate is shown in Figure 6.12

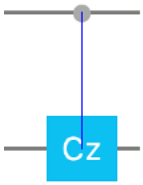

Figure 6.12 Strange symbol for the Controlled-Z gate

6.3.2 Measurements

We already talked about measurements before. In section 3.1, we explained that when a qubit is measured, it will always have either the value `0` or the value `1`. If the qubit was in a superposition state before the measurement, that superposition state is gone after the measurement. In many quantum simulators, including Strange, it is possible to do a measurement on a qubit in a program. Once a measurement is applied, the qubit is "destroyed" and the result is now a classical bit. Hence, it is not possible anymore to apply any gate that relates to superpostion. In Strange, you can do a measurement on a qubit by applying a "Measurement Operation" to it. While a measurement is not really a gate, Strange provides the `com.gluonhq.strange.gate.Measurement` class which extends from the `Gate` interface. The reason for this is that doing so, the `Measurement` class can benefit from functionality provided by the `Gate` interface and subclasses. We use the term *Measurement Operation* to make it clear that a measurement is not a gate.

When a Measurement Operation is applied to a qubit in Strange, the line representing the qubit flow will become a double line. The Measurement Operation itself is marked with an *M*, as

shown in Figure 6.13.

Figure 6.13 Strange symbol for the Measurement Operation

EXERCISE 6.1

You can now create a simple quantum circuit with 2 qubits. First, apply a Hadamard gate to the first qubit. Next, apply a Controlled-Z gate to the 2 qubits, with the first qubit being the control qubit. Finally, measure the 2 qubits.

6.4 Quantum teleportation

Because of its name, quantum teleportation is a concept that easily attracts attention. While we are not going to teleport a person from one physical location to another location, what we are going to discuss is an extremely important step towards a real quantum network.

6.4.1 The goal of quantum teleportation

In this section, you will send the information from a qubit that is held by Alice to a qubit held by Bob. Alice's qubit stays with Alice though, so we do not physically transfer the qubit. However, the end result is that you transfer some quantum information from Alice to Bob. That sounds very related to the core problem described in the previous sections: a qubit can not be cloned, but if we have a way to transfer its quantum information over a distance, we get much closer to a quantum network.

In the next sections, the algorithm for teleportation will be shown. You will step by step create a program that allows to achieve quantum teleportation, using the techniques and gates that were discussed before. It can be mathematically proven that the algorithm you are about to program indeed teleports the information from Alice to Bob. However, this prove is not in the scope of this book. We will rather artificially give some initial values to the qubit we want to teleport, so that we can later check if the teleportation was succesful.

6.4.2 Part 1, entanglement between Alice and Bob

The qubit held by Alice is shown in Figure 6.14 as qubit q.

Figure 6.14 Alice want to "send" her qubit q to Bob

A prerequisite for quantum teleportation is that Alice and Bob share an entangled pair of qubits, as shown in Figure 6.15:

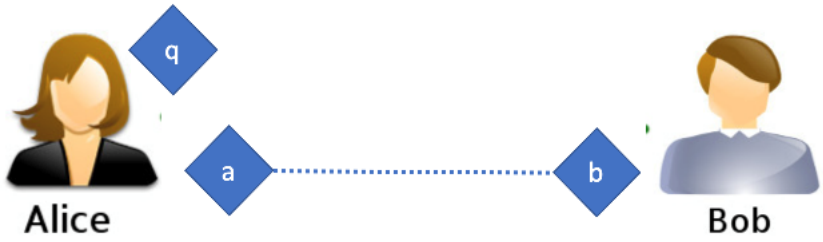

Figure 6.15 Alice and Bob share an entangled pair of qubits

From what you learned in the previous chapter, you can write the code to obtain this state. The follow code snippet shows how to achieve the situation from Figure 6.15:

Listing 6.2 Alice and Bob sharing an entangled pair

```
Program program = new Program(3);     ❶
Step step1 = new Step();
step1.addGate(new Hadamard(1));       ❷
Step step2 = new Step();
step2.addGate(new Cnot(1,2));         ❸
program.addStep(step1);               ❹
program.addStep(step2);
```

❶ This program contains 3 qubits: 'q', the qubit we want to teleport to Bob; 'a' and 'b' the entangled qubits

❷ Adding a Hadamard gate to qubit 'a'

❸ Adding a CNot gate to qubits 'a' and 'b'

❹ The steps (with the gates) created so far are added to the program.

Schematically, this code snippet is represented by the circuit in Figure 6.16

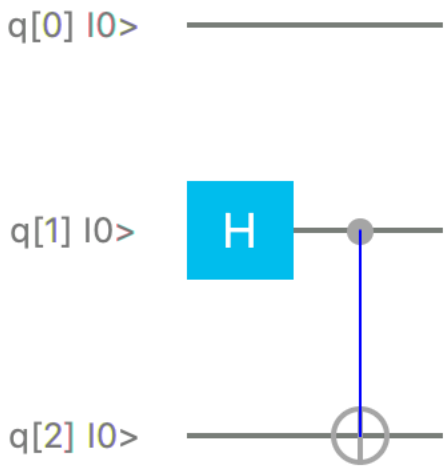

Figure 6.16 Circuit showing Alice with a qubit 'q' and Alice and Bob sharing an entangled pair.

Keep in mind that qubits `q[0]` (which is the qubit `q`) and `q[1]` (which is the qubit `a`) are located with Alice, where qubit `q[3]` (which is qubit `b`) is located with Bob.

6.4.3 Part 2, Alice operations

In the second part of the teleportation algorithm, Alice will let the qubit `q` interact with her component of the entangled qubit pair.

Schematically, this is shown in Figure 6.17:

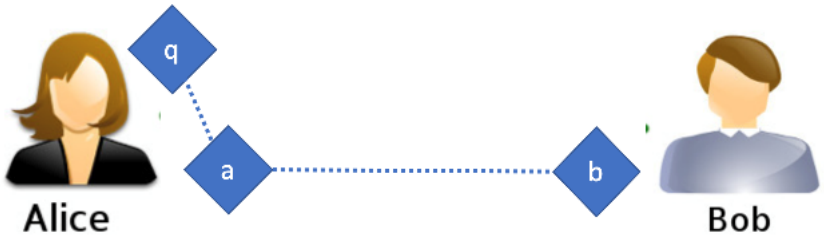

Figure 6.17 Alice let her qubit interact with her part of the entangled pair

It can be mathematically proven that the following steps will transfer the information from qubit `q` to the qubit `b` that is held by Bob. However, instead of the mathematical evidence, we will build the required code here, and then test it.

- Alice will first apply a CNot gate between her qubit q and her half of the entangled pair
- Next, Alice applies a Hadamard gate to her qubit.

The schematic representation of the quantum circuit we created so far is shown in Figure 6.18

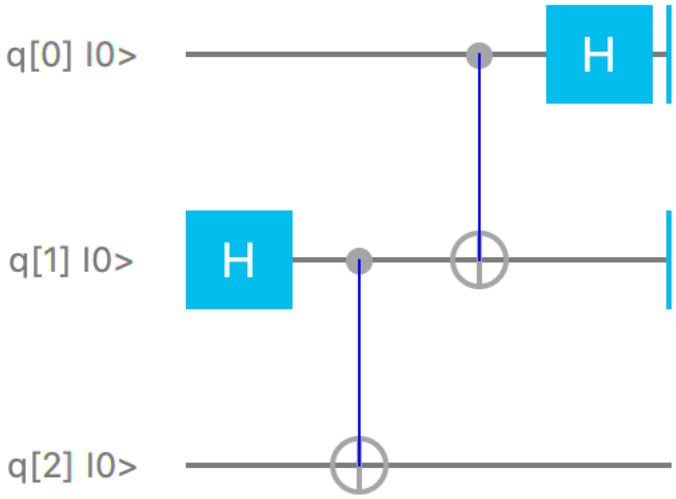

Figure 6.18 Adding the interaction between Alice's qubit and her half of the entangled pair

We added two steps to the quantum circuit, and in the code, this is achieved as follows:

```
Step step3 = new Step();
step3.addGate(new Cnot(0,1));          ❶
Step step4 = new Step();
step4.addGate(new Hadamard(0));        ❷
program.addStep(step3);                ❸
program.addStep(step4);
```

❶ Adding a CNot gate to the qubits q and a

❷ Adding a Hadamard gate to the qubit q

❸ The new steps (with the gates) we created are added to the program.

In the next step, Alice has to measure her 2 qubits.

The schematic representation of the quantum circuit we created so far is shown in Figure 6.19

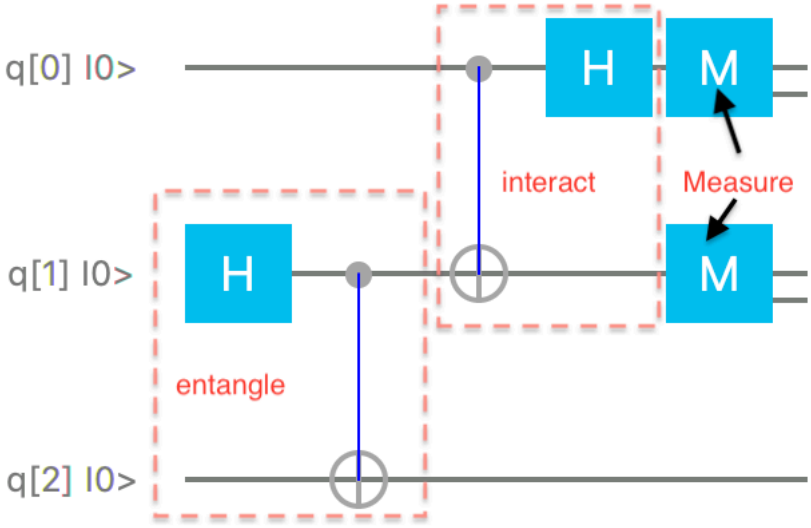

Figure 6.19 Alice measures her qubit and qubit A

The code that is required to perform these measurements is shown below:

```
Step step5 = new Step();
step5.addGate(new Measurement(0));
step5.addGate(new Measurement(1));
program.addStep(step5);
```

6.4.4 Part 3, Bob's operations

Finally, based on the measurements of Alice, Bob applies some operations on his qubit. If the measurement of the first qubit (the original qubit we want to teleport) is `1`, Bob will apply a Pauli-X gate to his qubit. If the measurement of the qubit A is `1`, Bob will apply a Pauli-Z gate to his qubit.

The schematic representation of the quantum circuit you created is shown in Figure 6.20

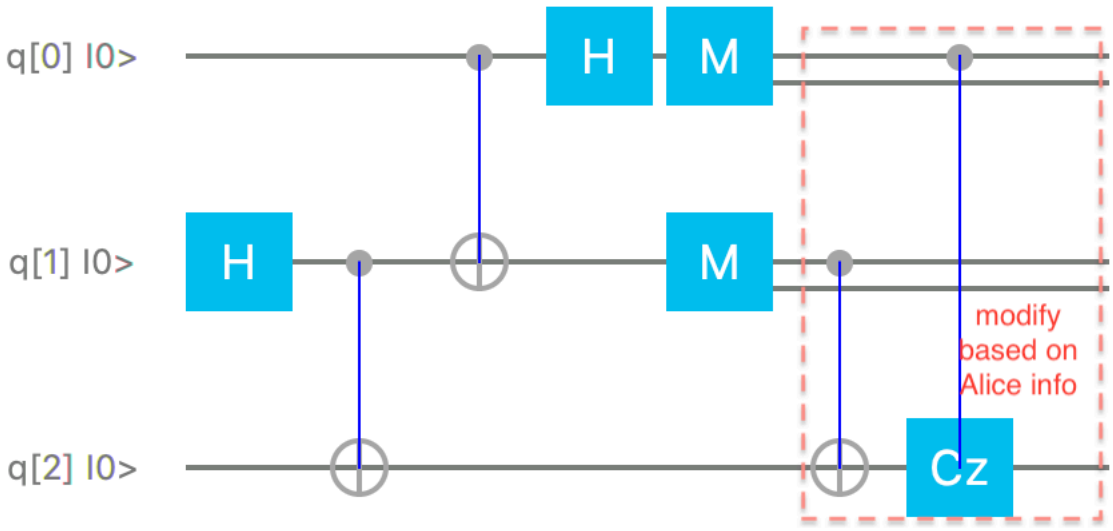

Figure 6.20 Depending on Alices measurements, Bob applies a Pauli-X and/or a Cz gate.

The code for these conditional steps is shown below:

```
Step step6 = new Step();
step6.addGate(new Cnot(1,2));                ❶
Step step7 = new Step();
step7.addGate(new Cz(0,2));                  ❷
program.addStep(step6);
program.addStep(step7);
```

❶ In case the qubit `q[1]` (which is `a`) is measured as `1`, apply a Pauli-X gate to `q[2]` (which is `b`)

❷ In case the qubit `q[0]` (which is `q`) is measured as `1`, apply a Pauli-Z gate to `q[2]`

That is all that is needed to teleport the information from qubit `q` to qubit `b`.

6.4.5 Running the application

You can now run the entire program, e.g. by running

```
mvn javafx:run
```

and you will see the output shown in Figure 6.21

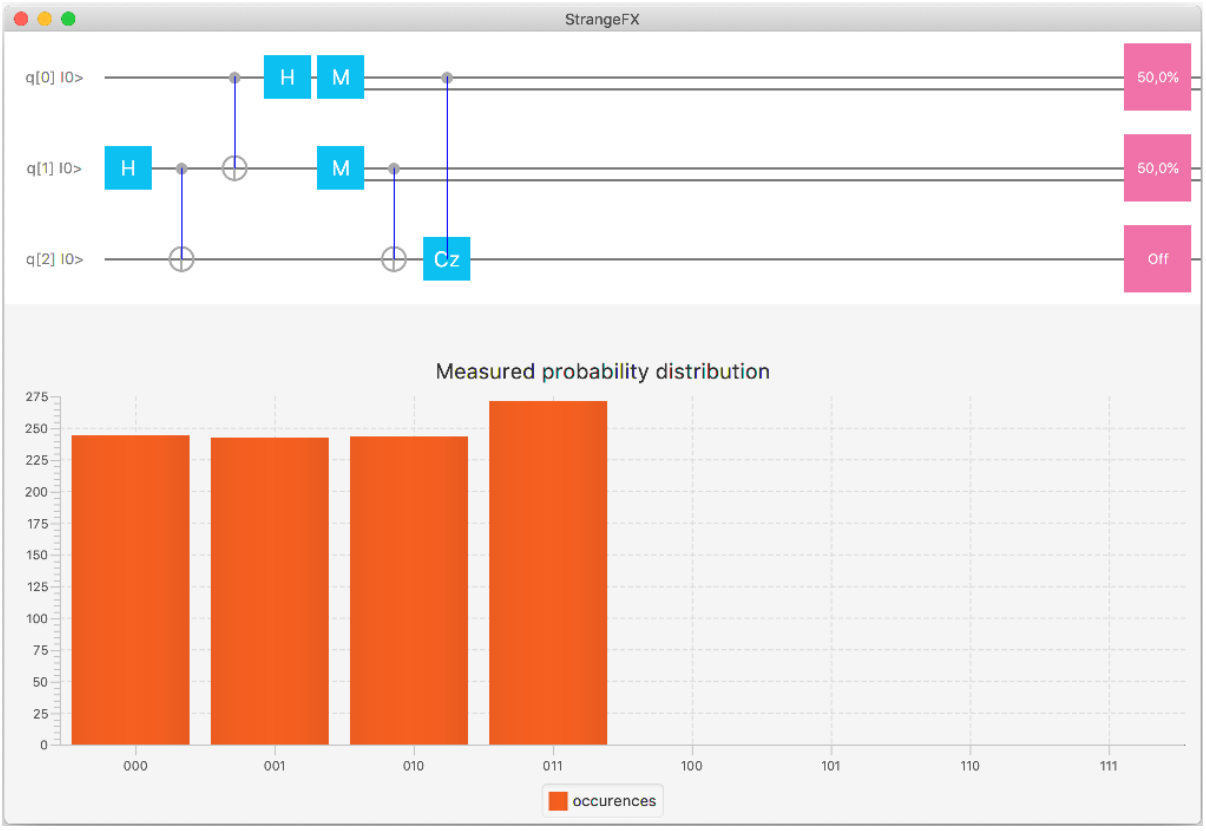

Figure 6.21 output from the teleport program

The output contains 2 parts. In the top half of the screenshot, you see the circuit with 3 qubits, with the probability to measure `1` at the right side. This probality result shows a number of things:

- there is 50% chance that the qubit `q` (denoted by `q[0]`) is measured as `1` and a 50% chance it is measured as `0`
- there is 50% chance that the qubit `q_A` (denoted by `q[1]`) is measured as
- the qubit `q_B` (denoted by `q[2]`) is guaranteed to be measured as `0`.

The last part is the most important one. Initially, `q` was holding the value `0`. At the end of the teleportation circuit, the value of `q` is not determined, but the value of `q_B` is now `0`. Hence, the information from qubit `q` is *teleported* to qubit `q_B`.

The same information can be obtained by analysing the bottom part of the figure. The bottom part shows the statistical results of 1000 simulated executions of our quantum teleportation

program. Of those 1000 runs, there were

- about 250 runs that had the outcome `000` (hence all qubits are measured `0`)
- about 250 runs with outcome `001`, which means `q` was measured `1` and both `q_A` and `q_B` were measured `0`
- about 250 runs with outcome `010`, which means `q` was measured `0`,
- about 250 runs with outcome `011`, which means `q` was measured `1`, `q_A` was measured `1` and `q_B` was measured `0`.

Note that in all those cases `q_B` is measured `0`. The other qubits can hold either `0` or `1`, but `q_B` is always 0.

These results seem to indicate that the algorihm you programmed is indeed teleporting a qubit from Alice to Bob. At least, when the value of Alice's qubit is `0`, the resulting value of Bob's qubit is `0` too.

EXERCISE 6.1

As an exercise, you can now check if the algorithm also works when Alice's qubit has the value `1`. In that case, we expect the end result of the algorithm always has the value of `q_B` to be `1` as well.

If you did the exercise correctly, you added a Pauli-X gate to the first qubit, so that Alice's qubit was holding a `1` before the whole teleportation algorithm started. This shows that the algorithm is also working when the qubit is in the `1` state, but what if it is in a superposition?

Fortunately, Strange has a capability to test this. Before a program is executed, you can initialize the value of a qubit, using the `Program.initializeQubut (int index, double alpha)` method. In this method, `index` specifies the index number of the qubit we want to set, and `alpha` specifies the value we want to give to $alpha$.

For example, add the following line to the program, before the `runProgram` is invoked:

```
program.initializeQubit(0, .4);
```

Doing this will set the α value of the original qubit `q[0]` to `0.4`. The probability to measure `0` is the square of α , which means there would be `16%` chance to measure `0` and hence `84%` chance to measure `1`.

If you run the modified program, you will now see something similar to the the output shown in Figure 6.22

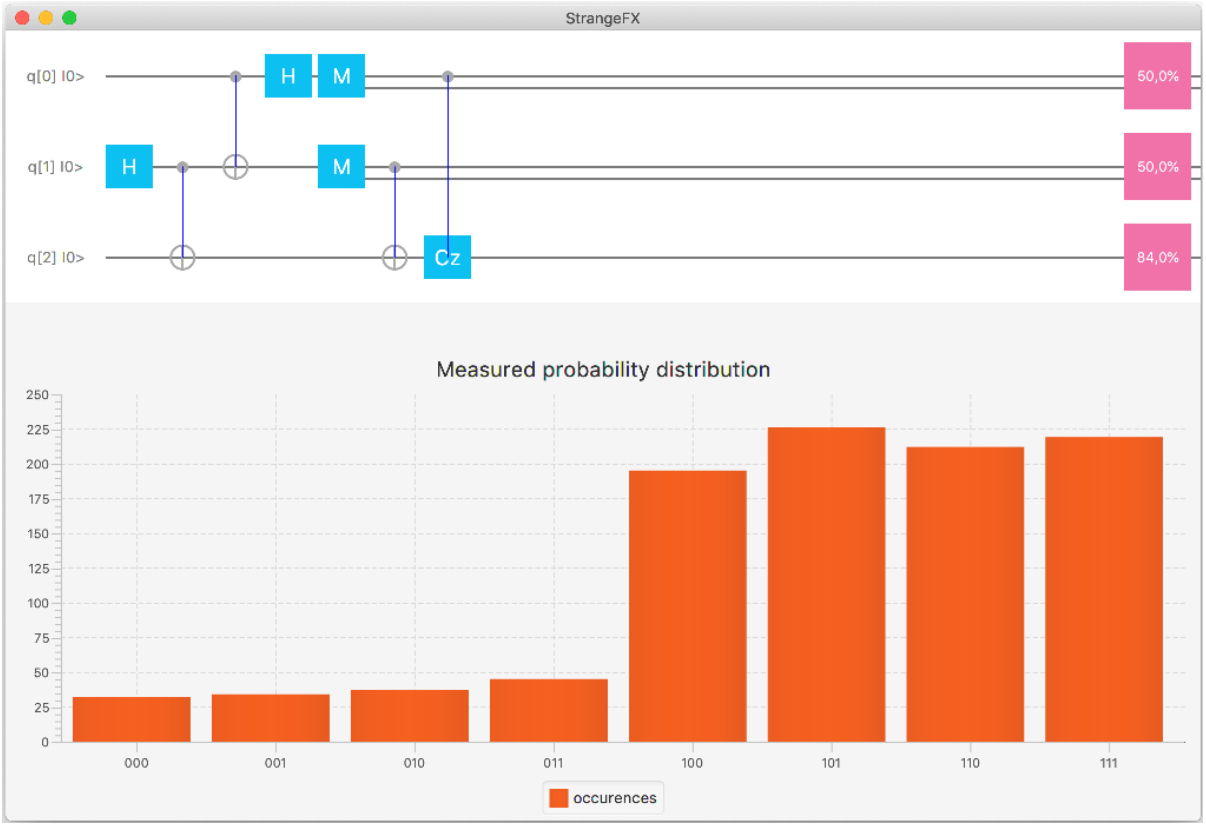

Figure 6.22 Output from the teleport program with an initialized qubit

From the top half of the output, we see that there is now `84%` chance that Bob's qubit `q[2]` will now be measured as `1`. The bottom part of the figure shows a similar result. This is exactly what we hoped for. It shows that the algorithm *teleports* the quantum information originally contained in the qubit held by Alice, to a qubit held by Bob. Again, note that we didn't provide mathematical evidence, which is beyond the scope of this book. Evidence can easily be find in a number of online resources, e.g. www.ryanlarose.com/uploads/1/1/5/8/115879647/quic02.pdf .

Congratulations, you just sent a qubit from one person to another!

6.4.6 Quantum and classical communication

A very important thing to note is that there is only very limited "quantum interaction" between Alice and Bob. In the first step of the algorithm, you created an entangled pair of qubits, were one part of the entangled pair is held by Alice, and the other part is held by Bob. Apart from this step, there is no quantum communication needed between Alice and Bob. Whether Bob needs to apply a Pauli-X gate, a Pauli-`Z` gate, or nothing at all, depends on the outcome of two measurements done by Alice. The result of a measurement is always a classical bit, hence the outcome can be transfered using a classical network.

Therefore, the communication aspects of the quantum teleportation algorithm can be split into two steps:

1. Make sure Alice and Bob each have a qubit that belongs to an entangled pair
2. Perform classical communication to send the 2 measurement result (either `0` or `1` from Alice to Bob.

Schematically, this is shown in Figure 6.23

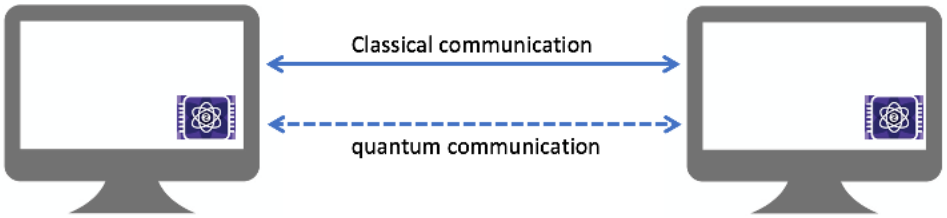

Figure 6.23 Communication between Alice and Bob is split between a classical and a quantum channel.

In this figure, it is shown that the entangled qubit travels over a quantum channel, while the outcome of the measurements travels over a classical channel.

As a result, if we have a device that can create an entangled pair of qubits, and if that device can send one of these qubits to Bob, we can transfer *any* qubit from Alice to Bob without additional requiring quantum interactions between Alice and Bob.

6.5 A quantum repeater

In the previous section, you managed to transfer the information from one qubit to another qubit, without violating the no-cloning theorem. If we can transfer entangled qubits, we can transfer the information contained in a qubit.

But what if Alice and Bob are located very far away from each other (e.g. more than 1000 km apart)? The classical data channel is not a problem. If the signal to noise ratio drops too much, classical repeaters can be used to amplify the signal. However, it becomes extremely difficult to send one of the entangled qubits from Alice to Bob.

In this section, you will create a software solution for this problem that leverages the code you already wrote earlier in this chapter.

We need to make use of *quantum repeaters*. A quantum repeater will not *amplify* the signal in a qubit (as that would require to measure it, hence destroy the information), but it can use the same quantum teleportation to transfer the information from one segment to the next segment as explained in Figure 6.24

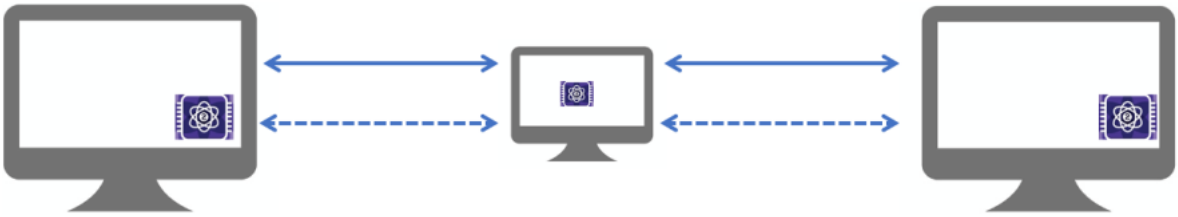

Figure 6.24 A quantum repeater separating the distance between Alice and Bob into segments, transfering the qubit from one segment to the other.

Before we show the code for creating this quantum repeater, let's have a look at the high level how we are going to do this. In the relative simple case where Alice and Bob are close to each other (that is, close enough to send half of an entangled pair from Alice to Bob), the situation is shown in Figure 6.15

The case with a quantum repeater in between Alice and Bob is shown in Figure 6.25

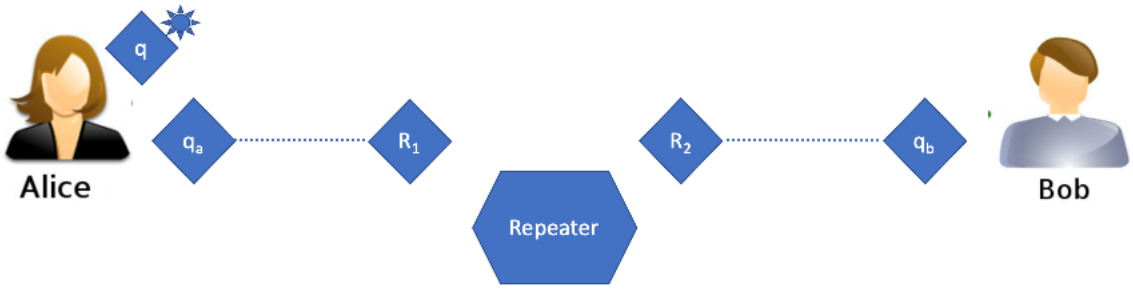

Figure 6.25 A quantum repeater between Alice and Bob

In this case, there are 2 entangled pairs:

- Alice and the Repeater share an entangled pair with qubits *qA* and *R1*
- The Repeater and Bob share an entangled pair with qubits *R2* and *qB*

The code for the quantum repeater can be found in the directory `ch06/repeater` of the sample repository. Since we are now dealing with 5 qubits (1 qubit which information we want to teleport, and 2 entangled pairs of qubits), the Program is now constructed as follows:

```
Program program = new Program(5);
```

The preparation of the system now requires the creation of 2 entangled pairs. In the quantum teleportation code, the entangled pair was created at the beginning of the code. We extend this now as follows (the lines we added are annotated):

```
Step step1 = new Step();
step1.addGate(new Hadamard(1));
step1.addGate(new Hadamard(3));                    ❶
Step step2 = new Step();
step2.addGate(new Cnot(1,2));
step2.addGate(new Cnot(3,4));                      ❷
```

❶ Add a Hadamard gate to `q[3]`

❷ Add a CNot gate between `q[3]` and `q[4]`

Note that we added a Hadamard gate to qubit 3 and a CNot gate between qubits 3 and 4, which creates an entangled pair between qubits 3 and qubits 4.

Apart from this preparation, the first part is to transfer the information from qubit *q* to qubit *R1*, using the teleportation algorithm created above.

The flow of this first part is shown in Figure 6.26

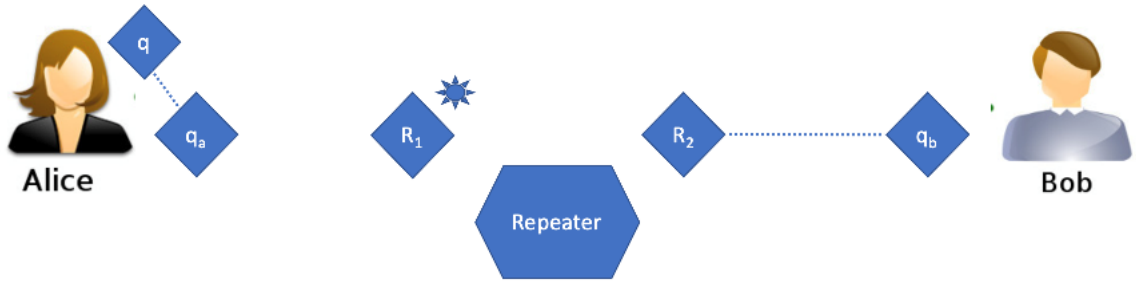

Figure 6.26 Alice interacts with the first entangled pair, teleporting her qubit to R1

The code for this is identical to the code we had before in the teleportation algorithm, and we won't duplicate it here.

The information that was originally in Alice's qubit *q* is now in the qubit *R1* at the Repeater. Next, we repeat the teleportation algorithm, this time teleporting the information in *R1* to *qB* The flow of this second part is shown in Figure 6.27

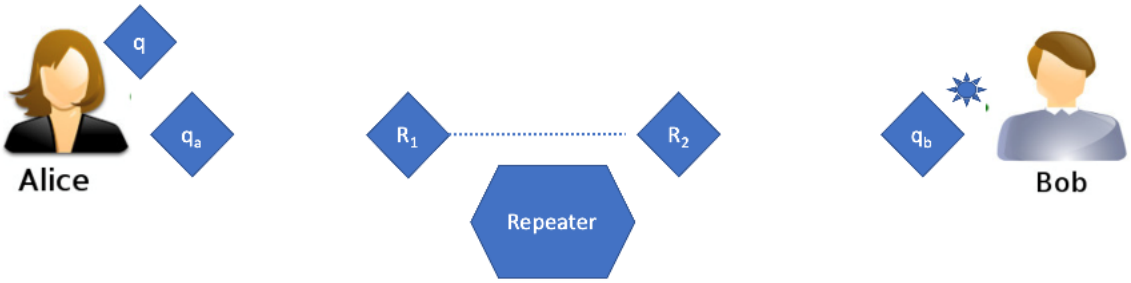

Figure 6.27 The repeater let the R1 qubit, interact with the second entangled pari, thereby teleporting the information in the qubit to qB

The code required to do this is very similar to the first part, but the gates now operate on different qubits. In the first part, steps 3 till 7 were doing the teleportation. You will now add steps 8 till 12 to perform similar operations to the other qubits:

```
Step step8 = new Step();
step8.addGate(new Cnot(2,3));                    ❶
Step step9 = new Step();
step9.addGate(new Hadamard(2));                  ❷
Step step10 = new Step();
step10.addGate(new Measurement(2));              ❸
step10.addGate(new Measurement(3));              ❹
```

```
Step step11 = new Step();
step11.addGate(new Cnot(3,4));                    ❺
Step step12 = new Step();
step12.addGate(new Cz(2,4));                      ❻
```

❶ Add a CNot gate between `q[2]` and `q[3]`

❷ Apply a Hadamard gate to `q[2]`

❸ Measure `q[2]`

❹ Measure `q[3]`

❺ In case `q[3]` is measured as `1`, apply a Pauli-X gate to `q[4]`

❻ In case `q[2]` is measured as `2`, apply a Pauli-Z gate to `q[4]`.

Note that in the code in the sample repository, we artificially initialized the original qubit (the one we want to transfer) similar to how we did it before so that it has `16%` to be measured as `0` and `84%` to be measured as `1`. This is achieved using the same code as in the quantum teleportation algorithm:

```
program.initializeQubit(0, .4);
```

Note that we only do this so that the results are easier to interpret.

When the program is now executed, the result should be similar to the output in Figure 6.28

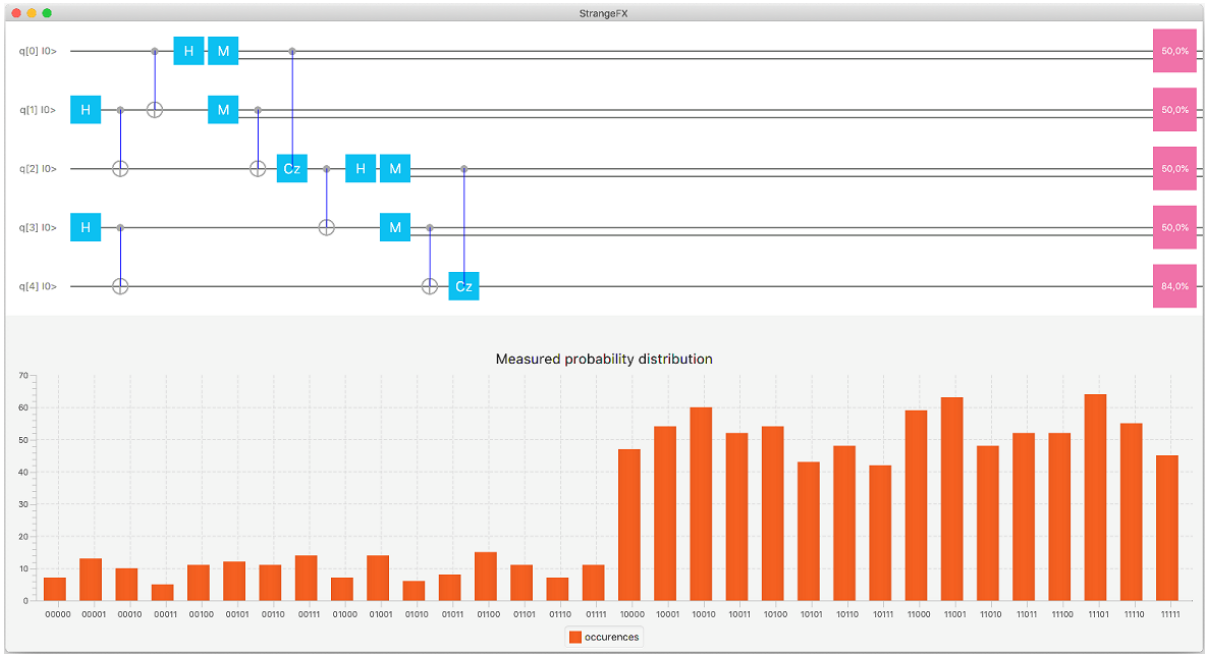

Figure 6.28 Result of the quantum repeater program

From this output, it is indeed clear that the information originally contained in Alice's qubit `q[0]` is transfered to the Qubit held by Bob, `q[4]`

6.6 Summary

Many interesting applications of quantum computing depend on quantum networking, where qubits are send from one quantum computer to another quantum computer.

In this chapter,

- you learned about the differences between classical and quantum networks.
- you learned the issues that are typically encountered with quantum networks.
- you created a Java application that is capable of *teleporting* the information from one qubit to another qubit
- you created the software code for a *quantum repeater* that allows to teleport qubits over a longer distance.

7

Our HelloWorld explained

This chapter covers:

- the different components between low-level quantum hardware and high-level programming languages
- the main options for developers leveraging quantum computing
- a brief overview of some quantum computing simulators and their approach
- a description of how Strange allows for high-level programming where no knowledge about quantum computing is involved, as well as low-level programming which requires more understanding of quantum computing, but which is more
- flexible
- how to debug quantum applications using Strange and StrangeFX
  the different runtime targets: local, cloud or real device

Tools in software development have specific goals. Some tools are helping with developer productivity, others help managing dependencies, or give easy access to specific frameworks. Developers using those tools should be very aware of what the tools they use can do, and what their limitations are. In this chapter, we explain the benefits of quantum computer simulators, and we explore some of the specific features of Strange that makes the use of Quantum Computing algorithms easy for existing (Java) developers. Strange, like any other quantum computer simulator, is not going to solve all your application issues by applying a quantum sauce to it. It will help developers though, to leverage benefits of quantum computing without being an expert in quantum computing. In order to maximally benefit from the advantages Strange is offering, some understanding of quantum computing tools in general is helpful. That is the focus of this chapter.

The Java code for the HelloWorld sample in chapter 2 is very familiar to Java developers. The goal of Strange is to provide a library that is both familiar to Java developer, but that is also capable to leverage quantum phenomenons discussed in the earlier chapters.

For some developers, quantum computing will be an implementation "detail" that they don’t

need to worry about. For others, leveraging the right quantum computing concepts in the right place, can be the main differentiator of their application.

With Strange, both options are possible. We will discuss the typical stack for a high-level programming language on top of quantum hardware. Before we do that, we make the analogy with classical stacks. There are 2 reasons why we do this:

1. The options to either write low-level code that allows to exploit specific hardware functionality or high-level code that is not dealing with any hardware specifics exists in the classical stack as well. We can learn from the classical approach in order to come up with options for a quantum stack.
2. We will explain why the quantum stack and the software stack are different and why we can't simply have a classical software stack on top of quantum hardware

7.1 From hardware to high-level languages

There are typically many steps between the hardware operations of a computer, and the high-level programming languages used by developers. Schematically, the flow in Figure 7.1 shows a classical software stack running on top of hardware (a CPU).

> **NOTE** the relevant hardware of a classical stack consists of more than only the CPU. However, the goal of this chapter is not to explain the classical hardware-software stack, hence we make this over-simplication.

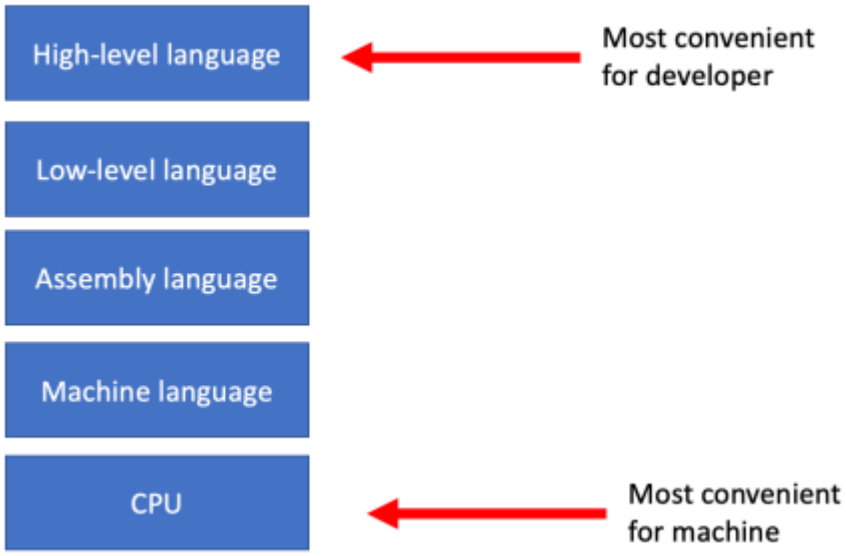

Figure 7.1 Typical software stack on top of a classical CPU. At the bottom, the hardware is depicted. The top of the stack is a high-level language used by software developers. The different layers in between allow for the next layer to be build on top of the previous layer.

The machine language is very integrated with the CPU. Different types of CPU's have different kinds of machine language. Assembly language is a more readable format, but it is still very dependent on the CPU type. Low-level languages abstract most of the CPU-specific architecture, but may still require differences for different types of CPU's (e.g. 32 bit or 64 bit based). A high-level language like Java does not depend on hardware at all.

Figure 7.2 shows the relative amount of code reuse for code written in the different layers in the stack for 2 different CPU's: AMD64 and AARCH64. At the very low level, the machine language for those CPU's is very different and there is no code reuse at all. The higher in the stack, the smaller the differences, and the more code can be reused. Finally, in the Java layer, there is 100% code reuse. A Java application that runs on an AMD64 CPU has the exact same sourcecode as a Java application that runs on an AARCH64 CPU.

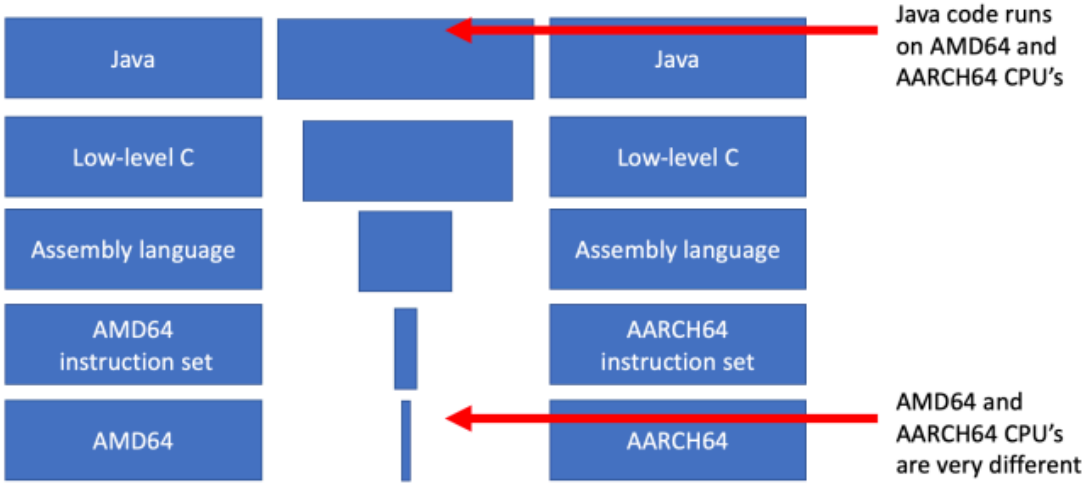

Figure 7.2 Comparison of code reuse when targeting 2 different CPU's. The wider the bar in the middle column, the more code reuse. Code for high-level languages typically runs on different kinds of hardware. Low-level code requires parts that are only relevant to a specific architecture (e.g. AMD64 or AARCH64).

Compilers and linkers make sure that the application written in a high-level programming language can ulitmately be executed leveraging the specific hardware that belongs to a specific computer. One of the reasons for the success of the Java Platform, is that it allows developers to write applications in a single language (e.g. Java) and then execute these application on all kinds of hardware, ranging from cloud servers over desktop running Windows, MacOS X or Linux to mobile and embedded devices. At the lower levels, there are a large number of differences between the different target systems, but developers are shielded from those differences.

7.2 Abstractions at different levels

Applications written in a high-level language like Java can leverage different types of hardware. For example, Java applications can be executed on a linux system with an AMD 64 CPU, but also on a Linux system with an AARCH64 CPU, or on a Windows system with an AMD64 CPU.

One may wonder if we can expect a quantum chip to replace the existing classical chips, and have existing applications running on top of those quantum chips. If that is the case, the schema in Figure 7.2 would also apply to the case where the CPU is actually a quantum computer. In that case, we can keep all our existing languages and libraries, and add another low-level abstraction layer that translates the high-level language (e.g. Java) into sort of an assembly language for quantum hardware.

However, as you learned in the previous chapters, there are a number of things that make quantum hardware very different from classical hardware, e.g. superposition (see Chapter 3) and entanglement (see Chapter 4). If we want to leverage the quantum capabilities of quantum processors, the layers above the hardware should leverage these capabilities. That means we need to leverage superposition and entanglement in the stack towards high-level application languages. Those concepts are not present in the classical assembly languages.

IMPORTANT If we want to leverage the real power of quantum computing, the core concepts (e.g. superposition and entanglement) need to be used inside the software stack. That does not mean that they have to be exposed at the top-level of any high-level language

There are a number of approaches for this:

1. Don’t make abstractions at all, and propagate the quantum characteristics to the high-level application language. In this case, developers need to understand and use the quantum concepts like superposition and entanglement.
2. Make abstractions at the low level, and have high-level languages leveraging these abstractions. In this case, developers do not need to understand anything about quantum computing. It requires high-level languages to be very aware of all aspects of quantum computing. Decisions about whether to do something using a classical or quantum approach need to be taken to the language.
3. Something in the middle.

The first approach is followed by Microsoft, where the third approach is followed by most other initiatives. With Strange, we use the third approach as well. The second approach would allow most developers to leverage quantum computing without even understanding the basics of it.

This is not unrealistic in the very long term but it will take a very long time before languages are capable enough to hide all quantum characteristics from the high-level development. Even then, there are use cases where it is beneficial to directly use quantum characteristics.

7.3 Other languages

Strange is not the only quantum computer simulator out there. There are a large and growing number of quantum simulators that follow the same or a different approach. A number of big IT companies (Microsoft, IBM, Google) created a quantum computer simulator as well.

Microsoft created a DSL (Domain Specific Language) called Q#, based on the analogy of C# and F#. The advantage of a DSL is that it allows to add very specific features in the language that can be leveraged by the developer. Doing so, it is possible to optimize applications for quantum features like superposition and entanglement. The drawback of this approach is that developers need to learn yet another new language, and it also requires a rather deep understanding of quantum computing.

IBM and Google took a different approach. They created a simulator in Python, which is clearly an existing language. The advantage of this approach is that Python developers do not need to learn a new language if they want to get started with Quantum Computing. This is the same advantage that Java developers have when using Strange.

7.3.1 Resources

As mentioned before, the area related to quantum computer simulators is rapidly growing. It is impossible to give a list today that would still be complete by the time this book is published. There are some online resources though that are kept up-to-date with new evolutions.

Below, we provide a few pointers to relevant resources, but keep in mind that these resources may become outdated, or moved to different locations.

- An exhaustive list of quantum simulators, sorted by programming language can be found at https://www.quantiki.org/wiki/list-qc-simulators
- The IBM Qiskit project is available at https://qiskit.org/
- Microsoft has information about its Q# programming language at https://docs.microsoft.com/en-us/quantum/
- Cirq, a quantum simulator in Python created by Google, can be found at https://cirq.readthedocs.io/

7.4 Strange: high-level and low-level approach

The HelloWorld sample that you created in Chapter 3 uses the top-level API of Strange. You also learned that the top-level API leverages a low-level API. For convenience, we repeat the high-level architecture diagram here in Figure 7.3.

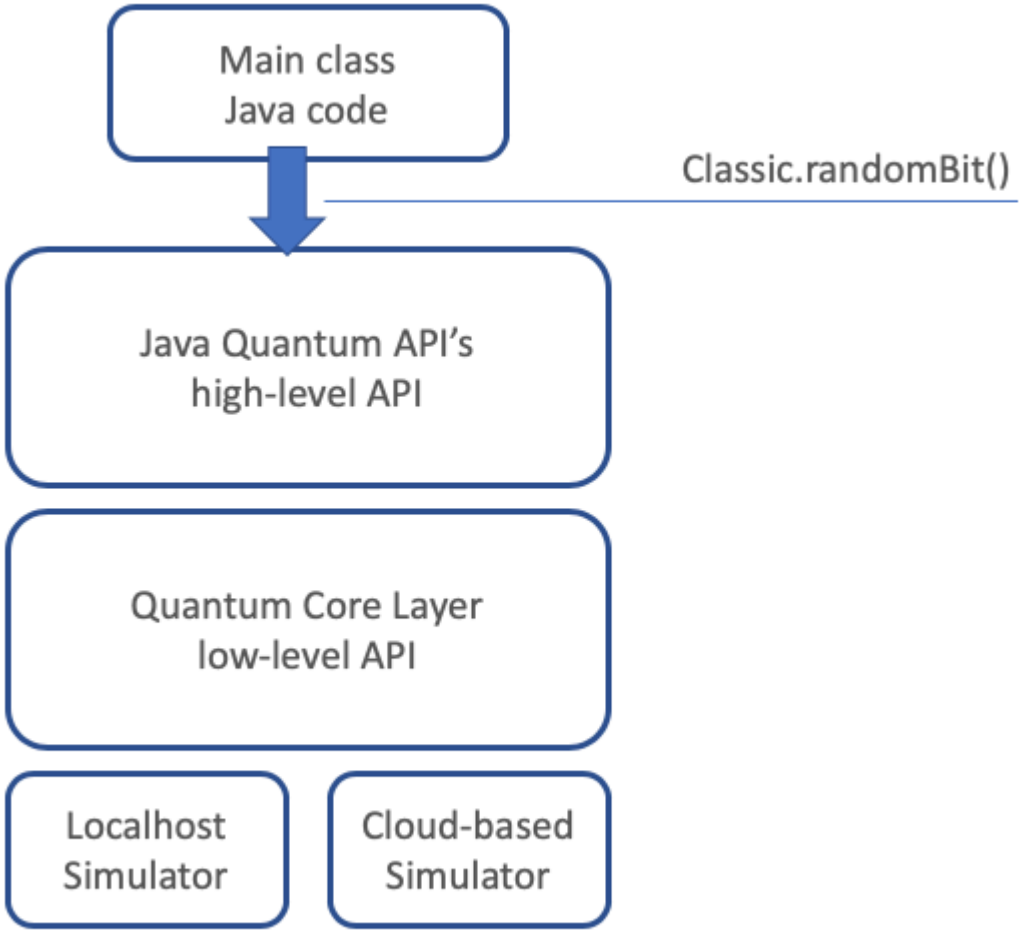

Figure 7.3 HelloWorld sample and Strange high and low level API's

The high-level API's focus on Java, where the low-level API's deal with quantum gates. If you want to develop using the high-level API's, you focus on Java code. If you want to develop using the low-level API's, you focus on quantum circuits with quantum gates. Internally, the implementation of the high-level API's depend on the low-level API's, as explained in Figure 7.4

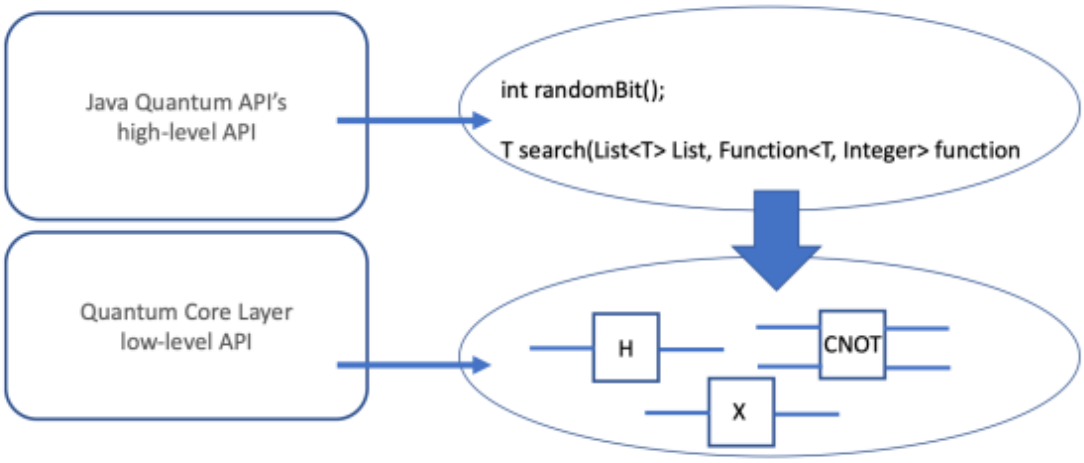

Figure 7.4 high-level and low-level API's and their implementation

Hence, the high-level and the low-level API's ultimately use the same low-level concepts. The difference is that the high-level API hides the complexity of these concepts to developers.

7.4.1 Top-level API

The Strange top-level API is a typical Java API, following the normal Java patterns. The API for this is in the class

`com.gluonhq.strange.algorithm.Class`

> **NOTE** At the time of writing, Strange is at version 0.0.9. Until the Strange 1.0 version is released, API's may change location.

Some examples of methods in this class are

public static int randomBit(); public static int qsum(int a, int b); public static<T> T search(List<T> list, Integer> function);

The API deals with some of the restrictions of quantum computing. For example, once a qubit is measured, it can not be used in a circuit anymore. This restriction comes from the real quantum world, where measuring the physical representation of a qubit destroys the information in that qubit. However, the Java developer should not worry about this restriction. The Strange top-level API's are created in a way that they can not enforce situations that are not compatible with real quantum systems.

Let's repeat the most important line of the `HelloWorld` sample from Chapter 2, where a random bit was generated:

int randomBit = Classic.randomBit();

The `Classic.randomBit()` doesn't throw an exception. Hence, the developer can assume that the implementation does everythng to ensure that there are no inconsistenties with the quantum world. Also, the concept of quantum gates is never exposed in the top-level API.

IMPORTANT The signature of the high-level API's do not depend on quantum specific objects. For example, the return value of a high-level API call is never a qubit.

7.4.2 Low-level API

The Strange low-level API's are spread over different packages. The typical approach that is followed when using those API's can be summarized as follows:

- create a quantum program with a given number of qubits
- add a number of quantum gates to this program
- run the program
- measure the qubits, or process the probability vector.

In Chapter 2, we showed how the high-level `Classic.randomBit()` method leverages the low-level API and we promised to go in more detail in this chaper. By now, you have already seen more low-level code samples, to the implementation of the `Classic.randomBit()` method will probably look more familiar.

Let's repeat the code here:

Listing 7.1

```
public static int randomBit() {
    Program program = new Program(1);                          ❶
    Step s0 = new Step();                                      ❷
    s0.addGate(new Hadamard(0));                               ❸
    program.addStep(s0);                                       ❹
    QuantumExecutionEnvironment qee =                          ❺
            new SimpleQuantumExecutionEnvironment();
    Result result = qee.runProgram(program);                   ❻
    Qubit[] qubits = result.getQubits();                       ❼
    int answer = qubits[0].measure();                          ❽
    return answer;                                             ❾
}
```

❶ A new quantum Program is created, using 1 qubit

❷ A new step is created that will added later to the Program

❸ A Hadamard gate, working on the first qubit (with index `0` is added to the new step

❹ The step is added to the Program

❺ A runtime is created.

❻ The quantum Program is executed

❼ The final state of the qubits is asked. Note that although this is an array, the array contains a single qubit only since the Program started with a single qubit

❽ The value of the qubit is measured. It will be either `0` or `1`.

❾ The resulting value is returned to the caller.

As you can see from this code, whenever the `randomBit()` function is invoked, a new quantum Program is created and executed. The return value, however, is a plain Java integer, and has no quantum information associated with it. This marks the clear separation between the low-level API's and the high-level API's.

It is another major difference between the high-level and the low-level API's, and it is explained in Figure 7.5.

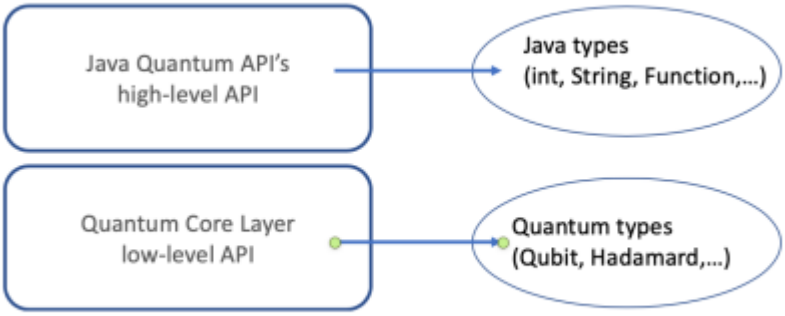

Figure 7.5 high-level and low-level API's and the types that they use

Java developers that wish to use existing Java types only, can do this by leveraging the high-level API's. Developers who are more familiar with quantum concepts, or want to experiment with those types, can use the low-level API's for this.

7.4.3 When to use what

Developers can choose to use the high-level API's or they can use the low-level API's. In the previous sections, you learned about the differences between the high-level API's and the low-level API's. Below, we summarize the different reasons for using either the high-level or the low-level API's. Keep in mind that it is very ok to use both approaches. There are cases where the high-level API's are more suitable, and cases where the low-level API's are more appropriate.

It is recommended to use the high-level API's in case

- You need to work on a project where an existing, well known and already implemented quantum algorithm can provide an advantage --- typically a performance advantage

- You want to experiment with classical code that could benefit from quantum algorithms.

It is recommended to use the low-level API's in case

- you want to learn about quantum computing
- you want to experiment with existing quantum algorithms
- you want to develop new quantum algorithms

7.5 StrangeFX, a development tool

The success of most popular classical programming languages is partly due to the availability of tools that allow developers to be very productive in that language. Almost all Java developers are using an IDE when they create applications.

Similarly, in order to make programming quantum applications productive, there is a need for tools that makes development easier. With StrangeFX, developers can easily visualise and debug their quantum applications.

7.5.1 Visualisation of circuits

The quantum circuits discussed in the previous chapters were relatively simple. The programming approach is easy to follow for Java developers. However, it often helps to get a visual overview of the created quantum circuit. Especially when programs become more complex, such a visualisation becomes more important. As we explained in Chapter 3, the StrangeFX library allows for a quick visualisation of quantum circuits. A call to
Renderer.renderProgram(program);

generates a window with a graphical overview of the circuit.

The quantum program in the "randombit" directory in the sample repository shows the visualisation. It also contains debug elements being discussed in the next section, and for now, we show the code without the debug elements:

Program program = new Program(dim); Step step0 = new Step(new Hadamard(0), new X(3)); Step step
Step(new Cnot(0,1)); program.addSteps(step0, step1); QuantumExecutionEnvironment qee = new
SimpleQuantumExecutionEnvironment(); Result result = qee.runProgram(program); Qubit[] qubits =
result.getQubits(); for (int i = 0; i < dim; i++) { System.err.println("Qubit["+i+"]: "+qubits[i].measure()
Renderer.renderProgram(program);

Running this program shows the measurements of the qubits, but also the circuit, including the probabilities for the qubits. The output of the measurements can be either

Qubit[0]: 0 Qubit[1]: 0 Qubit[2]: 0 Qubit[3]: 1

or

Qubit[0]: 1 Qubit[1]: 1 Qubit[2]: 0 Qubit[3]: 1

The visualisation of the circuit, which is shown in Figure 7.6 helps understanding these 2 different possible outcomes.

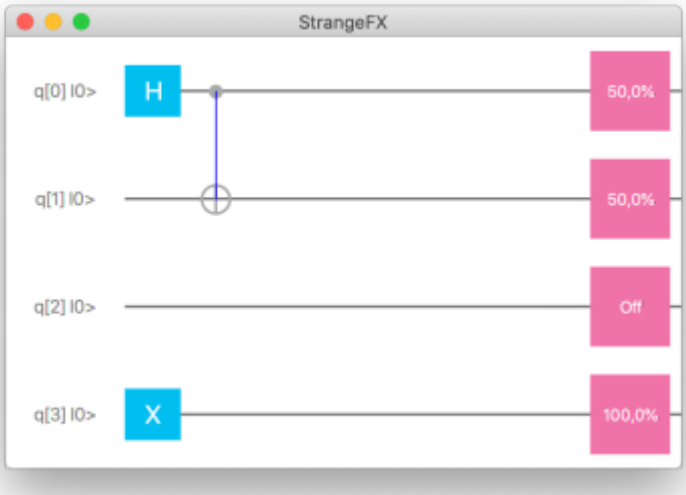

Figure 7.6 visualisation of the circuit

The visualisation shows that the quantum program starts with 4 qubits. In a first step, a Hadamard and a Not gate is added to the circuit. Next, a CNot gate is applied to gates *q0* and *q1*. At the right side of the picture, the resulting qubits are shown, with the probability that they are measured as `1`.

7.5.2 Debugging Strange code

In the previous chapters, you learned how to create simple and more complex quantum circuits. One of the most important restrictions of quantum circuits is that measuring a qubit influences its state. If a qubit is in a superposition state, and it is measured, it will fall back to either `0` or `1`. It can't go back to the state it was before it was measured.

While this provides great opportunities for security (as we will explain in the next chapter), this makes debugging quantum circuits hard. In a typical classical application, you often want to follow the value of a specific variable during the program flow. Debuggers are very popular with developers, and examining the change in a variable often provides valuable insight in why a specific application is not behaving in the way a developer expects it to behave.

However, if measuring a variable changes the behavior of the application --- as is the case in quantum computing --- this technique can not be used.

To make it more complex, even if we would be able to restore the original state of a qubit after measuring it, the measurement itself, being `0` or `1` doesn't give all information. As we explained

a number of times before, the real value in quantum programs is not only the measured value of a qubit, but mainly the probability distributions.

Fortunately, Strange and StrangeFX allow for a way to render the probability distributions. Strange allows to use a fictive gate, the `ProbabilitiesGate` which can be used to visualise the probability vector at a given moment in the program flow.

We'll reuse the program from the previous section, but this time we use the `ProbabilitiesGate` to render the probabilities after a given step.

The first part of the code is changed as folllows:

Listing 7.2

```
Program program = new Program(dim);
Step p0 = new Step (new ProbabilitiesGate(0));          ❶
Step step0 = new Step(new Hadamard(0), new X(3));       ❷
Step p1 = new Step (new ProbabilitiesGate(0));
Step step1 = new Step(new Cnot(0,1));
Step p2 = new Step (new ProbabilitiesGate(0));

program.addSteps(p0, step0, p1, step1, p2);
```

❶ A new step is created containing a ProbabilitiesGate

❷ The original steps are still created

Running the sample again shows the same circuit, but this time, you see a probability vector being displayed after each step, as shown in Figure 7.7.

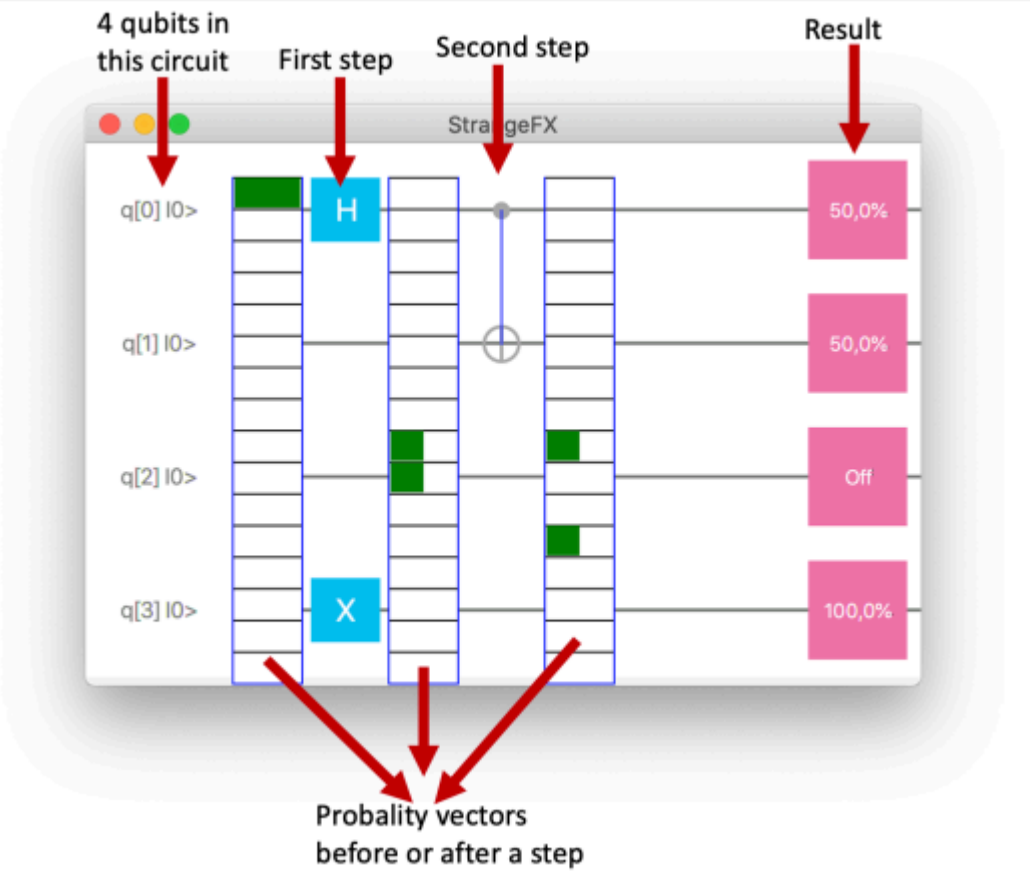

Figure 7.7 Visualisation of the circuit with probabilities. Before and after each step, the probability vector is shown. This gives an indication about what possible outcomes there are after each step, without measuring the qubits.

Let's have a closer look to what is happening.

The first step that is added to the program contains a `ProbabilitiesGate`. This does not change the probability vector at any point, but it triggers the renderer to display the vector. Zooming in on the left side of the visual output, we see in Figure 7.8 a probability vector immediately after the qubit declaration, and before the Hadamard and Not gates are applied.

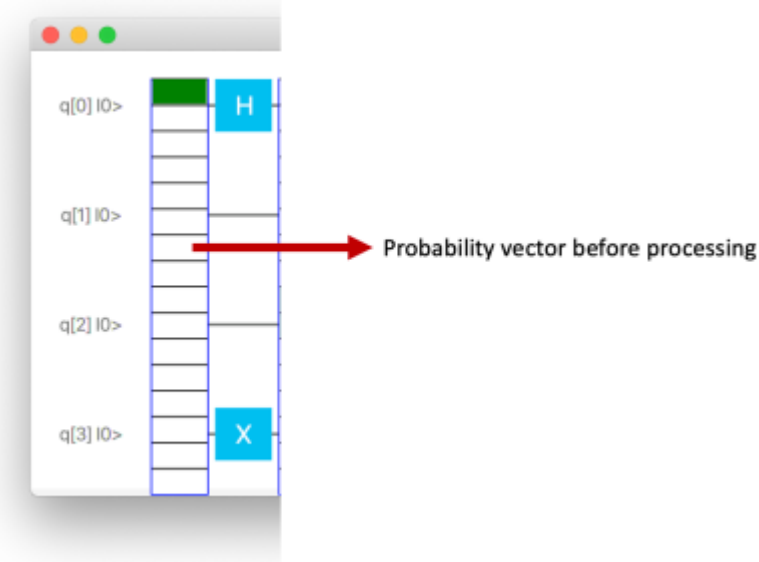

Figure 7.8 Probability vector before any processing is done

The probability vector is visualised as a rectangle divided in 16 parts. The first part represents the probability of measuring `0000` in case a measurement would be done at this point. The second part corresponds to the probability of measuring `0001` and so on.

The more a part is colored, the higher the corresponding probability. In this case, the first part is entirely colored, which means the probability of measuring `0000` is 100%. This is indeed what you would expect from a quantum circuit with 4 qubits and no gates. All qubits initially are in the `0000` state, and that is what you would measure.

After applying the first real step, which contains a Hadamard gate and a Cnot gate, another probability vector is rendered. This vector is highlighted in Figure 7.9 together with the corresponding qubit measurements.

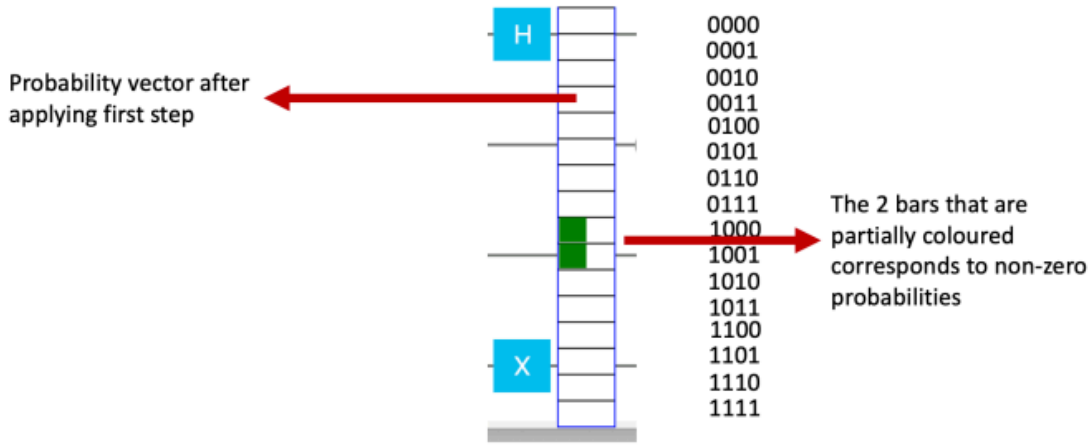

Figure 7.9 Probability vector after applying a Hadamard and a Not gate. The '1000' and '1001' states are the only possible ones after this step, wich an equal probability. Rendering this vector does not measure the qubits, so the processing can still continue.

This figure shows the probabilities of measuring one of the 16 combinations, in case a measurement would be made at this point. Again, keep in mind that we are talking about 16 probabilities, and not about the individual values of 4 qubits.

From the figure, it becomes clear that there are 2 possible outcomes for a measurement at this stage:

- 50% chance that we would measure `1000`
- 50% chance that we would measure `1001`

This corresponds to what we would expect when analysing the single step that has been applied to this circuit so far. Applying the NOT gate to the most significant qubit (q3) will cause that qubit to be measured as `1`. Without applying the Hadamard gate to qubit q0, the status would thus be `1000`. Applying the Hadamard gate to this qubit will result in 50% chance to measure this qubit as `0` and 50% chance to measure it as `1`. In summary, there will be 50% chance that after this step, the system is in the `1000` state and 50% chance the system will be in the `1001` state, exactly what is shown by the probability vector.

The second step applies a CNot gate on the qubits 0 and 1. The resulting probability vector is shown in Figure 7.10.

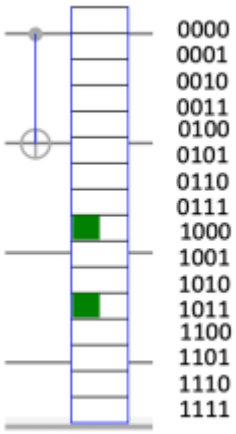

Figure 7.10 Probability vector after applying a CNot gate.

This figures indicates that there are 2 possible outcomes to measure at this point:

- 50% chance to measure `1000`
- 50% chance to measure `1011`

This corresponds with what you learned in Chapter 5, when creating a Bell state. Applying a Hadamard gate, followed by a CNot gate, brings the 2 involved qubits (q0 and q1) in an entangled state. Both qubits can be `0` and both qubits can be `1`. If both qubits are `0`, the total state of the quantum circuit is measured as `1000`. If both qubits are `1`, the total state is measured as `1011`.

This matches the final visualisation of the circuit outcome, shown in Figure 7.11.

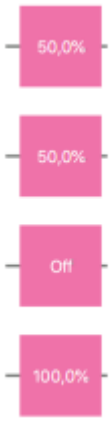

Figure 7.11 Different possible measurements

From this last figure, we know that q2 will always be measured as `0` and q3 will always be measured as `1`. The other 2 qubits, `q0` and `q1` can be either `0` and `1`.

At first glance, this might correspond to what we learned by looking at the probability vector, but there is some important information that is missing when looking at the possible measurements for the qubits only. Indeed, the probability vector states that the first two qubits can be either `00` or `11` but they can never be in `01` or `10` because they are entangled. There are only 2 possible combinations, not 4. This is something that can not be seen from simply looking at the potential outcomes for qubit measurements.

IMPORTANT The probability vector contains more information than the list of qubits with their individual probabilities. The latter misses the information about possible or impossible combinations, which is exactly what is available in the probability vector, as in this one, each entry deals with all qubits.

7.6 Simulators, cloud services and real hardware

Ultimately, the reason for creating quantum algorithms is to execute them on real quantum hardware. In order to take advantage of the special characteristics of quantum computing, we need to use real quantum devices. Understanding the real benefits of quantum computing, and writing algorithms and code that leverage quantum computing takes time. Using a quantum simulator, you can learn the principles of quantum computing, and you can create applications that can benefit from quantum hardware. If you master quantum algorithms by the time quantum hardware becomes available, you have a competitive advantage.

It is desired and expected that applications written for a quantum simulator can also work on real quantum hardware without any change, or only with limited changes, related to configuration. Therefore, as a developer, you focus on the application code, and not on the execution environment.

This is shown in Figure 7.12

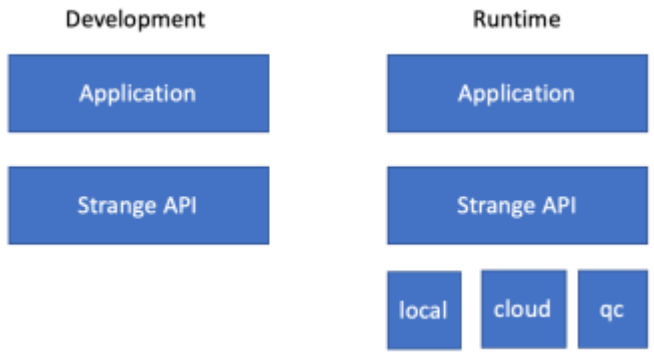

Figure 7.12 Development stack versus runtime stack

When developing applications that leverage quantum computing via Strange, developers use the public API's that are exposed by Strange. These can be the high-level API's, the low-level API's, or a combination.

When running those applications, there are a number of options:

- you run your application on a local simulator
- you run your application on a cloud simulator
- you run your application on real hardware (or) in a cloud

During the development phase, when you're writing the application, testing it, integrating it with other components, running the application on a local simulator is the easiest approach. You don't need quantum hardware for this, and you don't need to setup a connection to a cloud service. The drawback is clear: a local simulator requires more resources, and doesn't provide the performance that can be expected from a real quantum device.

The second option, running applications in a cloud simulator, is becoming available at the time of this writing. A number of cloud companies are offering cloud API's for doing quantum development. Inside the cloud, both simulators as well as real devices can be used. This adds another abstraction layer: applications can talk to cloud services, and their request might be served by real quantum hardware or quantum simulators.

This is shown in Figure 7.13.

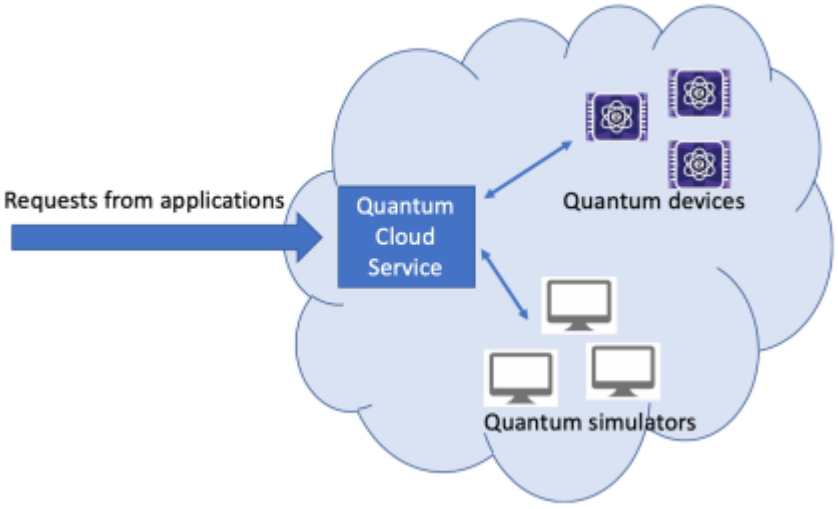

Figure 7.13 Cloud services with real quantum hardware or simulators.

In this figure, the Cloud service offers a single API. Based on a number of criteria, it can be decided to forward the request internally to a real quantum device, or to a quantum simulator in the cloud. Quantum simulators in cloud environments can benefit from the large scale and

virtualisation typical cloud providers offer. They can thus use more memory and CPU power compared to local quantum simulators.

Many experts expect that the first batch of commercial quantum computers will mainly be deployed in cloud environments. This will address some of the challenges introduced by the current prototypes for quantum computers, e.g. cooling the environment to almost zero Kelvin. Dealing with infrastructure requirements is easier done in a cloud facility then in on a deskopt or laptop in a private home environment. Obviously, a quantum processor on a mobile phone is even harder to create.

As long as cloud providers provide the same API's for accessing real quantum devices in their cloud as well as classical quantum simulators, the developer does not have to worry about it.

The Strange API's provide another level of abstraction. In Chapter 3, we talked about the interface `QuantumExecutionEnvironment`. We explained that this interface defines the methods that can be leveraged by developers for executing quantum applications, without having to specify where the program is being executed.

At this moment, Strange only contains a single implementation of this `QuantumExecutionEnvironment`, and that one is used throughout his book and samples: `SimpleQuantumExecutionEnvironment`. However, work is underway to provide more implementations that communicate with external cloud services for accessing third party quantum cloud environments. The major benefit for developers using Strange today, is that their applications will work tomorrow on third-party cloud services with simulated or real hardware. The only change that will be required is to change the `SimpleQuantumExecutionEnviroment` into `CloudQuantumExecutionEnvironment`

Based on the current draft work in Strange, this would require applications that are currently using the following snippet

Program program = new Program(...); ... QuantumExecutionEnvironment qee = new SimpleQuantumExecutionEnvironment(); Result result = qee.runProgram(program);

into using the new snippet:

Program program = new Program(...); Map<String, String> params; ... QuantumExecutionEnvironmen CloudQuantumExecutionEnvironment(params); Result result = qee.runProgram(program);

In this snippet, the `params` parameter provided to the `CloudQuantumExcecutionEnvironment` constructor contains information that allows Strange to select the most appropriate cloud service, and to provide relevant info (e.g. credentials) to this cloud service.

7.7 Summary

- you learned the difference between low-level and high-level quantum API's, and when they should be used. High-level quantum API's are easier to use than low-level quantum API's, and you should use them when they are directly applicable to your application. In case no high-level API exists (yet), you can combine the low-level API's to create your own algorithm.
- you wrote code that visualizes a quantum application, and that helps in understanding the programming flow.
- you debugged a quantum application by looking at the probabilities after each step. Using StrangeFX, you can render the probability vector, and check whether that matches with your expectations. Doing so, you don't need to manually recalculate each step.
- you learned about the different execution environments for running quantum applications. You learned that quantum offerings via a cloud service are already available.

8 Secure communication using Quantum Computing

This chapter covers

- solving the bootstrap problem of secure communication
- an introduction to Quantum Key Distribution
- a step-by-step composition of the BB84 algorithm
- a Java application that shows how Java developers can securely distribute shared keys between two parties.

In this chapter, you will create a useful quantum application. We will show that quantum computing allows you to create a secret key that can be shared between 2 parties in a very secure way. This so-called Quantum Key Distribution (QKD) is the basis for a number of encryption techniques that are proven to be secure — even the best quantum computer can't break this security!

8.1 The bootstrap problem

We started Chapter 6 by showing how classical networks are used to send classical information from one node (or computer) to another. We explained how different pieces of information travel from Java applications to a low level implementation, where they are sent as bits to the other node, and travel upwards again. This is shown in Figure 8.1.

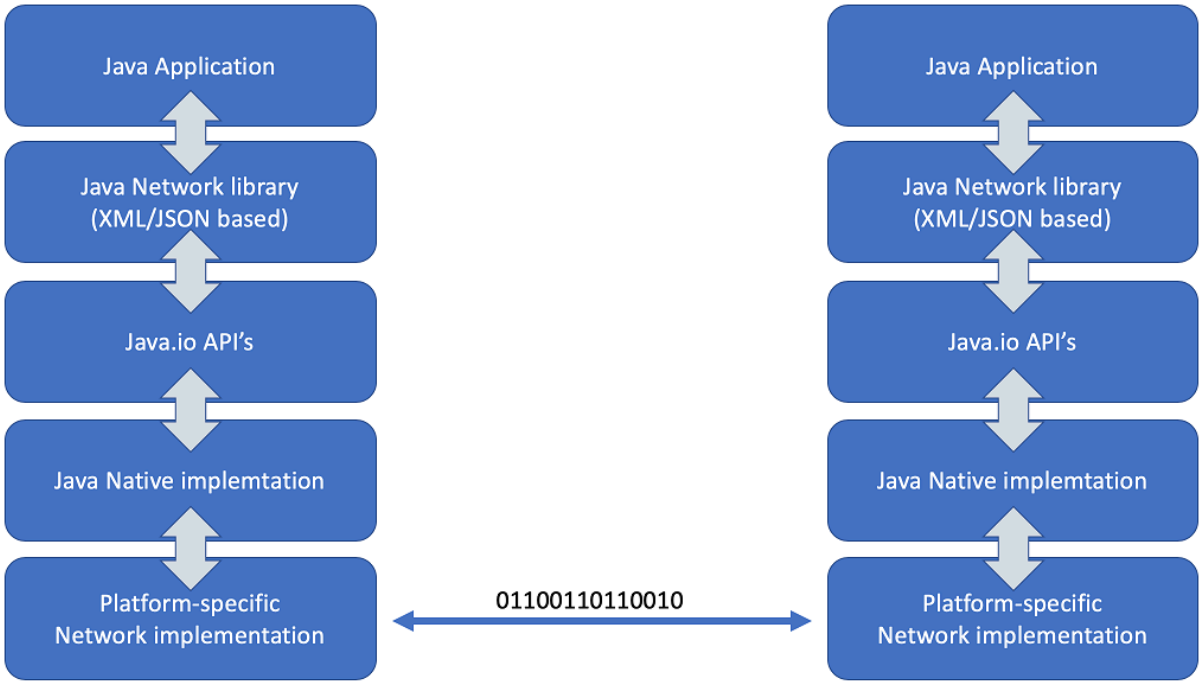

Figure 8.1 Java applications using classical communication.

8.1.1 Issues with sending bits over a network

In this chapter, we focus on what happens at the communication level between the 2 nodes. Bits are transfered over a network connection, e.g. over optical fiber.

How secure is this? Security and privacy are gaining importance, and for many applications, it is crucial that the bits that are sent over physical networks between computers are not intercepted by third parties, and can also not be altered by third parties.

The ideal situation is shown in Figure 8.2.

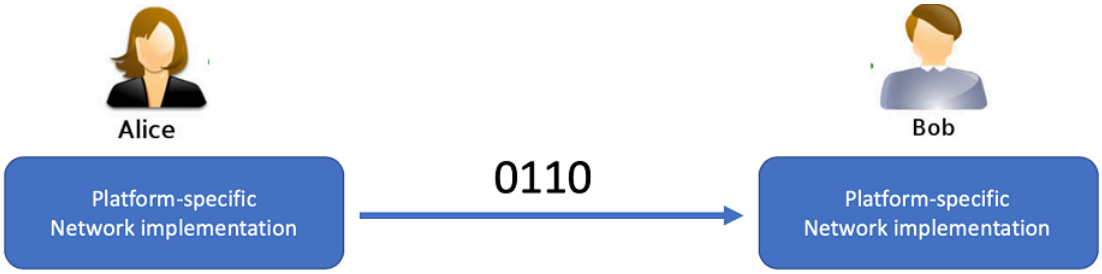

Figure 8.2 Ideal situation in network communication.

In this situation, the bits are sent from Alice to Bob, and nobody is listening or altering the bits. Alice can send a message to Bob, and Bob will receive this message. Nobody else received or modified the message.

READING MESSAGES

However, in practice, it is very well possible that there is an eavesdropper on the line, as shown in Figure 8.3.

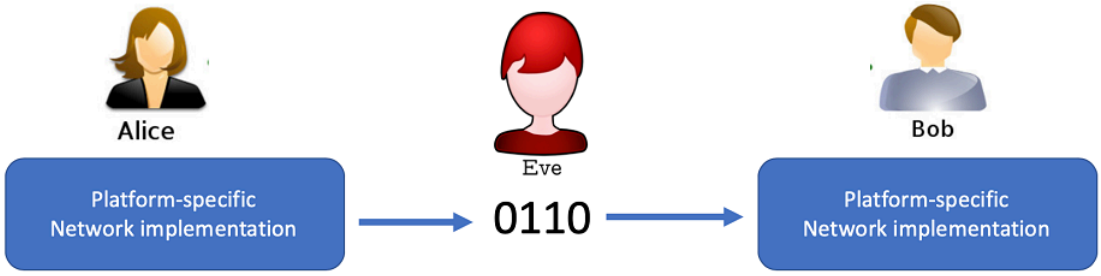

Figure 8.3 Eve, the eavesdropper, reads the network communication.

There are different scenarios how this can happen. The eavesdropper, who is often depicted as "Eve", can physically cut the network cable, listens to the incoming bits, write them down, and then send the same bits to the other part of the cable. Whatever technique Eve is using, the result is that she can read the bits that are sent over the communication channel between Alice and Bob.

Because the bits still arrive at Bob's end, neither Alice or Bob will know that Eve has been eavedropping. Alice and Bob think they communicated in a secure way, but Eve listened to everything they exchanged.

MODIFYING MESSAGES

The other problem that should be faced is that Eve might be altering the bits on the network line. For example, in Figure 8.4 it is shown that Eve switches the third bit from `1` to `0`.

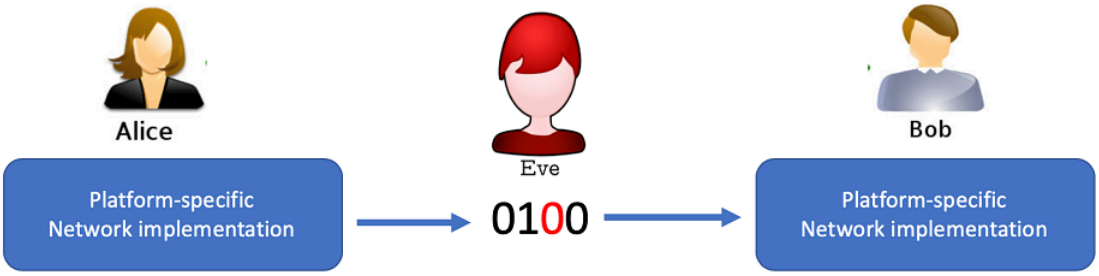

Figure 8.4 Eve, the eavesdropper, modified a bit

This can lead to serious problems. For example, suppose Alice is sending the following message to Bob: "Let's meet at 8 am." Eve intercepts the message, and alters it into "Let's meet at 5 pm.". Bob receives that message, and isn't aware of any manipulation by Eve. You can imagine Alice and Bob have some issues.

In many real-world situations, it is impossible to completely secure the physical channel that provides the low-level communinication between two parties. Rather than assuming this, we

need techniques that allow us to create secure communication channels built on top of insecure networking channels.

Fortunately, there are a number of classical techniques that improve the security and privacy of communication. We are not going to cover all these techniques, but we'll pick one that is very popular: the *one-time pad*.

8.1.2 One-time pad to the rescue

The state of every message, every object, every data that is used in classical computing can be written down as a sequence of bits. A one-time pad is a series of bits, at least equally long as the original message that needs to be transfered. If the source (Alice) and the receiver (Bob) of the message have access to the same one-time pad, and if Eve has no access to it, it is possible to securely encrypt the message in a way that only Alice and Bob can decrypt it. The "One-time" part of one-time pad means that the key should only be used once. If this is the case, it can be proven that the message encrypted with the one-time pad is really transmitted in a secure way.

Let's give an example.

> **NOTE** In the following samples, we are going to use very short sequences of bits, to keep things simple. Keep in mind though that the principles apply to very long sequences as well.

Suppose that Alice wants to send the following message (a bit sequence) to Bob:

```
0110
```

Before Alice and Bob started to communicate, they agreed on a secret key (a one-time pad). We will discuss later how they created this, but for now, let's assume this is their secret key:

```
1100
```

Only Alice and Bob know this key. Before Alice sends the message to Bob, she combines the message with the secret key: every bit in the original message is replaced by the XOR operation applied on the original bit and the corresponding bit in the key:

- if the original bit and the corresponding bit in the pad are equal (both `0` or `1`), the resulting bit is `0`
- if the original bit and the corresponding bit in the pad are opposite (hence `0` and `1` or `1` and `0`), the resulting bit is `1`

The result of this combination is an encrypted bit sequence.

```
0110  (original message)
1100  (one time pad)
----  (XOR operation)
```

```
1010  (result)
```

Alice now sends the encrypted bit sequence '1010' to Bob, who needs to decode the sequence. To do this, Bob also applies an XOR operation on every bit he receives with the corresponding bit in the key:

```
1010  (encrypted message, received from Alice)
1100  (one time pad)
----  (XOR)
0110  (original message)
```

As you can see, the result of this operation is the exact original message that is sent by Alice. It can be proven that this is not coincidence, and this always works regardless of what message is sent by Alice.

Schematically, the communication between Alice and Bob can now be represented by Figure 8.5

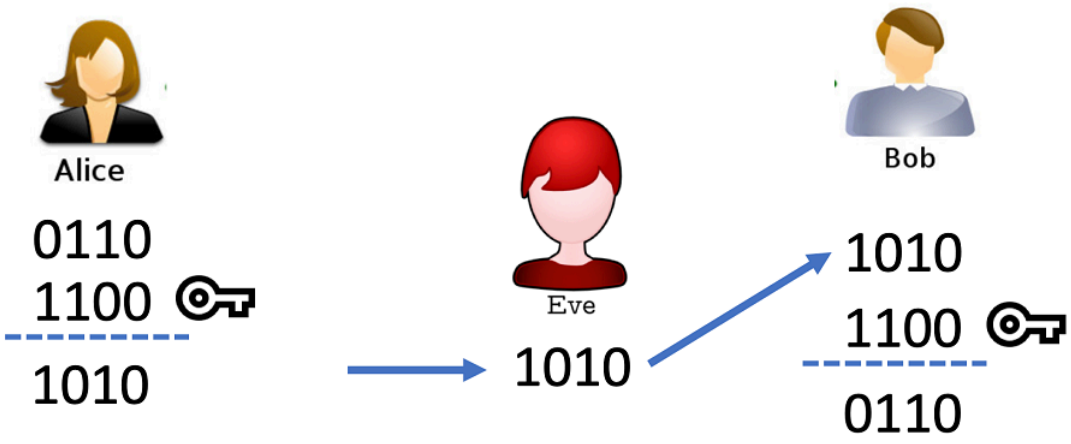

Figure 8.5 Alice and Bob communicating via a one-time pad

What happens if Eve still intercepts the message sent over the network? Instead of reading the original message ('0110'), she will read an encrypted message ('1010'), and since she doesn't have the key that Alice and Bob use, she can not decypher that message. Even if she knew that Alice and Bob used a secret key, and encrypted their message by using a bitwise XOR operation, without the secret key itself there is no way she can decrypt the message.

You just learned that if Alice and Bob share a secret key, a sequence of bits that has the same length as the original message, their communication can not be intercepted. Actually, it can be intercepted, but the eavesdropper can not decrypt the intercepted message.

8.1.3 Sharing a secret key

The difficult question though is how Alice and Bob can share a secret key. The naive approach would be to send a key over the network… but that brings us back at square one: we need a secret key to send our secret key in a secure way over the network.

This recursive problem is visualised in Figure 8.6

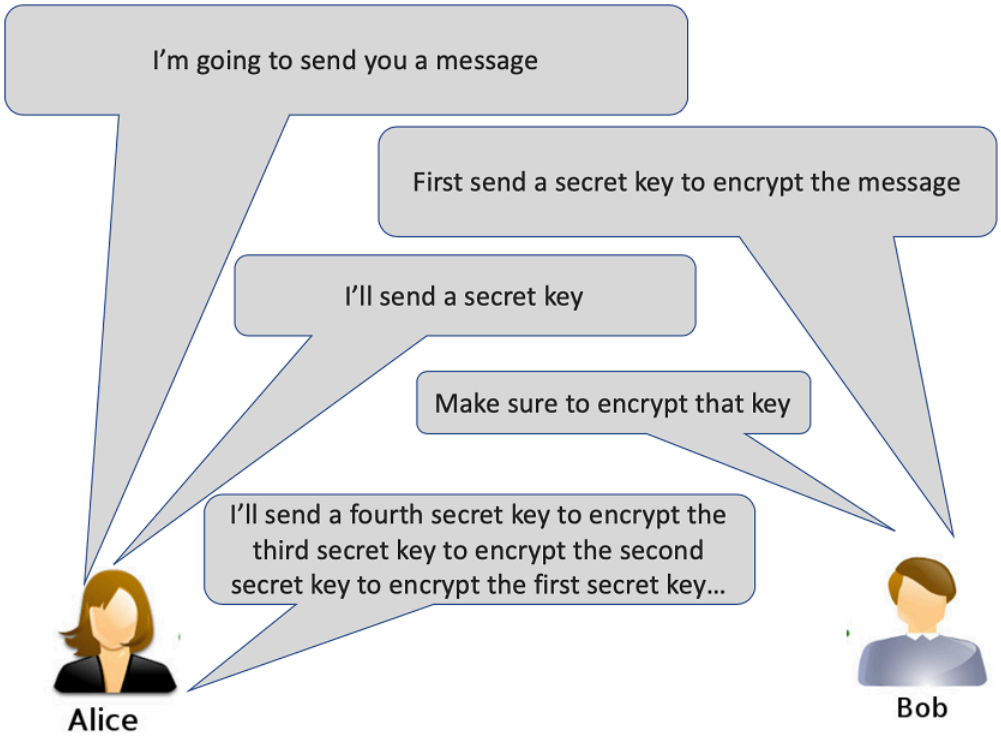

Figure 8.6 Alice and Bob discover the bootstrap problem

For real critical applications, secret keys are often not shared via the internet, but via traditional post or other ways.

In the remainder of this chapter, you will learn how quantum computing can fix this bootstrap problem.

8.2 Quantum Key Distribution

In this section, you'll learn how quantum computing can be used to generate a secret key and share it **in a secure way** between two parties, Alice and Bob. Once Alice and Bob have such a key, they can use it to encrypt the messages they want to send to each other. If we can share a secret key between Alice and Bob in a secure way, the bootstrap problem explained in the previous section is fixed.

The generation and distribution of such a secret key using quantum techniques is called Quantum Key Distribution, or QKD, and it is often considered one of the hot topics and key advantages of quantum computing.

There are a number of algorithms that can be used to generate QKD. Perhaps the most well-known algorithm is the BB84 algorithm, named after its inventors Charles Bennett and

Gilles Brassard who created it in 1984. We will eventually create this algorithm, but instead of starting from the physics behind the algorithm, we will use a software-oriented approach to arrive to the algorithm.

In the following section, we assume that we can *somehow* send qubits over a network. In Chapter 6, we discussed Quantum Teleportation, which allows us to send the status of a qubit over a classical network connection, provided the two parties share an entangled qubit before the classical communication starts. Later in this chapter, we will introduce a project that simulates (and eventually will provide) a real quantum network. Using this quantum network, we can send qubits from one node (or computer) to another node. Before we do that, though, the algorithms we develop will be executed on a single node. Keep in mind that the code you write and exectue on a single node will also be capable of being executed on nodes that are connected to each other.

8.3 Naive approach

In a naive approach, Alice would create a sequence of qubits that hold either the value of $|0\rangle$ or $|1\rangle$ and send those to Bob. Bob would then measure the qubits, thereby getting the original sequence of bits created by Alice. The sequence of qubits created by Alice (which can be done by using random bits) is the secret key, and after Bob measures the qubits, he has the same secret key as Alice. Alice and Bob can then use this secret key as a one-time pad, as explained in the previous section.

Schematically, this is explained in Figure 8.7.

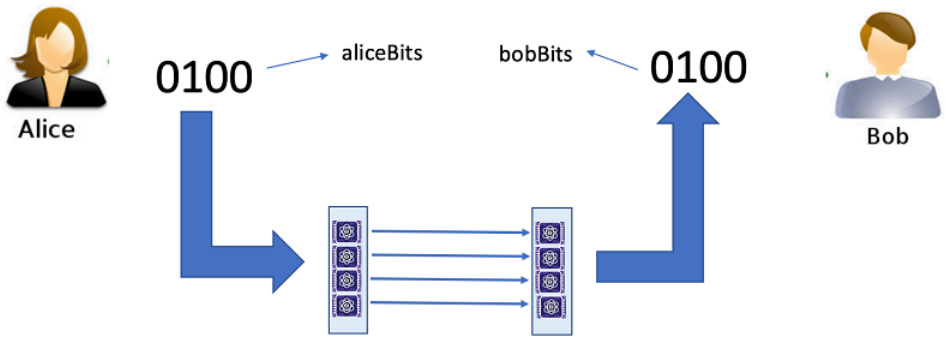

Figure 8.7 Alice generates random bits and uses qubits to sent the values to Bob.

Using the techniques you learned in the previous chapters, you can create an application that does this. The following code snippet is taken from the sample `ch08/native`:

Listing 8.1 Naive approach for generating a quantum key and sending it.

```
final int SIZE = 4;                                                    ❶
Random random = new Random();

boolean[] aliceBits = new boolean[SIZE];
for (int i = 0 ; i < SIZE; i++) {
    aliceBits[i] = random.nextBoolean();                               ❷
}

QuantumExecutionEnvironment simulator =
        new SimpleQuantumExecutionEnvironment();
Program program = new Program(SIZE);                                   ❸
Step step1 = new Step();
Step step2 = new Step();
for (int i = 0; i < SIZE; i++) {
    if (bits[i]) step1.addGate(new X(i));                              ❹
    step2.addGate(new Measurement(i));                                 ❺
}

program.addStep(step1);
program.addStep(step2);

Result result = simulator.runProgram(program);
Qubit[] qubit = result.getQubits();                                    ❻

int[] measurement = new int[SIZE];
boolean[] bobBits = new boolean[SIZE];

for (int i = 0; i < SIZE; i++) {
    measurement[i] = qubit[i].measure();                               ❼
    bobBits[i] = measurement[i] == 1;

    System.err.println("Alice sent "+(bits[i] ? "1" : "0") +
            " and Bob received "+ bobBits[i] ? "1" : "0");
}

Renderer.renderProgram(program);                                       ❽
```

❶ In this sample, you create a key with a fixed size: 4 bits.

❷ Alice generates the key, by assigning random values to each bit.

❸ You create a program that involves 1 qubit for every bit in the key

❹ When a bit is `TRUE`, a Pauli-X gate is applied to the corresponding qubit.

❺ All qubits will be measured in step 2.

❻ The program is executed, and the results are in an array of Qubits.

❼ The qubits are measured, and their value is printed next to the original value of the corresponding bit that used by Alice.

❽ The quantum circuit of this application is rendered.

This program contains only simple quantum operations. Alice first generates the secret key, a series of random classical bits. She then creates qubits based on those bits. Initialy, a qubit is in the value $|0\rangle$. When a qubit has to be created that corresponds to the bit `1`, Alice applies a Pauli-X gate to the qubit. Next, qubits are sent to Bob one by one. Bob performs a measurement, and reads the key bit by bit.

When you execute this program, e.g. by running `mvn javafx:run`,

you will see the following output on the console

```
Alice sent 0 and Bob received 0
Alice sent 1 and Bob received 1
Alice sent 0 and Bob received 0
Alice sent 0 and Bob received 0
```

> **NOTE** the exact output of this program is different each time you run it, since we used random values to initialize the bits used by Alice.

Also, the quantum circuit that is the representing the algorithm you created is shown:

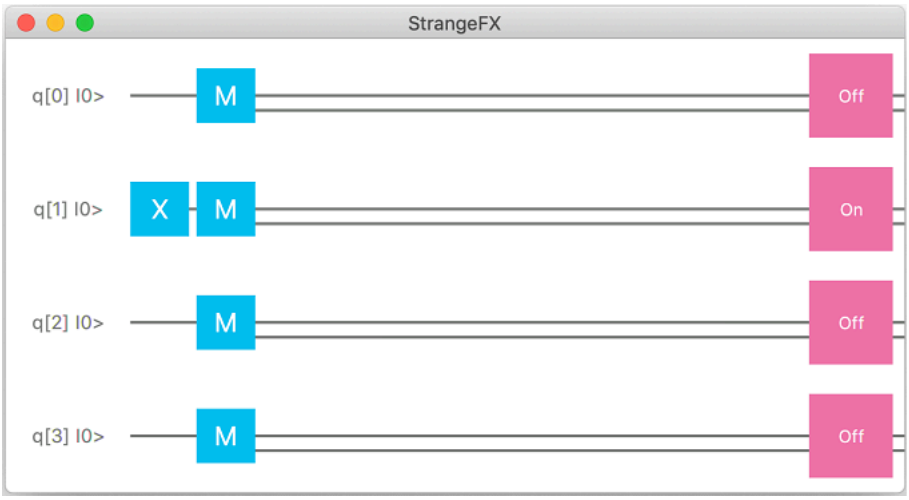

Figure 8.8 Quantum circuit showing the algorithm used by our application.

At the end of the previous section, we explained that for now, we will be running the samples on a single node. That means that both the part of the algorithm executed by Alice and the part executed by Bob are executed on the same node. Keep in mind though that there is an implicit point in the algorithm where we assume that the qubits are sent from Alice to Bob. This is shown in Figure 8.9.

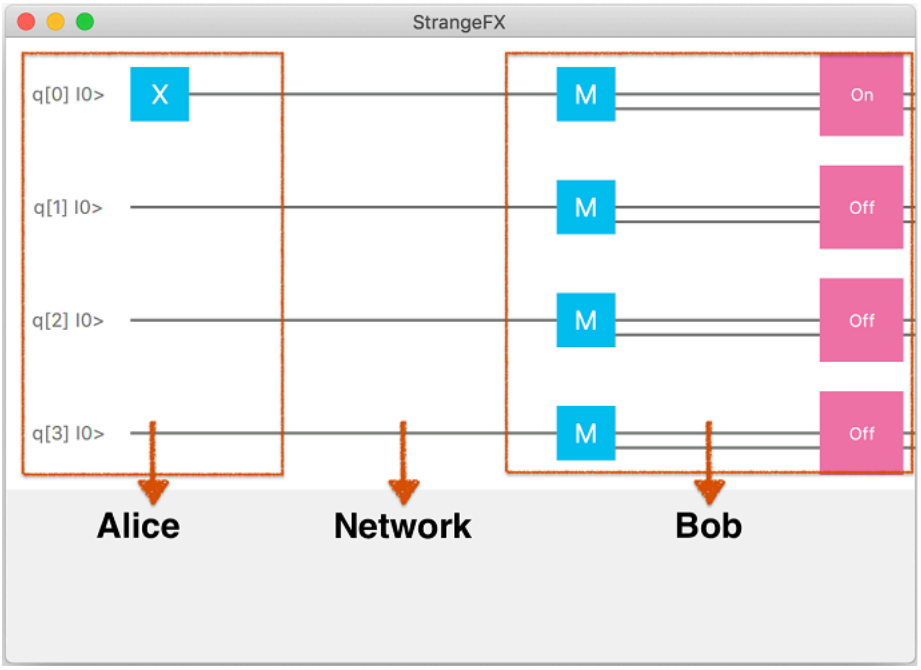

Figure 8.9 The first part of the algorithm is executed by Alice, then the qubits are sent over a quantum network to Bob, where the second part of the algorithm is executed.

The output from the application shows that Alice can create a sequence of random bits, and that Bob can receive the same sequence of random bits. You used qubits to transport the bits over a network cable.

In case the quantum network is reliable and secure, this approach should work. You learned before that qubits can not be cloned, and that once a qubit is measured, it falls back to one of its basic states. This behavior can be very helpful when dealing with quantum networks that should prevent eavesdropping.

However, the current application is far from secure. Suppose Eve is still in the middle, and she is measuring all qubit communication beteen Alice and Bob. We know that when Eve measures a qubit, the qubit will be either hold the value `0` or the value `1`. If it was in a superposition state, the information about that superposition is lost. But in the current algorithm, there are no qubits in a superposition state. Hence, Eve knows that when she is measuring `0`, the original qubit was in the state $|0\rangle$. She can then create a new qubit in the initial $|0\rangle$ state, and put that back on the wire towards Bob. Similar, when Eve measures a qubit and obtains the value `1`, she knows that the qubit was in the $|1\rangle$ state. She can create a new qubit in the $|0\rangle$ state, apply a Pauli-X gate to bring it in the $|1\rangle$ state, and send it to Bob.

This is shown in Figure 8.10

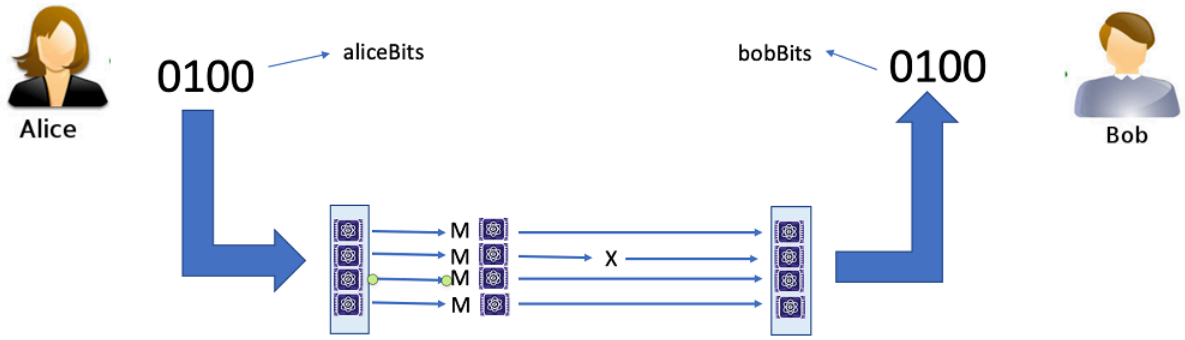

Figure 8.10 Eve is reading the qubits, and create new qubits based on what she measures.

In this figure, you see that the eavesdropping part happens in the network layer. When Eve has access to the network, she can obtain the (not so) secret key that Alice and Bob share. She measures the same values that Alice used to generate the qubits, and those values *are* the secret key. What is especially dangerous is that Alice and Bob are not aware of this. Bob receives qubits, measures them, and constructs the secret key. Bob and Alice successfully exchange a message encrypted with their secret key, but if Alice intercepts this message, she can decrypt it.

8.4 Leveraging superposition

So far, our attempts in using quantum technologies to generate a real secure secret key that is only shared by Alice and Bob were not succesful. But we didn't really leverage the fact that qubits are very different from classical bits. If we are guaranteed that the qubits are either in the $|0\rangle$ state or the $|1\rangle$ state, much of the advantages qubits offer, get lost.

After measuring, Eve can easily reconstruct the original qubit (or at least she can create a new qubit in the same state as the original qubit), if she knows that the original qubit is either $|0\rangle$ or $|1\rangle$. But if the qubit is in a superposition state, she will measure $|0\rangle$ or $|1\rangle$ without getting any information about the original state of the qubit. You will soon extend the initial naive algorithm by leveraging superposition. The qubits sent by Alice will no longer be in the $|0\rangle$ or the $|1\rangle$ state, but in a superposition of these states. We will explain how Bob can retrieve the original state of the qubit, after he received it from Alice.

8.4.1 Applying 2 Hadamard gates

Before we modify the algorithm, we need to explain an interesting fact about the Hadamard gate. It can be proven that when a Hadamard gate operates on a specific qubit, and another Hadamard gate operates on the result of this first operation, the resulting qubit will be in the same state it had originally.

Let's write some code to check if this is true. The code from `ch8/haha` does this, and the relevant snippet is shown below:

Listing 8.2 Applying two Hadamard gates in a row

```
QuantumExecutionEnvironment simulator =
    new SimpleQuantumExecutionEnvironment();

Program program = new Program(2);                    ❶
Step step0 = new Step();
step0.addGate(new X(0));                             ❷

Step step1 = new Step();                             ❸
step1.addGate(new Hadamard(0));
step1.addGate(new Hadamard(1));

Step step2 = new Step();                             ❹
step2.addGate(new Hadamard(0));
step2.addGate(new Hadamard(1));

program.addStep(step0);                              ❺
program.addStep(step1);
program.addStep(step2);

Result result = simulator.runProgram(program);       ❻
Qubit[] qubit = result.getQubits();                  ❼

Renderer.renderProgram(program);                     ❽
```

❶ We create a program with 2 qubits

❷ We flip the first qubit to be $|1\rangle$ while we keep the second qubit at $|0\rangle$

❸ We apply a Hadamard gate to both qubits

❹ We apply another Hadamard gate to both qubits

❺ All steps are added to the program

❻ The program is executed

❼ We measure the qubits

❽ The results are rendered graphically.

The result of this application is shown in Figure 8.11

Figure 8.11 Result of applying two Hadamard gates in a row

This figure shows that if the original qubit was in the state $|0\rangle$, we are guaranteed that the qubit

will be in the state |0⟩ again after applying two Hadamard gates. Similarly, if the original qubit was in the state |1⟩, it will without doubt be in the state |1⟩ again after the two Hadamard gates have been applied.

NOTE While we only proved that this holds for qubits that are initially in |0⟩ or |1⟩, it can mathematically be proven that the same applies to a qubit in any state.

What we learn from this code is the following: if Alice applies a Hadamard gate before she sends her qubit to Bob, and Bob applies another Hadamard gate before he measures the qubit, the qubit is back in the state that Alice prepared (so either in |0⟩ or |1⟩)

8.4.2 Sending qubits in superposition

We will now modify our original algorithm in order to leverage this superposition benefit. Alice will still create a key with qubits that are based on random bits, but before she sends a qubit to Bob (in the |0⟩ or |1⟩ state), she applies a Hadamard gate. When Bob receives the qubit, he will also first apply a Hadamard gate, which should bring the qubit back in the original state created by Alice.

SIDEBAR Before we show this schematically, we introduce a short notation for a qubit that is transformed from a base state into a superposition. In Chapter 4, you learned that applying a Hadamard gate to a qubit in the |0⟩ state brings the qubit in a new state:

$$\frac{1}{\sqrt{2}}(|0> + |1>)$$

Since this state is often encountered in algorithms, it can also be denoted by the shortcut |+⟩.

Similarly, applying a Hadamard gate to a qubit in the |1⟩ state brings that qubit into the following state:

$$\frac{1}{\sqrt{2}}(|0> - |1>)$$

The short notation for this state is |−⟩

We will use these notations throughout the text and figures in this book.

Schematically, the situation where Alice and Bob both apply a Hadamard gate is shown in Figure 8.12

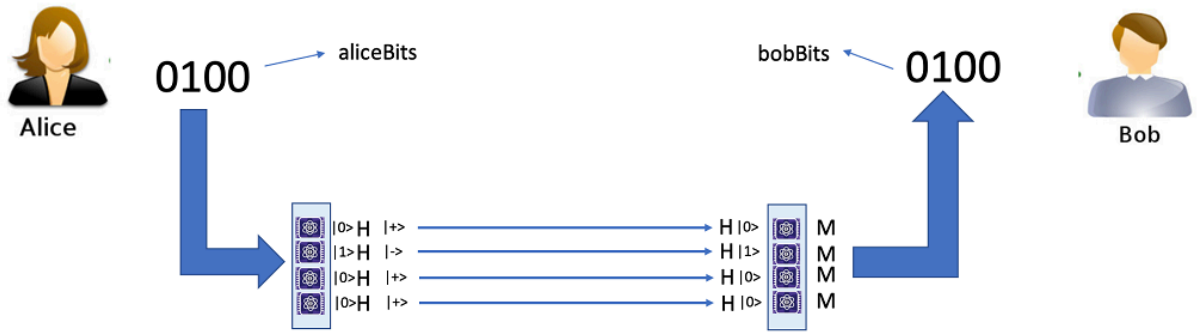

Figure 8.12 Alice applies a Hadamard gate before sending a qubit, Bob applies a Hadarmard gate before measuring the qubit.

The new code can be found in the sample `ch08/superposition` and the relevant part is shown below in Listing 8.3.

Listing 8.3 Using superposition to prevent easy reading of secret key

```
final int SIZE = 4;
Random random = new Random();

boolean[] aliceBits = new boolean[SIZE];
for (int i = 0 ; i < SIZE; i++) {
    aliceBits[i] = random.nextBoolean();                          ❶
}

QuantumExecutionEnvironment simulator = new SimpleQuantumExecutionEnvironment();
Program program = new Program(SIZE);
Step prepareStep = new Step();
Step superPositionStep = new Step();
Step superPositionStep2 = new Step();
Step measureStep = new Step();
for (int i = 0; i < SIZE; i++) {
    if (aliceBits[i]) prepareStep.addGate(new X(i));              ❷
    superPositionStep.addGate(new Hadamard(i));                   ❸
    superPositionStep2.addGate(new Hadamard(i));                  ❹
    measureStep.addGate(new Measurement(i));                      ❺
}

program.addStep(prepareStep);                                     ❻
program.addStep(superPositionStep);
program.addStep(superPositionStep2);
program.addStep(measureStep);

Result result = simulator.runProgram(program);                    ❼
Qubit[] qubit = result.getQubits();


int[] measurement = new int[SIZE];
boolean[] bobBits = new boolean[SIZE];
for (int i = 0; i < SIZE; i++) {
    measurement[i] = qubit[i].measure();                          ❽
    bobBits[i] = measurement[i] == 1;
    System.err.println("Alice sent "+(aliceBits[i] ? "1" : "0") + " and Bob received
                       "+(bobBits[i] ? "1" : "0"));
}
```

❶ Alice creates a key containing random bits.

❷ She initializes her qubits according to these random bits. A random bit of `0` will lead to a $|0\rangle$ qubit, while a random bit of `1` will lead to a $|1\rangle$ qubit

❸ Alice performs a Hadamard transformation to bring the qubit in a superposition and sends it over the network

❹ Bob receives the qubit and performs a second Hadamard transformation

❺ Bob measures the qubit

❻ The steps are added to the quantum program

❼ The program is executed

❽ Bobs bit are measured, and both the bits from Alice and Bob are printed. They should be bitwise equal.

When you execute this application, e.g. using `mvn javafx:run` you will see the following output (again, not the actual values are likely to be different since we generate the bits based on random values):

```
Alice sent 0 and Bob received 0
Alice sent 1 and Bob received 1
Alice sent 0 and Bob received 0
Alice sent 0 and Bob received 0
```

The application also shows the circuit that is created, and this is shown in Figure 8.13

Figure 8.13 Leveraging superposition to send qubits over a network

As expected, the qubits measured by Bob yield the same value that Alice used to prepare the qubits, before the double Hadamard gate was applied. Hence, from a functional point, this algorithm still provides Alice and Bob with the same key. But is it secure?

In case Eve is still able to listen on the network line, she can still measure the qubits sent by Alice. However, regardless of the original bit used to create the qubit by Alice is `0` or `1`, Eve will always have 50% chance to measure `0` and 50% chance to measure `1`. Hence, she won't be able to reconstruct the incoming qubit and send it to Bob --- at least not in the way she did before.

This is shown in Figure 8.14

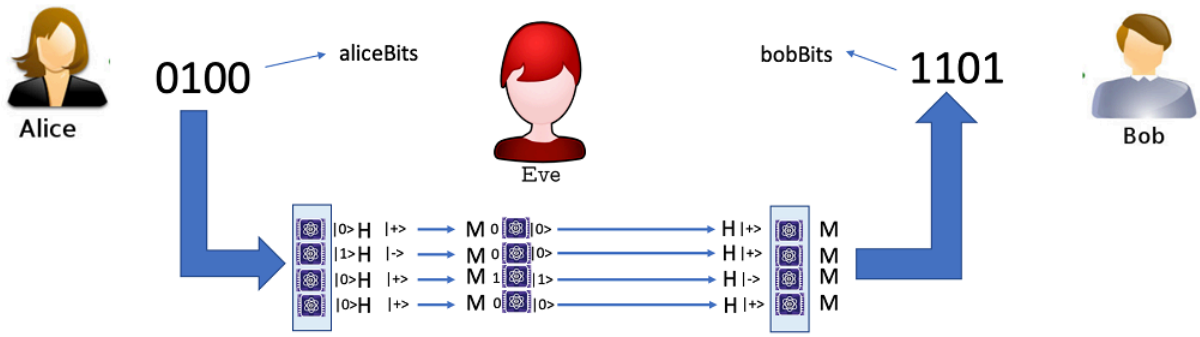

Figure 8.14 Eve measures the qubits sent by Alice, and send new qubits to Bob

As you can see from this Figure, things will go terribly wrong for Eve. When she measures the qubits from Alice, she will randomly obtain a value of `0` or a value of `1`. The qubits sent by Alice all are either in the `|+ >` state or in the $|-\rangle$ state. Both these states, when measured, will have 50% of having the value `0` and 50% change of having the value `1`. The real information is somehow *hidden* in the superposition composition. Eve is not aware of this, and the values she read might be correct, but they also might be wrong. For example, the first qubit in the picture, which originally was $|0\rangle$ is measured as $|0\rangle$ by Eve, so she is correct there. However, the second value, which originally was $|1\rangle$ is measured as $|0\rangle$ by Eve. Hence, Eve will not obtain the correct shared key using this approach. To make things worse, when Eve tries to hide her traces, she creates a new qubit based on her measurement, and send that to Bob. In the case of the first qubit, where she was lucky enough to measure `0`, she constructs a new qubit $|0\rangle$ and sends that to Bob. But Bob, not realising what happend, assumes Alice sent him a qubit in a superposition, and will apply a Hadamard gate. This now brings the qubit, sent by Eve, in a superposition. When Bob measurs this qubit, he can measure either $|0\rangle$ or $|1\rangle$. In the figure, Bob measured a `1`, which clearly is not what Alice has sent. In typical encryption algorithms, Alice and Bob use part of the transmitted bits to check if everything went correct. They share the value of those bits (which makes those particular bits useless as they are not secure anymore). If the value of the bits is different, they know something went wrong, and the whole key is not considered secure.

As a consequnce, it is clear that using this approach Eve can not obtain the secret without errors, or without being detected.

But Eve can learn as well. If Eve knows that Alice applied a Hadamard gate before sending the qubit over the wire, she might apply a Hadamard gate as well before measuring --- doing exactly what Bob is doing. This will give her the information that would otherwise be obtained by Bob:

the same bits that were used by Alice to prepare the qubits.

This would still not help Eve, since Bob won't receive qubits. By measuring them, Eve destroyed the superposition. However, now that Eve knows what Alice was doing, she can create new qubits doing exactly what Alice was doing. Hence, Bob would receive a qubit in the same state as it would have come from Alice. He applies a Hadamard gate, and then measures the qubit, and he gets the same bits that were used by Alice.

This is schematically shown in Figure 8.15

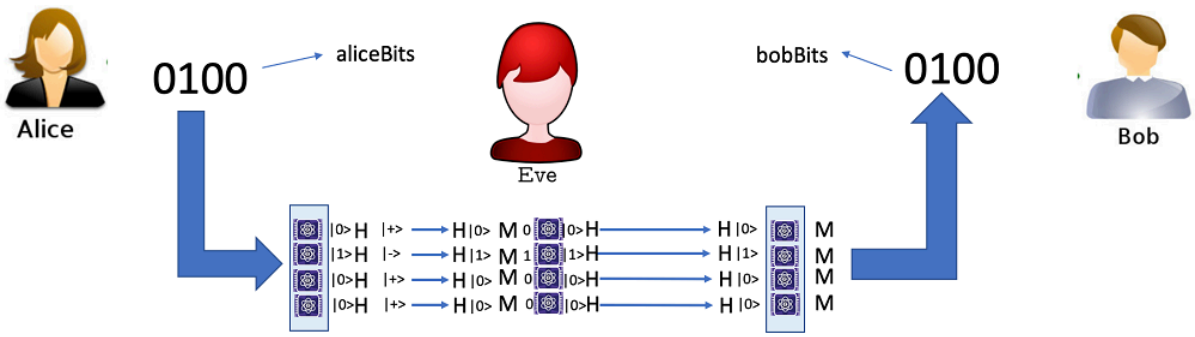

Figure 8.15 Eve applies a Hadamard gate before she measures the qubits sent by Alice, and send new qubis to Bob after she applied another Hadamard gate.

As a consequence, using this approach not only Alice and Bob share the bits of their secret key, but also Eve. Hence, this approach is not secure either.

8.5 BB84

The previous approach failed, because Eve knew upfront what Alice did and what Bob was going to do. In this section, we'll make it harder for Eve --- or actually impossible.

8.5.1 Confusing Eve

The reason that Eve can go undetected is because she manages to send a qubit in the same state to Bob as the one she intercepted from Alice. In case Alice only uses a Pauli-X gate or nothing at all before transfering her qubit to Bob, Eve can measure the qubit and she will obtain the original information. In case Alice also applied a Hadamard gate, Eve needs to apply a Hadamard gate as well before measuring the qubit.

But what if Eve doesn't know if Alice used a Hadamard gate or not? Should she apply a Hadamard gate herself, or not? Let's analyse that situation. We have 3 variables that can each take 2 options, leading to 8 scenarios.

- Alice sends a `0` or a `1`
- Alice applies a Hadamard or not
- Eve applies a Hadamard or not.

The example in `ch8/guess` simulates the possible outcomes for the 8 different scenarios.

The relevant part of this algorithm is shown in Listing 8.4

Listing 8.4 Using superposition to prevent easy reading of secret key

```
final int SIZE = 8;                                                    (1)

...

for (int i = 0; i < SIZE; i++) {
    if (i > (SIZE/2-1)) prepareStep.addGate(new X(i));                 (2)
    if ( (i/2) % 2 == 1) superPositionStep.addGate(new Hadamard(i));   (3)
    if (i%2 ==1 )superPositionStep2.addGate(new Hadamard(i));          (4)
    measureStep.addGate(new Measurement(i));
}
```

(1) We consider the 8 possible cases (numbered from 0 to 7)

(2) In the first 4 cases, Alice applies a Pauli-X gate

(3) In cases 2,3,6 and 7, Alice applies a Hadamard gate

(4) In cases 1, 3, 5 and 7, Eve applies a Hadamard gate

The code inside the `for` loop creates the 8 different scenarios. The visual output of the application, shown in Figure 8.16 clarifies this.

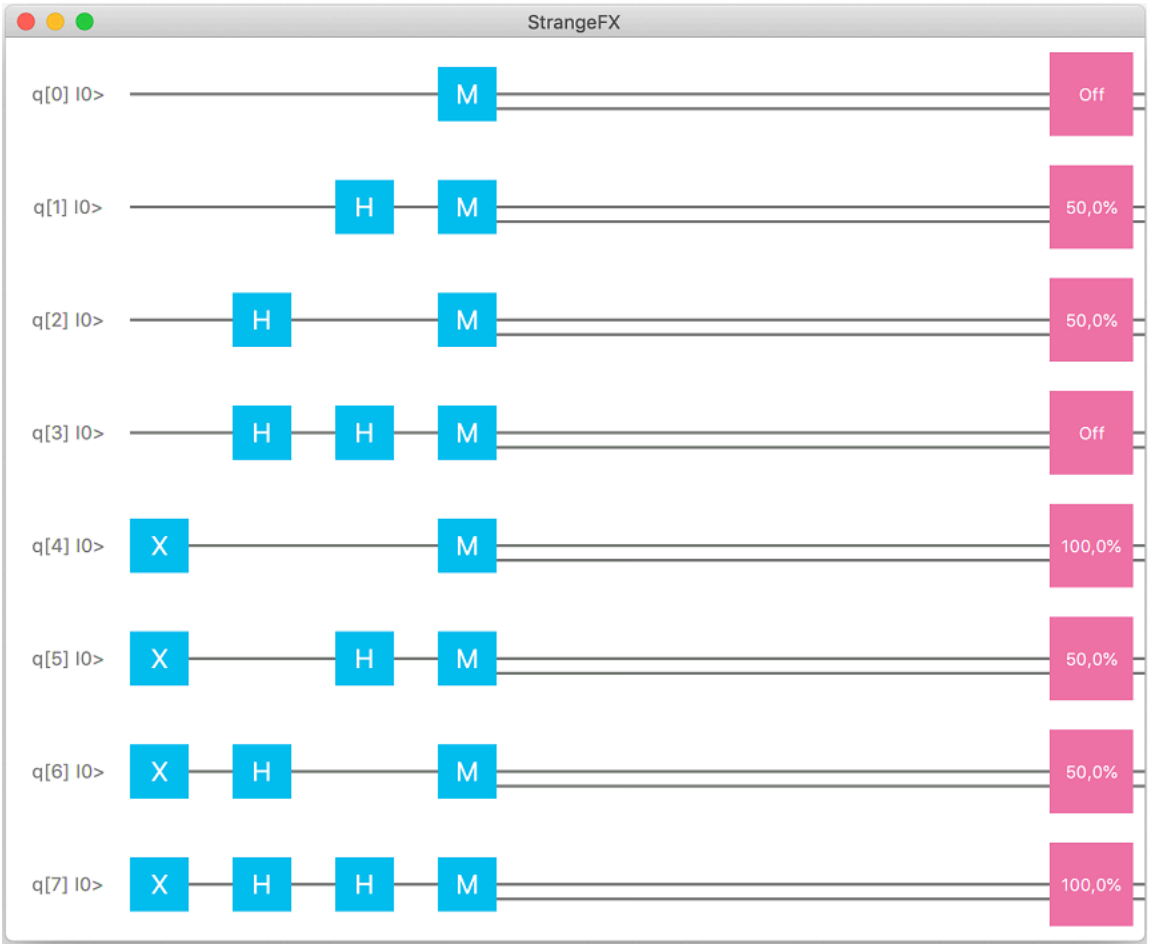

Figure 8.16 Different scenarios and their outcome

No Pauli-X gate was applied to the first 4 qubits, hence they are representing a bit value of `0`. Let's look at those 4 scenario's in a bit more detail. The analysis we will do here also applies to the last 4 qubits, with the difference that the initial value in that case is `1`.

Figure 8.17 shows the situation of the first 4 qubits.

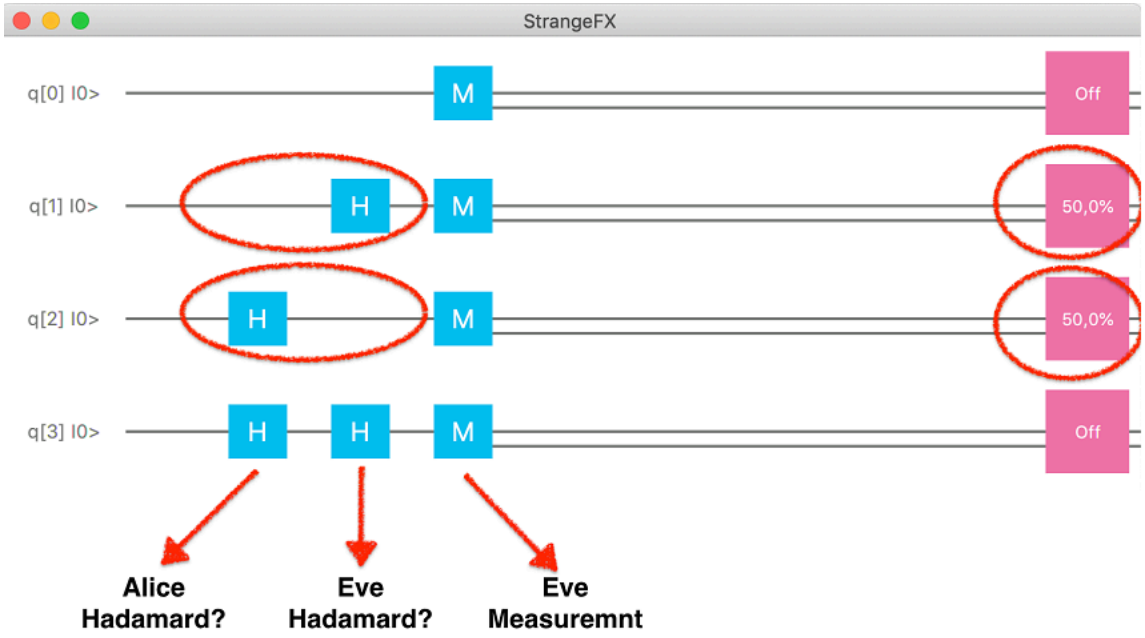

Figure 8.17 Alice is sending a 0 and the measurement depends on the presence of Hadamard gates.

In case both Alice and Eve apply a Hadamard operation or not, Eve will without doubt measure a `0` value as well. But if either of them applies a Hadamard operation while the other doesn't, there is a 50% chance that Eve will measure a `0` and a 50% chance tha Eve will measure a `1`. These cases are marked in red on Figure 8.17. The problem for Eve is that she can't tell if her measurement is correct or not. She doesn't know if Alice applied a Hadamard gate or not, so she can not tell with certainty which scenario applies. To make things worse for Eve, she is also unable to create a qubit in the same state as the original one.

For example, suppose Eve measures the qubit as a `0`. From Figure 8.16, it appears that there are 6 potential scenarios' that would lead to a measurement of `0`. The scenarios with `q[0]` and `q[3]` will absolutely lead to a measurement of `0`, but each of the scenarios `q[1]`, `q[2]`, `q[5]` and `q[6]` have 50% chance of resulting in a measurement of `0` as well. in the case of scenarios `q[5]` and `q[6]`, the original bit was `1` and in the other scenarios, the original bit was `0`. Since Eve knows if she applied a Hadamard gate or not, she can exclude half of the scenarios, but there is always the possiblity that the original bit was `0` and the possibility that the original bit was `1`. Since Eve doesn't know the original scenario, she can make a guess, and prepare a qubit that fits a valid scenario, but chances are this is the wrong scenario, and Bob will receive a qubit in a different state then the one sent by Alice. We will shortly show that this can be detected.

8.5.2 Bob is confused too

If Eve can't reconstruct the original scenario, the same must apply to Bob. Indeed, we assume that Alice and Bob have no upfront knowledge --- otherwise we would have fixed the bootstrap problem already.

In our algorithm, we will instruct Bob to randomly apply a Hadamard gate or not before measuring the incoming qubit. The situation for Bob is then very similar to the situation with Eve. In case Alice and Bob both applied a Hadamard gate, or in case none of them applied a Hadamard gate, the measurement of Bob would guaranteed correspond to the initial value of Alice. We can see this from Figure 8.16 and Figure 8.17 if we assume Bob is performing the second part instead of Eve. But if Alice applied a Hadamard transformation and Bob didn't, or if Alice didn't apply a Hadamard transformation but Bob did, the result can be wrong.

It seems that by making the situation complex for Eve, we made it equally complex for Bob. Both Alice and Bob randomly decided whether they apply a Hadamard gate or not. From the output of the previous application, it is clear though that if Alice and Bob both decided to apply a Hadamard gate, or if they both decided not to use a Hadamard gate, they can share a key. In that case, the initial value used by Alice is guaranteed to be the same value measured by Bob.

8.5.3 Alice and Bob are talking

From the previous discussion, it is clear that if Alice and Bob both used a Hadamard gate, or neither of them used a Hadamard gate, the original bit used by Alice and the measured bit used by Bob are guaranteed to be the same, and it can be used in a secret key.

But how do they know that? The answer is simple: they tell each other whether they applied a Hadamard gate or not.

IMPORTANT This might sound surprising. If Alice and Bob tell each other over a public channel whether they appled a Hadamard gate or not, Eve may be listening! The trick is this: Alice and Bob only share that information with each other **after** Bob has received and measured its qubit. At that moment, Eve can't do anything anymore. If Eve would have known the information upfront, she would be able to manipulate the system, since she could easily reproduce the qubit from Alice if she knew whether Alice applied a Hadamard gate or not. But she had to make a decision before sending a qubit to Bob. And that decision was made, based on a measurement of the qubit she intercepted from Alice. Hence, all information in that qubit is destroyed. Pity for Eve, but the public information is useless.

When Alice and Bob have each others information about the Hadamard gates, they simply remove the values measured on qubits that had non-matching Hadamard gates. The remaining values are guaranteed to be correct.

IMPORTANT Alice and Bob only share the information about the Hadamard gates, they do not share the initial value (in Alice's) case, or the measured value (in Bob's) case. They know though that those values are equal, and they can use those as part of a shared secret key.

Typically, Alice and Bob use a part of their secret key to check if the connection was eavesdropped or not. In case Eve isn't discouraged by the fact that she can't get the original key without getting noticed, she might still try to do so. But she will have to make wild guesses about whether to apply a Hadamard gate or not. In case she makes the wrong guess, she will send a qubit to Bob that is in a different state from the one Alice would have sent to Bob. Hence, there is a chance that Bob will measure a different value than Alice has been using. If both or none of Alice and Bob applied a Hadamard gate, but if the initial value from Alice is different from the measured value by Bob, Alice and Bob know that the connection was tampered with.

8.6 QKD in Java

The combined knowledge obtained in the previous sections allow you to create a Quantum Key Distribution application in Java. The example in `ch08/bb84` does exactly this.

8.6.1 The code

We won't paste the whole sample code, but will highlight a few important snippets.

INITIALIZE SOME VARIABLES

First of all, we initialize a number of arrays:

```
final int SIZE = 8;                          ❶
boolean[] aliceBits = new boolean[SIZE];     ❷
boolean[] bobBits = new boolean[SIZE];       ❸
boolean[] aliceBase = new boolean[SIZE];     ❹
boolean[] bobBase = new boolean[SIZE];       ❺
```

❶ We are going to create a key with maximum 8 bits. Keep in mind that we will have to remove the bits for which Alice and Bob used a different strategy (Hadamard or not), so on average the real length of the key is half of the size specified here.

❷ In this array, Alice keeps the random bits she generates, and those serve for the base she uses.

❸ In this array, Bob stores the bits he measured.

❹ When Alice decides to apply a Hadamard gate for a specific qubit, the corresponding value in this array will be set to true.

❺ When Bob decides to apply a Hadamard gate for a specific qubit, the corresponding value in this array will be set to true.

PREPARE THE DIFFERENT STEPS

The quantum application we create contains different steps. The first sets of these steps are performed by Alice, and the second steps are performed by Bob.

```
Step prepareStep = new Step();               ❶
Step superPositionStep = new Step();         ❷
Step superPositionStep2 = new Step();        ❸
Step measureStep = new Step();               ❹
```

❶ In this step, Alice will apply a Pauli-X gate in case the random bit under consideration is `1`.

❷ In this step, Alice will apply a Hadamard gate (or not)

❸ In this step, Bob will apply a Hadamard gate (or not)

❹ In this step, Bob will measure the result

FILL THE DIFFERENT STEPS

For each bit that can be part of the key, all steps will be created. Three of those steps depend on random values.

Based on a first random value, a Pauli-X gate will be applied to the `prepareStep`. There is 50% chance that the Pauli-X gate will be applied, causing the qubit to be in the $|1\rangle$ state and 50% chance that no gate will applied, and the qubit will stay in $|0\rangle$.

The second random value defines whether or not a Hadamard gate is applied to the `superPositionStep`, which is executed by Alice.

The next step, `superPositionStep2` uses a random value to decide wether or not Bob applies a Hadamard gate.

```
for (int i = 0; i < SIZE; i++) {                          ❶
    aliceBits[i] = random.nextBoolean();                  ❷
    if (aliceBits[i]) prepareStep.addGate(new X(i));      ❸
    aliceBase[i] = random.nextBoolean();                  ❹
    if (aliceBase[i]) superPositionStep.addGate(new Hadamard(i));

    bobBase[i] = random.nextBoolean();                    ❺
    if (bobBase[i]) superPositionStep2.addGate(new Hadamard(i));

    // Finally, Bob measures the result
    measureStep.addGate(new Measurement(i));              ❻
}
```

❶ The following steps will be applied for each bit that is a candidate for the secret key

❷ A random value determines whether Alice's bit will be `0` or `1`.

❸ If Alice's bit is 1, apply a X gate to the |0 state

❹ A random value (that will be stored in the aliceBase array) decides whether or not Alice will apply a Hadamard gate

❺ A random value (that will be stored in the bobBare array) decides whether or not Bob will apply a Hadamard gate.

❻ Finally, Bob measures the qubit.

EXECUTE THE APPLICATION

We now have to execute the application in our quantum simulator. This is done using the techniques you learned in previous chapters.

```
        QuantumExecutionEnvironment simulator =
                new SimpleQuantumExecutionEnvironment();          ❶
        program.addStep(prepareStep);                             ❷
        program.addStep(superPositionStep);
        program.addStep(superPositionStep2);
        program.addStep(measureStep);

        Result result = simulator.runProgram(program);            ❸
        Qubit[] qubit = result.getQubits();                       ❹
```

❶ Create a QuantumExecutionEnvironment

❷ Add the steps created in the previous phases

❸ Run the quantum program on the simulator

❹ Assign the results to an array of qubits

PROCESS THE RESULTS

Now that the program is executed, and the results are in, we can process those results. In this phase, we will decide whether a specific bit should be part of the key, both for Alice and Bob.

```
int[] measurement = new int[SIZE];
for (int i = 0; i < SIZE; i++) {                                  ❶
    measurement[i] = qubit[i].measure();
    bobBits[i] = measurement[i] == 1;                             ❷
    if (aliceBase[i] != bobBase[i]) {                             ❸
        System.err.println("Different bases used, ignore values "+aliceBits[i]+" and
                           "+ bobBits[i]);
    } else {                                                      ❹
        System.err.println("Same bases used. Alice sent " + (aliceBits[i] ? "1" :
                           "0") + " and Bob received " + (bobBits[i] ? "1" : "0"));
        key.append(aliceBits[i] ? "1" : "0");                     ❺
    }
}
```

❶ For each candidate-bit, we run the following steps that evaluate whether the bit should be part of the key or not.

❷ Set the bit in the bobBits array to the measurement value of the qubit.

❸ If the random bases chosen by Alice and Bob for this bit are different, ignore values and print a message.

❹ Otherwise, Alice and Bob used the same Hadamard strategy. The inital value from Alice matches the measurement from Bob.

❺ This bit now becomes part of the secret key

8.6.2 Running the application

The results shown when running this application vary each time you run the application, which you can do from the `ch8/bb84` directory with the command

```
mvn clean javafx:run
```

The following output is just one example of the many possibilities:

```
Same bases used. Alice sent 1 and Bob received 1
Same bases used. Alice sent 0 and Bob received 0
Same bases used. Alice sent 1 and Bob received 1
Different bases used, ignore values false and true
Same bases used. Alice sent 1 and Bob received 1
Different bases used, ignore values false and true
Same bases used. Alice sent 1 and Bob received 1
Different bases used, ignore values true and true
Secret key = 10111
```

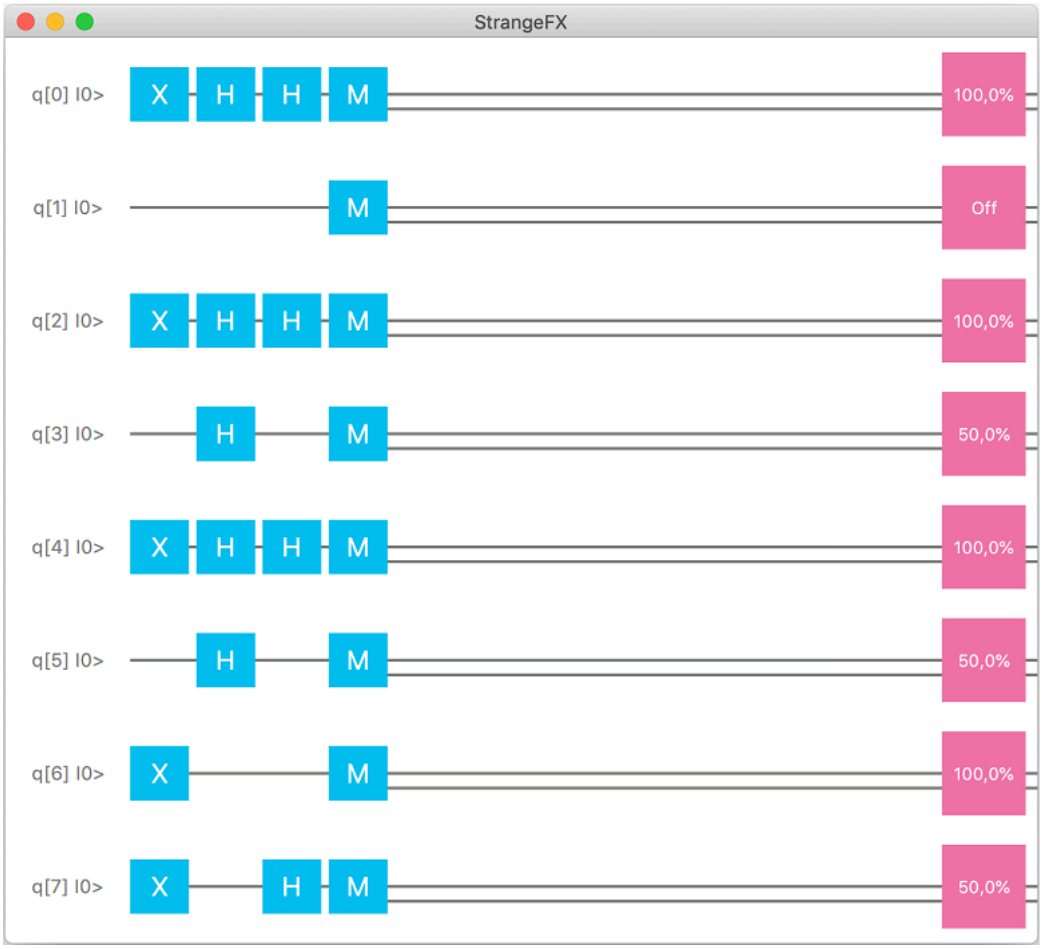

Figure 8.18 Resulting output obtained by running the bb84 application

From both the text output and the graphical output, it is clear that Alice and Bob used the same Hadamard strategy for bits 0, 1, 2, 4 and 6. Those 5 bits are thus part of the secret key. The other bits are useless since Alice and Bob used a different Hadamard strategy (either Alice applied one and Bob didn't, or the other way round).

8.7 Introducing Simulaqron

So far, all our code runs in a single Quantum Execution Environment. Doing so, we could explain the concepts that lead to the BB84 algorithm. In practice, though, secure communication requires two different nodes. We need to be able to send a qubit from one node to another if we want to generate a shared secret key between those two nodes. This requires a distributed version of a Quantum Execution Environment.

An interesting project that provides a way to transfer qubits from one node to another is the SimulaQron project from QuTech (qutech.nl). One of the goals of QuTech is to build a network of quantum computers, using fiber optic cables. From the code we have shown so far, it should be clear that even with a very limited number of qubits available, quantum networking already has huge benefits. A single qubit can be used to generate a shared secret bit between 2 parties. By repeating this process as often as required, a shared secret with as many bits as required can be obtained.

While work on the physical quantum network is being carried out, QuTech also worked on a protocol stack similar to a protocol stack of classical networking. Such a stack makes abstraction of the hardware implementation, and shields developers from the low-level implementations. Developers using the top-layer of such a protocol stack can create applications that can then run on different implementation of the protocols. This is not only useful to have the same code working with different kinds of hardware, it also allows code to be leveraging simulators while the hardware is not yet available.

SimulaQron provides a protocol called CQC that allows high-level programming languages (e.g. Java, Python, C,…) to interact with the implementation, and to leverage quantum networking functionality.

Support for the CQC protocol is being added to the Strange simulator. As a consequence, applications you write using Strange will work on a distributed system. In a first phase, this will be a network containing quantum simulators, but in a later phase --- once there will be real quantum nodes in a network --- this should also work on real hardware.

The current plan is to have a demo with a real quantum network connecting four Dutch cities in 2020.

8.8 Summary

This chapter explained one of the most interesting usecases of quantum computing that does not need a large number of qubits, and that has a potential to be used on a wider scale in the not too distant future.

In this chapter,

- you learned the security issues related to sending bits via a phyiscal connectivity layer
- you learned the basics of secure communication using a one-time pad
- you created a quantum algorithm that generates a one-time pad
- you gradually fixed the security issues in this algorithm

9 Deutsch-Jozsa algorithm

This chapter covers

- different ways to obtain information from classical functions
- the difference between function evaluations and function properties
- quantum gates that correspond to classical black box functions
- the Deutsch algorithm
- the Deutsch-Jozsa algorithm

9.1 When the solution is not the problem

Do you know if the number `168153` can be divided by 3? There are a number of ways to find out. For example, you can simply take a calculator and obtain the result:

```
168153 / 3 = 56051
```

The result of the division is `56051`. But that was not the question. Actually, we don't care about this result. Fortunately, thanks to this evaluation, we also know the real answer. There are no digits after the decimal point, hence we can conclude the number can indeed be divided by 3.

There is another simple approach to find the answer to this question, and you might know this simple trick: take the sum of the individual digits that compose the number, and see if that sum can be divided by 3. If so, the original number can be divided by 3 as well. Let's to that:

```
1 + 6 + 8 + 1 + 5 + 3 = 24
```

Since `24` can indeed be divded by 3, we can conclude that `56051` can be divided by 3 as well.

The first approach (using the calculator) gave us a result of a division, and it provided us with the real answer. The second approach (sum of the individual digits) only provided the real answer,

and not the outcome of the division.

The relevance of this is that in many cases, we are interested in a specific property of something (e.g. a number or a function). We are not interested in a function evaluation, but we somehow want to obtain information about the function. Evaluating the function is often the easiest way to do so, but it can be more efficient to indirectly look at the properties of the function, and draw conclusions from there.

This is of particular interest in quantum computing. A quantum computer with n qubits that needs to examine a specific function, can only evaluate the function one by one. Applying the quantum circuit to a given specific set of input qubits, will result in a modified state of these qubits. Measuring them gives a particular result, and if you want a new result, you need to run the circuit again. Even though we can apply Hadamard gates to the input qubits to bring them in superposition — which allows to evaluate the qubits in the `0` and the `1` state simultaneously — we can not magically create new qubits that will hold the information of the different cases. This is shown in Figure 9.1 for a system with 2 qubits.

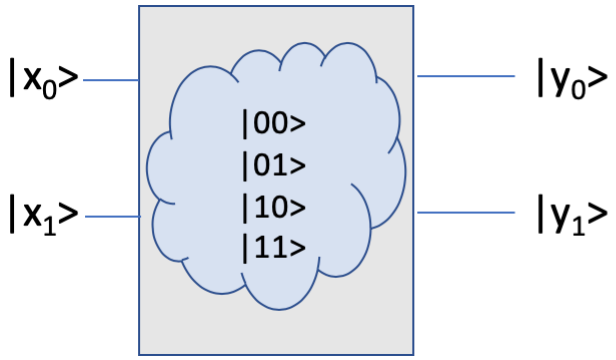

Figure 9.1 A quantum system with 2 qubits can do many evaluations, but only 2 qubits can be measured.

The internal computations can contain the equivalent of many evaluations, but we can not obtain those simultaneously.

We are limited to a result of n qubits. But that is often enough to solve problems. In the case of our number being a multiplicator of 3 or not, a single qubit is enough for storing the answer. We don't need to evaluate the division function.

In this chapter, we will show how this approach is done using quantum computing. We will investigate a property of a function f acting on n bits, without being interested in the individual function evaluations. We will show that retrieving the property in the classical way requires $2^{n-1} + 1$ function evaluations. With the quantum algorithm, the property can be obtained with a single evaluation.

The functions we will use are very simple, and there is no direct use case of this problem. But it demonstrates a very important aspect of quantum computing, and it explains why quantum

computing is often associated with "exponential" complexity. You can easily see that the more input bits the function has, the harder it becomes to solve the problem in a classical way. The number *n* is indeed in the exponent, and as we showed in chapter 1, exponential functions quickly result in huge values. If a quantum algorithm can fix the same problem in just a single evaluation (or in general, in less than an exponential number of equations), this is a huge advantage for a quantum computer.

We will gradually come to the algorithm that achieves this. We will follow the approach shown in Figure 9.2.

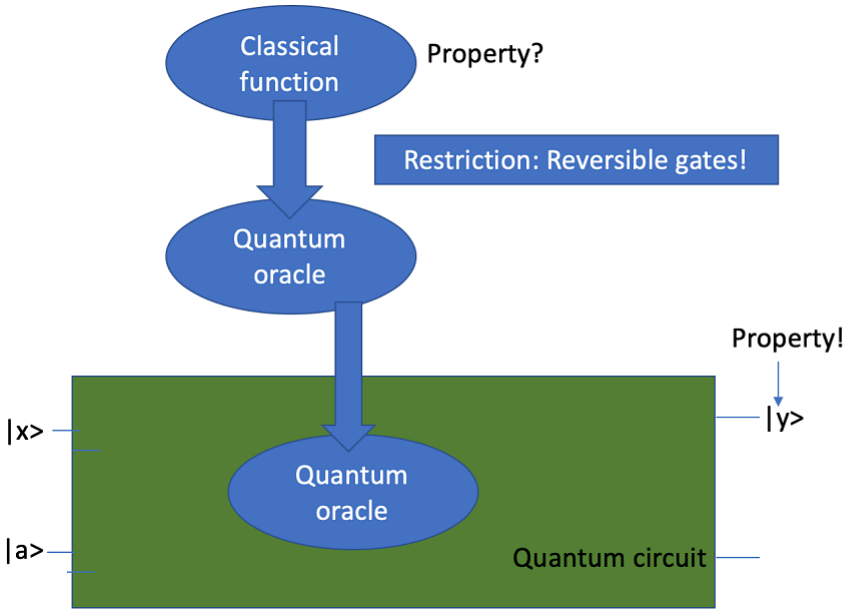

Figure 9.2 Finding the property of a function, approach

First, we'll talk about properties of functions, and how to obtain them in a classical way. Next, we convert the functions into quantum blocks called *oracles*. We will show that there are some requirements in doing this. Once you can create a quantum oracle that represents a classical function, this oracle can be used in a quantum circuit. Evaluating this quantum circuit once results in the property of the function we are looking for.

9.2 Properties of functions

In most typical cases where functions are involved, you are interested in finding the result of a function. For example, consider the function

$y(x) = x^2$

If we want to know the value of this function for e.g. $x = 4$ and $x = 7$, we need to evaluate the function:

$y(4) = 4^2 = 16$

$y(7) = 7^2 = 49$

In some cases though, the function evaluations are not important, but the characteristics of the functions are. In this area, quantum algorithms can be helpful. An example that we will discuss in a later chapter, is the periodicity of a function. We are not interested in the individual evaluations of a function, but we are interested in the periodicity. A periodic function is a function where the same pattern of values comes back with a fixed periodicty,

Table 9.1 Table example of a periodic function

x	0	1	2	3	4	5	6
y	7	9	5	7	9	5	7

From the table above, you can tell that the function has a periodicity of `3`: for every value of *x*, the result of the function is the same as the function applied to *x + 3*.

This is an example where the property of a function can be more interesting than the function evaluations themselves.

9.2.1 Constant and Balanced functions

In general, the functions we consider are noted as *f(x)* where *f* is the function that operates on an input variable called *x*. The evaluation of a function for a specific input is also called the *result* and sometimes noted as *y*, where *y = f(x)*.

In this chapter, we start with a very simple family of functions, that have simple properties. We will start with a function that has only a single input bit, and we will extend that later to a function with *n* input bits. In all cases, the result of the function is always either `0` or `1`.

The functions we will discuss here have a special property: they are either balanced functions or constant functions.

A function is called **constant** in case the result is not dependent on the input. In our case, that means that the result is either `0` for all input cases, or `1` for all input cases.

A function is called **balanced** in case the result is `0` in 50% of the cases, and `1` in the other cases.

The Deutsch algorithm, which we will discuss shortly, deals with a function, called f, that takes a single bit (a boolean value) as its input, and it produces a single bit as well. The function thus only operates on either '0' and '1' and it result is either '0' or '1',

The combination of 2 input options and 2 output options leads to 4 possible cases for this function, which we will name *f1*, *f2*, *f3* and *f4*:

f1:

f(0) = 0 and *f(1) = 0*

f2:

f(0) = 0 and *f(1) = 1*

f3:

f(0) = 1 and *f(1) = 0*

f4:

f(1) = 1 and *f(1) = 1*

From those defintions, it appears that *f1* and *f4* are constant functions, and *f2* and *f3* are balanced functions.

In many classical algorithms, it is important to know the output of a function for specific values. In many quantum algorithms, on the other hand, it is useful to know the properties of the function under consideration.

This is part of the "different thinking" that is required when thinking about quantum algorithms. Thanks to superposition, a quantum computer can evaluate many possibilities simultaneously, but since obtaining a result requires a measurement, the superposition is gone and we are back to a single value. Hence, the added value is in the function evaluation, and not in the result of those function evaluation.

In the Deutsch algorithm, a function is provided, but we don't know what function it is. We know it is *f1*, *f2*, *f3* or *f4* but that's all we know. We are now asked to find out if this function is constant or balanced. Our task is not to find out if the provided function is *f1*, *f2*, *f3* or *f4* . Hence, we are asked about a property of the function, not about the function itself.

How much function evaluations do we need before we can answer this question with 100% certainty? If we only make a single evaluation (we only calculate either *f(0)* or *f(1)*), we don't have enough information yet.

Suppose we measure *f(0)* and the result is `1`. From the table above, it seems that in this case, our function is either *f3* (which is balanced) or *f4* (which is constant). Hence, we don't have enough information. In case the result of measuring *f(0)* is `0`, the table above shows that the function is either *f1* (which is constant) or *f4* (which is balanced). Again, this proves that measuring *f(0)* is not enough to conclude whether the function is constant or balanced. It can be either.

EXERCISE 9.1

you can prove that measuring *f(1)* is not sufficient either.

It turns out we need 2 classical function evaluations before we can determine wether the provided function is constant or balanced.

Let's write a Java application that demonstrates this.

Listing 9.1 Two evaluations are needed to declare a function constant or balanced

```
static final List<Function<Integer, Integer>> functions = new ArrayList<>();

static {
    Function<Integer, Integer> f1 = (Integer t) -> 0;                  ❶
    Function<Integer, Integer> f2 = (Integer t) -> (t == 0) ? 0 : 1;
    Function<Integer, Integer> f3 = (Integer t) -> (t == 0) ? 1 : 0;
    Function<Integer, Integer> f4 = (Integer t) -> 1;
    functions.addAll(Arrays.asList(f1, f2, f3, f4));
}

public static void main(String[] args) {
    Random random = new Random();
    for (int i = 0; i < 10; i++) {                                     ❷
        int rnd = random.nextInt(4);
        Function<Integer, Integer> f = functions.get(rnd);             ❸
        int y0 = f.apply(0);                                           ❹
        int y1 = f.apply(1);
        System.err.println("f" + (rnd + 1 + " is a "                   ❺
                + ((y0 == y1) ? "constant" : "balanced")
                + " function"));
    }
}
```

❶ We prepare the 4 possible functions. This step needs to be done only once.

❷ We will do 10 experiments

❸ We pick a random function, not knowing anything about its implementation.

❹ Two function evaluations are performed, one for input `0` and one for input `1`

❺ If the results of those 2 evaluations are similar, the function is constant, otherwise the function is balanced.

In this code snippet, we create the four possible functions in a static block. The reason we do this is because we want to stress that the creation of the function, and the determination whether they are constant or balanced should be considered as two independent processes.

After the functions are created, the application really starts. Inside the `for` loop, a random function will be picked. Based on the two function evaluations, we can determine whether the function is constant or balanced.

A possible output of this application is the following:

```
f4 is a constant function
f4 is a constant function
f3 is a balanced function
f1 is a constant function
f2 is a balanced function
f2 is a balanced function
```

```
f1 is a constant function
f4 is a constant function
f3 is a balanced function
f2 is a balanced function
```

As expected, the application has the correct answers for every loop. But in every loop, we performed 2 evaluations of the function. As we showed above, a single evaluation would not be sufficient to conclude whether the function is balanced or not.

9.3 Reversible quantum gates

So far, we talked about classical functions and their properties. Before we can discuss the quantum equivalent of those functions, we need to look into a requirement of quantum gates in more detail.

In the previous chapters, you learned and used a number of quantum gates. These gates have many similarities with gates you encounter in classical computing. However, there are some fundamental differences.

Quantum gates are physically achieved using properties of quantum mechanics, and therefore must obey to the requirements and restrictions that are related to quantum mechanics.

One of the key requirements for a quantum gate is that it should be reversible. This means that, when a quantum gate is applied to a given begin status, there should exist another quantum gate that brings the result back to the begin status. In a quantum system, information can not simply disappear. The information that was in a system before a specifc quantum gate is applied, should be recoverable.

All the gates we discussed so far are reversible. Let's show that with a simple example: the Pauli-X gate. The gate that brings a system back in its original state after a Pauli-C gate is applied, is another Pauli-X gate.

We can explain this in two ways:

- experimental evidence
- mathematical proof

9.3.1 Experimental evidence

We will create a simple quantum application that applies a Pauli-X gate on a single qubit, followed by another Pauli-X gate. Instead of only taking the special cases into account where the qubit is either `|0>` or `|1>`, we aritifically initialize the qubit so that it has 75% chance to be measured as `1`.

The circuit is shown in Figure 9.3.

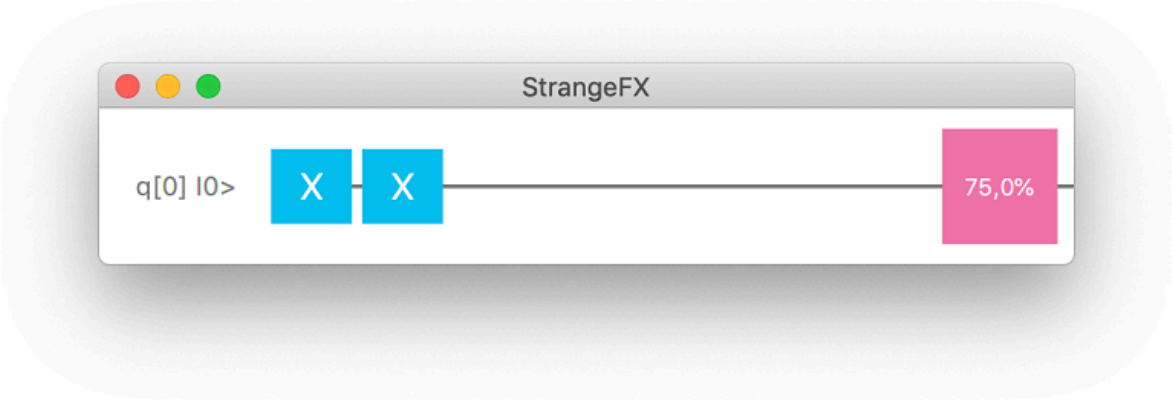

Figure 9.3 Quantum program containing 2 Pauli-X gates

The code for this sample can be found in the sample repository under `ch09/reversibleX` and the relevant snippet is shown in the listing below:

Listing 9.2 Two Pauli-X gates applied to a single qubit

```
QuantumExecutionEnvironment simulator =
        new SimpleQuantumExecutionEnvironment();
Program program = new Program(1);                          ❶
Step step0 = new Step();                                   ❷
step0.addGate(new X(0));

Step step1 = new Step();                                   ❸
step1.addGate(new X(0));
program.addStep(step0);                                    ❹
program.addStep(step1);
program.initializeQubit(0,.5);                             ❺

Result result = simulator.runProgram(program);             ❻
Renderer.showProbabilities(program,1000);                  ❼
Renderer.renderProgram(program);
```

❶ We create a quantum application with a single qubit

❷ In the first step (step0), a Pauli-X gate is applied to the qubit

❸ In the second step (step1), another Pauli-X gate is applied to the qubit

❹ The steps are added to the quantum program

❺ The single qubit is initialized with an alpha value of `0.5`, which leads to a probability of 25% of measuring `0`

❻ The quantum program is executed

❼ The statistical results of running this program 1000 times are rendered.

The result of running this circuit 1000 times is shown in Figure 9.4.

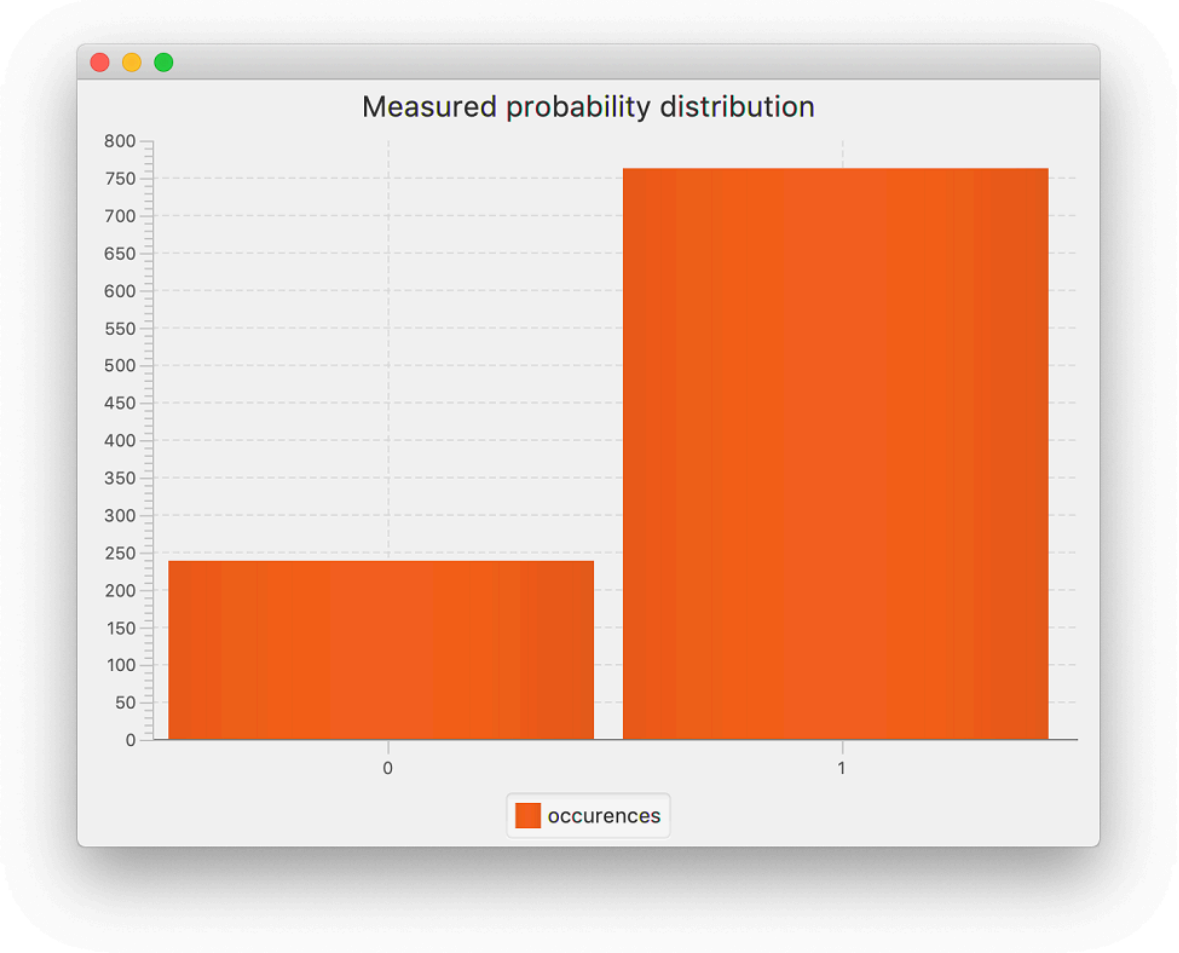

Figure 9.4 Statistical results of a quantum program containing 2 Pauli-X gates

As expected, after applying 2 Pauli-X gates, the probability of measuring `0` is about 25%, and there is about 75% chance that we will measure `1`. This matches the artificial initial value that we applied to the qubit.

9.3.2 Mathematical proof

In Chapter 4, we explained the mathematical equivalent of applying a gate to a qubit : we multiply the gate matrix and the probability vector of the qubit. Let's assume the initial qubit is described as follows:

$$\psi = \alpha|0> + \beta|1>$$

or in vector notation:

$$\psi' = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Applying two Pauli-X gates to this qubit brings it in the following state:

$$\psi' = XX \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

where X is the matrix defining the Pauli-X gate. From Chapter 4, we know the structure of the matrix, so we can write

$$\psi' = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Leveraging the matrix multiplication (which is explained in Appendix B), we can write this as follows

$$\psi' = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Hence, we have proven that the Pauli-X gate is indeed a reversible gate, and that the changes to a quantum system caused by applying a Pauli-X gate can be undone by applying another Pauli-X gate.

Now that you learned that the Pauli-X gate is a reversible quantum gate, you can use the same technique to proof that the other gates you learned so far are reversible as well.

> **NOTE** the gates we introduced so far have the special property that they are their own inverse. This is not always true, and there is no restriction that a quantum gate should be its own inverse.

9.4 Defining an Oracle

In many quantum algorithms, the term "oracle" is used. We will use an oracle when we create the Deutsch algorithm as well. An oracle is used to describe a "quantum black box". Internally, it is composed of one or more quantum gates, but we typically don't know which gates. By querying the oracle (e.g. by sending input and measuring the output), we can learn more about the properties of the oracle. Because an oracle is composed of quantum gates, the oracle itself needs to be reversible as well. An oracle can be considered as the quantum equivalent of the black-box functions we discussed earlier in this chapter. Both an oracle and a function perform some calculations, but we don't know the internal details about these calculations.

We will give an example of an oracle that is used in a simple quantum application. Using the

Strange simulator, you define an oracle by providing the matrix that is the mathematical representation of the oracle.

Let's consider the following snippet, which is taken from the sample in `ch09/oracle`

Listing 9.3 Introducing an oracle in quantum applications

```
QuantumExecutionEnvironment simulator = new SimpleQuantumExecutionEnvironment();
Program program = new Program(2);                                              ❶
Step step1 = new Step();
step1.addGate(new Hadamard(1));                                                ❷


Complex[][] matrix =  new Complex[][]{                                         ❸
        {Complex.ONE,Complex.ZERO,Complex.ZERO,Complex.ZERO},
        {Complex.ZERO,Complex.ONE,Complex.ZERO,Complex.ZERO},
        {Complex.ZERO,Complex.ZERO,Complex.ZERO,Complex.ONE},
        {Complex.ZERO,Complex.ZERO,Complex.ONE,Complex.ZERO}
};

Oracle oracle = new Oracle(matrix);                                            ❹

Step step2 = new Step();                                                       ❺
step2.addGate(oracle);

program.addStep(step1);                                                        ❻
program.addStep(step2);

Result result = simulator.runProgram(program);                                 ❼
Renderer.showProbabilities(program,1000);
Renderer.renderProgram(program);
```

❶ We create a quantum program that requires 2 qubits

❷ In a first step, we apply a Hadamard gate to the first qubit

❸ We create a matrix containing complex numbers.

❹ We create an oracle based on this matrix

❺ We create a second step in which the oracle is applied

❻ Both steps are added to the quantum program

❼ The program is executed, and we display its circuit and the results of 1000 runs.

The circuit is shown in Figure 9.5.

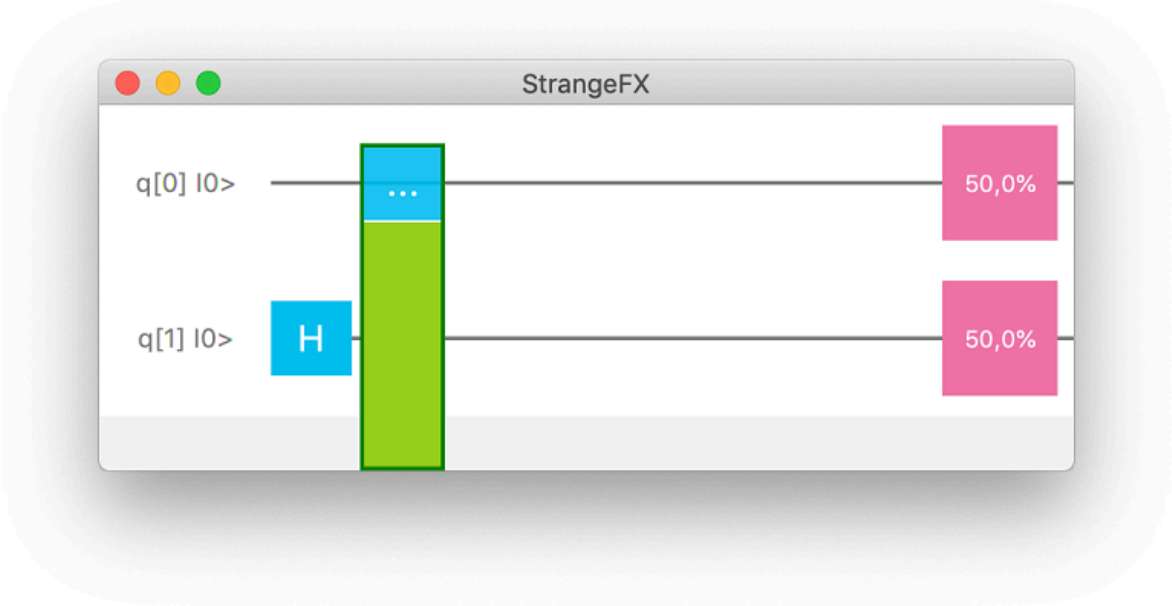

Figure 9.5 Circuit containing an Oracle

The result of running this circuit 1000 times is shown in Figure 9.6.

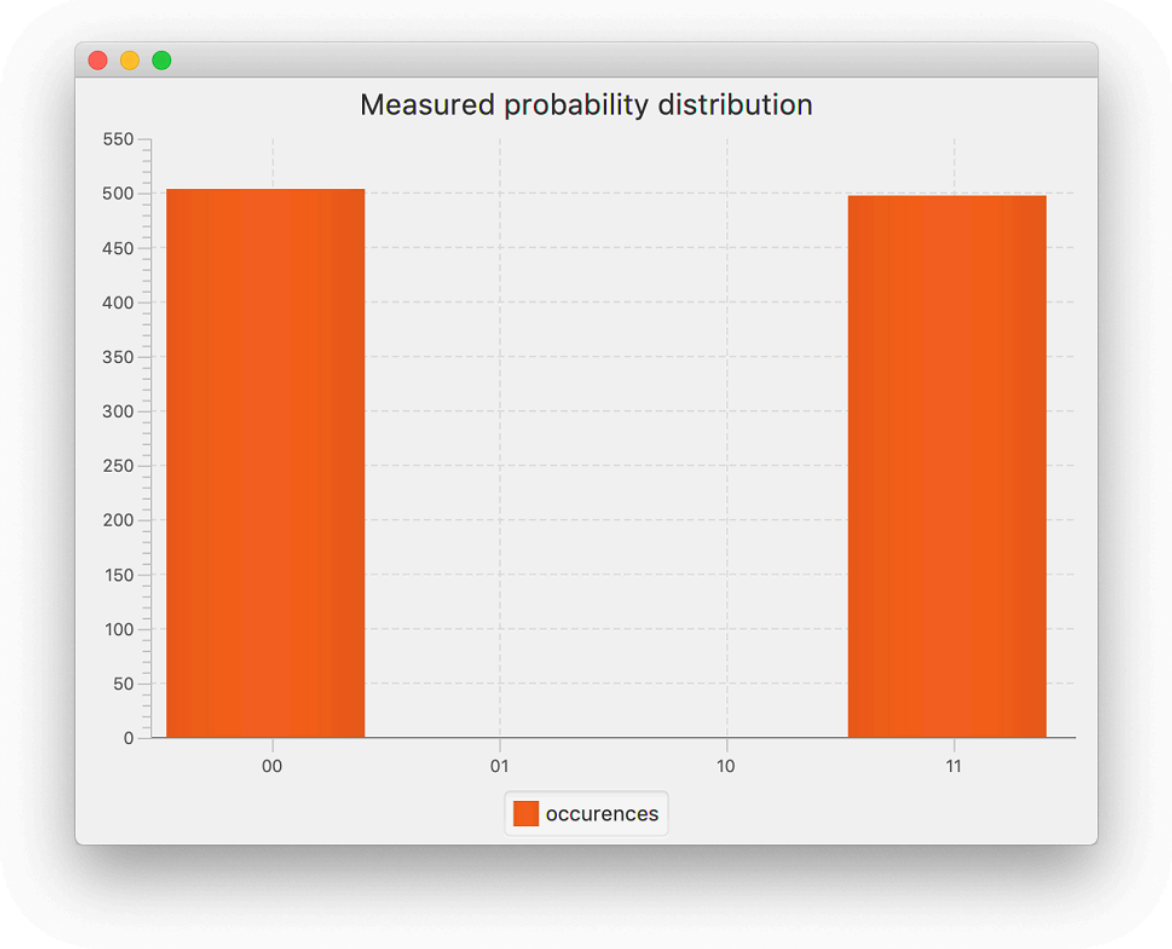

Figure 9.6 Statistical results of a quantum program containing an oracle

If you look at those statistical results, it seems that there are only 2 possible outcomes: `|00>` or `|11>`. Remember from Chapter 5 that a circuit with 2 entangled qubits has the same probabilities: there is 50% that we will measure `|00>` and 50% that we will measure `|11>`. This is an indication that the Oracle we created, combined with the initial Hadamard gate, results in 2 entangled qubits. In Chapter 5, you created 2 entangled qubits by applying a CNot gate after a Hadamard gate. Hence, we learn that the oracle we created behaves like a CNot gate. If we cheat, and we look at the contents of the oracle, we see that indeed the matrix that is representing the oracle matches the matrix of the CNot gate.

This exercise shows that we can apply oracles to quantum circuit. We can apply an oracle Without knowing the internal details of the oracle.

9.5 From functions to Oracle

In the Deutsch algorithm, we will show that a single evaluation is enough to find out if a provided function is constant or balanced. Before we can do that, we need to convert the classical function into a quantum operation.

We can not simply apply a function to a qubit. Remember we explained that all quantum gates need to be reversible. A function that, after being applied makes it impossible to retrieve the original input, can not be used in a quantum circuit. Therefore, the function first need to be transformed into a reversible Oracle.

IMPORTANT The creation of the function, and the creation of the oracle is should be considered as a totally separated process. In the upcoming algorithms, we assume someone created an oracle for us. The algorithm itself has no clue about how the oracle was created, how complex or simple it is etc. This is often a confusing part, since in order to demonstrate the algorithm, we obvisouly need an oracle. However, the complexity to create the oracle should not be considered as part of the complexity of the algorithm. Just assume someone (yourself, another developer, a real piece of hardware, or nature itself) created the oracle and provided it to you.

In this section, we will demonstrate how oracles can be created, that can then be used in the Deutsch algorithm we explain in the next section. Similar to how a classical function is handed to the classical algorithm, an oracle is handed to the quantum algorithm.

Every classical function that we described earlier in this chapter can be represented by a specific oracle. Since we had 4 possible functions, we also have 4 possible oracles.

The general way to construct an oracle based on a function is shown in Figure 9.7.

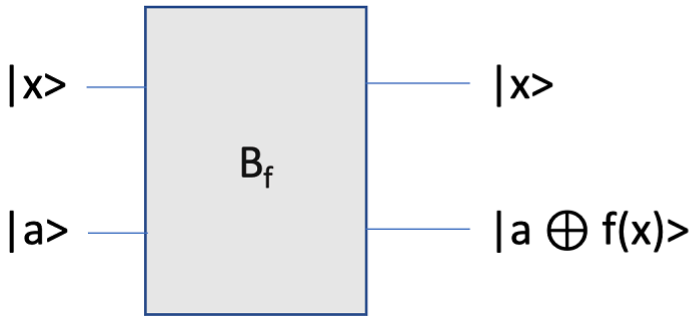

Figure 9.7 Oracle used in Deutsch algorithm

In this appraoch, we have an input qubit called `|x>`, and an additional qubit named `|a>`

> **SIDEBAR** Many quantum algorithms require additional qubits for making operations reversible, or for storing temporary results etc. These qubits are often called **ancilla** qubits. We often denote them as `|a>`

The oracle leaves the `|x>` qubit in its original state, and the `|a>` qubit is replaced by the XOR operation between `a` and `f(x)`.

Let's examine this oracle in a bit more detail. We will investigate how the oracle looks like for the 4 functions that we defined before.

9.5.1 Constant functions

The first function, *f1*, is a constant function that returns `0` regardless of the input. Hence, since $f(x) = 0$ for any value of x, the output status of the second wire can be simplified as follows:

$a \ f(x) = a \ 0 = a$

The resulting oracle can thus be pictured as in Figure 9.8.

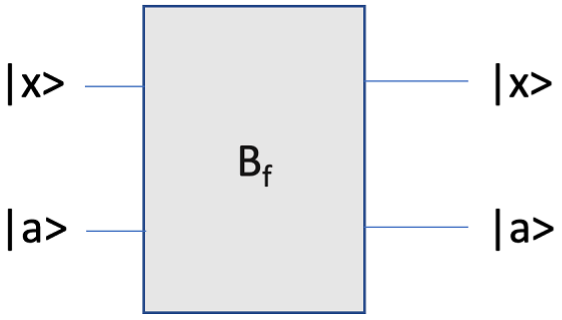

Figure 9.8 Oracle used in Deutsch algorithm for f1

In this case, the oracle is simply the identity matrix. Both `|x>` and `|a>` are unaltered between the input and the output. The hidden logic inside the oracle can thus be represented by the scheme in Figure 9.9.

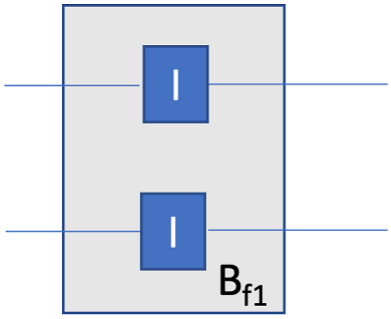

Figure 9.9 Oracle circuit for f1

As a result, the matrix representing the oracle is written as

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

NOTE the circuit shown in 9.9 isn't the only possible circuit that results in the identity matrix. There are many other circuits that operate on 2 qubits and return the 2 qubits in the same state. For example, applying 2 Pauli-X gates on each qubit results in the exact same state. This is part of the black box aspect of an oracle: we don't know the internal details, and we are typically not interested in those. We want to investigate a specific property of the oracle, not its internal implementation.

The fourth function, *f4* is a constant function that always returns the value `1`, regardless of the input. As a consequence, the output of the second wire after applying the oracle can be written as follows:

$a \; f(x) = a \; 0 = \bar{a}$

The vertical bar above a variable indicates that this is the inverted variable, which in this case corresponds to a Pauli-X gate being applied to `|a>`.

Hence, this oracle can schematically presented by Figure 9.10.

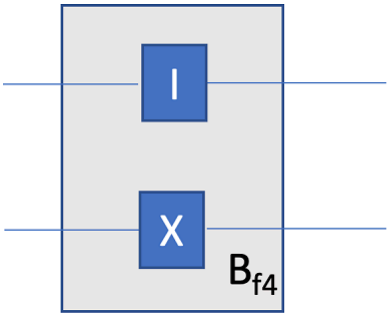

Figure 9.10 Oracle circuit for f4

The matrix corresponding to this oracle can therefore be written as

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

9.5.2 Balanced functions

Let's have a look at the second classical function, *f2*. This function is defined as follows:

f(0) = 0

f(1) = 1

This can also simply be written as

f(x) = x

Using this in the general description of the oracle, as shown in Figure 9.7, the scheme simplifies to Figure 9.11.

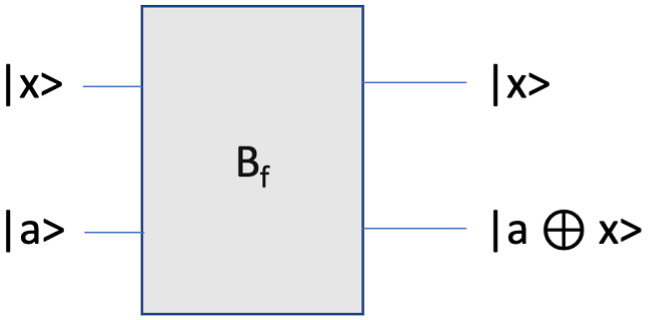

Figure 9.11 Oracle used in Deutsch algorithm for f2

This is exactly the state that would be obtained if the oracle has a C-NOT gate. Hence, a possible circuit of the oracle corresponding to the *f2* function is shown in Figure

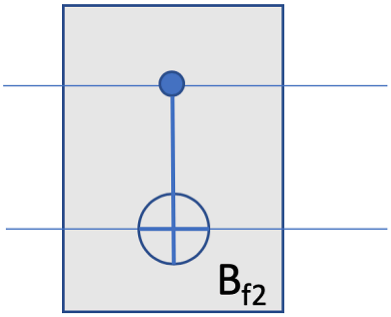

Figure 9.12 Oracle circuit for f2

The matrix representation of this Oracle is thus simply the matrix representation of the C-NOT gate:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

TIP As an exercise, you can now calculate the matrix representation of the oracle corresponding to the *f3* function. The result you should obtain is this:

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

9.6 Deutsch algorithm

The Deutsch algorithm requires only a single evaluation of the oracle in order to know whether the function under consideration is constant or balanced.

Let's start with a naive approach, and assume that all we need is to apply the oracle and measure the result.

The code for this is in `ch9/applyoracle`. Before we explain the algorithm, we first point to the part of the code where the different oracles are created. As we stated before, the creation of an oracle is **not** part of the algorithm that tries to find out if a function is balanced or not. Although for practical reasons we create the oracle in the same Java class file as the algorithm, it should be stressed that the creator of the oracle (who might now the outcome of the problem) and the creator of the algorithm are not the same.

From the previous section, it should be clear that there are 4 different types of oracles. There are an infinite number of oracles that can be used to represent the simple functions we discussed in the beginning of this chapter, but they all corresponds to one of the 4 gate matrices that we showed in the previous section.

The algorithm will ask to pick a random oracle, and the code that constructs this random oracle is shown in 9.4

Listing 9.4 Create an oracle

```
static Oracle createOracle(int f) {                                    ❶
    Complex[][] matrix = new Complex[4][4];                             ❷

    switch (f) {
        case 0:                                                         ❸
            matrix[0][0] = Complex.ONE;
            matrix[1][1] = Complex.ONE;
            matrix[2][2] = Complex.ONE;
            matrix[3][3] = Complex.ONE;
            return new Oracle(matrix);
        case 1:                                                         ❹
            matrix[0][0] = Complex.ONE;
            matrix[1][1] = Complex.ONE;
            matrix[2][3] = Complex.ONE;
            matrix[3][2] = Complex.ONE;
            return new Oracle(matrix);
        case 2:                                                         ❺
            matrix[0][1] = Complex.ONE;
            matrix[1][0] = Complex.ONE;
            matrix[2][2] = Complex.ONE;
            matrix[3][3] = Complex.ONE;
            return new Oracle(matrix);
        case 3:                                                         ❻
            matrix[0][1] = Complex.ONE;
            matrix[1][0] = Complex.ONE;
            matrix[2][3] = Complex.ONE;
            matrix[3][2] = Complex.ONE;
            return new Oracle(matrix);
        default:                                                        ❼
            throw new IllegalArgumentException("Wrong index in oracle");
    }
}
```

❶ When this function is called, an integer needs to be provided indicating what oracle type needs to be returned

❷ In all cases, the result is a 4 by 4 matrix with complex numbers

❸ In case the called provided `0`, an oracle with the matrix corresponding to *f1* will be returned

❹ In case the called provided `1`, an oracle with the matrix corresponding to *f2* will be returned

❺ In case the called provided `2`, an oracle with the matrix corresponding to *f3* will be returned

❻ In case the called provided `3`, an oracle with the matrix corresponding to *f4* will be returned

❼ If we reach here, the caller provided a wrong value, and we throw an exception.

Now that we have the code that returns us an oracle corresponding with a value we provide, we can focus on the algorithm that should detect if the oracle is linked with a constant function or a balanced function.

We start with the simple naive approach where we just apply the Oracle to 2 qubits that are initially 0, and we hope that the result will tell us with 100% confidence if the underlying function is balanced or not. The code for this is shown below:

Listing 9.5 Apply the oracle

```
static void try00() {                                                    ❶
    QuantumExecutionEnvironment simulator = new SimpleQuantumExecutionEnvironment();
    Program program = null;
    for (int choice = 0; choice < 4; choice++) {                         ❷
        program = new Program(2);                                        ❸

        Step oracleStep = new Step();                                    ❹
        Oracle oracle = createOracle(choice);
        oracleStep.addGate(oracle);
        program.addStep(oracleStep);

        Result result = simulator.runProgram(program);                   ❺
        Qubit[] qubits = result.getQubits();

        boolean constant = (choice == 0) || (choice == 3);               ❻

        System.err.println((constant ? "C" : "B") + ", measured = "
                + qubits[0].measure() + ", " + qubits[1].measure());
    }
}
```

❶ This function is called try00 as it applies the oracles to 2 qubits that are in their initial state of `|0>`.

❷ We will iterate over the 4 possible oracle types

❸ We create a quantum program that contains 2 qubits

❹ We create the oracle corresponding to the loop index *choice* and add it to the program

❺ The program is executed, and the results are obtained

❻ Based on the loop index *choice* we know if the oracle corresponds to a balanced or constant function. We print that information together with the measurements of the 2 qubits.

The result of this application is shown below:

```
C, measured = 0, 0
B, measured = 0, 0
B, measured = 1, 0
C, measured = 1, 0
```

Note that since we didn't use superposition, these results are always the same.

Let's investigate these results. There are 2 possible outcomes we can measure: the result is either `0, 0` or `1, 0`. Unfortunately, a single result doesn't tell us whether the function was constant (as indicated by the `C`) or balanced (as indicated by the `B`). For example, if we measure `0, 0` the function is either *f1* or *f2*, but since the first is constant and the second is balanced, we don't have an answer to our question.

We can be more clever, and try to run the application again, but this time we first flip one of the qubits, or both, to the `|1>` state using a Pauli-X gate. This is shown in the code in the same file, and it is a good exercise to create this code yourself before looking at the sample.

Doing so, we run 4 versions of our simple program, each version corresponding to a different initial state of the 2 qubits. The result is shown below.

```
Use 00 as input
C, measured = 0, 0
B, measured = 0, 0
B, measured = 1, 0
C, measured = 1, 0

Use 01 as input
C, measured = 1, 0
B, measured = 1, 0
B, measured = 0, 0
C, measured = 0, 0

Use 10 as input
C, measured = 0, 1
B, measured = 1, 1
B, measured = 0, 1
C, measured = 1, 1

Use 11 as input
C, measured = 1, 1
B, measured = 0, 1
B, measured = 1, 1
C, measured = 0, 1
```

If you analyse this result, you will conclude that none of those versions is sufficient to detect wether the oracle corresponds to a constant or a balanced function by doing a single evaluation.

But so far, we didn't use the powerful superposition. We will now do that.

Before we write the code for the algorithm, we show the quantum circuit that leads to the result in Figure 9.13.

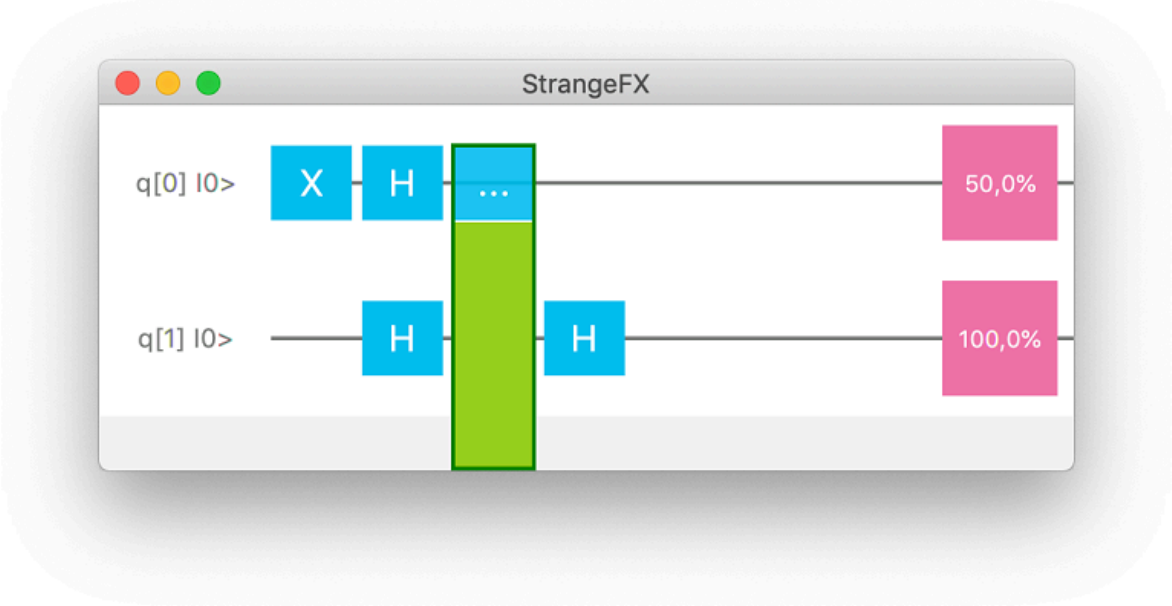

Figure 9.13 Quantum circuit of the Deutsch algorithm

We start with 2 qubits. The first qubit is the one that will be evaluated. But instead of evaluating twice, the first time in the state `|0>` and the second time in the state `|1>`, we apply a Hadamard transform to it to bring it in a superposition. You can think of this that this allows us to evaluate both possible values in a single quantum step.

The second qubit, which is initally `|0>` as well, is first flipped into `|1>` by applying a Pauli-X gate. Both qubits are then used as input to the Oracle we discussed in the previous section. After the Oracle has been applied, we discard the second qubit. On the first qubit, a Hadamard gate is applied, and the qubit is measured. If the measurement is `0`, we are guaranteed that the function that is represented by the oracle is balanced. If the measurement is `1`, we know that the considered function is constant.

It can be proven mathematically that the probability to measure `0` for the first qubit after the circuit is applied is given by

$$\left(\frac{1}{2}\left((-1)^{f(0)} + (-1)^{f(1)}\right)\right)^2$$

In case f is a constant function, this will always result in `1`.

In case f is a balanced function, this will always result in `0`.

The interesting part of this algorithm is that we managed to make the first qubit dependend on the evaluation of all values. We don't have the measurements for all these evaluations, but that was not the original question. The original goal was to determine if a given function is constant

or balanced.

We will now show the code that implements the algorithm:

```
QuantumExecutionEnvironment simulator = new SimpleQuantumExecutionEnvironment();
   Random random = new Random();
   Program program = null;
   for (int i = 0; i < 10; i++) {                                    ❶
       program = new Program(2);                                     ❷
       Step step0 = new Step();
       step0.addGate(new X(0));                                      ❸

       Step step1 = new Step();                                      ❹
       step1.addGate(new Hadamard(0));
       step1.addGate(new Hadamard(1));

       Step step2 = new Step();
       int choice = random.nextInt(4);
       Oracle oracle = createOracle(choice);                         ❺
       step2.addGate(oracle);                                        ❻

       Step step3 = new Step();
       step3.addGate(new Hadamard(1));                               ❼

       program.addStep(step0);                                       ❽
       program.addStep(step1);
       program.addStep(step2);
       program.addStep(step3);
       Result result = simulator.runProgram(program);                ❾
       Qubit[] qubits = result.getQubits();                          ❿
       System.err.println("f = "+choice+", val = "+qubits[1].measure());
   }
```

❶ The following loop will be executed 10 times, each time with a random oracle.

❷ We create a program with 2 steps

❸ In a first step, we apply a Pauli-X gate to the first qubit

❹ In the second step, we apply Hadamard gates to both qubits

❺ A random oracle is choosen (from a predefined list)

❻ The oracle is added to the quantum circuit

❼ Another Hadamard gate is applied to the second qubit

❽ The steps are added to the quantum program

❾ The quantum program is executed

❿ The second qubit is measured, and based on its value we know the oracle corresponded with either a constant or a balanced function.

9.7 Deutsch Josza algorithm

The Deutsch algorithm shows that a specific problem that requires 2 evaluations in a classical approach can be solved by a single evaluation using a quantum algorithm. While this may sound a bit disappointing, the principle is very promising. The Deutsch algorithm can easily be extended to the Deutsch Josza algorithm, in which the input function is not operating on a single boolean value, but on *n* boolean values.

In this case, the function can be represented as

$f(x_0, x_1, ..., x_{n-1})$

which indicates that the function uses *n* bits that are either '0' or '1' as input. We are give such a function, and we are told that again the function is either constant (which means it always returns '0' or it always returns '1') or balanced (which means in half of the cases it returns '0' and in the other half it return '1').

The Deutsch algorithm is a special case of this situation, where $n = 1$. In that case, that are only 2 possible input scenario's. In case $n = 2$, there are 4 possible input scenarios. In general, there are 2^n scenarios when there are *n* input bits.

How much classical evaluations do we need to do before we really know 100% certain that the function is either constant or balanced? Suppose that we evaluate half of the possible scenarios (hence, $2^n/2$ which is 2^{n-1}). If at least one of the results is *0* and at least one of the results is *1*, we know that the function is not constant, so it must be balanced. But what can we conclude in call all evalutions resulted in the value '1'? In that case, it looks like the function is constant. But we still need one additional evaluation, as there is a probability that all the other evaluations will result in '0'. Hence, in order to be 100% certain, a function with *n* bits as input requires 2^{n-1} evaluations before we can conclude that the function is either balanced or constant.

However, using a quantum circuit similar to the one in the Deutsch algorithm only a single evaluation is required. The importance of this is that it shows that quantum algorithms are great for problems that require exponential complexity using a classical approach.

The Deutsch Josza algorithm is very similar to the Deutsch algorithm. It is shown in `ch09/deutschjozsa` and the relevant snippet is shown below:

```
        static final int N = 3;                                                    ❶

...

        QuantumExecutionEnvironment simulator = new SimpleQuantumExecutionEnvironment();
        Random random = new Random();
        Program program = null;
        for (int i = 0; i < 10; i++) {
            program = new Program(N+1);                                            ❷
            Step step0 = new Step();
```

```
            step0.addGate(new X(N));                                    ❸

            Step step1 = new Step();
            for (int j = 0; j < N+1; j++) {                             ❹
                step1.addGate(new Hadamard(j));
            }

            Step step2 = new Step();
            int choice = random.nextInt(2);
            Oracle oracle = createOracle(choice);                       ❺
            step2.addGate(oracle);

            Step step3 = new Step();
            for (int j = 0; j < N; j++) {
                step3.addGate(new Hadamard(j));                         ❻
            }

            program.addStep(step0);
            program.addStep(step1);
            program.addStep(step2);
            program.addStep(step3);
            Result result = simulator.runProgram(program);              ❼
            Qubit[] qubits = result.getQubits();
            System.err.println("f = "+choice+", val = "+qubits[0].measure());
        }
```

❶ Here we define how many input bits we use (in this case, we use 3 input bits).

❷ We create a progam with `N+1` qubits. We need `N` qubits for the input bits, and an additional ancilla qubit

❸ A Pauli-X gate is applied to the ancilla qubit

❹ A Hadamard gate is applied to all qubits, bringing them into superposition.

❺ A random oracle is added to the circuit.

❻ A Hadamard gate is applied to all input qubits (not to the ancilla qubit)

❼ The program is executed and the result of the first qubit is measured.

As an exercise, you can proof that the Deutsch-Jozsa algorithm is exactly the same as the Deutsch algorithm in case `N = 1`.

The circuit for this algorithm, in case we have 3 input qubits, is shown in Figure 9.14.

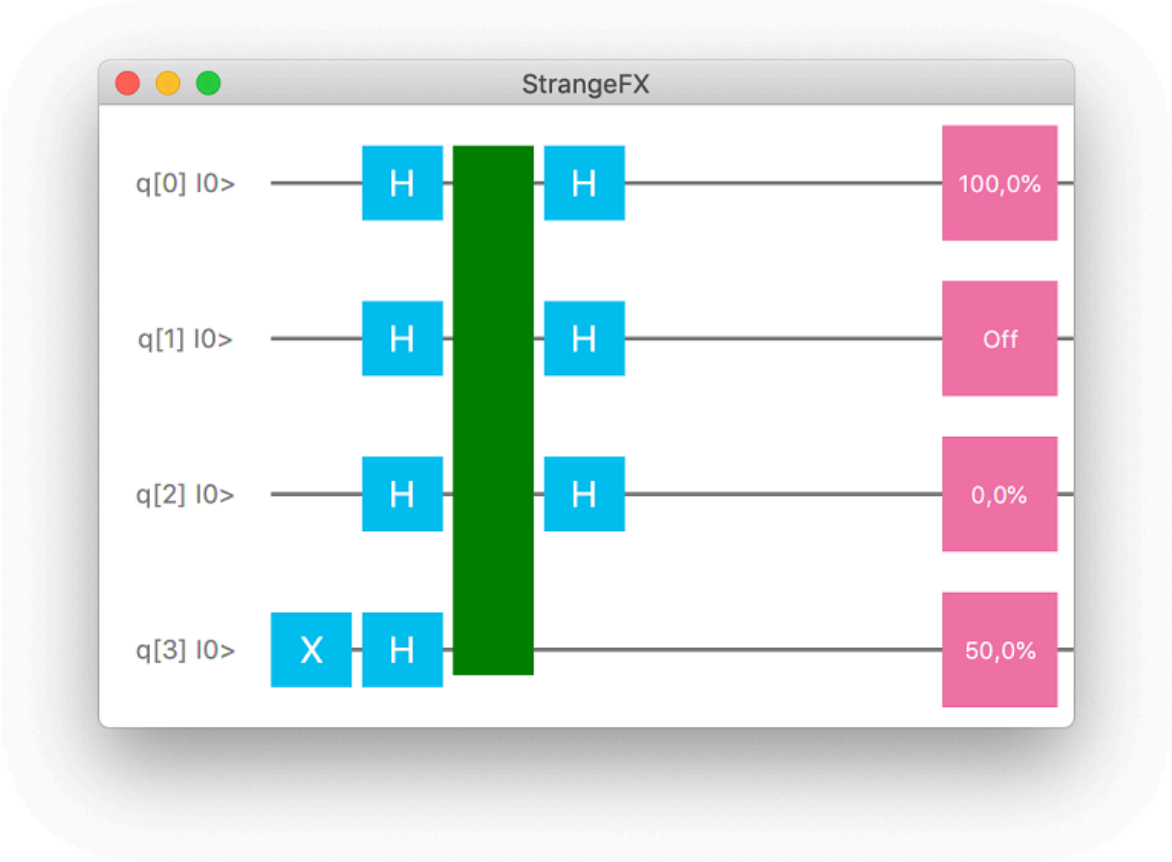

Figure 9.14 Quantum circuit of the Deutsch-Jozsa algorithm

If we apply this circuit, it can again be proven that the probability of measuring `0` in the first qubit is given by the following equation:

Similar to the Deutsch algorithm, in case *f(x)* is a constant function, this equation shows the probability to measure `0` on the first qubit is 100%.

In case *f(x)* is a balanced function, the first qubit will always be measured as `1` (as the probability to measure `0` is null).

The code in the sample will randomly pick one of 2 predefined Oracles. The first oracle corresponds to the Identity gate, and that corresponds to a constant function that always returns 0. The second oracle corresponds to a CNot gate, where the ancilla qubit is swapped in case the last input qubit is 1.

Again, our goal is not to create those oracles. You can assume that these are somehow provided to you, and you have to find out of these oracles correspond to either constant or balanced functions.

9.8 Summary

In this chapter, you created the Deutsch-Jozsa algorithm. While there is no direct practical usage of this algorithm, you achieved a major milestone. For the first time in this book, you created a quantum algorithm that can execute a task much faster than a corresponding classical algorithm. The real speedup can only be seen if you use a real quantum computer, but the algorithms you created clearly show that a single evaluation is required to fix a specific problem, whereas in classical computing an exponential number of evaluations is required.

Two very important but challenging parts of quantum computing are

- to come up with quantum algorithms like this that are proven to be faster than corresponding classical algorithms
- to find practical use cases for such algorithms.

In the next chapter, we will discuss two algorithms that satisfy these requirements.

In this chapter,

- you learned that problems can be solved without calculating end results
- you learned the difference between function evaluations and function properties
- you explored balanced and constant functions, and wrote a classical algorithm to detect whether a provided function is balanced or constant
- you learned how to convert a classical function into a quantum oracle
- you wrote a quantum algorithm that detects whether a supplied oracle corresponds to a balanced or a constant function.

10 Grover's Search Algorithm

This chapter covers

- the positioning of Grover's search algorithm relative to existing data storage systems
- an explanation of cases and problems where Grover's search may be applicable.
- classical examples showing cases where Grover's search could be applicable too
- how to use Grover's search from classical Java code
- an intuitive explanation on the internal working of Grover's search algorithm

In this chapter, we will answer two important questions that developers have:

1. When is it a good idea to use Grover's search algorithm?
2. How does the algorithm work?

Grover's search algorithm is one of the most popular and well-known quantum algorithms. Despite its name, this algorithm is not really a replacement for search algorithms that are used today in classical software projects. In this chapter, we will explain what kind of problems could benefit of Grover's search algorithm. After reading this chapter, you will be able to determine if a particular application you are dealing with can leverage Grover's search algorithm. If so, you can immediately use the classical API in Strange that allows to use Grover's algorithm.

10.1 Do we need yet another search architecture?

There are many excellent libraries, protocols, techniques available for searching structured and unstructured data systems. Grover's search algorithm doesn't compete with those technologies.

10.1.1 Traditional search architecture

Searching a database is one of the most popular tasks delegated to computers. Many IT applications are architected in a 3-tier approach, as shown in Figure 10.1:

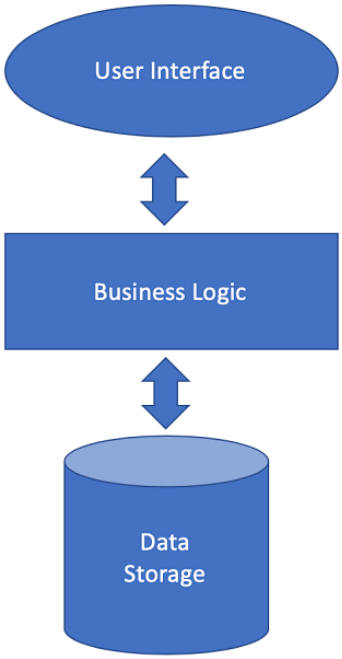

Figure 10.1 Typical three-layered architecture for classical applications.

The 3 layers that are involved in this approach are

- a user interface (or presentation layer) allows interaction, requests input and renders output. This is typically a standalone desktop application, a mobile app, or a website.
- a middle tier deals with business logic, rules, and processes. This tier handles requests from the presentation layer, and might require access to data in order to process incoming requests.
- the data layer makes sure all data is stored and retrieved in data storages. Very often, the data is made available via a developer-friendly API, that allows to search or modify data based on different criteria in a performant way.

In many, the middle tier needs to query the data layer, to find some specific data, based on specific requirements. The quality and performance of many applications strongly depends on how flexible, reliable and performant these queries are handled. Therefore, the area of data storage and retrieval is a very important one in todays IT industry.

There are many different approaches to store and retrieve data, and this domain is constantly evolving. There are relational and non-relational databases, and SQL versus NoSQL approaches. Quantum Computing, and Grover's search algorithm in particular, does not provide a new architecture for storing and retrieving data.

CAUTION Although the name *Grover's search algorithm* implies it deals with search techniques, it does not cover the aspects that are typically discussed when talking about search architecture.

As a consequence, Grover's search algorithm is not a replacement for existing search software. It

can be useful, however, in existing or new software libraries and projects that implement search functionality. The algorithm can thus be used in different database techniques.

10.1.2 What is Grover's search algorithm?

Now that we discussed what Grover's search algorithm is **not** about, it is probably relevant to explain what it is about then. We will briefly mention this now, and the link between search applications and Grover's algorithms will become clear in the subsequent sections.

> **SIDEBAR** Suppose we are provided a black box that requires an integer number as input, and that will return an output. The output is always `0`, except for one specific input value (often noted as *w*), in which case the output is `1`. Grover's search algorithm allows to retrieve this specific input value in a performant way.

The concept of the black box is shown in Figure 10.2.

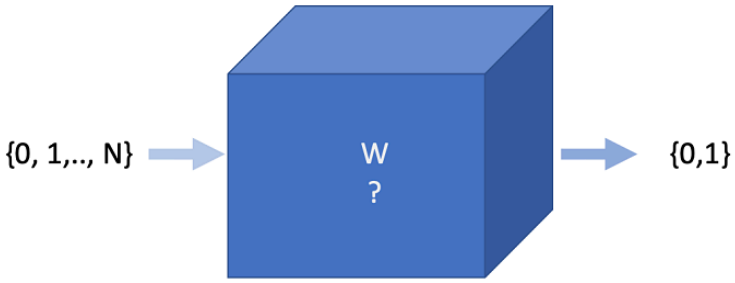

Figure 10.2 Black box returning either 0 or 1, based on an integer number.

Somehow, the black box checks if the provided input equals *w*. If so, the output will be `1`. Otherwise, the output will be `0`. It is important to realise that we don't know how the black box works internally. It might contain a very simple or a very complex algorithm. You have to assume someone created the black box, and handed it over to you — you were not involved in how it is created. Indeed, if you were the creator of the black box, there would be no point in writing algorithms to retrieve the value of *w*, as you would have used that value to create the black box.

Somehow, the black box contains information about the value *w*, and by querying it in a smart way, Grover's algorithm is capable of retrieving that value.

The input of Grover's search algorithm is not a number, a search query, a SQL string,… but it is the black box that we just discussed. This is explained in Figure 10.3.

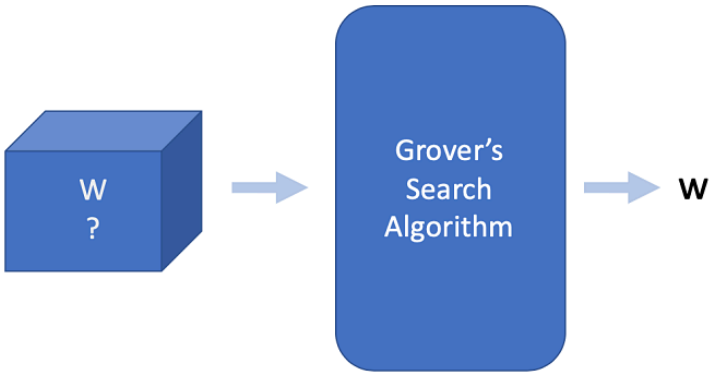

Figure 10.3 Grover's search algorithm takes a black box as input, and returns the number w that causes the black box to evaluate to 1.

We will now explain how Grover's algorithm can help with traditional search problems. We will start with a classical search problem, and refine that until we are at the point where we can introduce Grover's search.

10.2 Classical search problems

Most enterprise IT cases use a number of databases with tables and rows. For demo purposes, we create a simple data storage containing people, and we store their age and country. The following table describes our data:

Table 10.1 Table data used in our samples

	age	country
Alice	42	Nigeria
Bob	36	Australia
Eve	85	USA
Niels	18	Greece
Albert	29	Mexico
Roger	29	Belgium
Marie	15	Russia
Janice	52	China

Search applications can leverage this data to provide answers to questions like these:

- Who is 36 years old and lives in Australia?
- How many of those people live in Russia
- Is there someone named "Joe" who lives in Greece?
- Give me the names of all people older than 34 years.
- …

We will create an application that answers one specific question:

```
Find the person who is 29 years old and lives in Mexico
```

From the table above, you can see that the answer to this question is `Albert`.

We will first address this question using the classical approach. Next, we will reformulate the question, so that we can deal with it in a more functional approach, as this comes closer to Grover's algoritm. Finally, we will use Grover's algorithm to implement the functionality for this search. This corresponds to the following mental model:

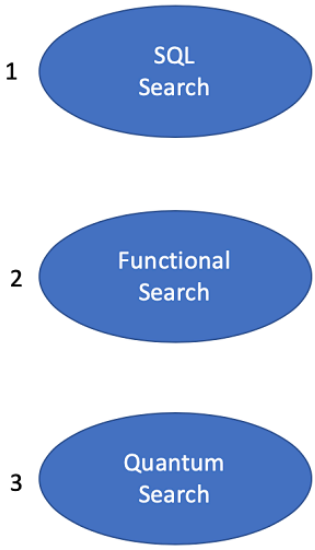

Figure 10.4 From SQL search to quantum search.

After reading this chapter, you will have a good idea about when Grover's search algoritm might be a good fit in your projects, and when it is less relevant.

10.2.1 General preparations

The upcoming samples share common code, and we will not repeat this common code in each sample.

THE PERSON CLASS

Before we start writing the search functionality, we will define the data we are talking about. All rows in the table represent a person, hence we create a Java class named `Person`

Listing 10.1 Definition of a Person

```
public class Person {

    private final String name;                                          ❶
    private final int age;                                              ❷
    private final String country;                                       ❸

    public Person(String name, int age, String country) {               ❹
        this.name = name;
        this.age = age;
        this.country = country;
    }

    public String getName() {                                           ❺
        return this.name;
    }

    public int getAge() {                                               ❻
        return this.age;
    }

    public String getCountry() {                                        ❼
        return this.country;
    }
}
```

❶ A person has a name

❷ A person has an age

❸ A person lives in a country

❹ When a Person object is created, the three properties need to be defined

❺ The class provives a method for returning the name of the person.

❻ The class provives a method for returning the age of the person.

❼ The class provives a method for returning the country of the person.

We will use this Person class in all upcoming samples.

CREATING THE DATABASE

While there are a large number of high-quality database libraries available in Java, we will stick to a very simple database representation in this sample. All instances of the `Person` class will be stored in a simple Java `List` object, since this is a standard class in the Java platform and we want to avoid introducing dependencies that are not essential to understand quantum computing. As we mentioned before, the goal of Grover's search algorithm is not to create another classical database library. At the contrary, we will explain the algorithm without depending on a particular type of database.

The following code snippet will populate our database: we simply add a number of `Person` instances into the `List` that is then our data store.

Listing 10.2 Creating the database

```
List<Person> prepareDatabase() {
    List<Person> persons = new LinkedList<>();
    persons.add(new Person("Alice", 42, "Nigeria"));
    persons.add(new Person("Bob", 36, "Australia"));
    persons.add(new Person("Eve", 85, "USA"));
    persons.add(new Person("Niels", 18, "Greece"));
    persons.add(new Person("Albert", 29, "Mexico"));
    persons.add(new Person("Roger", 29, "Belgium"));
    persons.add(new Person("Marie", 15, "Russia"));
    persons.add(new Person("Janice", 52, "China"));
    return persons;
}
```

You will use this method in all our samples. When you invoke this method, you will receive a `List` of `Person` items that match the predefined table at the beginning of this chapter.

10.2.2 Searching the list

We now write the code for the typical approach for searching our data store for an answer to the original question:

```
Find the person who is 29 years old and lives in Mexico
```

Given a list of persons who might be the answer to this question, the following approach iterates over all persons until the one satisfying the criteria is detected:

```
Person findPersonByAgeAndCountry(List<Person> persons, int age, String country) {
    boolean found = false;                                          ❶
    int idx = 0;                                                    ❷
    while (!found && (idx<persons.size())) {                        ❸
        Person target = persons.get(idx++);                         ❹
        if ((target.getAge() == age) &&                             ❺
            (target.getCountry().equals(country))) {
            found = true;                                           ❻
        }
    }
    System.out.println("Got result after "+idx+" tries");           ❼
    return persons.get(idx-1);                                      ❽
```

❶ We keep a boolean variable that indicates if we already got the result

❷ We also keep an index that tells us the position of the element we are investigating

❸ As long as we don't have a result, and the index is still lower than the total number of elements, we execute the following loop

❹ The element under consideration is obtained from the list

❺ The properties (age and country) of that element are checked

❻ If the properties match, we flip the boolean variable to true so that the loop is not needlessly executed.

❼ The number of evaluations is printed here.

❽ The result is returned to the caller.

NOTE the same could have been achieved by using the Java Streams API:

```
return persons.stream()
 .filter(p -> {return (p.getAge() == age && p.getCountry().equals(country));})
 .findFirst().get();
```

However, in this case the procedural aproach is easier to explain and it allows to count how many evaluations are required.

In case the list of persons contains the answer, we are guaranteed that this function returns the correct result. If we are lucky and the correct person is the first one in the list, we will get the answer after a single execution inside the `while` loop. If we have bad luck and the correct person is the last one in the list, the function requires `n` evaluations, with `n` being the number of elements in the list.

On average, the algorithm will require `n/2` evaluations before returning the correct result. You can verify this by executing this applications many times, and look at the number of evaluations that are printed by the program.

10.2.3 Searching using a function

The sample code from the previous section is very flexible, since we can easily modify the search criteria by providing a different age or a different country to the `findPersonByAgeAndCountr` method. Unfortunately, this is not how Grover's search algorithm work. With Grover's algorithm, we don't provide search parameters, but we have to provide a single function that evaluates to `1` for exactly 1 input case, and it evaluates to `0` in all other cases.

In the following, we often use the variable *w* for indicating the input case that will result in a function evaluation of `1` --- in other words, *w* is the value we are looking for. We can define this as follows:

$f(w) = 1$

$f(x) = 0,\ x \neq w$

In this section, we will modify the classical sample from the previous sample so that it is using a functional approach, which can then conceptually being mapped to the quantum algorithm in the next section.

We will perform the same search query, but instead of checking the entries by examining their properties one by one, we will apply a function to each entry. When the function evaluates to `1`, we know we have the correct entry.

We use the Java Function API to achieve this. First, we need to create the function. The code for this is shown in Listing 10.3:

Listing 10.3 Creating the function

```
Function<Person, Integer> f29Mexico                                          (1)
   = (Person p) ->                                                           (2)
       ((p.getAge() == 29) && (p.getCountry().equals("Mexico"))) ? 1 : 0;    (3)
```

(1) We create a Function that takes a `Person` as input, and returns an `Integer` as output

(2) When the function is applied, the parameter `p` contains the supplied person

(3) In case the age of the supplied person is `29` and the country is `Mexico`, the function returns `1`. In all other cases, the function returns `0`.

Note that this function is a fixed function for a particular problem. If we want to retrieve a person whose age is `36`, we need to create a new function.

Now that we have this function, we can write some Java code that iterates over the list of persons, and applies the function over each entry until the function returns `1` --- which means the correct answer has been found.

The code for this is shown below:

```
Person findPersonByFunction(List<Person> persons, Function<Person, Integer> function) {
    boolean found = false;
    int idx = 0;
    while (!found && (idx<persons.size())) {
        Person target = persons.get(idx++);
        if (function.apply(target) == 1) {
            found = true;                                                    (1)
        }
    }
    System.out.println("Got result after "+idx+" tries");
    return persons.get(idx-1);
}
```

(1) Instead of checking the properties of the target, like we did in the previous sample, we apply our function to it. When the function returns `1`, we know the target is the correct result.

While the approach is different, the required amount of time is similar to the previous algorithm. We still iterate over every person in the list, and check if the age and country of the considered person satisfies our criteria.

The second approach is closer to the quantum approach we will discuss in the following section. Instead of providing a number of parameters, we provide a function. The search algorithm does not have to *create* that function, but it has to *evaluate* it.

The differences between the two approaches are highlighted in Figure 10.5.

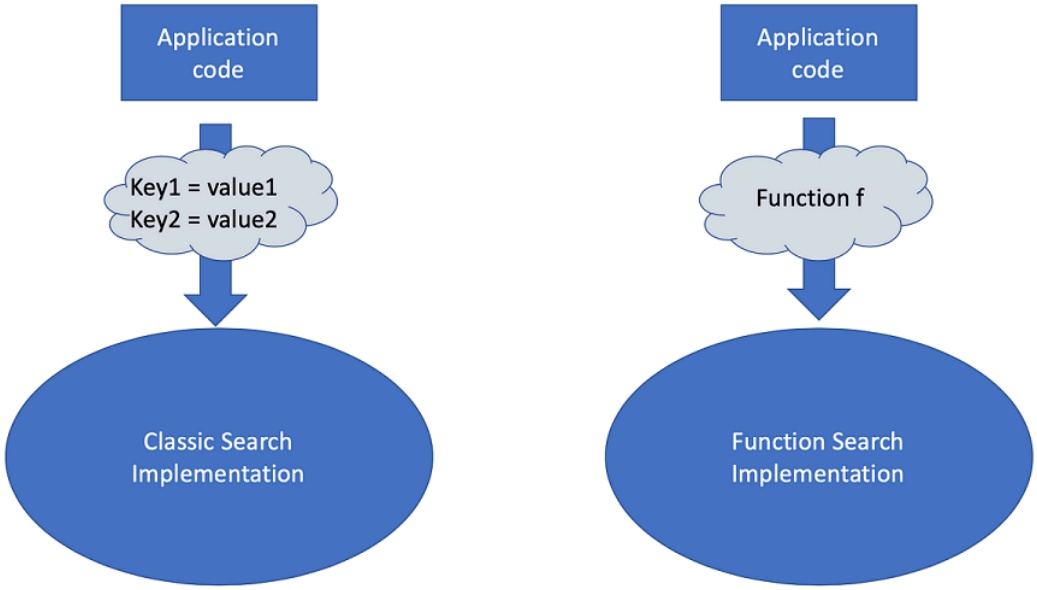

Figure 10.5 Searching using a classic approach versus using functions.

Note that the *Function Search Implementation* as we call it in the figure is still using classical code. However, it brings us closer to how Grover's search algorithm works on a quantum computer.

10.3 Quantum search: Using Grover's search algorithm

Grover's algorithm has some similarities with the function-based search discussed in the previous section.

When a black box (or a quantum oracle) is provided that is linked to a function like the one described in the previous section, Grover's algorithm can return the unique input w that would result in a function evaluation of `1` in about $\sqrt{n}$ steps, where each step requires a single evaluation of the oracle.

IMPORTANT In other words, while the classical search algorithm requires on average `n/2` function evaluations, Grover's algorithm achieves the same goal in n function evalutions.

For small lists, this is not impressive. A list with 8 elements requires on average 4 function evaluations for the classical case, and about 3 evaluations for the quantum case. However, for large lists, the advantage becomes clear. A list with 1 million elements might require 1 million classical evaluations, but the same result can be obtained with only 1000 quantum evaluations.

The difference between the required evaluations for a classical search versus quantum search using Grover's algorithm is visualised in Figure 10.6.

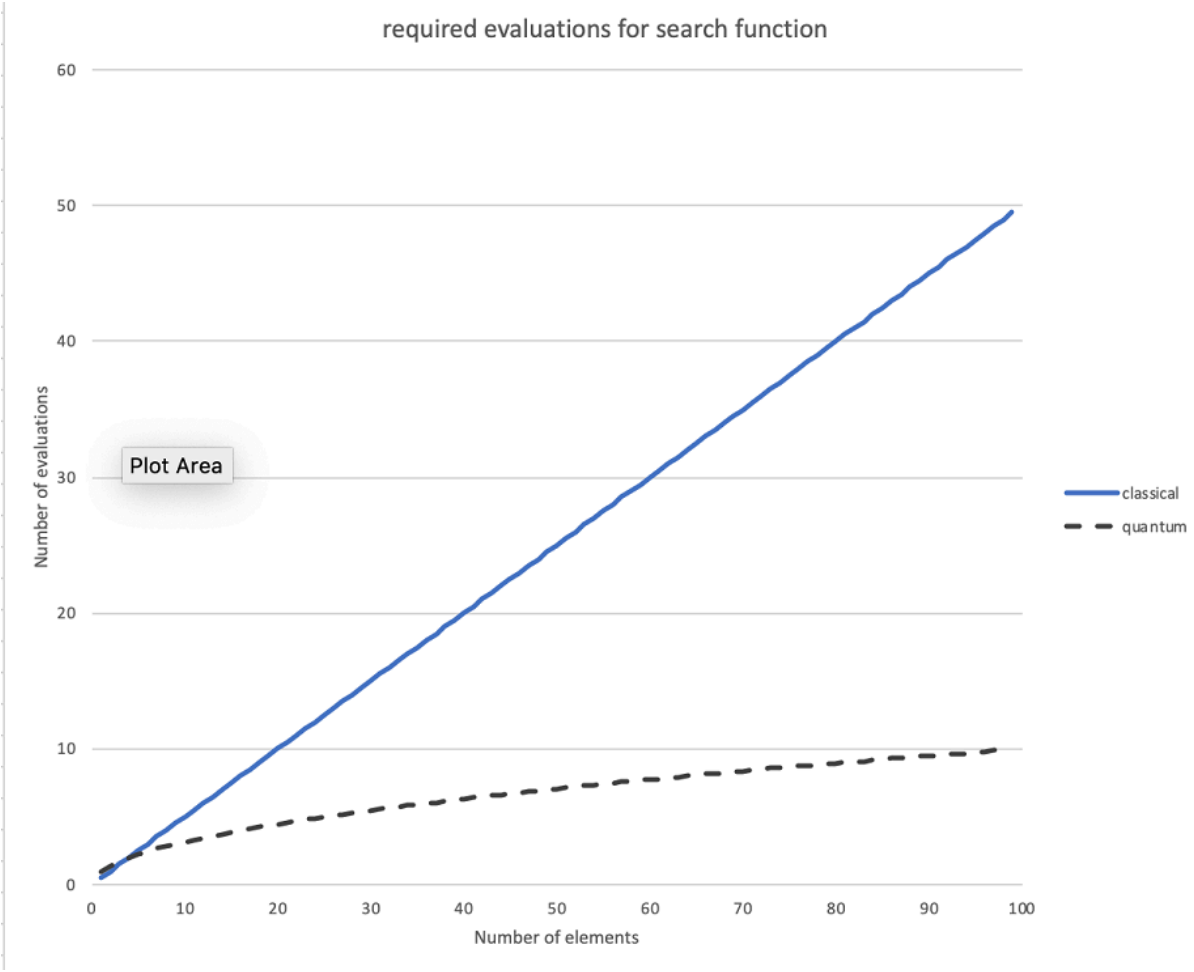

Figure 10.6 Required number of function evaluations as a function of number of items

In this figure, we only showed the difference between the classical search and Grover's search algorithm for lists of up to 100 elements. As you can see from this figure, the larger the size of the list, the more remarkable the differences become. Hence, it becomes clear that Grover's algorithm is particularly useful for lists with a huge number of elements.

SIDEBAR **quadratic speedup**

It is often said that Grover's algorithm provides a quadratic speedup compared to classical search algorithms. This is indeed true, since for a given amount of evaluations (e.g. `N`) Grover's algorithm can deal with a list of N^2 elements, while a classical algorithm can only deal with lists of N elements.

At the end of this chapter, we will explain how Grover's algorithm works. For developers, it is often more important to realise when an algorithm is applicable rather than how it works. Therefore, we first explain how to use the builtin Grover functionality in Strange, which hides the underlying implementation.

The Strange quantum library contains a classical method that under the hood uses Grover's search algorithm to implement a search operation. The signature of the classical method resembles the signature of the sample we discussed earlier in this chapter:

```
public static<T> T search(List<T> list, Function<T, Integer> function);
```

This method takes two parameters as input:

- a list with elements of type `T`, where `T` can be a `Person`, or any other Java class.
- a function which takes as input an element of type `T` and returns either `1` (in case the provided input is the one we are looking for) or `0` (in all other cases)

The following code snippet, which is taken from the sample in `ch10/quantumsearch` shows how we can leverage this method.

```
void quantumSearch() {
    Function<Person, Integer> f29Mexico                          ❶
          = (Person p) -> ((p.getAge() == 29) &&
            (p.getCountry().equals("Mexico"))) ? 1 : 0;
    List<Person> persons = prepareDatabase();                    ❷
    Collections.shuffle(persons);                                ❸
    Person target = Classic.search(persons, f29Mexico);          ❹
    System.out.println("Result of function Search = "            ❺
          + target.getName());
}
```

❶ We create a function, similar to the function in the previous sample

❷ We create the initial database again

❸ The elements in the database are randomly shuffled

❹ The search method that is under the hood invoking Grover's search is called

❺ The result is printed.

It is important to emphasize that the provided function is created outside the algorithm. In this case, the function is called `f29Mexico` and the `Classic.search` method does not need to know anything about the internals of that function. The function will be evaluated, but to the algorithm, this evaluation is a black box.

10.4 The algorithm behind Grover's search

The `Classic.search` method that is available in Strange allows to leverage Grover's search algorithm using classical computing only. One of its main benefits is that it allows developers to understand what type of problems could benefit from Grover's search.

The internal working of the quantum algorithm is less relevant to most developers, but there are some reasons why a basic understanding will benefit developers:

- Gover's search algorithm doesn't require a classical function as input, but a Quantum

Oracle that corresponds to a classical function. We will discuss this oracle in the upcoming sections.

- Understanding how Grover's search algorithm leverages quantum computing characteristics can help in creating or understanding other quantum algorithms.

In the following sections, we will explain the different parts of Grover's search algorithm. The mathematical evidence will be omitted though.

10.4.1 Running the sample code

The code in `ch10/grover` allows you to run Grover's search algorithm step by step. We will use this code to explain the algorithm.

The implementation of the algorithm is done in a method called `doGrover` which has the following signature:

```
private static void doGrover(int dim, int solution)
```

The first argument to this method, `dim`, specifies how many qubits should be involved. The second argument, `solution`, specifies the index of the element we are searching for. Again, note that in a real scenario, it doesn't make sense to provide the answer we are looking for (the `solution`) to the problem. Someone should provide us with a black box function. In this case though, the code will use the `solution` to construct the black box.

The main method of this sample is very short:

```
public static void main (String[] args) {
    doGrover(6,10);
}
```

We simply call the `doGrover` method and specify that we have a system with 6 qubits (hence, we can accomodate a list with $2^6 = 64$ elements) and the target element is at index `10`.

When running the sample, the quantum circuit will be shown, demonstrating the different steps of the algorithm, along with the probality vectors after each step. The image looks similar to Figure 10.7.

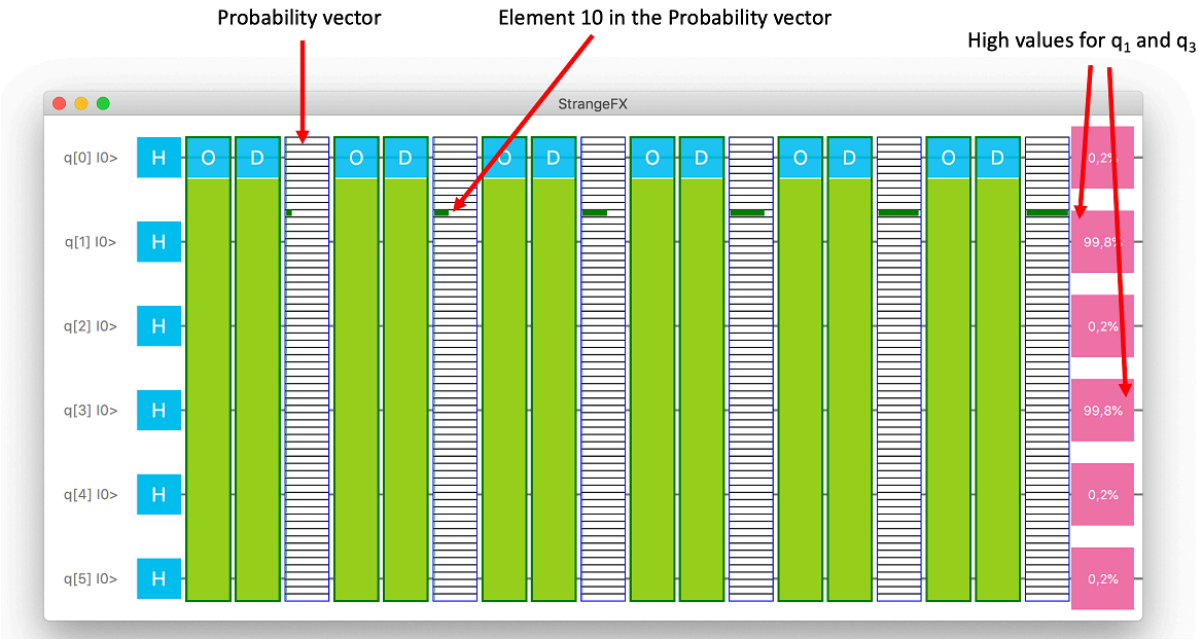

Figure 10.7 Running the Grover sample

The right side of this picture shows the qubits after the algorithm has been applied. Qubits 1 and 3 (denoted as q[1] and q[3]) have a very high probability of being measured as 1 (99,8%), while all other qubits have a very low probability of being measured as 1 (0,2%). As a result, there is a very high probability that the following sequence will be obtained when the qubits are measured:

```
0 0 1 0 1 0
```

This is the binary representation of the number `10`. Hence, Grover's search algorithm resulted in returning the index of the element we are searching for.

From Figure 10.7 it is also clear that the algorithm contains a step that is repeated a number of times. The flow is shown in Figure 10.8.

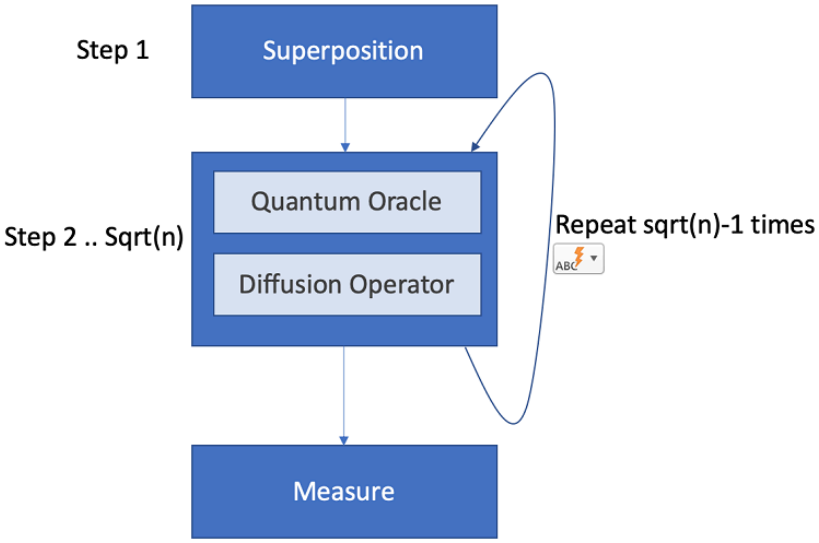

Figure 10.8 Flow of Grover's search algorithm

Each invocation of the step applies a quantum oracle denoted by an `O` and a diffusion operator denoted by a `D`. As we will explain later the `O` is the quantum oracle that corresponds to the function we have been given, and the `D` is the diffusion operator that we will introduce later.

After every invocation of this step, the probability vector is rendered.

After the first step, which applies a Hadamard gate to each qubit, all options have the same probablity. After the second step, all options have a rather low probablity. The element at index 10 has a higher probability than the others, but it is still low. Hence, if we would measure the system after this first step, there is a fair chance that we would measure the correct answer `10` but there is an even bigger chance that we would measure something else.

With each step, however, the probability of measuring `10` increases. At the last step, the probability of measuring 10 is 99,8%.

The sample in the `stepbystepgrover` is very similar to the sample discussed above, but it deals with a list of 4 items only, hence it can be handled by 2 qubits. This makes it easier to explain, and we will use that sample in the following subsections. Note that we also added more probability visualisations in this sample. After every step, the probability vector is rendered. The result is shown in Figure 10.9.

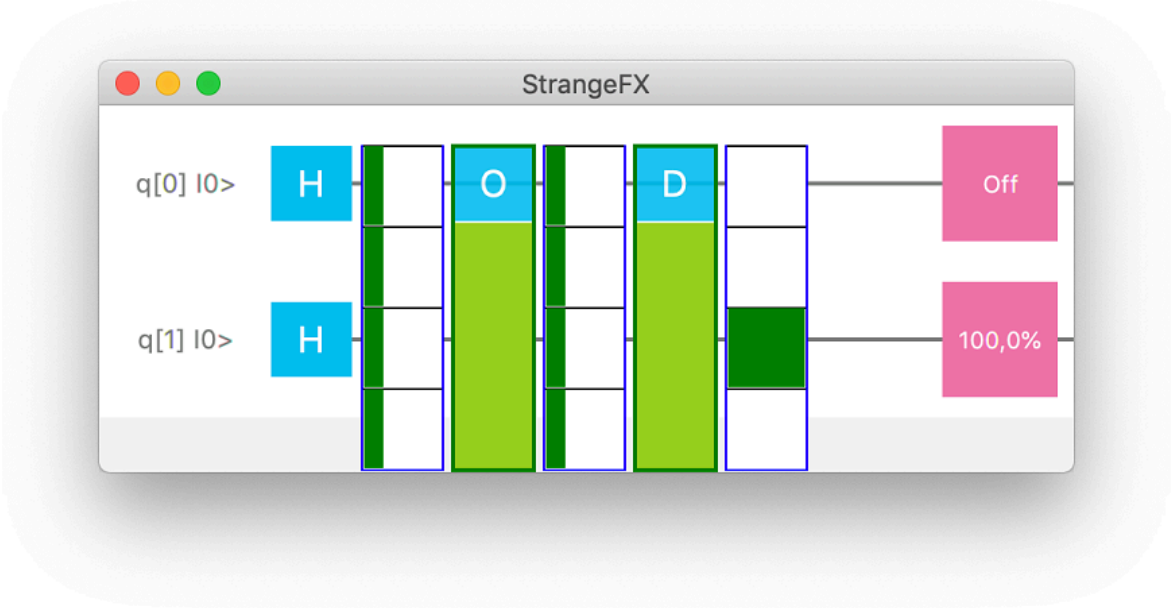

Figure 10.9 Running the Grover sample with only 2 qubits

10.4.2 Probabilities and amplitudes

Throughout this book, we emphasized the importance of *probabilities*. After applying a quantum circuit, we are left with a number of qubits, that can be in different states. The probability vector describes how likely it is to measure a specific value. The goal of many quantum algorithms is to manipulate the probabilty vector in such a way that the measured outcome is likely to be very relevant to the original question.

In Grover's search algorithm, we need n qubits if we want to search in a list of 2^n elements. For example, if our list has 128 elements, we need 7 qubits. If we have to deal with 130 elements, we need 8 qubits, and so on. After applying Grover's algorithm, we hopefully obtain a set of qubits that, when measured, return the index of the element in the list we are searching for.

This is explained in Figure 10.10 for a list with up to 64 elements.

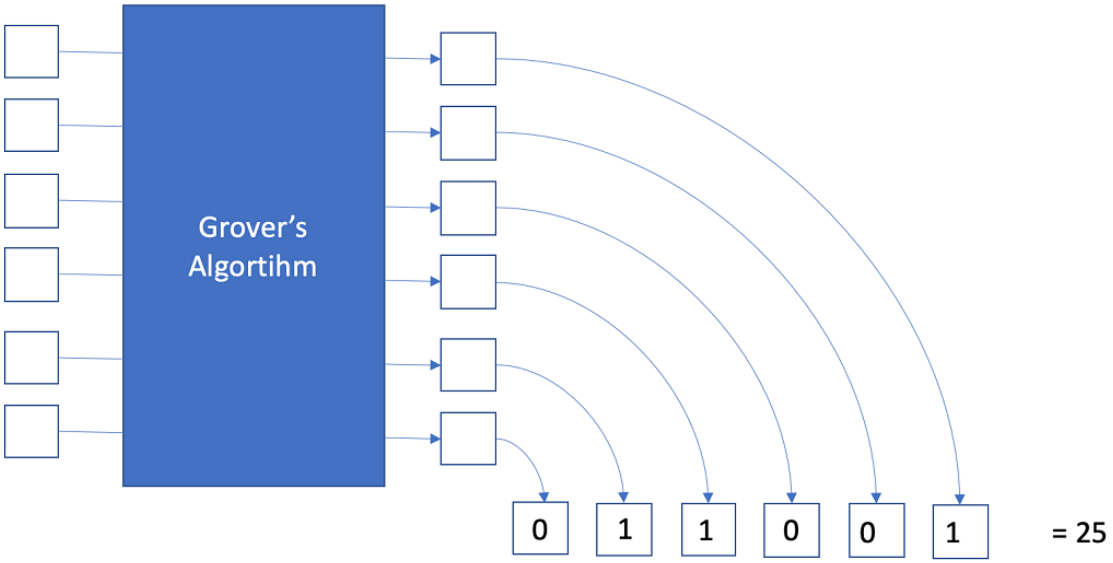

Figure 10.10 High level overview of what Grover's search algorithm achieves.

Suppose that the element we are searching has the index 25. Initially, all qubits have the value `0`. After applying Grover's algorithm, we measure the qubits, and we hope there is a very high probability to measure the values 0, 1, 1, 0, 0 and 1 which is the binary representation of 25.

With 6 qubits, there are $2^6 = 64$ possible outcomes, hence we have a probability vector with 64 elements. The goal of Grover's algorithm is to maximize the value of element 25, and to minimize the value of all other elements. We will show that in most cases, the resulting probability for the correct value is very high, but not 100%.

Grover's search algorithm requires a number of steps. We will show that with each step, the probability to find the correct answer is increasing.

Probabilities can be indicated by numbers, or by corresponding horizontal bars. A probability of 1 indicates that this specific outcome is guaranteed to be measured, and it corresponds with a filled horizontal bar. A probability of 0 means that there is no change that this specific outcome can be measured, and it correcponds with an empty horizontal bar. Numbers between 0 and 1 correspond with partially filled horizontal bars.

In case we have a system with 3 qubits, we have 8 possible outcomes. Figure 10.11 shows two different probability sets for this system, both with a vector with numbers and a notation with bars.

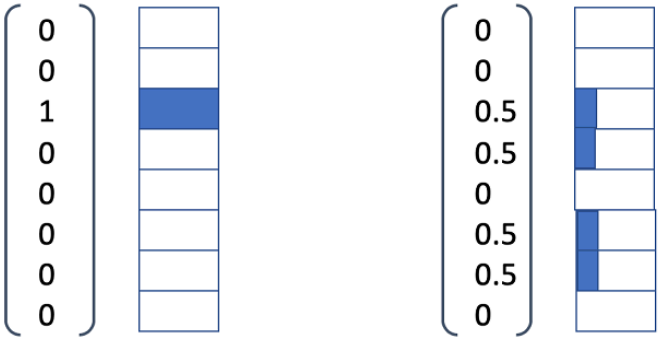

Figure 10.11 Probabilities in a vector and in a bar notation

Note that the sum of all probabilities should equal to 1.

The probabilities in the probability vector are based on the amplitudes While amplitudes can be positive as well as negative, probabilities are always positive. That is why the square is used. In the following sections, we will deal with amplitudes as well though.

We will now discuss the different steps in the algorithm, as shown in Figure 10.8.

10.4.3 Superposition

The first step in Grover's algorithm brings all qubits in a superposition state. This approach is often used in quantum algorithms, as it allows the processing to be applied to different cases simultaneously.

In our 2-qubit sample, the two qubits initially have the state |0>. Hence, the initial state vector is described by

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The probability vector, which is obtained by each element taking the square of the corresponding element in the state vector, looks exactly the same. Indeed, the square of 1 is 1, and the square of 0 is 0.

The first element in this vector corresponds with the probability of measuring the two qubits in the |00> state, which is exactly the initial state.

After applying a Hadamard gate to both qubits, the state vector becomes

$$\begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}$$

Each element in this vector has an amplitude of `1/2` or `0.5`. In the corresponding probability vector, each element equals `1/4` or `0.25`, which is the square of `1/2`.

Therefore, the probability vector after applying the Hadamard gates is written as

$$\begin{bmatrix} \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \end{bmatrix}$$

TIP the state vector shows *amplitudes* while the probability vector shows *probabilities*. For real numbers, the probability is the square of the amplitude.

This is also what can be seen from the probability infogate in Figure 10.12.

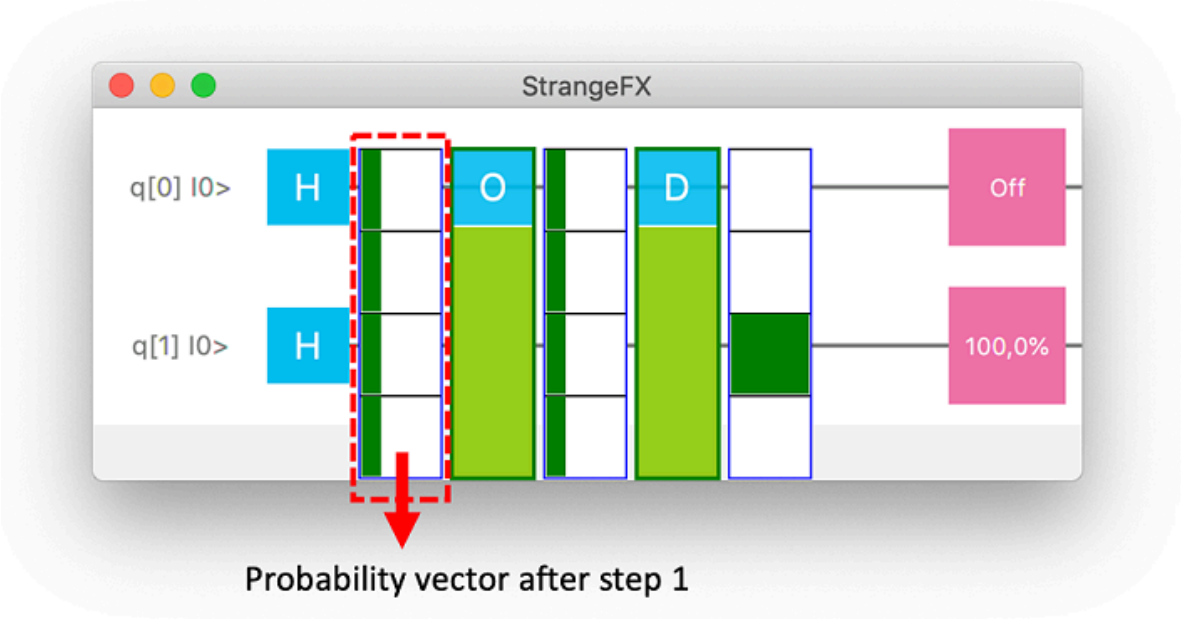

Figure 10.12 After step 1, all probabilities are equal

10.4.4 Quantum Oracle

The main requirement for the classical variant of Grover's search algorithm is that we are given a function that for a single specific value returns `1` and for all other values returns `0`.

You learned that Grover's search algorithm considers this function as a black box, and it does not have any knowledge about the internals of this function.

However, that is a classical function, and if we want to really leverage the quantum algorithm, we need a quantum oracle that is linked to this classical function.

> **SIDEBAR** **Remember the quantum oracle from Deutsch algorithm**
>
> This is very similar to what you learned in the previous chapter about Deutsch algorithm. Remember that in the case of Deutch algorithm, we were dealing with a function that was either constant or balanced. You created an Oracle that operated on two qubits, where the first qubit was left intact, and the second qubit was transfered via an operation that depends on the function evaluation.

Figure 10.13 schematically shows the difference between the classical version of Grover's search algorithm and the quantum version: in the classical version, the black box is realised by a classical function, while in the quantum version, the black box is realised by a quantum oracle.

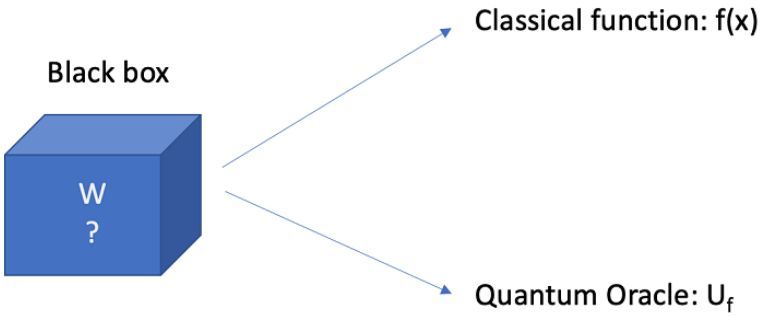

Figure 10.13 Black box in a classical versus quantum context

Obviously, there is a relation between the classical function representing the black box and the quantum oracle representing the same black box.

The quantum oracle related to a classical function f(x) does the following: for any value of |x> that is not the specific value `w`, f(x) is `0`, hence the original value |x> will be returned. In case the value `w` is passed through the Oracle, the result will be -|x> .

Let's give a concrete example. Suppose we have a list with 4 elements. In that case, we require 2 qubits (as $2^2 = 4$). The element that we hope to find has index 2. Hence, the function that we would pass to a classical algorithm is the following:

$$
\begin{aligned}
(1) \quad f(0) &= 0 \\
(2) \quad f(1) &= 0 \\
(3) \quad f(2) &= 1 \\
(4) \quad f(3) &= 0
\end{aligned}
$$

This corresponds to the oracle defined by the following matrix:

$$
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & -1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
$$

Multiplying this matrix and the state vector that was obtained after applying the Hadamard gates results in the following:

$$
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & -1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{bmatrix}
\frac{1}{2} \\
\frac{1}{2} \\
\frac{1}{2} \\
\frac{1}{2}
\end{bmatrix}
=
\begin{bmatrix}
\frac{1}{2} \\
\frac{1}{2} \\
-\frac{1}{2} \\
\frac{1}{2}
\end{bmatrix}
$$

Note that the third element in this vector, corresponding to the state |10> (which is the value `2`) is now negative. If we look at the probability vector, though, all elements in this vector are still equal to `1/4`, which is shown in Figure 10.14

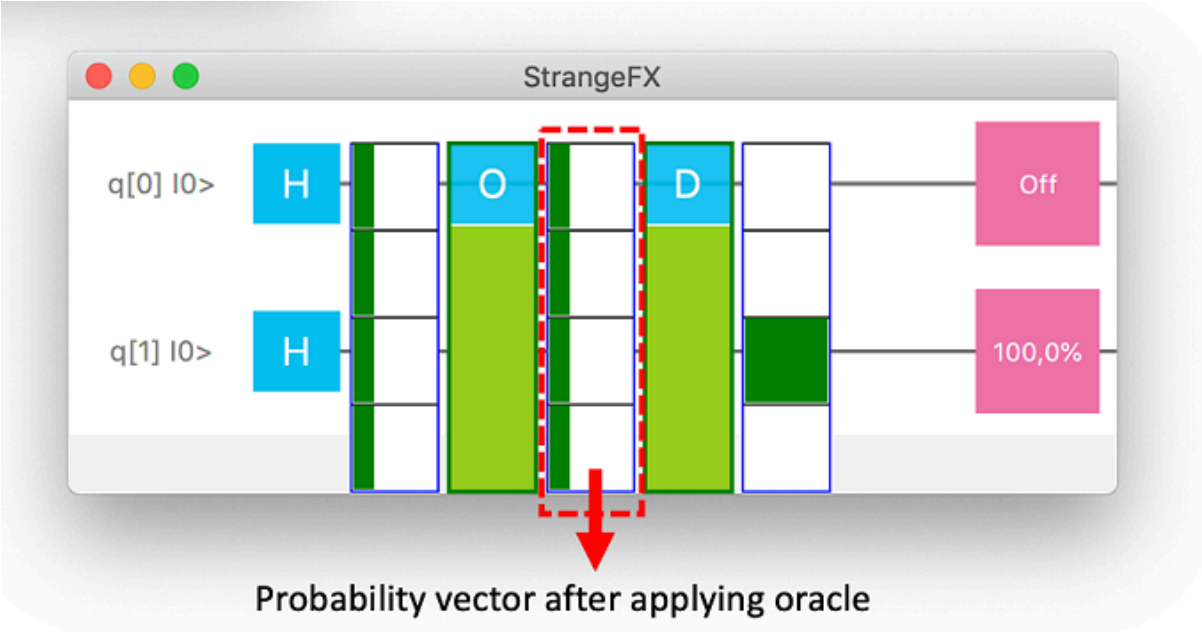

Figure 10.14 After applying the quantum oracle, all steps still have an equal probability

The quantum oracle does not change the probabilities. If we would measure the system now, we would have equal chances of measuring any value. However, the quantum circuit itself works with the amplitudes, which are modified. In the next step, we will take advantage of this.

This situation shows a very important difference between the state vector, which contains amplitudes, and the probability vector, which contains probabilities. We typically talk about probabilities, but in this case, let's have a deeper look at the amplitudes.

Figure 10.15 shows the state vector after applying the quantum oracle, and we show the 4 different amplitudes as horizontal lines. A line to the right indicates a positive amplitude, a line to the right indicates a negative amplitude.

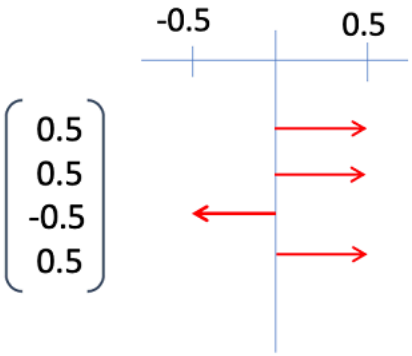

Figure 10.15 Visualisation of the state vector after applying the quantum oracle

> **NOTE** We stressed in the previous chapter that the creation of the Oracle is out of scope for the algorithm. It is assumed that we somehow are given an Oracle, and that we are asked to find out something about it. This is also the case for Grover's search: we are given an Oracle, and our goal is to find the value `w` by doing as few evaluations as possible.

This corresponds to the classical problem: we are given a Function, and we are asked for which value the function evaluation is `1`. Where in the classical case, on average `n/2` function evaluations are needed to get the answer, in the quantum case `sqrt(n)` oracle evaluations are needed to get the answer.

10.4.5 Grover Diffusion Operator: Increasing the probability

The next step of Grover's search algorithm applies a Diffusion Operator to the state of the system. This Diffusion Operator can be constructed by applying quantum gates or by creating its matrix. The interested reader can look at the implementation in the code either in Strange or in the grover sample.

The Diffusion Operator does a so-called "inversion about the mean", which means the following:

- all values in the state vector are summed
- the average is calculated
- all values are replaced with the value that would be obtained my mirroring the value about the mean

Let's calculate what that does with the state vector we currently have. The 4 elements in the state vector are `1/2`, `1/2`, `-1/2`, `1/2`. The sum of those elements is thus `1`:

$$\frac{1}{2} + \frac{1}{2} - \frac{1}{2} + \frac{1}{2} = 1$$

Hence, the average is *1/4*.

We now need to "mirror" the elements (which are either `1/2` or `-1/2`) around this value of `1/4`.

As shown on figure 10.16, mirroring 1/2 results in 0.

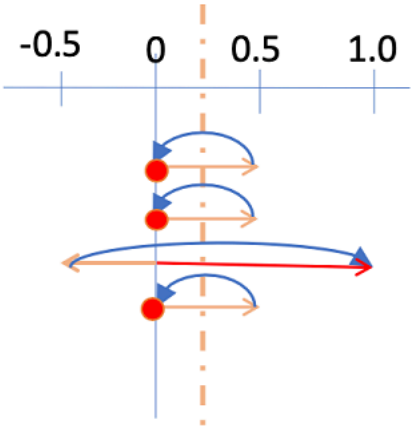

Figure 10.16 Visualisation of the state vector after applying the diffusion operator

Interestingly, mirroring -1/2 results in 1.

The real power of Grover's search algorithm comes from the combination of the quantum oracle, which flips the sign of the amplitude of the target value, and the diffusion operator, which inverts all amplitudes over their mean, thereby amplifying the negative amplitude into the largest element.

In this particular case, with 2 qubits only, a single step is sufficient to find the correct answer to the original problem. We were provided with an Oracle, and a single evaluation of that Oracle was enough to determine that the element at index 2 was given the correct answer to the original function.

In case there are more than 2 qubits, the probability for measuring the correct answer is larger than the probability of measuring any of the other options, but it is not 100%. In that case, the quantum oracle and the diffusion operator have to be applied multiple times. It can be proven mathematically that the number of steps that provides the optimal result is the value closest to $\sqrt{N}\ \Pi / 4$.

10.5 Conclusion

Grover's search algorithm is one of the most popular quantum algorithms. In this chapter, you learned that while the algorithm itself is not related to searching a database, it can be leveraged in applications that require searching through unstructured lists.

As is often the case with quantum algorithms, Grover's search algorithm increases the probability of measuring the correct response, and lowers the probability of measuring the wrong response.

Without any upfront knowledge, all possible answers have the same probability. After applying 1 step of the algorithm, the correct answer will already have a higher probability then the other possible outcomes. After applying the optimal amount of steps (the number closest to sqrt(N) *

Pi/4), the correct answer will have the highest probability.

In this chapter

- you learned about the problem area Grover's search algorithm addresses
- you wrote classic code that searches for an element in an unstructured list using a Java Function
- you wrote classic code that performs a similar search, this time using Grover's search algorithm under the hood
- you learned step by step how Grover's search algorithm is working
- you learned that quantum algorithms can work by increasing the probability that the value measured at the end of the process is indeed the value you intend to find.

11 Shor's Algorithm

This chapter covers

- An explanation of Shor's Algorithm and why it is relevant
- A classical approach to solve integer factorization
- The quantum approach to solve integer factorization, implemented in a
- classical way The same approach, implemented using quantum computing techniques.

In this chapter, we will discuss one of the most famous quantum algorithms that is currently known. More important than the results from this algorithm is the approach that is taken to come to this algorithm. The mental model shown in Figure 11.1 outlines the chapter.

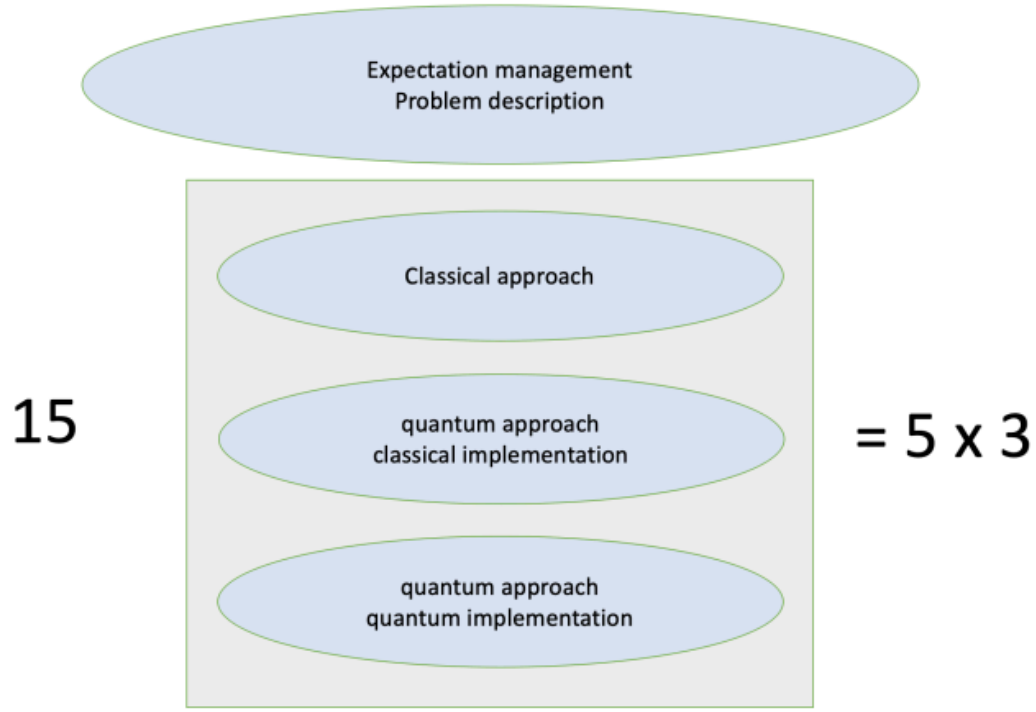

Figure 11.1 Mental model for this chapter. We will gradually develop a Java application that leverages quantum computing to factor 15 in 5 and 3.

11.1 A quick sample

Before we explain Shor's algorithm and discuss it, let's have a look at some real Java code that invokes Shor's algorithm on a quantum computer simulator.

The sample in ch11/quantumfactor has everything you need. We will discuss the sample later, for now it is important to know that the sample uses Strange to simulate the behavior of a real quantum computer. If you run it, you will see the following output:

```
Factored 15 in 3 and 5
```

That's it. The main application in the last chapter of this book factors 15 in 3 and 5. While that is something you could easily do with a classical computer as well, or even by head, it is a great example on where quantum computers can make a real difference, and why. As we said before, the results of the code in this chapter are not impressive. But there are two important reasons why we put so much emphasis on this algorithm:

- once there are quantum computers with enough high-quality qubits, the results of Shor's algorithm will be very impressive, and actually very threatening to many current encryption techniques
- the approach taken by Shor to implement this problem on a quantum computer might help others to find similar approaches for different problems.

IMPORTANT You don't need a quantum computer, or a quantum computer simulator to find out that 15 = 3 x 5. However, by understanding how a quantum computer can do this, you can work on similar problems that will benefit from quantum advantages once quantum computers are really powerful. For example, optimization algorithms, and some machine learning algorithms might leverage the same quantum techniques as the ones that we describe in this chapter.

11.2 The marketing hype

In talks about quantum computing, the question is often raised what areas are expected to change considerably because of quantum computing. One of the most common answers is encryption. Often, when asking people what they know about quantum computing, the answer is "It will break encryption". While that is not necesarily a wrong answer, it should be placed in the right context. Clearly, it is an answer that sparks discussions, and therefore often increases interest in quantum computing. However, there are a few caveats to this:

- There are many other impressive targets for quantum computing, apart from breaking encryption
- It is expected to take a number of years before quantum computers are powerful enough to break the most common encryption techniques used today.

The second caveat should again be taken with a grain of salt. Indeed, current quantum computers are by no means capable of decrypting messages sent with a 2048 bit RSA key. However, those encrypted messages can be stored on disk today, and once quantum computers are powerful enough, they can be decrypted. Maybe in 10 years from now, some secrets from today will be unveiled.

The basic idea behind the statement that "quantum computing will break current encryption" is that many encryption techniques used today rely on the assumption that it is extremely hard to factor a large integer. Until today, the largest number that has been factored has 829 bits and the process required about 2700 core-years using Intel Xeon Gold 6130 CPU's. Since the currently best performing algorithms still are in the sub-exponential time complexity class, adding a single bit makes it almost exponentially harder for classical computers to factor the target. It is therefore assumed that e.g. a 2048 bit key is very secure.

However, in 1994, Peter Shor wrote a paper in which he explained how a quantum computer would be able to factor integers in a much faster, and especially more scalable way than the best possible classical computers.

The algorithm, which was coined "Shor's algorithm" after its inventor, has been implemented on quantum computer simulators (e.g. Strange) and also on real quantum computers. The results

might look unimpressive, but since the algorithm has polynomial time complexity, the real benefits will only become visible once the problem is harder, and once there are more stable qubits in a quantum computer.

> **NOTE** we discussed time complexity in Chapter 1. It might be good to refresh that information, as we will talk about polynomial and exponential time complexity throughout this chapter.

11.3 Classic factorization versus quantum factorization

Many encryption algorithms rely on the assumption that it is very hard for computers to factor large numbers. But is it really that hard?

Since breaking encryption is a rewarding exercise, many research has been done in order to find the best possible algorithm to factor large numbers. Currently, the best known algorithm for doing this is of the sub-exponential time complexity class.

Shor's algorithm solves the problem in polynomial time. The absolute numbers depend on many factors, but the general idea should be clear from Figure 11.2 where we compare the sub-exponential curve and the polynomial curve.

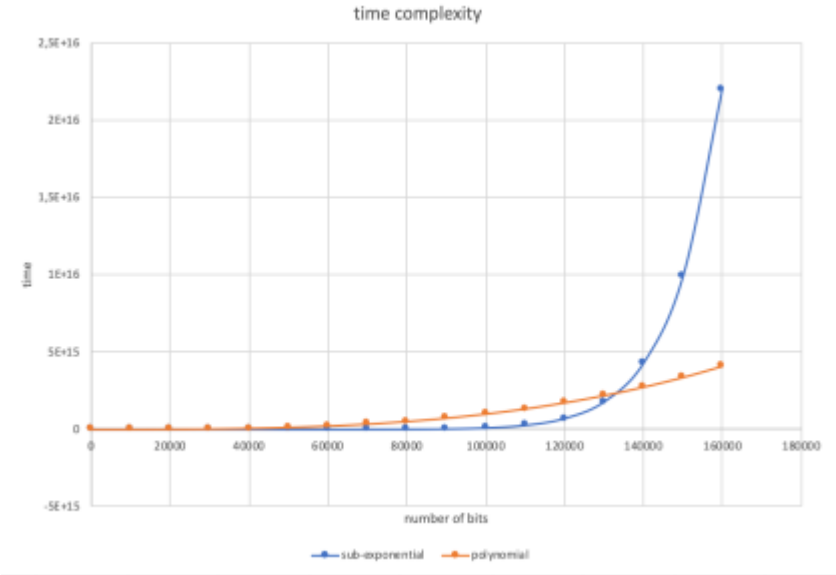

Figure 11.2 Required computing time for sub-exponential versus polynomial algorithm as a function of number of bits.

The values on the axes are not relevant, and they are just an indication. The main observations of this comparison are:

- for small numbers of bits, the sub-exponential approach (e.g. the classic algorithm) is working very well, and maybe even better than the polynomial approach (quantum

algorithm)

- once the number of bits becomes large enough, adding a single bit makes the problem much harder using the classic algorithm compared to using the quantum algorithm

In conclusion, Shor's algorithm really shows its power when we have to factor large numbers, which is typically the case when dealing with encryption. This requires a larger number of qubits than available on todays quantum computers, so we don't see the real benefit *yet*.

11.4 A multi-disciplinary problem

There are a number of views to Shor's algorithm. Obviously, it leverages properties specific to quantum physics, otherwise it wouldn't benefit from quantum computing. The algorithm itself is based on linear algebra, and mathematical equations. Finally, in order to be of practical use, it should be written in a programming language, and integrated with other software components.

This multi-disciplinary approach is shown in Figure 11.3

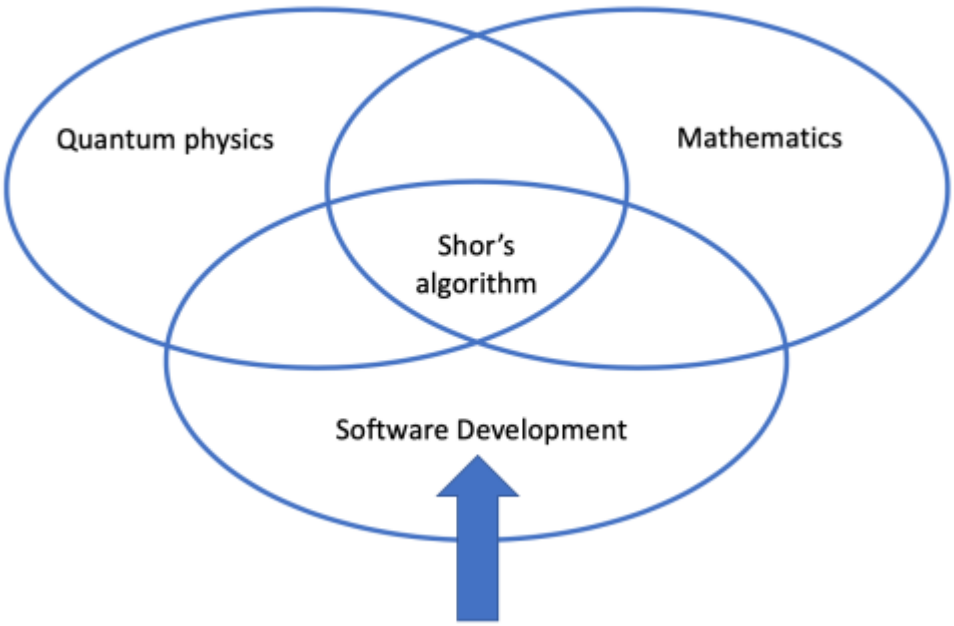

Figure 11.3 Different expertise fields are required for Shor's algorithm. In this chapter, we focus on the field of software development, but that does not mean the other fields are less important!

The success of Shor's algorithm is mainly related to the promising performance. The ultimate performance is a combination of the properties in the three fields shown in the 11.3. Lots of research has been done in order to find the best approach, taking into account a number of characteristics:

- how many qubits are required?

- how many elementary gates are required?
- what is the depth of the algorithm (related to how many gate operations can be executed in parallel)?

The answers to these questions are related to the number of bits in the integer that we need to factor. It can be proven that both the number of qubits as well as the amount of gates required are polynomial with the number of bits.

In this chapter, we will focus on the software development parts of Shor's algorithm. The mathematical and physical background of the algorithm is rather complex. The interested reader can find more information in the following paper from Stephane Beauregard:

_Circuit for Shor's algorithm using 2n+3 qubits _

which can be found at

https://arxiv.org/abs/quant-ph/0205095

11.5 Problem description

The core concept of many encription techniques is the concept of prime numbers. An integer is a prime number if it can only be divived by 1 or itself. For example, 7 is a prime number, but 6 is not—as 6 can be divided by 1, 2, 3 and 6.

Suppose that you know 2 prime numbers, e.g. 7 and 11. Calculating the product of those 2 prime numbers is easy:

```
7 x 11 = 77
```

The reverse operation is more complex: given a number that is the product of 2 unknown prime numbers, come up with those 2 prime numbers. In the simple case above, it is still easy:

```
77 = 7 x 11
```

We say that 77 can be factored in 7 and 11. You don't even need a calculator for this.

However, once the prime numbers become bigger, the problem becomes more complex. A slightly more difficult number is 64507. Can you quickly say if this number can be factored in two prime numbers? That is already harder to answer than factoring 77. The opposite question would be much easier:

What is the product of 251 and 257?

```
251 x 257 = 64507
```

This is one of the basic rules for encryption: it is easy to go from A to B (or from factors to

number), but very hard to go from B to A (or from number to factors).

Hence, the core problem we try to solve in this chapter is the following:

```
Given an Integer N, find 2 integers a and b so that N = a * b with both
a and b > 1
```

We first solve this problem in a purely classical way. This corresponds to the first approach shown in the mental model, as highlighted in Figure 11.4

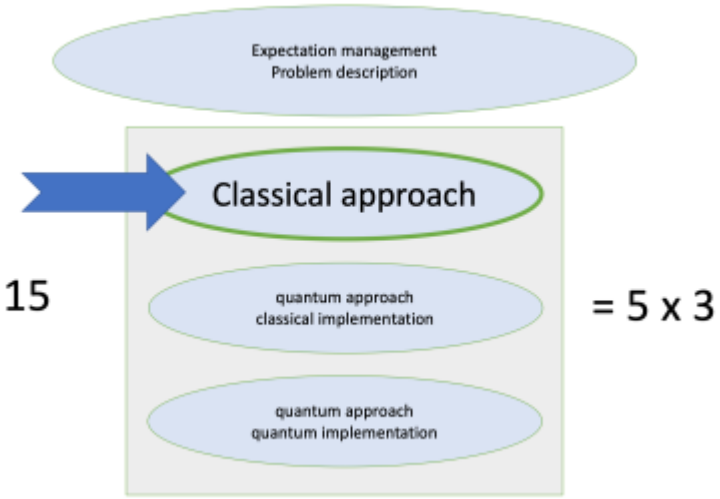

Figure 11.4 Referring to the mental model, we are now going to explain the classic approach.

In general, the classical way for doing integer factorization is straightforward, and it is shown in Figure 11.5.

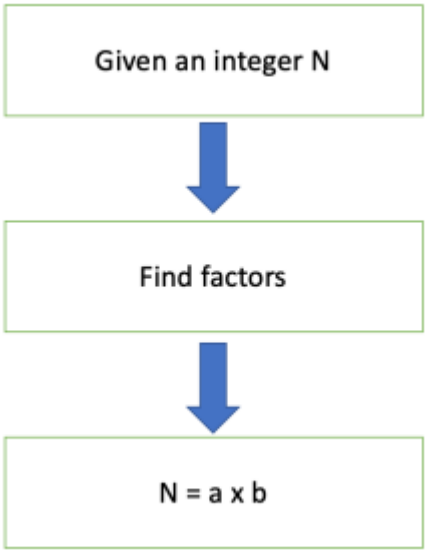

Figure 11.5 Classic flow for factoring integers. A classic algorithm will focus on finding the factors for a given number, and return those factors.

The approach will immediately try to find the factors for the given integer and return them. While this may sound very obvious, we will show later that the quantum approach takes a different path.

A naive approach for doing this is provided in the sample `ch11/classicfactor`

The main method is shown in 11.1 and looks as follows:

Listing 11.1 source code for classic main method

```
public static void main (String[] args) {
    int target = (int)(10000 * Math.random());                          ❶
    int f = factor (target);                                            ❷
    System.out.println("Factored "+target+" in "+f+ " and "+target/f);   ❸
}
```

❶ Pick a random integer between 0 and 10000

❷ Invoke the `factor` method to obtain one factor of the picked integer

❸ Print the obtained factor, and the corresponding one that when mulitplied return the original picked integer.

The main method delegates the work to the `factor` method, which looks as follows:

```
public static int factor (int N) {
    int i = 1;
    int max = (int) Math.sqrt(N);                                       ❶
    while (i++ < max ) {                                                ❷
```

```
        if (N%i == 0) return i;                                    ❸
    }
    return N;                                                      ❹
}
```

❶ We don’t need to check potential factors larger than the square root of the original number, as those would correspond with factors smaller than the square root of the original number

❷ Try every integer, starting from 2 that is smaller than the square root of the original number

❸ In case the candidate divides the original integer (in which case the modulus of the division is 0), return that candidate

❹ If no number was successful, return the original number

Clearly, this is a very naive approach, and more performant approaches exist.

11.6 The rationale behind Shor’s algorithm

The mathematical details for Shor’s algorithm are beyond the scope of this book. However, the rationale behind those details is very important, as it applies to many potential quantum algorithms. Shor’s algorithm translates the original problem into another problem: finding the *periodicity* of a function.

A function is called a periodic function if its evaluations are repeated at regular intervals. The length of this interval is called the periodicity of the function.

To give you a more tangible idea of what a periodic function based on modular exponentiation looks like, consider the case where `a = 7` and `N = 15`.

In that case, Figure 11.6 shows the value for y = 7^x \mod 15y = 7^x \mod 15.

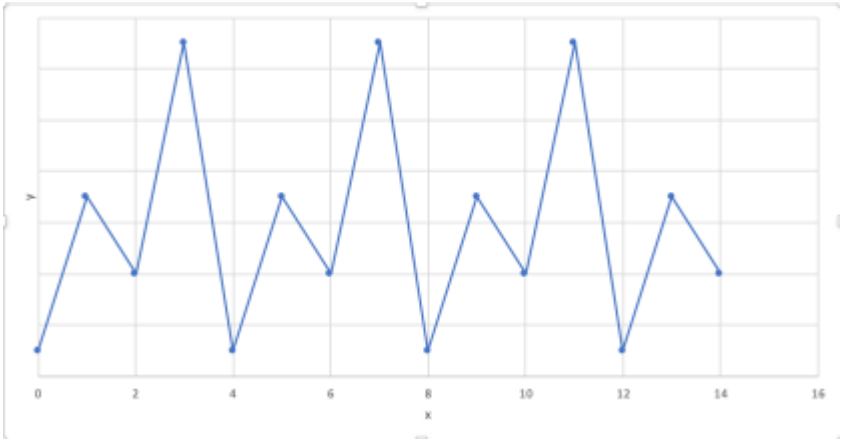

Figure 11.6 Example of a periodic function. The periodicity of this function is 4: the same pattern in the y values comes back after every 4 evaluations. For example, the peaks in the function occur at x values of 3, 7 and 11.

There is a pattern in this function, that repeats itself. Whenever we increase `x` by `4`, the value of the function is the same as the original value. For example

y(0) = 1 y(1) = 7 y(2) = 4 y(3) = 13 y(4) = 7 y(5) = 7 y(6) = 4 y(7) = 13 y(8) = 1

From these observations, it turns out that the periodicity of this function is `4`

Finding the periodicity of a function is a problem that can be solved by a quantum computer in polynomial time. After the quantum computing part, the result needs to be translated to the original problem again. This flow is shown in Figure 11.7.

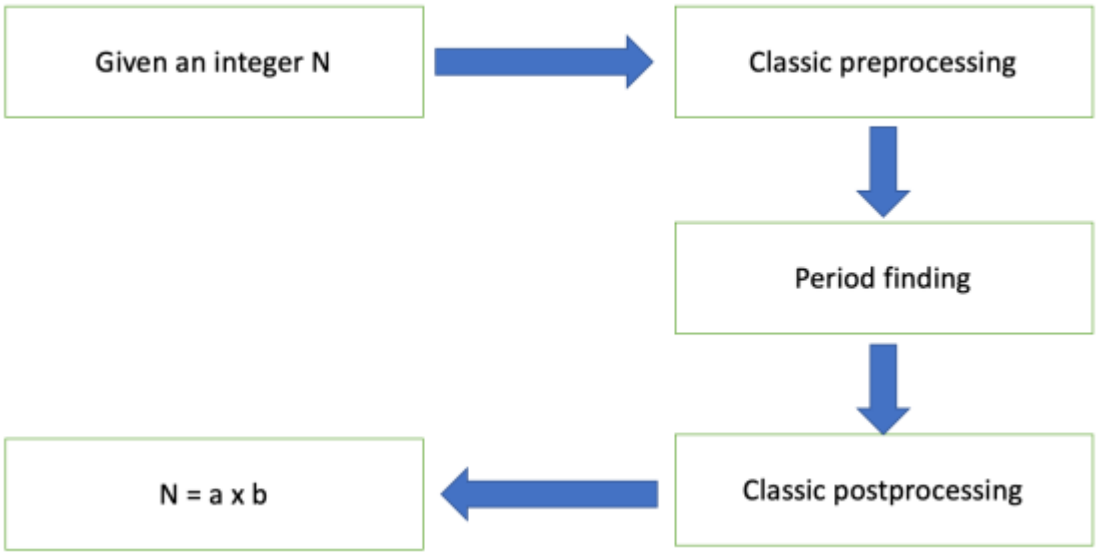

Figure 11.7 Solving a different problem. Instead of directly finding the factors for the number N, we will translate the original problem to a different problem, solve that, and translate the result back to the original question which can then be answered.

IMPORTANT Quantum computers can provide a huge speedup for some algorithms, but not for all algorithms. Therefore, the key in creating quantum applications is often in finding a way to translate the original problem into a problem that can be solved easily by a quantum computer (e.g. in polynomial time instead of exponential time), and then transform it back to the original domain.

The original problem is to find two integers that, when multiplied together, yield the integer `N` that we want to factor.

The problem that we will actually solve using a quantum computer looks very different, and is formulated as follows:

```
Given an integer A, and an integer N, find the periodicity of the function
latexmath:[a^x \mod N]
```

While that problem looks very different from the original problem, it can be proven mathematically that they are related. Once we find the periodicity of this function, we can find the factors for `N` easily.

We are not going to provide the mathematical proof, but we will show the relation by looking at some Java code.

The flow that we will follow in this code is illustrated in Figure 11.8.

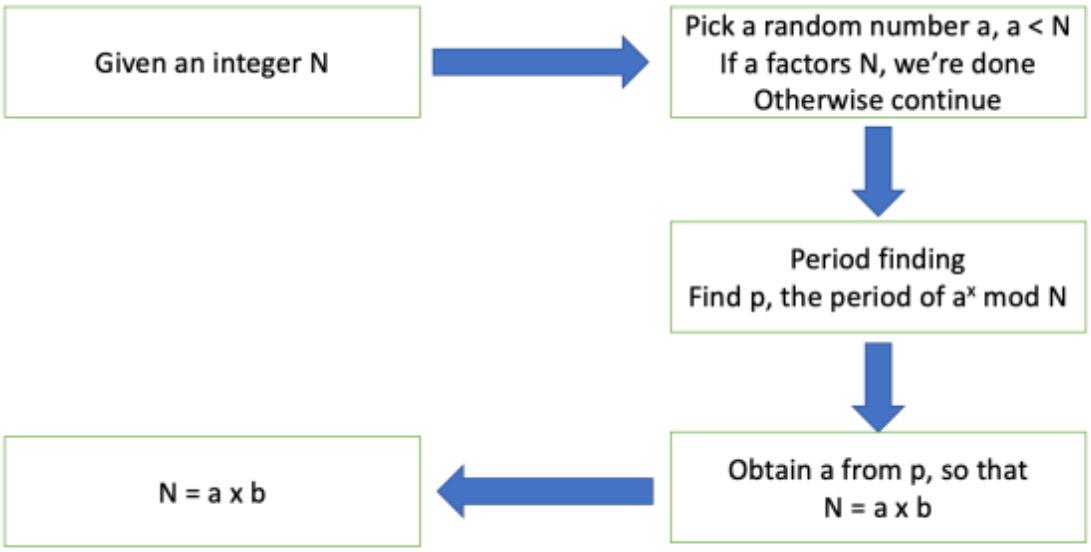

Figure 11.8 Detailed flow for both classic and quantum implementation. The pre-processing and post-processing, which transforms the original problem into the problem of period finding, is similar for the classic and the quantum implementation. The period finding itself can be implemented in a classic way and in a quantum way.

In the next section, we will explain how the critical part, finding the periodicity of the modular exponentiation, is achieved using a quantum algorithm. In this section, we will write out the complete algorithm using classical computing. Things brings us to the second approach in our mental model, as shown in Figure 11.9

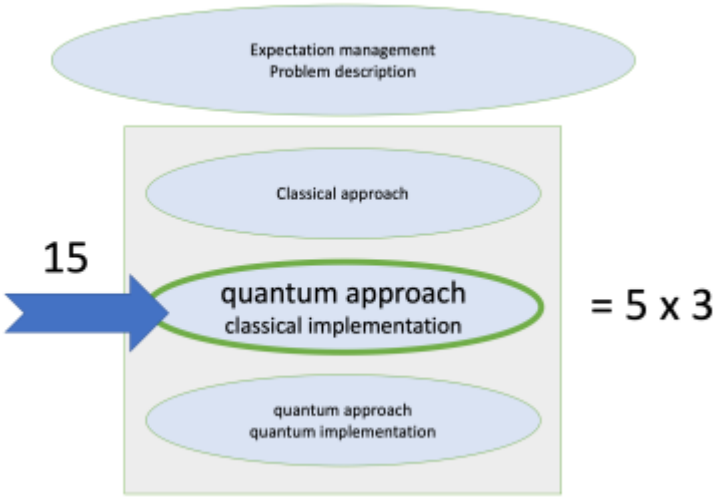

Figure 11.9 Mental model, classic implementation of the quantum approach. In this approach, we don't use direct factorization techniques, as we did in the classic approach. We use the technique of period finding, which is part of the quantum approach, but we will first develop that in a classic way.

The code sample in the `ch11/semiclassicfactor` directory contains a classical implementation of Shor's algorithm. Before looking into the code, let's run the example by running

```
mvn compile javafx:run
```

The result will be something like this:

```
We need to factor 9500
Pick a random number a, a < N: 5841
calculate gcd(a, N):1
period of f = 150
Factored 9500 in 2 and 4750
```

This means that the algorithm discovered that 9500 could be written as the product of 2 and 4750.

Let's have a look at the main method for this sample.

```
public static void main (String[] args) {
    int target = (int)(10000 * Math.random());
    int f = factor (target);
    System.out.println("Factored "+target+" in "+f+ " and "+target/f);
}
```

This code is straightforward and it is exactly the same as the main method shown in 11.1. A random integer between 0 and 10000 is generated, and then the `factor` method is called to find a divider of this integer. Finally, the divider and the other divider are printed.

This method delegates the bulk of the work to the `factor` method, so let's have a look at that one, shown in 11.2. It is worth keeping an eye on 11.8 while looking at the code.

Listing 11.2 factor method for the classical implementation of the quantum approach.

```
public static int factor (int N) {
    // PREPROCESSING                                                    ❶
    System.out.println("We need to factor "+N);
    int a = 1+ (int)((N-1) * Math.random());                            ❷
    System.out.println("Pick a random number a, a < N: "+a);
    int gcdan = gcd(N,a);                                               ❸
    System.out.println("calculate gcd(a, N):"+ gcdan);
    if (gcdan != 1) return gcdan;                                       ❹

    // PERIOD FINDING
    int p = findPeriod (a, N);                                          ❺

    // POSTPROCESSING
    System.out.println("period of f = "+p);
     if (p%2 == 1) {                                                    ❻
        System.out.println("bummer, odd period, restart.");
        return factor(N);
    }
    int md = (int)(Math.pow(a, p/2) +1);                                ❼
    int m2 = md%N;
    if (m2 == 0) {
        System.out.println("bummer, m^p/2 + 1 = 0 mod N, restart");
        return factor(N);
    }
    int f2 = (int)Math.pow(a, p/2) -1;
    return gcd(N, f2);

}
```

❶ Here, the preprocessing part begins.

❷ Pick a random number a between 1 and `N`

❸ Calculate the greatest common denominator (GCD) between `a` and `N`

❹ In case this GCD is not `1`, we are done, since that means the GCD is a factor of `N`

❺ Find the periodicity of the modular exponentiatio function. This is the bulk of the work, and it will be detailed in the next listing.

❻ If the period turns out to be an odd number, we can't use it and have to repeat the process

❼ Perform some minor mathematical operations on the period to obtain a factor of `N`.

The `factor` method will call the `findPeriod` method to obtain the periodicity of the function a^x mod N. This is the function that can be executed *fast* on a quantum computer.

> **NOTE** when we say the function can be executed fast, we actually mean this is done in polynomial time.

We can of course achieve this on a classical computer as well, but for large numbers this will

take a very long time. However, for small numbers (e.g. smaller than 10,000 like we have in our code), it works very well on classical computers.

A naive approach for a classical java function that does this is shown below:

Listing 11.3 Classic implementation for finding the period of a modular exponential function

```
public static int findPeriod(int a, int N) {
    int r = 1;
    long mp = (long) (Math.pow(a,r)) % N;
    BigInteger bn = BigInteger.valueOf(N);
    while (mp != 1) {
        r++;
        BigInteger bi = BigInteger.valueOf(a);
        BigInteger mpd = bi.pow(r);
        BigInteger mpb = mpd.mod(bn);
        mp = mpb.longValue();
    }
    return r;
}
```

This function (finding the periodicity of a modular exponentiation) can also be implemented on a quantum computer, using a quantum algorithm that leverages quantum properties. It is the core of Shor's algorithm, and we will discuss it in the next section.

11.7 The quantum-based implementation.

Let's go back to the original sample in this chapter, which is in the `ch11/quantumfactor` directory. That sample contains a `main` method which is very simple:

```
public static void main (String[] args) {
    int target = 15;
    int f = Classic.qfactor (target);
    System.out.println("QFactored "+target+" in "+f+ " and "+target/f);
}
```

The only real work in this method is the invocation of the `Classic.qfactor` API. The `qfactor` method in Strange returns a factor of the supplied integer. This single factor allows us to calculate the other factor as well. For example, if we ask a factor of the integer `15` and we are returned the value `3` we know that `5` is another factor, as it is `15/3`.

The `qfactor` method uses classical computing for the pre-processing and the post-processing steps, very similar to what we did in the previous section. But the findPeriod method is implemented very differently in this approach.

In our mental model, we are now about to discuss the third approach, as shown in Figure 11.10

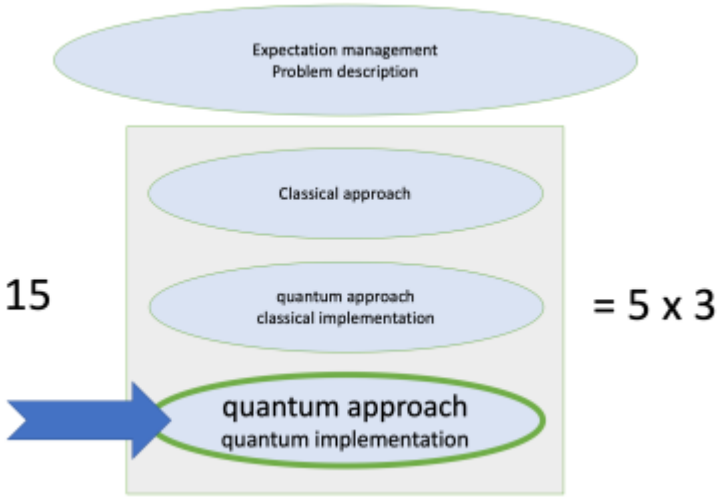

Figure 11.10 Mental model, quantum implementation of the quantum approach. In this approach, the pre-processing and post-processing are done in a classic way, but the period finding is done using a quantum algorithm.

The implementation of `Classic.qfactor` can be found in the source code of Strange, but it is extremely similar to the code snippet shown above.

The only difference is the implementation of the `findPeriod` method. In the following, we will explain how this method is implemented in Strange, leveraging a quantum algorithm.

From the code snippet above, the challenge we are facing is to find the periodicity of the following function:

f = a^x \mod N f = a^x \mod N

Instead of calling the classical function `findPeriod` shown in 11.3, we will now use the quantum implementation, which can be found in `Classic.findPeriod (int a, int mod)`. Note that the signature of this method is exactly the same as the one containing the classical implementation.

The general trick that we will apply again here is the following: we try to evaluate the function for all integers between 0 and N at once by creating a superposition state, and then manipulate the system so that when we measure the result, a useful value is obtained.

We split that challenge into 2 issues:

- create a periodic function
- calculate the periodicity based on a measurement

Schematically, the flow for this is shown in Figure 11.11

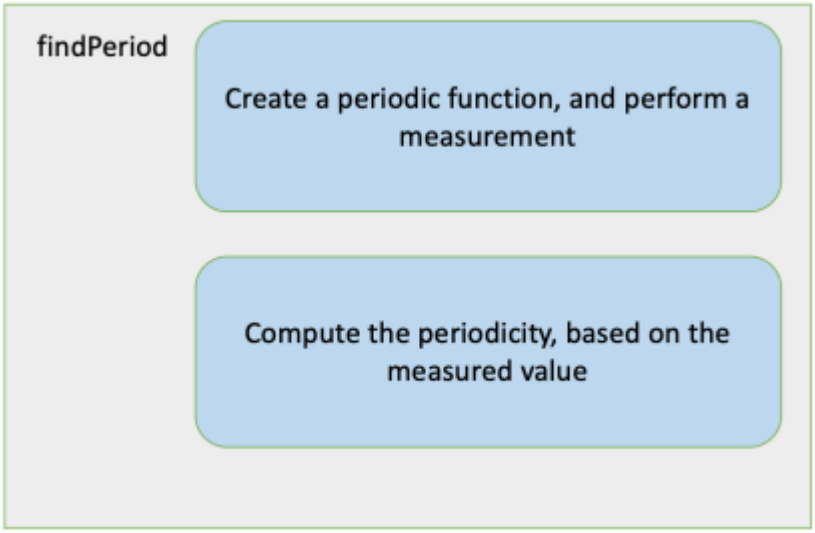

Figure 11.11 High-level flow for period finding in Shor's algorithm

The first part (creating the periodic function) will be done with quantum code, while the second part is done using classic code only.

The implementation of the `findPeriod` function illustrates this approach:

```
public static int findPeriod(int a, int mod) {
    int p = 0;
    while (p == 0) {
        p = measurePeriod(a, mod);                    ❶
    }
    int period = Computations.fraction(p, mod);       ❷
    return period;
}
```

❶ the periodic function is prepared, and a measurement is done

❷ the measured value is used to compute the periodicity

Note that the first part may not return a useful result. In that case, the `measurePeriod` function will return `0` and the function will be invoked again.

11.7.1 Creating the periodic function

As we said before, we won't go into the mathematical details that prove the correctness of Shor's algorithm. In this section, we'll give an intuitive approach that explains why the approach works.

This part of the algorithm contains different steps as well. We will highlight those steps by showing the flow of the algorithm, as well as the quantum circuit that realizes this step.

The flow for creating the perdiodic function is shown in Figure 11.12.

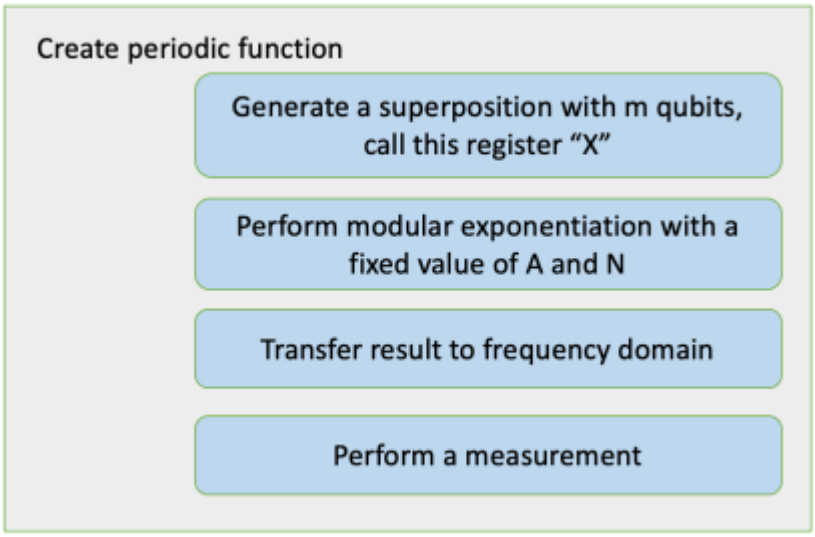

Figure 11.12 Flow for creating a perdiod function

Since this part is done using a quantum algorithm, there is a quantum circuit involved. This circuit is shown in Figure 11.13.

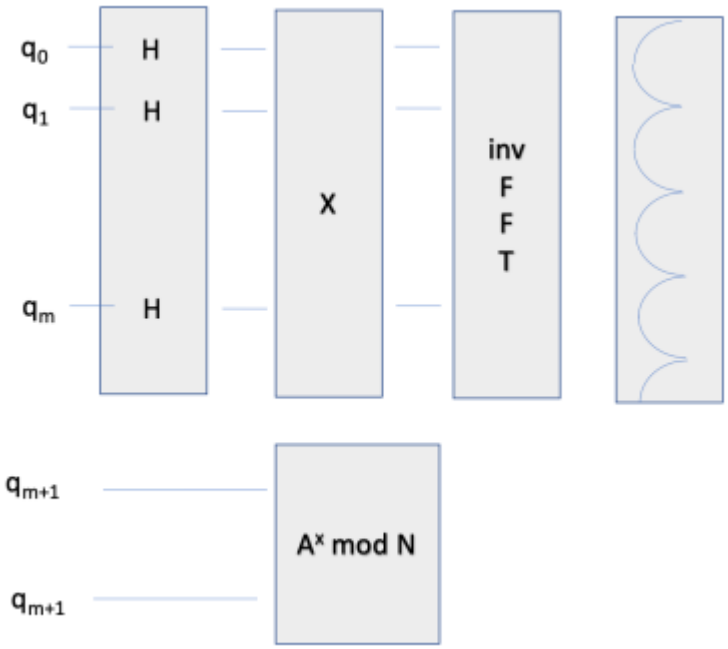

Figure 11.13 Quantum circuit that creates a periodic function

The qubits involved in this schema are divided in two *registers* where a register is just a set of qubits that together have a conceptual meaning. The top register is called the input register, and the bottom register is called the *ancilla register*.

CREATING A SUPERPOSITION

We first apply a Hadamard Gate to each qubit in the input register. This brings the input register in a superposition, and the upcoming calculations can thus be done with a combination of all possible values that the input register can contain.

This is something that can’t be done on a classical computer, and it is an intuitive indication why a quantum computer can handle this problem much faster. However, keep in mind that even though we can create a superposition of all possible values, we can only make a single measurement on the system. Hence, we need to apply some smart steps so that the single measurement we obtain is actually useful.

PERFORM MODULAR EXPONENTIATION

Next, the input register is used to calculate the modular exponentiation a^x \mod Na^x \mod N and that result is computed and stored in the ancilla register. As a consequence of this operation, the input register is now a periodic function with periodicity *r*, where *r* is the periodicity of [a^x \mod Na^x \mod N.

This is interesting, as we now have a periodic function. However, we can only make a single measurement, and whatever we would measure, we would not get much information about the periodicity of the function.

APPLYING AN INVERSE QUANTUM FOURIER TRANSFORM

By applying an inverse quantum Fourier transform, the periodic function is transformed into a function with peaks at some specific "frequencies". It can be proven that the probability vector has exactly *r* peaks where the first peak occurs at the value |0> and the other peaks are evenly spread.

After this, the probability matrix will show a number of peaks, as shown in Figure 11.14.

Figure 11.14 Probability distribution with 8 peaks

11.7.2 Calculate the periodicity

If we could measure the probability vector, we would be able to count how many peaks it has. Unfortunately, we can't do that. We can only measure the qubits, and a single measurement corresponds to a single entry in the probability vector. However, we know that there is a very high chance that the entry we measure is in one of the peaks of the probability vector. Based on that information, it is possible to determine the number of peaks with a high probability.

We use the continued fraction expansion algorithm for doing this. This algorithm takes the measured value as input, along with the maximum value of the answer, and will return the periodicity. This algorithm is implemented in `Computations` and its signature is

```
public static int fraction (double d, int max);
```

In this algorithm, `d` is the measured value divided by the maximum value, so it is a value between `0` and `1` and `max` is the maximum number that can be returned. Based on the knowledge that the measured value is on or near a peak in the probability distribution, the result of this algorithm returns the number of peaks in the probablility distribution.

11.8 Implementation challenges

Shor's algorithm relies heavily on modular exponentiaton. While this may look like just an implementation detail, it is actually a big challenge.

Mathematical operations on a quantum computer are not very trivial to implement, since all gates need to be reversible, as we explained before.

For example, an addition operation in a classical circuit could be implemented as shown in Figure 11.15.

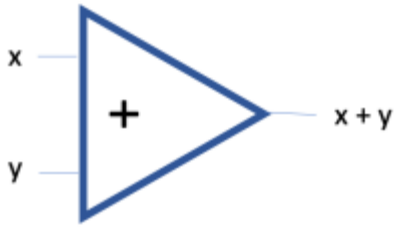

Figure 11.15 Classic addition operation

The input to this gate are two values `x` and `y`, which are probably bits, and the output is a single value `x + y`.

This won't work in the quantum world, as there is no way to go back from `x + y` to `x` and `y`. For example, if `x + y = 1`, we don't know if either `x` was `0` and `y` was `1` or if on the other hand `y` was `0` and `x` was `1`. Therefore, a quantum adder gate is rather implemented as shown in Figure 11.16.

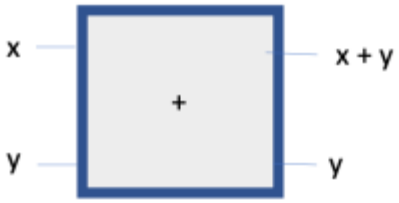

Figure 11.16 Quantum addition operation

The result of this gate keeps the original `y` value, so based on `x + y` and `y` it is possible to obtain the original `x` value. Hence, this gate has an inverse gate that brings back the original state.

The addition gate is the basis for the multiplication gate, which in turn is the basis for the exponential gate. Adding to the complexity is the modular aspect of the arithmetic operations. If we want to create a circuit for modular exponentiation, we need to be able to perform modular multiplication, which means we need to be able to perform modular addition.

While there is no direct advantage in doing basic arithmetic operations on a quantum computer, it is important to realise that those operations are available. For example, as you just learned,

Shor’s algorithm heavily depends on modular exponentiation.

Hence, if you need those operations in your own algorithms, you can leverage the arithmetic operations in the `com.gluonhq.strange.gate` package.

11.9 Summary

- The goal of Shor’s algorithm is to find the factors of an integer.
- Shor’s algorithm shows that it can be beneficial to transform a specific problem into another problem, one that can be solved more easily (faster) by a quantim computer. In particular, Shor’s algorithm transforms the problem of factorization into the problem of finding the periodicity of a periodic function.
- You implemented integer factorisation in a classical way
- You implemented integer factorisation using Shor’s algorihm, with a classic way for calculating the periodicity of a function
- You implemented integer factorisation using Shor’s algorihm, with a quantum approach for calculating the periodicity of a function
- You learned that in order to leverage the benefits of quantum computing, it is often required to transform the original problem into a problem that can be dealt with more efficient by a quantum computer.

Appendix: Installing Strange

A.1 Requirements

Strange is a modular Java library, leveraging the module concepts introduced in Java 11. In order to run applications using Strange, the Java 11 runtime is needed. Developing applications requires the Java 11 SDK to be installed, which also includes the Java 11 runtime. The Java 11 SDK can be downloaded from jdk.java.net/11. Make sure to download the version that matches your platform.

Most Java developers use an integrated development environment (IDE) to create Java applications. The most common IDE's for Java development are Eclipse (https://www.eclipse.org/ide), Apache NetBeans (https://www.eclipse.org/ide) and IntelliJ IDEA (https://www.jetbrains.com/idea). Since Strange is a modular Java library following the same rules and conventions as any other Java library, it can be used out of the box on those IDE's, since they provide support for the Java modular system.

Apart from IDE's, some developers prefer to use command-line tools to create, maintain and execute applications. Those applications typically use a build tool like maven or gradle, and dependencies are declared in specific files, e.g. a pom.xml file for maven or a build.gradle file for gradle.

All IDE's provide support for maven and gradle. We assume the reader is familiar with how his/her IDE supports gradle. This allows us to use command-line driven gradle projects for our examples. Developers have the choice to either run the examples using the command-line approach, or run the examples in their favourite IDE leveraging the IDE-specific gradle integration.

A.2 Obtaining and installing the demo code

The samples and demoes in this book are available in a git repository located at You can get a local copy of the samples by cloning the repository via command-line git commands, e.g.

```
git clone https://github.com/johanvos/quantumjava.git
```

or via the git support offered by your favourite IDE.

Cloning the repository creates a directory called `quantumjava` on your local filesystem. You will notice this directory contains a number of subdirectories that correspond to the chapters of this book. For example, the samples for Chapter 2 are located in the `ch02` directory.

A.3 The HelloStrange program

The `ch02` directory contains the samples used in this chapter. The first sample we will run in the `hellostrange` sample. Like all other samples, this sample can be opened in your favourite IDE. As discussed before, we will use the gradle command line approach in this book. However, if you prefer to run the samples from your IDE, that should work equally wel.

A.3.1 Running the program

All samples contain wrapper scripts that will first check if the correct gradle version is already installed on the system. If this is not the case, the wrapper script will automatically download and install the required version of gradle.

If you are using Linux or MacOS X, the gradle wrapper script is invoked using

```
./gradlew
```

If you are using Windows, the gradle wrapper script should be invoked via

```
gradlew.bat
```

Running the `hellostrange` demo application is very straightforward by supplying the `run` task to gradle. On Linux and MacOS X, this is done by calling

```
./gradlew run
```

and on windows, this is achieved using

```
gradlew.bat run
```

The result of this action depends on whether you already have the required gradle version or not.