To Codri and Celina.

To all the Little Lispers ever.

Soaked in Java: An Introduction to Programming
Text by: Adrian German
Illustrations: Nickolas Boyce

Prelude

What computers can do (and how).
Programming languages.
Java.

Computers can do lots of things.	For today's computers to perform a complex task, we need a precise and complete description of how to do that task in terms of a sequence of simple basic procedures.
This instructing has to be exact and unambiguous.	In life, of course, we never tell each other exactly what we want to say; we never need to, as context, body language, familiarity with the speaker, and so on, enables us to "fill the gaps" and resolve any ambiguities in what is said.
Computers, however, can't yet "catch on" to what is being said, the way a person does. They need to be told in *excruciating* detail exactly what to do.	I can see the emphasis.
Perhaps one day we will have machines that can cope with approximate task descriptions, but in the meantime we have to be very *prissy* about how we tell computers to do things.	A computer program tells a computer, in minute detail, the sequence of steps that are needed to fulfill a task. The act of designing and implementing these programs is called computer programming. In this course, you will learn how to program a computer – that is, how to direct a computer to execute tasks.
Today's computer programs are so sophisticated that it is hard to believe that they are all composed of *extremely primitive* operations.	Only because a program contains a huge number of such operations, and because the computer can execute them at great speed, does the computer user have the illusion of smooth interaction.
To use a computer you do not need to do any programming. You can drive a car without being a mechanic and toast bread without being an electrician.	Many people who use computers every day in their careers never need to do any programming.

Of course, a professional computer scientist or software engineer does a great deal of programming.	Since you are taking this first course in computer science, it may well be your career goal to become such a professional.
Or maybe not—but let's pretend!	OK.
To write a computer game with motion and sound effects or a word processor that supports fancy fonts and pictures is a complex task that requires a team of many highly skilled programmers.	Your first programming efforts will be more mundane. But we'll still be able to program a video game by the end of the semester.
The concepts and skills you learn in this course form an important foundation, and you should not be disappointed if your first programs do not rival the sophisticated software that is familiar to you.	Actually, you will find that there is an immense thrill even in simple programming tasks. It is an amazing experience to see the computer precisely and quickly carry out a task that would take you hours of drudgery, to make small changes in a program that lead to immediate improvements, and to see the computer become an extension of your mental powers.
OK. Why is programming fun?	I don't know.
I think the reason I like it so much is that it gives me a world I can control.	Could be. You a tad insecure?
No. (Snicker). The laws of nature in that world are published, and knowing them, you can make things happen to your liking.	I agree.
There are *few* limits to what you can accomplish if you just think hard enough.	Not quite like the real world.
The real world?	"...the location of non-programmers and activities not related to programming" ([1], [2] p. xvii)

Very good. What is programming?

Programming is a solution to a problem like this:

> "You are given two different length strings that have the characteristic that they both take exactly one hour to burn. However, neither string burns at a constant rate. Some sections of the strings burn very fast; other sections burn very slowly. All you have to work with is a box of matches and the two strings. Describe an algorithm that uses the strings and the matches to calculate when exactly 45 minutes have elapsed."

That's *parallel* programming[1].

I agree. What *is* programming?

[1] Burning a string from both ends would make the string last only $\frac{1}{2}$ hour, wouldn't it?

Programming is a solution to a problem like this:	That's *logic* programming.

> "A farmer lent the mechanic next door a 40-pound weight. Unfortunately, the mechanic dropped the weight and it broke into four pieces. The good news is that, according to the mechanic, it is still possible to use the four pieces to weight any quantity between one and 40 pounds on a balance scale. How much did each of the four pieces weigh? (Note: You can weigh a 4-pound object on a balance by putting a 5-pound weight on one side and a 1-pound weight on the other)."

Quite true.	Is *programming* related to solving a problem like this?
Where's the problem?	Take it easy now. Here it is:

Ah! *Sequential* programming.

> "A captive queen weighing 195 pounds, her son weighing 90 pounds, and her daughter weighing 165 pounds, were trapped in a very high tower. Outside their window was a pulley and rope with a basket fastened on each end. They managed to escape by using the baskets and a 75-pound weight they found in the tower. How did they do it? The problem is anytime the difference in weight between the two baskets is more than 15 pounds, someone might get killed. Describe an algorithm that gets them down safely."

A bit conservative, I agree[2].

Do you have more examples?	Sure. But how about programming in Java?
Very well. What is this: 5	A number. Java calls that an `int`.
What is this: 3 + 5	Java calls that an *expression*.
What's the value of this expression? $$2 - 3 + 5$$	It's different from $$2 - (3 + 5)$$
Very good. What's the value of $$2/3 * 6$$	Shouldn't this work the same? $$6 * 2/3$$
No, and that's the whole point.	Precisely.

[2]This problem also illustrates the notion of a *named procedure*—just like the solution to the previous puzzle (using a *balanced ternary system*) can help bring up the topic of arithmetic with positional number systems. Solving the puzzles reveals even more.

Let's move on.	How do you calculate $$3 + 5 - 2$$
First an 8 gets created.	Where do we store it?
I don't know, it hangs around	So things can be built in stages:

```
int result = 3 + 5;
result = result - 2; // gives us 6
```

What's `result`?	A *name*. The name of a *variable*.
What is a variable?	A location with a name (and a type).
What's this?	That's a cupholder.[3]

```
class Pair {
  int x;
  int y;
}
```

Could be a `Point2D`.	Or a `Fraction`.[4]
They *look* similar.	They just *behave* differently.
How do you create a new cupholder?	You say: `new Pair()`
How do you place the cups in?	Easy. Start by giving it a name:

```
Pair a = new Pair();
a.x = 3;
a.y = 5;
```

Then use the name of the cupholder to place the individual cups.

[3] Did somebody say McDonald's? (I didn't think so.)
[4] Really?

I see...

You could have more than one cupholder.

Indeed.

And you'd be accessing them in the same way.

Like this:

```
Pair a, b;
a = new Pair();
b = new Pair();
a.x = 3;
a.y = 5;
b.x = 1;
b.y = -2;
```

How many *kinds* of variables do we have in Java?

Four: local, instance, `static` variables, and also parameters.

Cups are *instance* variables.

Do you understand this?

$$f(x) = x + 1$$

x is a parameter.

Do you understand this?

$$f(x) = \begin{cases} \frac{x}{2} & \text{if } x \text{ is even} \\ 3x + 1 & \text{otherwise} \end{cases}$$

Ah! `if` statements.

Do you understand this?

$$f(x) = \begin{cases} 1 & \text{if } x = 1 \\ x + f(x-1) & \text{otherwise} \end{cases}$$

This one is a loop ... $f(10)$ is 55.

How do you do this in Java?

Ask Alan Kay[5].

Seriously...

Easy. Turn the page.

[5] "The ability to start with an idea and see it through to a correct and efficient program is one prerequisite for a great software designer. A second is to see the value of other people's good programming ideas. In 1961 Kay worked on the problem of transporting data files and procedures from one Air Force air training installation to another and discovered that some unknown programmer had figured out a clever method of doing the job. The idea was to send the data bundled along with its procedures, so that a program at the new installation could use the procedures directly, even without knowing the format of the data files. The idea that a program could use procedures without knowing how the data was represented struck Kay as a good one. It formed the basis for his later ideas about objects." ([3], p. 41)

Here's `sum` as defined above.

Programming uses longer names.

```
int sum(int x) {
  if (x == 1) return 1;
  else return x + sum(x - 1);
}
```

Not bad...

Show me more.

Here's a `Fraction`

It knows how to `add`.

```
class Fraction {
  private int num, den;
  Fraction(int num, int den) {
    this.num = num;
    this.den = den;
  }
  Fraction add(Fraction other) {
    return new Fraction(this.num * other.den +
                        this.den * other.num,
                        this.den * other.den);
  }
}
```

Wow. You lost me.

I thought so. Let's back up...

To interact with a human user, a computer requires so-called peripheral devices.

The computer transmits information to the user through a *display* screen, *loudspeakers*, and *printers*. The user can enter information and directions to the computer by using a *keyboard* or a pointing device such as a *mouse*.

Some program instructions will cause the computer to place dots on the display screen or printer or to vibrate the speaker.

As these actions happen many times over and at a great speed, the human user will perceive images and sound.

Some program instructions read user input from the keyboard or mouse.

The program analyzes the nature of these inputs and then executes the next appropriate instructions.

On the most basic level, computer instructions are extremely primitive.

Java is a *high-level* programming language. In Java the programmer expresses the idea behind the task that needs to be performed in a language that resembles both natural language (somewhat) and (to a greater extent) mathematics.

Then, a special computer program, called a *compiler* translates the higher-level description into machine instructions (called *bytecode*) for the Java virtual machine.	Compilers are sophisticated programs.
Thanks to them programming languages are independent of a specific computer architecture.	Still, they are human creations, and as such they follow certain conventions. To ease the translation process, those conventions are much stricter than they are for human languages.
When you talk to another person, and you scramble or omit a word or two, your conversation partner will usually still understand what you have to say.	Compilers are less forgiving.
Just as there are many human languages, there are many programming languages.	This provides a useful source of analogy. Let me ask you this: which is the *best* language for describing something? Say: a four-wheeled gas-driven vehicle.
We needn't introduce democracy just at the level of words. We can go down to the level of alphabets.	What, for example, is the best alphabet for English? That is, why stick with our usual 26 letters?
Everything we can do with these, we can do with three symbols – the Morse code, dot, dash, and space.	So we see that we can choose our basic set of elements with a lot of freedom, and all this choice really affects is the efficiency of our language, and hence the sizes of our books; there is no "best" language or alphabet– each is logically universal, and each can model any other. Same with programming languages, and Java is no exception.
Like C (another popular programming language), the Java language arose from the ashes of a failing project.	In the case of Java, the situation was an anticipated market that failed to materialize[6].
The HotJava browser, which was shown to an enthusiastic crowd at the SunWorld exhibition in 1995, had one unique property: It could download programs, called *applets*, from the web and run them.	Applets let web developers provide a variety of animation and interaction and can greatly extend the capabilities of the web page. In 1996 both Netscape and Microsoft supported Java in their browsers. Since then Java has grown at a phenomenal rate.
Programmers have embraced the language because it is simpler than its closest rival, C++. In addition to the programming language itself, Java has a rich library that makes it possible to write portable programs that can bypass proprietary operating systems.	At this time Java has already established itself as one of the most important languages for general-purpose programming as well as for computer science instruction.
Was Java designed for beginners?	No.

[6]Perhaps we'll tell you the story at some point.

Java is an industrial language. And because Java was not specifically designed for students, no thought was given to make it really simple to write basic programs. A certain amount of technical machinery is necessary in Java to write even the simplest programs.	To understand what this technical machinery does, you need to know something about programming.
This is not a problem for a professional programmer with prior experience in another programming language, but not having a linear learning path is a drawback for the student.	As you learn how to program in Java, there will be times when you will be asked to be satisfied with a preliminary explanation and wait for complete details in a later chapter.
Furthermore, you cannot hope to learn all of Java in one semester. The Java language itself is relatively simple, but Java contains a vast set of library packages that are necessary to write useful programs. There are packages for graphics, user interface design, cryptography, networking, sound, database storage, and many other purposes.	Even expert Java programmers do not know the contents of all the package—they just use those that are needed for particular projects.
Taking this class, you should expect to learn a good deal about the Java language and about the most important packages.	Keep in mind though that the purpose of this course is not to make you memorize Java minutiae, but to teach you how to think about programming.
All right, let's see a program written in Java.	How about this one?

```
public class Hello
{ public static void main(String[] args)
  { System.out.println("Hello, and welcome to A201!");
  }
}
```

What can it do?	It displays a simple greeting.
I'd like to see that.	You need to create a program file, compile it and then run it.

Here's the session in Unix:

```
frilled.cs.indiana.edu%pico Hello.java
frilled.cs.indiana.edu%javac Hello.java
frilled.cs.indiana.edu%java Hello
Hello, and welcome to A201!
frilled.cs.indiana.edu%
```

What's `pico`?

It's a (very small) Unix editor. That's how it all gets started: you enter the program statements into a text editor. The editor stores the text and gives it a name such as `Hello.java` which you then compile.	Yes, with `javac`.
When you compile your program, the compiler translates the Java *source code* (that is the text, or statements that you wrote) into so-called *bytecode* which consists of virtual machine instructions and some other pieces of information on how to load the program into memory prior to execution.	The bytecode for a program is stored in a separate file with extension `.class` for example the bytecode for the program we wrote will be stored in `Hello.class` and you should look for this file on your system after compilation.
What's `frilled`?	Just the prompt on the Unix machine we were on at the time. On your computer it might be `C:\>` or some such thing.
What's next?	The Java bytecode file contains the translation of your program in Java virtual machine terms. A *Java interpreter* loads the bytecode of the program you wrote, starts your program, and loads the necessary library bytecode files as they are required.
That's... `java`!	Precisely.
Your programming activity centers around these steps: you start in the editor, writing the source file. Compile the program—look at the error messages. Go back to the editor and fix the syntax errors. When the compiler succeeds—run the executable file.	If you find an error, you try to debug your program to find the cause of the error. Once you find the cause of the error, you go back to the editor and try to fix it. You compile and run again to see whether the error has gone away.
If not, you go back to the editor.	You bet.
This is called the edit - compile - debug loop, and you will spend a substantial amount of time in this loop in the months and years to come.	Can you draw a picture of that?
Sure. I could draw a *flowchart*.	I thought so.

Problems and Pain

The inevitable `fun()`

Let's take a look at some examples.	Examples of what?
Of what programming is like, of course.	Fine, just to eliminate any misunderstandings.
Exactly. Here's Problem No. 1	Go for it.
Write down detailed rules for multiplication, then find someone willing to help you. Hand the rules you wrote to that person. Ask her (or him) to perform one or two multiplications following the rules you wrote.	Ask the person to follow the rules *exactly*, without any innovations or implicit assumptions.
Can you give me an example of a multiplication?	Sure: 123×4578 is an example.
To make it even clearer, here's a second problem.	Problem No. 2
Exactly.	Let's jump right into it.
Very well: write down detailed rules for taking the square root of a positive integer.	For example: $\sqrt{123}$
Or $\sqrt{8}\ldots$	Or $\sqrt{18452998501835243075937l}$ for that matter.
It should be a little bit harder to come up with the rules in this case, or recall them	But that's what makes the problem an even better example than the previous one.
Yes, it's easier to test your rules on people now.	Yes, because they're not likely to take square roots on a daily basis (the way one uses multiplication).

Throughout the book, I will suggest some problems for you to play with. You might feel tempted to skip them. If they're too hard, fine. Some of them are pretty difficult!	But you might skip them thinking that, well, they've probably already been done by somebody else; so what's the point? Well, of course they've been done! But so what? Do them for the fun of it. That's how to learn the knack of doing things when you have to do them.
Let me give you an example. Suppose I wanted to add up a series of numbers, $1+2+3+4+5+6+7+\ldots$ up to, say, 62. No doubt you know how to do it; but when you play with this sort of problem as a kid, and you haven't been shown the answer... it's *fun* trying to figure out how to do it.	Then, as you go into adulthood, you develop a certain confidence that you can discover things; but if they've already been discovered, that shouldn't bother you at all. What one fool can do, so can another, and the fact that some other fool beat you to it shouldn't disturb you: you should get a kick out of having discovered something.
Most of the problems I give you in this book have been worked over many times, and many ingenious solutions have been devised for them.	But if you keep proving stuff that others have done, getting confidence, increasing the complexities of your solutions—for the fun of it—then one day you'll turn around and discover that nobody actually did that one! And that's the way to become a computer scientist.
I don't want to become a computer scientist...	You don't say!
But I think I solved the square root problem.	Hot ziggity, how'd you do it!?
OK, let's say we want to calculate $\sqrt{5}$. I start with an initial guess x_0 for $\sqrt{5}$. I choose a positive number.	Fascinating. What do you do with it?
Well, chances are that $x_0 \neq \sqrt{5}$, so I will use this guess to produce a new and better guess x_1.	How do you do that?
Here's the procedure. If $x_0 = \sqrt{5}$ we're done.	Sure. Even I could verify *that*.
If $x_0 \neq \sqrt{5}$ then x_0 is either (strictly) smaller or greater than $\sqrt{5}$.	That much I agree with.
In the former case we have $\sqrt{5}x_0 < 5$, that is, $$\sqrt{5} < \frac{5}{x_0}$$ for $x_0 \neq 0$.	On the other hand, if $x_0 > \sqrt{5}$, then $$\sqrt{5} > \frac{5}{x_0}$$

<table>
<tr><td>Thus we have either

$$x_0 < \sqrt{5} < \frac{5}{x_0}$$

or

$$\frac{5}{x_0} < \sqrt{5} < x_0$$</td><td>So, if we take the average of x_0 and $5/x_0$, namely

$$x_1 = \frac{1}{2}\left(x_0 + \frac{5}{x_0}\right)$$

the resulting value will lie midway between x_0 and $5/x_0$ and so will, hopefully, be a better approximation to $\sqrt{5}$ (although it might not).</td></tr>
<tr><td>So we use this value as our next “guess”.</td><td>Now you’re on to something.</td></tr>
<tr><td>Continuing, we form the successive averages

$$x_2 = \frac{1}{2}\left(x_1 + \frac{5}{x_1}\right)$$

$$x_3 = \frac{1}{2}\left(x_2 + \frac{5}{x_2}\right)$$

. . . and so on.</td><td>I got it.</td></tr>
<tr><td>Intuitively, the sequence of numbers $x_0, x_1, x_2, \ldots$ should eventually approach $\sqrt{5}$.</td><td>Pretty neat. And a lot of fun.</td></tr>
<tr><td>Certainly. But pretty difficult too.</td><td>Life is difficult, too. Life is a series of problems.</td></tr>
<tr><td>What makes life difficult is that the process of confronting and solving problems is a painful one.</td><td>Discipline is the basic set of tools that we require to solve life’s problems, and these tools are basically techniques of suffering: Means by which we experience the pain of problems in such a way as to work them through and solve them successfully, learning and growing in the process.</td></tr>
<tr><td>The tools of discipline are four: delaying of gratification, acceptance of responsibility, dedication to truth, and *balancing*.</td><td>What is *balancing*?</td></tr>
<tr><td>The exercise of discipline is not only a demanding but also a complex task, requiring both flexibility and judgment.</td><td>Courageous people must continually push themselves to be completely honest, yet must also possess the capacity to withhold the whole truth when appropriate.</td></tr>
</table>

To be free people, we must assume total responsibility for ourselves, but in doing so we must possess the capacity to reject responsibility that is not truly ours. To be organized and efficient, to live wisely, we must daily delay gratification and keep an eye on the future; yet to live joyously we must also possess the capacity, when it is not destructive, to live in the present and act spontaneously.	In other words, discipline itself must be disciplined.
Precisely. This kind of meta-discipline is what we call balancing. It is the type of discipline required to discipline discipline.	This is not hard; it is *very* hard.
But it is the kind of discipline that gives us flexibility.	Since you are taking A201, A597, or I210, or simply reading this book, it may be that you want, or need to learn Java—or programming in general. Since this is a first experience for you I deeply hope it will come easy, but be prepared if it does not.
In fact it really won't be easy at all, unless you approach it with patience, perseverence and determination.	If you treat it superficially it will be downright difficult from the beginning, and will continue to be that way until the very end, no matter how much we'll try to make it easy or understandable or obvious or intuitive or immediate or easy to grasp.
But you can help, and I am sure you will.	Because there is some risk involved, I wish you luck.
And because the act of entering programming as a beginner *and* a non-major is basically an act of courage, you have my admiration	The difficulty in learning programming has two clearly identifiable components: $\frac{3}{4}$ of it is of a very genuine mathematical nature. The other half ($\frac{1}{2}$) is psychological. You'll need to bridge the two.
That was from Yogi Berra, wasn't it?	And don't forget: whatever happens, you're still simply the best! Should someone fail to see this evidence, with *patience* prove it beyond any conceivable doubt.

"You can observe a lot by watching."
Yogi Berra (who also said: "If the people don't want to come out to the ballpark, nobody's going to stop them")

"Failures, repeated failures, are finger posts on the road to achievement. One fails forward toward success."
Charles F. Kettering (1876-1958, American Engineer, Inventor)

Getting Started

Edit, compile, run.
Our first laboratory experiments.
Our first Java programs.

In Lecture One I have shown you a videotape[7].	That's "Wallace and Gromit"[8] isn't it?.
Yes. We'll discuss some of it below.	I am sure we will get to that.
But first, I would like to give you a challenge.	What is it?
Can you write a program that produces this:	Is it easy, or hard?

```
         .8.           8 888888888o          ,o888888o.
        .888.          8 8888    ‘88.       8888       ‘88.
       :88888.         8 8888     ‘88    ,8 8888        ‘8.
      . ‘88888.        8 8888     ,88    88 8888
     .8. ‘88888.       8 8888.   ,88’    88 8888
    .8‘8. ‘88888.      8 8888888888      88 8888
   .8’ ‘8. ‘88888.     8 8888    ‘88.    88 8888
  .8’   ‘8. ‘88888.    8 8888      88    ‘8 8888       .8’
 .888888888. ‘88888.   8 8888    ,88’       8888     ,88’
.8’       ‘8. ‘88888.  8 888888888P          ‘8888888P’
```

I don't know, what do you think?	How do you do, Mr. Chinny Chin Chin?
You don't need to write this program...	I don't?
Just think whether you can write it or not (and *how*).	Oh, then that's easy!

[7] It was a portion of the "The Wrong Trousers".
[8] An Aardman production.

Also compile and run this program... (as shown in lecture)

```
import java.awt.Color;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.SwingConstants;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;

public class SliderTest
{
  public static void main(String[] args)
  {
    SliderFrame frame = new SliderFrame();
    frame.setTitle("SliderTest");
    frame.show();
  }
}

class SliderFrame extends JFrame
{
  public SliderFrame()
  {
    final int DEFAULT_FRAME_WIDTH = 300;
    final int DEFAULT_FRAME_HEIGHT = 300;
    setSize(DEFAULT_FRAME_WIDTH, DEFAULT_FRAME_HEIGHT);
    addWindowListener(new WindowCloser());
    colorPanel = new JPanel();
    ColorListener listener = new ColorListener();

    redSlider = new JSlider(0, 100, 100);
    redSlider.addChangeListener(listener);

    greenSlider = new JSlider(0, 100, 70);
    greenSlider.addChangeListener(listener);


    blueSlider = new JSlider(0, 100, 100);
    blueSlider.addChangeListener(listener);

    JPanel southPanel = new JPanel();
    southPanel.setLayout(new GridLayout(3, 2));
    southPanel.add(new JLabel("Red", SwingConstants.RIGHT));
```

```
    southPanel.add(redSlider);
    southPanel.add(new JLabel("Green", SwingConstants.RIGHT));
    southPanel.add(greenSlider);
    southPanel.add(new JLabel("Blue", SwingConstants.RIGHT));
    southPanel.add(blueSlider);

    Container contentPane = getContentPane();
    contentPane.add(colorPanel, "Center");
    contentPane.add(southPanel, "South");

    setSampleColor();
  }

  public void setSampleColor()
  {
    float red = 0.01F * redSlider.getValue();
    float green = 0.01F * greenSlider.getValue();
    float blue = 0.01F * blueSlider.getValue();

    colorPanel.setBackground(new Color(red, green, blue));
    colorPanel.repaint();
  }

  private JPanel colorPanel;
  private JSlider redSlider;
  private JSlider greenSlider;
  private JSlider blueSlider;

  private class ColorListener implements ChangeListener
  {
    public void stateChanged(ChangeEvent event)
    {
      setSampleColor();
    }
  }

  private class WindowCloser extends WindowAdapter
  {
    public void windowClosing(WindowEvent event)
    {
        System.exit(0);
    }
  }
}
```

What does it do?

It doesn't matter. But you should have no problem creating, compiling, and running it.

And when you see it running you should feel happy about it (and I hope you will).

The goal for this semester's work is to understand thoroughly a program such as this[9].	A second large program will be discussed, you can work with it here[10].
And now write a program as described below.	In your program:

1. Create a square room composed of 100 tiles (10 x 10, that is).
2. Create a `Penguin`, and add it to the room in the 8th line and 3rd column.
3. That means tile (7, 2) given our numbering convention.
4. So, please take some time to review the numbering convention now.
5. Now ask the penguin to turn around and move to tile (2, 2).
6. Then ask the penguin to turn right, then move to tile (2, 7).
7. Then make the penguin turn right, and have it move to tile (7, 7).
8. Now ask the penguin to move to tile (2, 7).
9. And finally have it return to (7, 2), passing through (2, 2).

What you will need is described below.	Yes, let me do the next chapter.
Be my guest. Here is a snapshot.	

[9] http://www.cs.indiana.edu/classes/a201-dger/spr2001/labs/nine/alienU.html
[10] http://www.cs.indiana.edu/classes/a348/t540/lectures/iceblox/iceblox.html

Your First Java Program

Visiting Karel the Robot.

What do we need to get started?	Take a look:

```
frilled.cs.indiana.edu%ls -l
total 28
-rw-------   1 dgerman  faculty       489 Oct 25 20:56 ActionList.java
-rw-------   1 dgerman  faculty      2376 Oct 25 22:05 AppletFrame.java
-rw-------   1 dgerman  faculty      1355 Oct 25 22:12 Dance.java
-rw-------   1 dgerman  faculty       883 Oct 25 20:56 NoFlickerApplet.java
-rw-------   1 dgerman  faculty      2261 Oct 25 22:11 One.java
-rw-------   1 dgerman  faculty      4052 Oct 25 22:04 Penguin.java
-rw-------   1 dgerman  faculty      4011 Oct 25 22:16 Rink.java
-rw-------   1 dgerman  faculty      9719 Oct 25 20:56 iceblox.gif
frilled.cs.indiana.edu%
```

You need six files, one of which[11] is a picture[12].	Let me explain each one in turn.
First, there is `Rink`, which is an applet.	No need to understand its inner workings now.
The only requirement is to understand how it's used.	Then there's `Penguin`[13]
Create two files, put the code in, and wait.	Then, there are three more ancillary files.
`ActionList` is every `Penguin`'s agenda.	`NoFlickerApplet` is a double-buffer.
These are all concepts we'll learn inside out.	By the end of the semester, of course, not today.

[11] `http://www.cs.indiana.edu/classes/a348/CTED/moduleFour/lectures/iceblox/iceblox.gif`
[12] Used by permission (courtesy Karl Hörnell).
[13] Which is a `Thread`.

`AppletFrame` simply provides a context.	Essential, but not central for now.
Then there's `One.java` which is the first example.	And `Dance.java` which is the second.
	Now:

A `Penguin`:

1. can be created (and it will be facing south by default)
2. can be added to a `Rink` at a certain location (you have to ask the rink for that, though)
3. can be asked to turn left, or right (regardless of the direction it's facing)
4. can be asked to move one cell forward (regardless of the direction it's facing)

A Rink:

1. can be created (with the size (columns, rows) of your choice)
2. can hold one penguin at a time, added with add
3. adding a penguin to a rink is done by specifying
   (a) the column (vertical slices), and
   (b) the line (lines are horizontal rows in the rink) in that order, in the method `add`.

Note though that when the Rink shows, it labels the cells by first printing the line, then the column, for each one of its tiles.	The reason for which this numbering is also important is due to it being the numbering used in 2 dimensional arrays in Java (of the kind that we will encounter a bit later).

Note that `x` and `y` still keep the meaning that they originally had:

1. `x` is the number of columns to the left, and
2. `y` is the number of lines above (to the top)

OK.

When we create the `Rink`, and when we add a `Penguin` to it, we mention `x` first, and `y` next, almost as we do in analytical geometry.	However when we refer to the table of cells that the `Rink` is, we can also denote the cells in the array by printing `(y, x)`, that is, by specifying the line first, and the column next. The point being that both notations are well-established, and we need to be aware of them both.
You should now write the program with no problems.	That's what I was going to say...

Here's Rink. It looks like a semester project to me.

```
/*
  <applet code="Rink.class" width=300 height=300>
    <param name="columns" value= "10">
    <param name="rows"    value= "10">
    <param name="penguin" value="yes">
    <param name="pengo_x" value=  "1">
    <param name="pengo_y" value=  "7">
  </applet>
*/
import java.applet.*;
import java.awt.*;
import java.net.*;
import java.awt.image.*;
import java.awt.event.*;
public class Rink extends NoFlickerApplet implements KeyListener {
    int columns, rows;
    Thread animation;
    Image small[];
    int cellWidth = 30, cellHeight = 30;
    public Rink() { }
    public Rink (int columns, int rows) {
        this.columns = columns;
        this.rows = rows;
    }
    int wide, tall;
    public void init() {
        if (columns == 0)
            this.columns = Integer.parseInt(this.getParameter("columns"));
        if (rows == 0)
            this.rows = Integer.parseInt(this.getParameter("rows"));
        this.wide = columns * cellWidth + cellWidth / 2;
        this.tall = (1 + rows) * cellHeight + cellHeight / 2;
        this.setSize(this.wide, this.tall);
        String pictureURL = "http://www.cs.indiana.edu/classes/a348/CT" +
                            "ED/moduleFour/lectures/iceblox/iceblox.gif";
        MediaTracker tracker = new MediaTracker(this);
        Image collection;
        try {
            collection =
                Toolkit.getDefaultToolkit().getImage(new URL(pictureURL));
        } catch (Exception e) {
            collection = Toolkit.getDefaultToolkit().getImage("iceblox.gif");
        }
        tracker.addImage(collection, 0);
        try {
            tracker.waitForID(0);
        } catch (InterruptedException e) { }
        ImageProducer collectionProducer = collection.getSource();
```

```
    int smalls = 48;
    small = new Image[smalls];
    int k = 0, i = 0, j = 0;
    ImageFilter filter;
    while (k < smalls) {
        filter = new CropImageFilter(j * 30, i * 30, 30, 30);
        small[k] = createImage(
                     new FilteredImageSource(
                       collectionProducer, filter));
        tracker.addImage(small[k], 1);
        k++; j++;
        if (j == 8) {
            j = 0; i++;
        }
    }
    try {
        tracker.waitForID(1);
    } catch (InterruptedException e) { }
    if (this.getParameter("penguin").equals("yes")) {
        this.add(new Penguin(),
                 Integer.parseInt(this.getParameter("pengo_x")),
                 Integer.parseInt(this.getParameter("pengo_y")));
    }
    this.addKeyListener(this);
}
int fontSize = 10; // in pixels
Font digitsFont = new Font("Serif", Font.PLAIN, fontSize);
Penguin skater;
void add(Penguin p, int x, int y) {
    this.skater = p;
    p.placeIn(this, x, y);
    skater.start();
}
public void paint(Graphics g) {
    ((Graphics2D)g).setFont(digitsFont);
    g.setColor(Color.black);
    g.fillRect(0, 0, columns * cellWidth, rows * cellHeight);
    g.setColor(Color.gray);
    for (int i = 0; i <= rows; i++) {
        g.drawLine(0, i * cellHeight, columns * cellWidth, i * cellHeight);
    }
    for (int i = 0; i <= columns; i++) {
        g.drawLine(i * cellWidth, 0, i * cellWidth, rows * cellHeight);
    }
    g.drawRect(0, 0, cellWidth * columns, cellHeight * rows);
    for (int j = 0; j < columns; j = j + 1)
        for (int i = 0; i < rows; i++)
            g.drawString(i + ", " + j,
                         j * cellWidth + 2, i * cellHeight + fontSize);
    if (skater != null) {
```

```
            int x = skater.x,
                y = skater.y;
            g.setColor(Color.black);
            g.fillRect(x, y, 31, 31);
            skater.draw(g);
        }
    }
    public void keyPressed(KeyEvent e) {
        switch(e.getKeyCode()) {
        case KeyEvent.VK_L: // left
            // System.out.println("left");
            skater.action.put("turnLeft");
            break;
        case KeyEvent.VK_F: // forward
            // System.out.println("forward");
            skater.action.put("moveForward");
            break;
        case KeyEvent.VK_R: // right
            // System.out.println("right");
            skater.action.put("turnRight");
            break;
        case KeyEvent.VK_B: // backwards
            // System.out.println("back");
            skater.action.put("backwards");
            break;
        }
    }
    public void keyReleased(KeyEvent e) { }
    public void keyTyped(KeyEvent e) { }
}
```

Here's Penguin in great detail. The second part of the semester project.

```
import java.awt.*;
public class Penguin extends Thread {
    ActionList action;
    public void run() {
        while (true) {
            perform(action.get());
        }
    }
    private void _pause() {
        try {
            this.sleep(speed * 10);
        } catch (InterruptedException e) { }
    }
    private void _think() {
        try {
            this.sleep(100);
        } catch (InterruptedException e) { }
    }
```

```
private void _turnLeft() {
    _think();
    if (direction.equals("south")) {
        direction = "east"; look = 12;
    } else if (direction.equals("east")) {
        direction = "north"; look = 5;
    } else if (direction.equals("north")) {
        direction = "west"; look = 7;
    } else {
        direction = "south"; look = 2;
    }
    report();
}
private void _wave() {
    for (int i = 0; i <  4; i++) {
        _think(); look =  0; report();
        _think(); look = 39; report();
    }
    _think(); look =  0; report();
}
private void _happy() {
    _think();
    if (direction.equals("south")) {
        _wave(); _think(); look = 2; report();
    } else if (direction.equals("east")) {
        look = 2; _wave(); _think(); look = 12; report();
    } else if (direction.equals("north")) {
        look =  7; report(); _think();
        look =  2; report(); _wave(); _think();
        look =  7; report(); _think();
        look =  5; report(); _think();
    } else { // west
        look = 2; report(); _wave();
        look = 7; report(); _think();
    }
}
private void _turnRight() {
    _think();
    if (direction.equals("south")) {
        direction = "west"; look = 7;
    } else if (direction.equals("east")) {
        direction = "south"; look = 2;
    } else if (direction.equals("north")) {
        direction = "east"; look = 12;
    } else {
        direction = "north"; look = 5;
    }
    report();
}
void _moveForward() {
```

```
    for (int i = 0; i < 5; i++) {
        _think();
        if (direction.equals("south")) {
            y += dy; look = animP[12 + (i + 1) % 4];
        } else if (direction.equals("east")) {
            x += dx; look = animP[ 4 + (i + 2) % 4];
        } else if (direction.equals("north")) {
            y -= dy; look = animP[ 8 + (i + 1) % 4];
        } else { // west
            x -= dx; look = animP[ 0 + i % 4];
        }
        report();
    }
}
public void perform(String action) {
    if (action.equals("turnLeft")) { this._turnLeft();
    } else if (action.equals("turnRight")) { this._turnRight();
    } else if (action.equals("moveForward")) { this._moveForward();
    } else if (action.equals("backwards")) {
        _turnLeft(); _turnLeft(); _moveForward(); _turnRight(); _turnRight();
    } else if (action.equals("pause")) { this._pause();
    } else if (action.equals("think")) { this._think();
    } else if (action.equals("happy")) { this._happy();
    } else System.out.println("Don't understand " + action);
}
void turnRight() { action.put("turnRight"); }
void moveForward() { action.put("moveForward"); }
void wave() { action.put("wave"); }
void pause() { action.put("pause"); }
void think() { action.put("think"); }
void happy() { action.put("happy"); }
int animP[] = {    7,  8,  9,  8, //  left ( west)
                  10, 11, 12, 11, // right ( east)
                   4,  5,  6,  5, //    up (north)
                   1,  2,  3,  2  //  down (south)
              };
Rink location;
void placeIn(Rink placement, int x, int y) {
    this.location = placement;
    frames = this.location.small;
    this.x = x * 30; this.y = y * 30;
    this.report();
}
Image[] frames;
void report() { location.repaint(); }
int x, y, dx = 6, dy = 6, look = 2;
void draw(Graphics g) {
    g.drawImage(frames[look], x, y, location);
}
int speed = 100;
```

```
        Penguin() {
            action = new ActionList();
            this.speed = 100;
        }
        void turnLeft() { action.put("turnLeft"); }
        String direction = "south";
    }
```

Don't forget the *agenda*.

How could I[14]?

You're right. . .

Perhaps we should say something about it.

Such as?

This is a FIFO[15] structure.

YDS[16]!!

OMWOH[17].

And it's also synchronized. Amazing!

```
    import java.util.*;
    public class ActionList {
        private Vector agenda;
        public synchronized String get() {
            while (agenda.size() == 0) {
                try {
                    // wait for producer
                    wait();
                } catch (InterruptedException e) { }
            }
            String value = (String) agenda.remove(0);
            notifyAll();
            return value;
        }
        public synchronized void put(String value) {
            agenda.addElement(value);
            notifyAll();
        }
        public ActionList() {
            agenda = new Vector();
        }
    }
```

Next the double-buffer.

And then the applet frame.

[14] Even if I wanted.
[15] First-in, first-out.
[16] You don't say!
[17] On my word of honor.

This one here is due to Calin Tenitchi.	A very clever concept.

```
import java.awt.*;
import java.applet.*;
class NoFlickerApplet extends Applet {
    private Image     offScreenImage;
    private Graphics  offScreenGraphics;
    private Dimension offScreenSize;
    public final void update(Graphics theGraphicsContext){
        Dimension dim = this.getSize(); /* size() originally... */
        if( (offScreenImage == null) ||
            (dim.width != offScreenSize.width) ||
            (dim.height != offScreenSize.height)) {
            this.offScreenImage = this.createImage(dim.width, dim.height);
            this.offScreenSize = dim;
            this.offScreenGraphics = this.offScreenImage.getGraphics();
        }
        this.offScreenGraphics.clearRect(0,
                                         0,
                                         this.offScreenSize.width,
                                         this.offScreenSize.height);
        this.paint(offScreenGraphics);
        theGraphicsContext.drawImage(this.offScreenImage,
                                     0,
                                     0,
                                     null);
    }
}
```

Next one is due to Gary Cornell and Cay Horstmann.	A little-used[18] trick.
Indeed, as you will see it will provide a lot of flexibility.	I certainly hope so.

```
import java.awt.*;
import java.net.*;
import java.awt.event.*;
import java.applet.*;
import java.util.*;
import java.io.*;
public class AppletFrame extends Frame implements AppletStub,
                                                  AppletContext,
                                                  WindowListener {
    AppletFrame(Applet a) {
        setTitle(a.getClass().getName());
        add("Center", a);
        a.setStub(this);
```

[18] And yet again very refreshing.

```
        a.init();
        a.start();
        setSize(((Rink)a).wide, ((Rink)a).tall); // note the casting
        show();
        this.addWindowListener(this);
    }
    // AppletStub methods
    public boolean isActive() { return true; }
    public URL getDocumentBase() { return null; }
    public URL getCodeBase() { return null; }
    public String getParameter(String name) {
        if (name.equals("columns")) return "10";
        else if (name.equals("rows")) return "10";
        else if (name.equals("penguin")) return "no"; // could be yes
        else if (name.equals("pengo_x")) return "1";  // don't matter...
        else if (name.equals("pengo_y")) return "7";
        else return "";
    }
    public AppletContext getAppletContext() { return this; }
    public void appletResize(int width, int height) { }
    // AppletContext methods
    public AudioClip getAudioClip(URL url) { return null; }
    public Image getImage(URL url) { return null; }
    public Applet getApplet(String name) { return null; }
    public Enumeration getApplets() { return null; }
    public void showDocument(URL url) { }
    public void showDocument(URL url, String target) { }
    public void showStatus(String status) { }
    public void setStream(String key,
                          InputStream stream) throws IOException { }
    public InputStream getStream(String key) { return null; }
    public Iterator getStreamKeys() { return null; }
    // WindowListener methods
    public void windowActivated(WindowEvent e) { }
    public void windowClosed(WindowEvent e) { }
    public void windowClosing(WindowEvent e) { System.exit(0); }
    public void windowDeactivated(WindowEvent e) { }
    public void windowDeiconified(WindowEvent e) { }
    public void windowIconified(WindowEvent e) { }
    public void windowOpened(WindowEvent e) { }
}
```

With this we can write our first two programs.	The first one, is presented below.
Remember: if you have the five plus one files listed above, you're ready to write new programs.	One such program is listed here.
What does it do?	It asks `Pixel Pete` to move around.

```
import java.awt.*;
import java.net.*;
import java.awt.event.*;
import java.applet.*;
import java.util.*;
import java.io.*;

public class One {

    public static void main(String[] args) {

        Rink ballroom = new Rink(10,  // number of columns
                                 10); // number of rows

        /* Note the Rink created is called 'ballroom'. We'll
           have to use this name to refer to it thereafter.*/

        new AppletFrame(ballroom); // ask me why you need this...

        Penguin p = new Penguin(); // create a Penguin, call it
                                   // ... 'p' (what's in a name?)

        ballroom.add(p, 1, 7); // add the Penguin to our Rink, in
        // column 1 and line 7 (and remember our numbering scheme)

        p.pause(); p.turnLeft(); // control the Penguin
        /* Remember 'The Wrong Trousers' (the video)? */

        p.moveForward();
        p.moveForward();
        p.moveForward();
        p.happy();
        p.moveForward();
        p.moveForward();
        p.moveForward();
        p.pause();
        p.turnLeft();
        p.pause();        // commands are issued in sequence

        p.moveForward(); p.moveForward(); p.moveForward(); p.happy();
        p.moveForward(); p.moveForward(); p.moveForward(); p.pause();

        p.turnLeft(); p.pause();

        p.moveForward(); p.moveForward(); p.moveForward(); p.happy();
        p.moveForward(); p.moveForward(); p.moveForward(); p.pause();

        p.turnLeft(); p.pause();
```

```
        p.moveForward(); p.moveForward(); p.moveForward(); p.happy();
p.moveForward(); p.moveForward(); p.moveForward(); p.pause(); p.turnLeft();
p.pause(); p.moveForward(); p.turnRight(); p.moveForward(); p.turnLeft(); p.
moveForward(); p.moveForward(); p.turnLeft(); p.turnLeft(); p.moveForward();
p.moveForward(); p.turnLeft(); p.turnLeft(); p.moveForward(); p.moveForward();
 p.turnLeft(); p.turnLeft(); p.moveForward(); p.moveForward(); p.turnLeft();
p.turnLeft(); p.moveForward(); p.moveForward(); p.turnLeft(); p.turnLeft();

        /* Can you still say you know where the Penguin is right now?

           Remember that not only the computer reads your programs!

           Write your programs as if they were essays.

           Make your code crystal clear.

         */

        p.moveForward();
        p.moveForward();

        p.turnLeft();

        p.happy();

    }
}
```

Here's the second one. Called `Dance.java` it's a bit shorter.

```
class Dance {
    public static void main(String[] args) {

        Rink ballroom = new Rink(6, 6);

        new AppletFrame(ballroom); // again, ask me why you need this...

        Penguin p = new Penguin();

        ballroom.add(p, 1, 4);

        p.pause();

        p.turnLeft();

        p.turnLeft();    p.moveForward(); // go left
        p.turnRight();   p.moveForward(); // go right
        p.turnLeft();    p.moveForward(); // go left
        p.turnRight();   p.moveForward(); // go right
        p.turnLeft();    p.moveForward(); // go left
```

```
        p.turnRight();   p.moveForward(); // go right
        // now stop, rotate once, stay some more
        p.pause();       p.turnRight();   p.pause();
        // come south three tiles
        p.moveForward(); p.moveForward(); p.moveForward();
        // stop and catch your breath
        p.pause();
        // pirouette
        p.turnLeft();    p.turnLeft();    p.turnLeft();    p.turnLeft();
        // stop, for applause
        p.pause();
        // another pirouette, followed by immediate movement west
        p.turnRight();   p.turnRight();   p.turnRight();   p.turnRight();
        p.turnRight();   p.moveForward(); p.moveForward(); p.moveForward();
        // stop
        p.pause();
        // turn left, then stop
        p.turnLeft();

        p.pause();
        // one final pirouette, after which just thank the audience
        p.turnLeft(); p.turnLeft(); p.turnLeft(); p.turnLeft();

        p.happy();
        // Don't worry(), be happy().
    }
}
```

Algorithms

Algorithms.
Programs.
Mechanics of implementation in Java.

Before we look at the mechanics of implementing computations let us consider the planning process that precedes the implementation.

If you can't give instructions for someone to solve a problem by hand, there is no way the computer can magically solve the problem.

The computer can only do what you do by hand. It just does it faster, and it doesn't get bored or exhausted.

I'd like to see an example.

OK, let's consider the following investment problem:

> You put $10,000 into a bank account that earns 5% interest per year. Interest is compounded annually. How many years does it take for the account balance to be double the original?

One could solve this problem by hand.

Sure, let's do that. After the first year you earn $500 (5% of $10,000). The interest gets added to your bank account and your balance becomes $10,500.00. Next year, the interest is $525 (5% of $10,500)...

...and your balance is $11,025.

You can keep going that way and build a table:

Very good!

Year	*B*alance	
0	$\$10,000.00 =$	$\$10,000.00$
1	$\$10,000.00 + 0.05 \times \$10,000 =$	$\$10,500.00$
2	$\$10,500.00 + 0.05 \times \$10,500.00 =$	$\$11,025.00$
3	$\$11,025.00 + 0.05 \times \$11,025.00 =$	$\$11,576.25$
4	$\$11,576.25 + 0.05 \times \$11,576.25 =$	$\$12,155.06$

You keep going until the balance goes over $20,000.00 and when it does...

...which it does (doesn't it?!)...

...you look into the "Year" column and you have the answer. Of course, carrying out this computation is intensely boring.

Yes, but the fact that a computation is boring or tedious is irrelevant to a computer. Computers are very good at carrying out repetitive calculations quickly and flawlessly.

What is important to the computer is the existence of a systematic approach for finding the solution. The answer can be found just by following a series of steps that involves no guesswork.

Here's such a series of steps:

Step 1 Start with the table

Year	*B*alance	
0	$10,000.00 =	$10,000.00

Step 2 Repeat steps **2a**, **2b**, **2c** while the balance is less than $20,000.00

Step 2a Add a new row to the table.

Step 2b In column 1 of the new row, labeled "Year", put one more than the preceding year value.

Step 2c In column 2 of the new row, labeled "Balance", place the value of the preceding balance value, multiplied by $1.05 = (1 + 5\%)$

Step 3 Read the last number in the year column and report it as the number of years required to double the investment.

Of course, these steps are not yet in a language that a computer can understand, but you will learn soon how to formulate them in Java.

Is this collection of steps an *algorithm*?

Yes, because the description is unambigous...

There are precise instructions what to do in every step and where to go next. There is no room for guesswork or creativity.

...executable, and terminating.

Because each step can be carried out in practice, and the computation can be shown to come to an end: with every step, the balance goes up by at least $500.00, so eventually it must reach $20,000.00

Now we start looking at the mechanics of implementing computations in Java.	Let's analyze our first program.

```
public class Hello
{ public static void main(String[] args)
  { System.out.println("Hello, World!");
  }
}
```

You know that you should make a new program file and call it `Hello.java`, enter the program instructions, then compile and run the program.	That's clear, the contents is more intriguing.
It is composed of words and symbols separated by spaces.	The words and symbols are important and atomic: they're like words and symbols in an English sentence. *Bring me a glass of water!*
Yes, but it's a lot stricter. Java is *case-sensitive*. You must enter upper- and lowercase letters exactly as they appear in the program listing.	On the other hand Java has *free-form layout*. Spaces and line breaks are not important, except to separate words.
However, good taste dictates that you lay out your programs in readable fashion, so you should follow the layout in the program listing.	Now that we've seen the program, it's time to understand its makeup.
The [part] [in] [boxes] introduces a class,	... called `Hello`.

```
[public class Hello]
[{] public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
[}]
```

	In Java, classes are the central organizing mechanism for code. You can't do *anything* in Java without defining at least a class.
That is why we introduce the `Hello` class	... as the holder of the `main` method.
Java, like most programming languages, requires that all program statements be placed inside *methods*.	A method is a collection of programming instructions that describe how to carry out a particular task.

The part in boxes further defines the `main` method.

```
public class Hello
{ public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

Every Java application must have a main method.

Most Java programs contain other methods beside `main`, but it will take us a while to learn how to write other methods. For the time being, simply put all instructions that you want to have executed inside the `main` method of a class.

I have them all boxed. (There's only one, here).

```
public class Hello
{ public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

The instructions or *statements* in the body of the main method (that is, the statements inside the curly braces `{ }`) are executed one by one. Note that each statement ends in a semicolon (`;`).

Our method has a single statement:

```
System.out.println("Hello, World!");
```

Yes, but it, too, has a structure.

The statement is supposed to print a line of text. I presume the text is enclosed by double quotes (`"`).

Yes, a sequence of characters in quotation marks is called a *string*. You must enclose the contents of the string inside quotation marks so that the compiler considers them plain text and does not try to interpret them as program instructions.

To print the text you call a method `println` as if you'd call Papa John's for a large pizza. But which Papa John's? You need to precisely locate it. Suppose you say: the one on 3rd Street. But which 3rd: most towns have a 3rd Street. So you need to add Bloomington, and then IN.

All of this is apparent in their telephone number:

812	323	PAPA
`System`	`out`	`println`

From this analogy it looks like `System` contains `out` which contains `println`?

Yes, we call the `println` method that is part of the `out` object, that is part of the `System` class, and we pass it the string that we wanted printed.

So that's what the parentheses are for...

Yes, in fact that's how we tell that `out` is merely data, and not a method: method names are always followed by a pair of parentheses.

Why is `System` uppercased and `out` lowercased?

This is only a convention, that object variables (or names) start with a lowercase letter, while classes names should start with an upper case. Using this convention is strongly encouraged, as part of the style guide.

So let's summarize: designers of the Java libraries defined a `System` class in which they've put useful objects and methods. One of these objects is called `out` and it lets you access the terminal window (also called the *standard output*). To use the `out` object in the `System` class you must refer to it as `System.out`; it has a method inside it, by the name of `println` which we can use, and so we do.

That's correct.

Asking the computer to execute a method is also known as *calling* or *invoking* the method.

When we call a method we can pass it information in between parentheses. If we pass no information there will be an empty pair of parentheses.

In this case we pass one string.

Have you looked at the exercises yet?

What for?

How would you print `Hello, "World"!` ?

You need to escape the quotation marks inside the string with a backslash (\), like that:

```
public class Hello
{ public static void main(String[] args) {
    System.out.println("Hello, \"World\"!");
  }
}
```

It becomes harder to read, but it's also more precise.

The computer won't mistake any of the two escaped double quotes as being the end of the string.

What other escape sequences are there?

Let's mention a couple…

Since the backslash character is used as an escape character, it needs to be escaped itself, if we need it in output. Another escape sequence used ocasionally is `\n` which is the same as new line or line feed character.

Are there any other things that we could pass to `println` for printing?

Yes, for example arithmetic expressions.

Such as:

```
3 + 4
(2.5 - 1) / 4
(3 + 4) * (2 - 5)
```

Yes. What's the asterisk (`*`) for?

It's for multiplication, and / for division.

Very good. You could even "add" strings by listing them with a `+` between them. `println` will actually *concatenate* them (or string them together).

OK, maybe I won't do that right away. What's the difference between `println` and `print`?

The `out` object contains a second method called `print`. You can see the difference between the two methods if you consider the following program:

```
public class Test
{ public static void main(String[] args)
  { System.out.println("Hello, ");
    System.out.println("World! ");
  }
}
```

What does it do?

The `println` method prints a string or a number and then starts a new line. The `print` method does the same printing, without starting a new line afterwards.

I see... putting all of the things we've learned together I could write the same program as follows:

```
public class Test
{ public static void main(String[] args)
  { System.out.print("Hello, \n");
    System.out.print("World! \n");
  }
}
```

Yes, and in fact in many other ways.

I won't count how many...

Simple Programs

Objects and classes, reference types, symbolic names, variables declaration and their initialization through assignment.

Objects and classes are central concepts for Java programming.	It might take you some time to master these concepts fully, but since every Java program uses at least a couple of objects and classes, it is a good idea to have a basic understanding of these concepts right away.
An *object* is an entity that you can manipulate in your program, generally by calling *methods*.	You should think of an object as a "black box" with a public interface—the methods you can call, and a hidden implementation—the code and data that are necessary to make these methods work.
Different objects support different sets of methods (in general).	OK, enough of that, let's see some examples. Have we seen any object yet?
`System.out`	What methods does it have?
`println` and `print`	Yes, and we call them by identifying the object, followed by the name of the method, and then by parentheses.
Yes, don't you ever forget the parentheses.	Good. What other objects have we seen?
None that I know of.	Exactly. You've seen one but you didn't know it was an object.
`"Hello, World!"`	Indeed. It is a `String` object (its type).
Does every object have a type?	Yes.

Then what type does `System.out` have?

`PrintStream` and its class is defined in the package `java.io`. But this shouldn't tell you too much just yet. (Full name is `java.io.PrintStream`.)

Nor could I have answered this question by myself with what we know so far.

Although you could have looked it up in the online documentation as part of class `System`.

```
http://java.sun.com/products/jdk/1.2/docs/api/java/lang/System.html#out
http://java.sun.com/products/jdk/1.2/docs/api/java/lang/System.html
http://java.sun.com/products/jdk/1.2/docs/api/overview-tree.html
```

...which is defined in the package `java.lang` (and there are so many other packages besides it).

Let's go back to `"Hello, World!"`.

What methods does it have?

No `println` or `print`, I don't think...

To find out what methods it supports we need to look up its class (`String`) into the on-line documentation.

Of all the methods it has, we choose to take a look at the `length()` method.

```
http://java.sun.com/products/jdk/1.2/docs/api/java/lang/String.html
```

For any object of type `String` the `length` method counts and returns (reports) the number of characters in the string.

So `"Hello, World!".length()` evaluates to 13

...as the quotation marks are not counted.

I'd like to see that.

Very well, then try this:

```
System.out.println("Hello, World!".length());
```

How does this work?

Same as before, only more work is to be done before `println` can output its argument.

OK, let's move on. Without classes there would be no objects. Let's take a look at classes.

A class is a holding place (or a container) for static methods and objects.

That's old news: the `Hello` class holds the static `main` method. The `System` class holds the static `out` object.

A class is also a factory for objects. It contains the blueprint of all objects of that kind, and can be used to generate new objects.

I'd like to see that.

To see how a class can be an object factory, let us turn to another class: the `Rectangle` class in the Java class library.

Objects of type `Rectangle` describe rectangular shapes.

This is where you can read about `Rectangles`.

```
http://java.sun.com/products/jdk/1.2/docs/api/java/awt/Rectangle.html
```

Great—that's off the same URL you gave me earlier.

Note that a `Rectangle` object isn't a rectangular shape—it is just a set of numbers that describe the rectangle. Each rectangle is described by the x and y coordinates of its top left corner, its width and height.

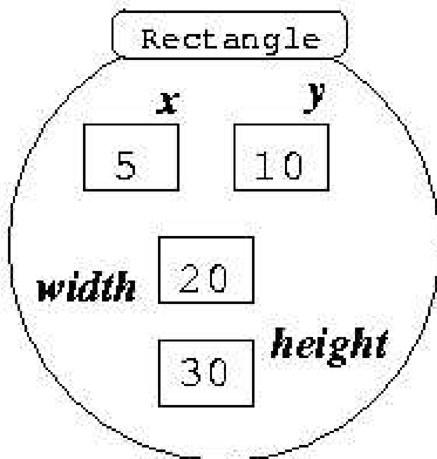
I think I'd like to see a picture of that.

We need to work an example first. You can make a new rectangle with top left corner at (5, 10) and with a width of 20 and height 30. To make a new rectangle you need to specify these four values, and in that order.

```
new Rectangle(5, 10, 20, 30)
```

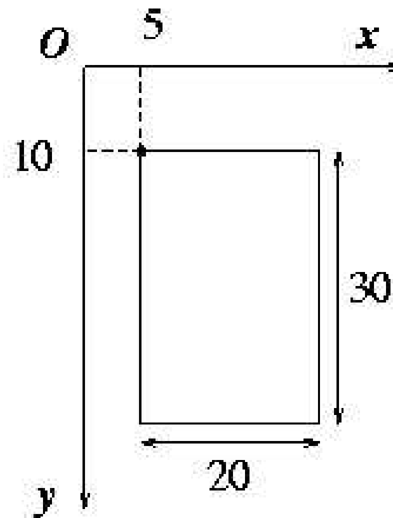
The `new` operator causes the creation of an object of type `Rectangle`. The process of creating a new object is called *construction*. The four values 5, 10, 20, 30 are called the *construction parameters*.

What does the new object look like?



What's the rectangular shape that is described by this `Rectangle` object?

Now you can draw your picture.



Why did you draw the referential upside down?

Because that's how it is in computer graphics.

To find the x you just measure how far you are from the left margin of the screen.

To find out the y coordinate you measure how many lines of pixels are there in between that point and the top of the screen.

This is a side-effect of thinking that in English we write from left to right, and from top to bottom	... so a character in a text message could be located by the number of the column in which it appears (the x coordinate) and the line in which it appears (the y coordinate).

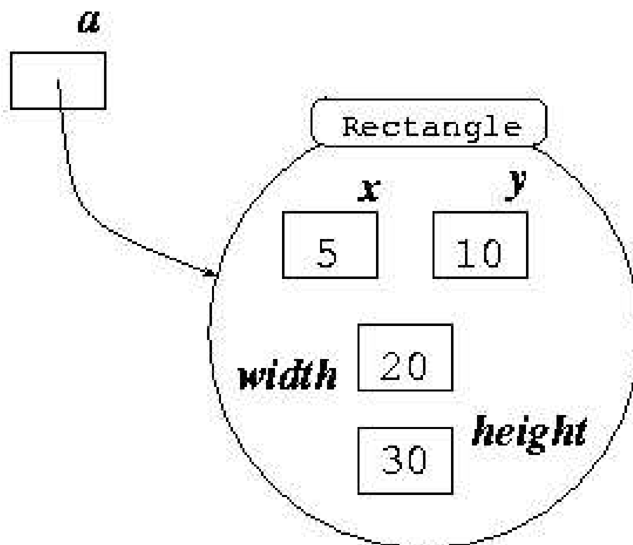
To construct any object, you do the following:

1. use the `new` operator
2. give the name of the class
3. supply construction parameters (if any) inside parentheses

Different classes will require different construction parameters, and some classes will let you construct objects in multiple ways.	For example, you can also obtain a `Rectangle` object by supplying no construction parameters at all: `new Rectangle();`
But you must still use the parentheses. This constructs a (rather useless) rectangle with top left corner at the origin (0, 0), width 0 and height 0.	How do I know that?
You have to read up the documentation see what the designers of the class had in mind for it.	So it's not something I could have deduced, or inferred?
No.	What can you do with a `Rectangle` object?
What can you do with a number?	What number?
Say, `"abc".length()`	I could print it, to see if it comes out as 3 or not.
Can you print a `Rectangle` object?	I could try. Can it be printed?
You should try it.	I'd rather *draw* it.
We'll learn that in a few chapters.	OK, how about this:

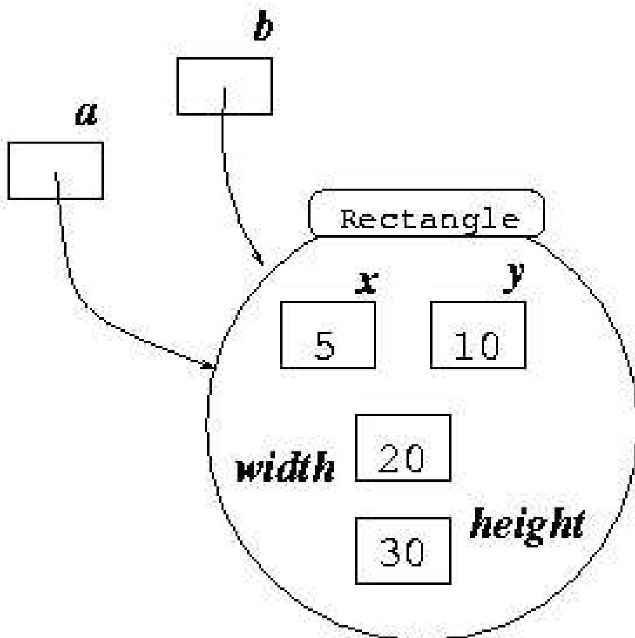
```
System.out.println(new Rectangle(5, 10, 20, 30));
```

How does it work?	The code creates an object of type `Rectangle` then passes the object to the `println` method, and finally forgets that object.

To remember an object give it a name; hold it in an *object variable.*	An *object variable* is a storage location that stores not the actual object but information about the object's location.
Can you give it any name?	Variable names in Java can start with a letter, an underscore (`_`), or a dollar sign (`$`). They cannot start with a number. After the first character, your variable names can include any letter or number.
Once you decide on a name for a variable, to declare it you need to place the name of the class in front of it, followed by the variable name, and a semicolon (`;`) at the end.	Like this? `Rectangle a;`
Yes. This is a declaration statement. It says that the name `a` will be used for a variable that will hold the address to a `Rectangle` object.	So far, the variable does not refer to any object at all. To make it refer to an actual object you could copy in it the address of an actual object reference, as returned by `new`: `Rectangle a = new Rectangle(5, 10, 20, 30);}`
It is very important that you remember that `a` does not contain the object. It contains the address of the object (and refers to this object).  	I have a picture for that:
Very good.	I think I got the hang of it.

You could have two object variables refer to the same object.	Like this? `Rectangle b = a;`
Yes. The equal sign (=) acts more like an arrow from right to left, as it represents the copying of the value on the left into the location that the right hand side denotes.	So the value on the right, which is the contents of the variable `a` (an address) is copied into the location that `b` denotes.
`b` has just been declared. And at the time of its declaration (and also allocation) we copy in it the address of the anonymous object that `a` points to.	May I draw a picture of that?

Definitely.

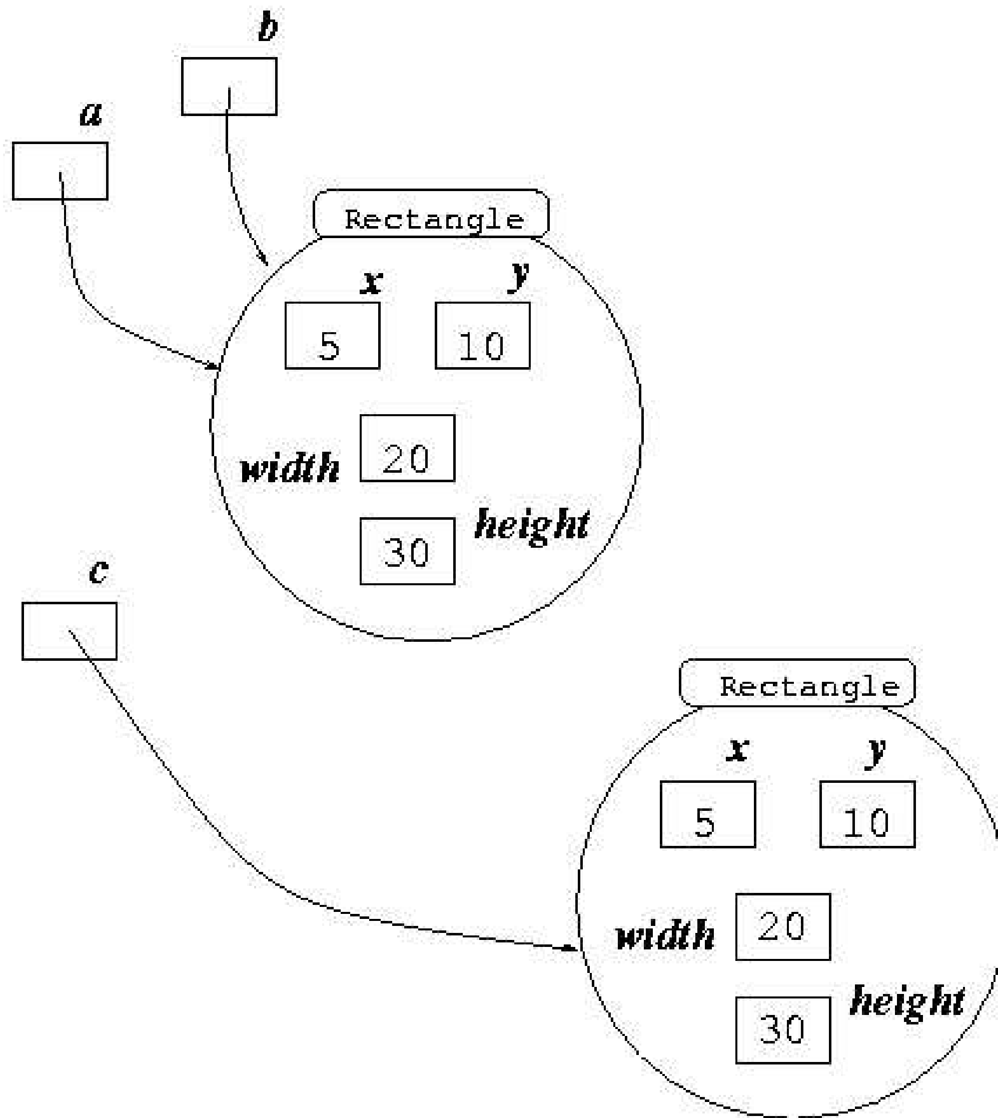


Now how does the picture change if we add

```
Rectangle c = new Rectangle(5, 10, 20, 30);
```

A new object will come into the picture, and its address will be stored in the variable with the name `c`.

Isn't it identical to the object pointed to by `a` and `b`?	Yes, it's like a twin. Identical but not the same.
Let me see it.	There you go.



What else can you do with `Rectangle` objects?	The `Rectangle` class has over 50 methods, some useful, some less so.
To give you a flavor of manipulating `Rectangles`, let us look at a method of the `Rectangle` class.	The `translate` method *moves* a rectangle by a certain distance in the x and y directions.
For example `a.translate(15, 25);`	...moves the rectangle by 15 units in the x direction and 25 units in the y direction.

Moving a rectangle doesn't change its width or height, but it changes the coordinates of the top left corner.	Can I see that?
Let's write a program and test it. As with the `Hello` program, we need to carry out three steps:	1. Invent a new class, say `RectangleTest` 2. Supply a `main` method 3. Place instructions inside the `main` method
Correct. Let's see the program.	How about this one:

```
public class RectangleTest
{ public static void main(String[] args)
  { Rectangle a = new Rectangle(5, 10, 20, 30);
    System.out.println(a);
    a.translate(15, 25);
    System.out.println(a);
  }
}
```

It would work well, but	...if you try to compile it you will run into an error.

```
frilled.cs.indiana.edu%pico RectangleTest.java
frilled.cs.indiana.edu%javac RectangleTest.java
RectangleTest.java:3: Class Rectangle not found.
{ Rectangle a = new Rectangle(5, 10, 20, 30);
  ^
RectangleTest.java:3: Class Rectangle not found.
  { Rectangle a = new Rectangle(5, 10, 20, 30);
                      ^
2 errors
frilled.cs.indiana.edu%
```

Does it always come out in boxes?

Not really. But the point here is that for this program there is an additional step that you need to carry out: you need to *import* the `Rectangle` class from a *package*	...which is a collection of classes with a related purpose. All classes in the standard library are contained in packages. The `Rectangle` class belongs to the package `java.awt`. Thus the full name of the `Rectangle` class is really `java.awt.Rectangle`.
The abbreviation `awt` stands for "Abstract Windowing Toolkit". To use `Rectangle` from the `java.awt` package simply place the following	

```
import java.awt.Rectangle;
```

...at the top of your program.

You never need to import classes from the `java.lang` package. All classes from this package are automatically imported.	That's why we can use the `String` and `System` classes without ever needing to import them.
Ready for one final question?	Certainly.

What's the output of the following snippet of code?

```
Rectangle a = new Rectangle(5, 10, 20, 30);
Rectangle b = a;
a.translate(10, 10);
b.translate(10, 10);
System.out.println(a);
```

	Easy: `a` and `b` both point to the same object. We basically translate it twice, once by using its `a` name and once by using its `b` name.
In the end printing `a` or `b` is the same.	You will see the original rectangle that has been translated twice.
So the top left corner is now at (25, 30) but the width and height are unchanged.	Good. Can we move on now?
We sure can.	Great. I think it was about time.

```
import java.io.*; // I/O package needed

public class ConsoleReader {
    public ConsoleReader(InputStream inStream) { // constructor
        reader = new BufferedReader(
                    new InputStreamReader(
                      inStream));
    }
    public String readLine() { // instance method
        String inputLine = "";
        try {
            inputLine = reader.readLine();
        } catch (IOException e) {
            System.out.println(e);
            System.exit(1);
        }
        return inputLine;
    }
    public int readInt() { // instance method
        String inputString = readLine();
        int n = Integer.parseInt(inputString);
        return n;
    }
    public double readDouble() { // instance method
        String inputString = readLine();
        double x = Double.parseDouble(inputString);
        return x;
    }
    private BufferedReader reader; // instance method
}
```

Types and I/O

Numbers. Strings.
Reading input with `ConsoleReader`*.*
`ConsoleReader` *revealed.*

What's 23?	A number. An integer.
Java calls that an `int`.	Most of the times.
What's 3.5?	A number with a decimal part.
Java calls that a floating-point number.	I see. Isn't there a keyword for that, like `int`?
There are two of them: `double` and `float`.	In Java there are two kinds of numbers: integers and floating point numbers.
Integers have no fractional part.	And floating point numbers, which have a decimal point and therefore a fractional part.
So `2.0` is a floating-point number	... while `2` is an integer.
The second one does not have any fractional part, while the first one's is zero.	Not missing, but zero.
In practice this can make a big difference.	There are two reasons for having separate types for numbers: one philosophical and one pragmatic.
The philosophy is to use whole numbers when you can't have or don't need a fractional part.	It is generally a good idea to choose programming solutions that document one's intentions.
Pragmatically speaking, integers are more efficient than floating-point numbers.	They take less storage and are processed faster.

<table>
<tr><td>How do you use <code>int</code>, <code>double</code>, <code>float</code> in practice?</td><td>Like <code>Rectangle</code> they're types.</td></tr>
<tr><td>Unlike objects of type <code>Rectangle</code> numbers are not objects.</td><td>Yes, <code>Rectangle</code> is a reference type. <code>int</code>, <code>double</code>, <code>float</code> (and 5 other) are primitive types.</td></tr>
<tr><td>So we can declare a variable of type <code>int</code>?</td><td>Yes. It's like in algebra, except names have types in Java.</td></tr>
<tr><td>In Java each variable has a type.

By defining

<code>int a;</code>

you proclaim that <code>a</code> can hold only integer values.</td><td>Even though initialization is optional, it is a good idea always to initialize variables with a specific value.</td></tr>
<tr><td>You should always supply an initial value for every variable at the time you define it.</td><td>So I could, for example, write:

<code>int a = 3;</code></td></tr>
<tr><td>Could you have written

<code>int a = 3.5;</code>

instead?</td><td>No that is a contradiction in terms. I would have broken my own rule of proclaiming that <code>a</code> won't need a fractional part.</td></tr>
<tr><td>Can we write

<code>double b = 3.4;</code></td><td>Yes, but how about:

<code>double b = 3;</code></td></tr>
<tr><td>That would work well, since there's no loss of information. Seeing the missing fractional part of 3 Java will initialize <code>b</code> with <code>3.0</code></td><td>Symbolic names like <code>a</code> and <code>b</code> are meant to make the program more readable and manageable. Keep in mind, however, that you can only declare and initialize a symbolic name just once in every method.</td></tr>
<tr><td>Why is it good to initialize variables as soon as we define them?</td><td>So that we don't forget to initialize them at all.</td></tr>
<tr><td>If you try to use an uninitialized variable the compiler will notice and complain.</td><td>All but the simplest programs use variables to store values. Variables are locations in memory that can hold values of a particular type.</td></tr>
</table>

They're called variables because once you store a value in them you can change it later at will.	I'd like to see that.

Very well, take a look:
`int a = 5;`
`a = 7;`
`System.out.println(a);`
	That prints 7 and it works as follows:
	`// a is declared and initialized to 5,`
	`// but later the value 7 is assigned to it.`

Yes, that was an example of an assignment statement.	Can I see others?

Yes. Here's a more complicated scenario:
`int a = 5;`
`int b = a + 5;`
`System.out.println(b);`
	This prints 10
	`// as what's being stored in b is the value of a plus 5.`

So a symbolic name acts as a storage location (or address) on the left hand side of the equals sign (which is used for assignment statements)	. . . while when it appears on the right hand side it is replaced by its value.
Can one name appear on both sides of an assignment?	Show me an example.
Here's a scenario: `int a = 10; a = a + 10; System.out.println(a);`	By the rule you formulated, that should print 20.
Yes, since the expression on the right is evaluated first, then the resulting value is stored in the location named `a`.	What other expressions can we use?
All of arithmetic, if you refer to numbers.	However, in general, when you make an assignment of an expression into a variable, the expression on the right, . . .
. . . which can contain method invocations, . . .	. . . is first evaluated, and the resulting value has to be of a compatible type with the type of the variable.
OK, I will try to remember that.	Let's work some more examples.
Here's a challenge first. Given: `int x, y; x = 5; y = 3;` . . . how do you swap the values of `x` and `y`?	Easy. I need a third (temporary) location:

```
int x = 5, y = 3, temp;
temp = x;
x = y;
y = temp;
```

Is there another way?

What do you mean?

Supose you *can't* use another variable.

Yet x and y are numbers?

Yes.

Then it's trickier, but fancier:

```
int x = 5, y = 3;
x = x + y;
y = x - y;
x = x - y;
```

Well done!

I know. Isn't that nice?

Let's work out some more examples.

Why is this legal?

```
int a = 3;
double b = a;
```

Because b becomes 3.0, so we acknowledge the lack of fractional part of a by writing a 0 (zero) for it in b.

Is this legal?

```
double b = 3.5;
int a = b;
```

No, because a doesn't have any room for a fractional part (0.5) in it.

Can we just ignore that, the fractional part?

You can, but Java won't do that for you unless you specifically request it.

How do I do that?

You *cast* the floating point value to an integer:

```
double b = 3.5;
int a = (int) b;
```

This has the effect of discarding the fractional part.

What is (int)?

It is the *cast to an int* operator. It acts like the unary minus sign. For example, the expression `-5 + 3`	. . . which yields `-2`,
and `-(5 + 3)`	. . . which yields `-8`,
. . . is the same as the difference between `(int)3.6 + 3.6`	. . . which yields `6.6`,
. . . and `(int)(3.6 + 3.6)`	. . . which yields `7`.
There is a good reason why you must use a cast in Java when you convert a floating point number to an integer: The conversion *loses information*.	You must confirm that you agree to that information loss. Java is quite strict about this. You must use a cast whenever there is the possibility of information loss.
A cast is always of the form `(` *typename*`)` for example `(int)` or `(double)`.	There are a few methods in class `Math` that have a related functionality.
`Math.round(3.7)` evaluates to. . .	`4`
`Math.round(-3.7)` evaluates to. . .	`-4`
`Math.round(3.2)` evaluates to. . .	`3`
`Math.round(x)` evaluates to the closest integer to `x` (represented as a `long`).	What's `long`?
Another kind of integer. We'll talk about it before *too* long.	OK. Hit me with more `Math`

`Math.ceil(3.7)` evaluates to...	`4.0`
`Math.ceil(-3.7)` evaluates to...	`-3.0`
`Math.ceil(3.2)` evaluates to...	`4.0`
`Math.ceil(x)` evaluates to the smallest integer greater or equal to `x` (as a `double`).	`ceil` is short for "ceiling". There is also a mathematical "floor" function.
`Math.floor(3.7)` evaluates to...	`3.0`
`Math.floor(-3.7)` evaluates to...	`-4.0`
`Math.floor(3.2)` evaluates to...	`3.0`
`Math.floor` evaluates to the largest integer less than or equal to `x` (as a `double`).	`Math` is a class that is defined in the `java.lang` package.
The `Math` class groups together the definitions of several useful mathematical methods such as: `sqrt`, `pow`, `sin`, `cos`, `exp`, `log`, `abs`, `round`, `ceil`, `floor`	...and many others. All these methods are *static* (or class) methods (unlike `print` and `println` of the `System.out` object). They belong to the class `Math`.
Beginners (or uninitiated) might think that in `Math.round(3.7)` the `round` method is applied to an object called `Math`, because `Math.` precedes `round`	...just as `System.out.` precedes `print`. That's not true. `Math` is a class, not an object.
A method such as `Math.round` that does not operate on any object is known as a `static` method; another example is `main`.	Static methods do not operate on objects, but they are still defined inside classes, and you must specify the class to which the `round` method belongs.
How can you tell whether `Math` is a class or an object.	You really can't.
Then how do we know?	It is certainly useful to memorize the names of the more important classes (such as `System` and `Math`). You should also pay attention to capitalization.
All classes in the Java library start with an uppercase letter (such as `System`).	Objects and methods start with a lowercase letter (such as `out` and `println`).

You can tell objects and methods apart because method calls are followed by parentheses.	Therefore
	
	`System.out.println()`
	
	denotes a call of the `println` method on the `out` object inside the `System` class. On the other hand
	
	`Math.round(price)`
	
	denotes a call to the `round` method...

...inside the `Math` class. This use of upper- and lowercase letters is merely a *convention*, not a rule of the Java language.	It is, however, a convention that the authors of the Java class libraries follow consistently. You should do the same in your programs.
You can use all four basic arithmetic operations in Java: addition, subtraction, multiplication, and division.	Parentheses are used just as in algebra: to indicate in which order the subexpressions should be computed.
Just as in regular algebraic notation, multiplication and division *bind more strongly* than addition and subtraction.	So `3 + 5 * 2` yields `13` while `(3 + 5) * 2` yields `16` as the parentheses come into play.
Division works as you would expect, as long as at least one of the numbers involved is a floating-point number.	However, if *both* numbers are integers, then the result of the division is always an integer, with the remainder discarded.
Here are some examples: `17 / 4`	...evaluates to: `4`
`10 / 3`	`3`
`13 / 7`	`1`
and `6 / 9` evaluates to...	`0`
If you're interested only in the remainder,	...you can use the `%` operator.
But for that we need to turn the page.	Exactly.

Here are some examples.	...evaluates to:
`17 % 4`	`1`
`10 % 3`	`1`
`13 % 7`	`6`
and `6 % 9` evaluates to...	`6`
The symbol `%` has no analog in algebra. It was chosen because it looks similar to `/`, and the remainder operation is related to division. `(a / b) * b + a % b`	Is it true that... ...is the same as `a` (for `a` and `b` positive integers)?
Yes. But can you *prove* it?	OK, let's move on.
What is `16 / 5 * 5`	`15`, since all operands are integers.
How about `(16 / 5) * 5`	Still `15`, and in the same way.
What then is `5 * (16 / 5)`	`15` (multiplication is commutative).
OK, drop the parentheses: `5 * 16 / 5`	The result is now `16` as we have to do the multiplication first.
What property requires this?	Left-to-right associativity.
Very good.	Let's move on.
Next to numbers *strings* are the most important data type that most programs use. A string is a sequence of characters, such as `"Hello"`	In Java strings are enclosed in quotation marks, which are not themselves part of the string.

You can declare variables that hold strings:

```
String name = "John";
```

Use assignment

```
name = "Carl";
```

. . . to place a different string into the variable.

The number of characters in a string is called the *length*. For example, the length of "Hello!" is 6.

You can compute the length of a string with the `length` method:

```
int n = name.length();
```

That would place 4 in `n`.

For our example, yes.

By the way, a string of length 0 (zero), containing no characters, is called the *empty* string

. . . and is written as `""`.

Also note that unlike numbers, strings are objects.

`Rectangles` were objects too. You can tell that `String` is a class because it starts with an uppercase letter. The basic types `int` and `double` start with a lowercase letter.

Once you have a string, what can you do with it?

You can extract substrings, and you can glue smaller strings together to form larger ones.

To extract a string use the `substring` operation.

```
s.substring(start, pastEnd)
```

returns a string that is made up of . . .

. . . the characters in the string `s` starting at character with index `start`, and containing all characters up to, but not including, the character with index `pastEnd`. Let's see an example.

```
String a = "automaton";
String b = a.substring(2, 8);
// b is set to "tomato"
```

In Java there are two ways of writing comments.

We already know (and have used it above) that the compiler ignores anything that you type between `//` and the end of line.

The compiler also ignores any text between a `/*` and `*/`. The `//` comment is easier to type if the comment is only a single line long.

If you have a comment that is longer than a line or two, then the `/* ...*/` comment is simpler.

So we could also have:

```
String c = "appearance";
String d = c.substring(2, 6);
/* d is "pear" as the substring operation makes a string
   that consists of four characters taken from string c */
```

A curious aspect of the `substring` operation is the numbering of starting and ending positions.

Starting position 0 (zero) means ”start at the beginning of the string”.

For technical reasons that used to be important but are no longer relevant, Java string position numbers start at 0 (zero).

The first position is labeled 0 (zero), the second one is labeled 1 (one), and so on.

For example here are the position numbers

```
appearance
0123456789
```

in the "appearance" string.

The position number of the last character (a for the "appearance" string) is always 1 less than the length of the string.

How do you extract the substring "Bird" from

```
"Larry Bird, Indiana"
           1         1
 0123456789012345678
```

Count characters starting at 0, not 1. You find that B, the 7th character, has position number 6. The first character that you *don't* want, a comma, is the character at position 10.

Therefore the appropriate substring command is

```
String m = "Larry Bird, Indiana";
String n = m.substring(6, 10);
```

It is curious that you must specify the position of the first character that you do want and then the first character that you don't want.

There is one advantage to this setup. You can easily compute the *length* of the substring: it is

```
pastEnd - start
```

If you omit the second parameter of the substring method, then all characters from the starting position to the end of the string are copied.

For example:

```
String u = "Larry Bird, Indiana";
String tail = u.substring(6);
```

...sets tail to the string

```
"Bird, Indiana"
```

This is equivalent to the call

```
String tail = u.substring(6, u.length());
```

I see.

Now that you know how to take strings apart, let us see how to put them back together.

Given two strings, such as "India" and "napolis", you can *concatenate* them to one long string.

```
String one = "India";
String two = "napolis";
String city = one + two;
```

The + operator concatenates two strings.

How do you get

```
"Larry Bird"
```

out of "Larry" and "Bird"?

```
"Larry" + " " + "Bird"
```

Very good, with a *blank* in the middle.

The concatenation operator in Java is very powerful.	If one of the expressions, either to the left or the right of a `+` operator, is a string, then the other one is automatically forced to become a string as well, and both strings are concatenated.
For example:	...evaluates to:
`"Agent" + 7`	`"Agent7"`
`"2" + 3`	`"23"`
`2 + 3`	`5`
Of course when associativity comes into play you have to be a bit more careful.	...evaluates to:
`2 + "" + 3`	`"23"`
whereas...	...evaluates to:
`2 + 3 + ""`	`"5"`
Concatenation is very useful to reduce the number of `System.out.print` instructions.	Fine. What if you want to convert a string like "23" that contains only decimals into a number?
To *convert* a string into a number you have two possibilities: to convert to an `int` use...	`Integer.parseInt(...)`
...and to convert to a `double` use...	`Double.parseDouble(...)`
So, ...	...evaluates to:
`"23" + 5`	`"235"`
but `Integer.parseInt("23") + 5`	`28`
Likewise `"2.3" + 5`	`"2.35"`
while `Double.parseDouble("2.3") + 5`	`7.3`

How can you get a "2.3" string when you need a number?	If you're a user and type a number such as 2.3 you will realize that you, in fact, have to type three characters:
...the digit 2, the period, and the digit 3.	The whole thing is a string of characters.
And that's what you will have to start from when dealing with user input.	How can we write programs that accept user input?
We will use a non-standard Java class, that we will thoroughly discuss later on, and whose purpose is to make processing keyboard input easier and less tedious.	We recommend that you use the `ConsoleReader` class in your own programs whenever you need to read console input.
Simply place the `ConsoleReader.java` file together with your program file, into a common directory.	The purpose of the `ConsoleReader` class is to add a friendly interface to an input stream such as `System.in` and here's how you use it.

To accept user input in a program you first need to construct a `ConsoleReader` object, like this:

```
ConsoleReader console = new ConsoleReader(System.in);
```

Next, simply call one of the following three methods:

```
String line = console.readLine(); // read a line of input
int n = console.readInt();        // read an integer
double x = console.readDouble();  // read a floating-point number
```

Let's see an example.	OK, let's write a program that asks a user for a name (that will be recorded as a `String`), an amount of dollars (that the user has) and the rate of exchange between the British pound and the American dollar.
The program asks all this of the user, and then computes and tells the user how many British pounds the user would get in exchange for the amount of dollars the user has.	Why not leave this for tomorrow?
Good idea.	See you in the lab.

Reference vs. Primitive Types

Some simple programs. Some sample problems.

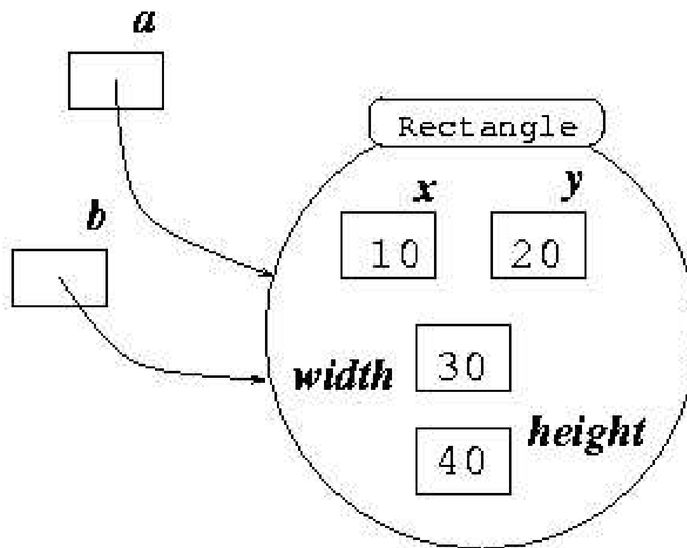
Here's a diagrammatic description of the fundamental difference between primitive and reference types.

Reference types first (using `Rectangle`)

Suppose you have

```
Rectangle a = new Rectangle(10, 20, 30, 40);
Rectangle b = a;
```

The picture looks as follows:



Now if you have

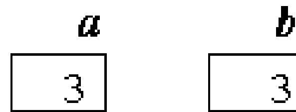
```
a.translate(3, 3);
System.out.println(b);
```

You will be able to see the change. Both `a` and `b` point to one and the same thing, an essentially anonymous object, to which we refer as both `a` and `b`. So changes made using the `a` name can be seen by looking at the object using the other name, that is, `b`.

Primitive types (using `int`)

```
int a = 3;
int b = a;
```

The picture looks as follows:



The big difference is that the primitive value is copied *into* the storage location. So each location has its own copy. It just works that way with numbers (as primitive types) but doesn't work the same with objects (reference types).

That's simply how it works. With reference types, what the variable holds is a *reference*, the arrow. With primitive types, the variable holds a *value*, a copy of the actual value.

Now if you have

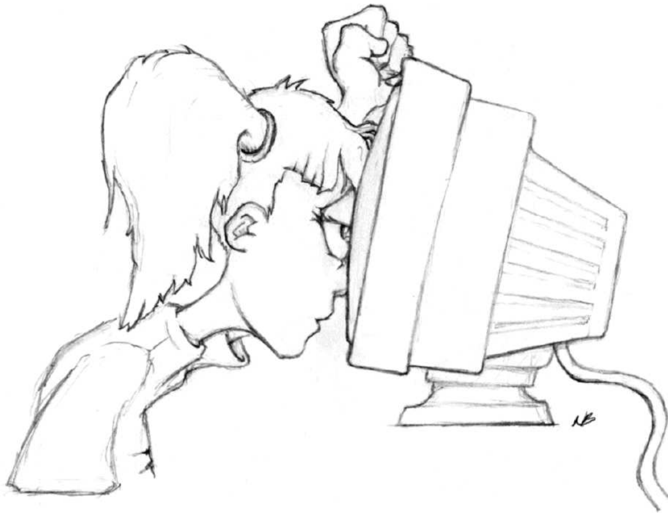
```
a = 10;
System.out.println(b);
```

you will see that the value of `b` has not been updated.

Each location has its own (copy of the) value, and changing one does not affect the other. This is an important difference, we refer to it as the by value vs. by reference access to variables, and it will be useful for you to remember it from now on when you reason about and design programs.

Now, let's move on to exercises.

Yes, we'd better be doing that.



I need to do more exercises!

Check the website for more!

Syntax

The structure of `main`

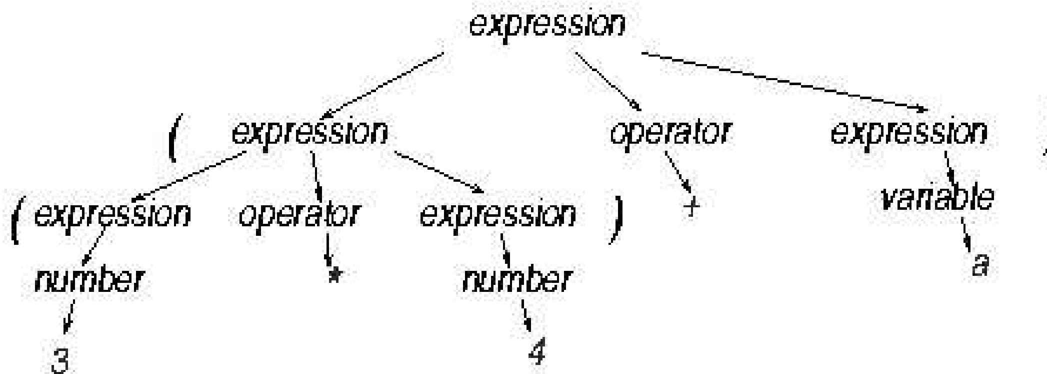
What is a *number*?	That much we know.
Let's say: an *integer* or a *floating point number*.	We know all about *integers* and *floating point numbers*, I'd say.
What is an *operator*?	You want a long answer or a short one?
Short.	`+`, `-`, `*`, `/` are *operators*.
What is an *expression*?	A *number*, is a (very simple) expression. A *symbolic name* is also a (very simple, or *atomic*) expression.
What is an *expression* followed by an *operator* followed by another *expression*?	That would also be an expression, wouldn't it?
Yes, indeed. What do we have so far?	We have this table:

Term defined	Composition
expression	*number*
	variable
	(*expression operator expression*)

Do we need the parentheses?	Not really, but we want to emphasize the structure.
Correct. Now, is `((3 + 4) * a)` an expression?	It is. Do you want to show why with a diagram?

I sure do.

Very well, there you go:



That's interesting that we can define a concept (such as *expression*) in terms of itself.

This is called a *recursive* definition. An important part of a recursive definition is specifying a set of *fixed points*,

...without which defining something in terms of itself could go on for ever.

Exactly. Thank goodness for *numbers* and *variable names* without which our definition would be irreversibly circular.

What is an *assignment statement*?

Its structure is as follows: *variable name*, followed by an *equal sign*, followed by an *expression*, and then by a *semicolon*.

```
a = 3 + 5;
a = b;
r = new Rectangle(2, 2, 10, 10);
```

What are we trying to get at?

The structure of all Java programs.

What is a *declaration*?

```
int m;
Rectangle r;
double x, y, z;
```

It is a *type*, followed by a *variable name* and a *semicolon*...

I see you can declare several variables at the same time.

Yes, that's relatively obvious by now, I hope.

. . . and it can also be a *type*, followed by an *assignment statement* (as defined above).

You should be building a few examples following these definitions, or take some statements and analyze them the way we analyzed the simple expression a few lines above.

```
int m = 25;
int n = 34, p;
Rectangle r = new Rectangle (5, 10, 8, 16);
```

OK, I see what you're getting at, let's move on.

Are we really going to be extremely precise and cover all possible cases?

Not really. For that we'd have to sacrifice some of the intuitive structure of all these.

But we'll go far enough for you to get a good grasp of the general structure.

Are you ready for something really deep?

Go ahead.

What is a *function call*?

A *function name* followed by *open parenthesis*, followed by zero or more *arguments* separated by commas, followed by *closed parenthesis*.

```
System.out.println();
System.out.println("Wow!");
Math.sqrt(2);
r.translate(3, 3);
```

What is an *argument*?

I suppose any *expression* would work as one.

```
System.out.println("tom" + "ate".substring(0, 2) + "a dog".charAt(3));
```

I've seen `length()` and `substring()` being invoked on `String`s, but what's `charAt()`?

Exactly what you're thinking that it might be.

Based on its name?

Yes. And we'll talk more about `char`s tomorrow.

OK, let's summarize what we have so far:

Literals appear in boxes.

Term defined	Composition
expression	number variable (expression operator expression)
funCall	funName ( ) funName (arguments)
arguments	expression expression, arguments

Any questions about function names?	Not really, I suppose they're basically of the "*absolute path*" kind, like `System.out.println`
Good assumption.	Are function calls expressions themselves?
I'm glad you asked. The answer is "yes" if the function returns a value.	Then we are now dealing with an even bigger infinity of expressions.
Note though that not all functions return values.	Yes, `println` from `System.out` does not return a value, but `sqrt` from class `Math` returns the square root of the argument, so it can be used in an expression.

Let's update our table. We'll put a star (*) next to remind ourselves that it's a *logic* error to use a *function call* in an expression if the function does not return a value. I didn't box the terminals this time around.

Term defined	**Composition**
expression	*number* *variable* (*expression operator expression*) *funCall*(*)
funCall	*funName* ( ) *funName* (*arguments*)
arguments	*expression* *expression*, *arguments*

The update is minimal, but I think I need to look at more examples before I become too dizzy.

You've already seen examples of expressions, as defined by the table above.	This table is a table of syntactic categories described in terms of other syntactic categories,
. . . an enterprise aimed at describing the (grammatical structure of our simple) language.	Here are some more examples of expressions.
Take them apart, fit them in the table, don't just accept them and then move on. `Math.sqrt(a + Math.sqrt(b))`	Don't worry. Let's take a look: This is easy: $\sqrt{a+\sqrt{b}}$
`Math.sqrt(Math.sqrt(Math.sqrt(Math.sqrt(a))))`	Just as easy: $\sqrt{\sqrt{\sqrt{a}}}$

`Math.pow(a, (1.0 / 16.0))`	Same as above: $a^{\frac{1}{16}}$

```
Math.sqrt(Math.pow(Math.sqrt(Math.pow(Math.sqrt(2)), 2)), 2)
```

Let's rewrite this as follows:

```
Math.sqrt(
  Math.pow(
    Math.sqrt(
      Math.pow(
        Math.sqrt(2)), 2)), 2)
```

A bit messy, perhaps, but easy:

$$\sqrt{\left(\sqrt{(\sqrt{2})^2}\right)^2}$$

`(6 + (5 + (4 + (3 + (2 + 1)))))`	Neat.

What is a *statement*?

A *statement* is a

- *declaration* or
- an *assignment statement*,

Or a *function call*, as in:

```
System.out.println("Hello, world!");
```

What is this?

```
public class Template
  { public static void main(String[] args)
    {
      <methodBody>
    }
  }
```

(Notice how I boxed the non-terminal this time.)

Exactly.

It's the template we're using, and `methodBody` is composed of one or more *statements*.

That is, *declarations* and *statements* in any order, with the following final table describing the entire structure of the language (so far).

Term defined	Composition
statement	*declaration* *assignment* *funCall*
declaration	*type variable* ; *type variable* = *assignment* ;
assignment	*variable* = *expression* ;
expression	*number* *variable* (*expression operator expression*) *funCall*(*)
funCall	*funName* ( ) *funName* (*arguments*)
arguments	*expression* *expression*, *arguments*

This table covers the syntax of the Java programs that we are going to be writing for a while.

Note that not all syntactically correct programs are logically correct.

For example declaring a variable `a` twice is a *semantic* error, although having two declarations in a program is not a syntactic error...

... but if the variable is one and the same the semantic part of the compiler will signal that.

Suppose you compile and run the program below.

```
class One {
  public static void main(String[] args) {
    int m = 2;
    System.out.println(m);
    int m = 3;
    System.out.println(m);
  }
}
```

What's its output?

The program won't compile.

One cannot declare a variable twice.

(One cannot declare *the same* variable twice.)

All right, let's move on. Have you heard of the latest late policy on assignments and such?

There's no new policy, everything is still the same: you need to turn everything on time. Try your best to meet the deadlines.

Just trying to get your attention.

Sure. Let's move on.

Fine. What is an `int`?

It's an integer number between -2^{31} (which is about -2 billion) and $(2^{31} - 1)$. If you need to refer to these boundaries in your program, use the constants `Integer.MIN_VALUE` and `Integer.MAX_VALUE`

... which are defined in a class called `Integer` like `Math.PI` is defined in the `Math` class.

Convention says: name your constants using all caps for the name of the constant.

How do you *define* constants?	Mark them `final` when you declare them.
Here's a program, what does it produce: (see code below)	Overflow.

```
public class Test
{ public static void main(String[] args)
    { int n = Integer.MAX_VALUE;
      System.out.println(n);
      n = n + 1;
      System.out.println(n);
    }
}
```

Have you run it?	Yes, it's an eye-opening experience.
Why does it happen?	Representation is finite.
What if we need bigger integers?	Use `long`.
What's a `long`?	It's a type that allows for the representation of bigger integers. The range is now -2^{63} (which is about -9 billion billions) to (2^{63} - 1).
What if we need bigger integers?	Then work with objects of class `BigInteger`.
How do I do that?	We'll see that in a minute. What is a floating-point number?
A `double` or a `float`.	`float` spans the range from -3.4E38 to 3.4E38. `double` is much wider: from -1.7E308 to 1.7E308 but both suffer from the same problem:
... precision. (see code below)	Yes—what's the output of this program?

```
public class Test
{ public static void main(String[] args)
    { double a = 30000000000000000000000.0;
      System.out.println(a - (a - 0.5));
    }
}
```

It should be `0.5` by algebra.	Yes, but is that what the program prints?
Try it.	Also, try to initialize the `double` variable with a value that doesn't contain the decimal point.
In that case the compiler catches the overflow before it happens.	Overflow happens even for `double` type.

The following program produces `Infinity`.	You're kidding me!

Try it:

```
public class Test
{ public static void main(String[] args)
    { double a = 1.5E308;
      System.out.println(a * 10);
    }
}
```

For most of the programming projects in this book, the limited range and precision of `int` and `double` are acceptable.	Just bear in mind that overflow or loss of precision occur.
Another kind of loss of precision occurs in what is known as a *roundoff error*.	What's *that*?
In the processor hardware, numbers are represented in the binary number system, not in decimal.	You get roundoff errors when binary digits are lost, they just may crop up at different places than you might expect.
Here's an example: `System.out.println(4.35 * 100);`	Another eyeopener.
What do we do?	Keep a cool head. For example in this last case first round then *cast* to an `int`...
...if you want an `int`, ...	...and especially if you want the right one.
How about `System.out.println(Math.pow(Math.sqrt(2), 2));`	Slightly off...
A bit different, but related.	Yes, and we'll discuss how floating-point numbers should be tested for equality soon.
How do you use big numbers?	If you want to compute with really large numbers, you can use *big number* objects.

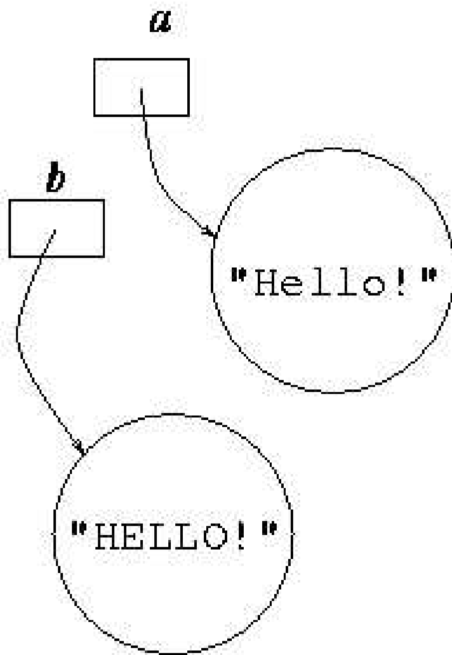
Big number objects are objects of the `BigInteger`[19] and `BigDecimal`[20] classes in the `java.math`[21] package.	Unlike `int` or `double`, big numbers have essentially no limits on their size and precision,
...however, computations with big number objects are much slower than those that involve primitive types. To perform operations you need to use methods such as...	• `add(...)` • `subtract(...)` • `multiply(...)` and • `divide(...)`
(...that big number objects have.)	Here's `a + b * c` with big numbers.

```
import java.math.*;
public class Test
{ public static void main(String[] args)
  { BigInteger a = new BigInteger("1000000000000000000000000000000000000000");
    BigInteger b = new BigInteger("2000000000000000000000000000000000000000");
    BigInteger c = new BigInteger("3000000000000000000000000000000000000000");
    BigInteger d = a.add(b.multiply(c));
    System.out.println(d);
  }
}
```

This is good exercise in object notation.	You bet. We've seen rectangles, strings, and now big numbers. We're getting even better.
So what should we have achieved by now?	I think we understand integer and floating point numbers.
We're able to write arithmetic expressions in Java.	We can appreciate the importance of comments and good code layout.
We can define and initialize variables and constants.	We know of the limitations of `int` and `double` types,
...and the overflow and roundoff errors that result.	We can read program input with `ConsoleReader`.
We know how to print program output.	We understand the structure of a `method body`,
...which contains statements, such as declarations, assignments, and function calls.	We've also talked about expressions.
We have used the `String` class...	...to define and manipulate strings.

[19] http://java.sun.com/products/jdk/1.2/docs/api/java/math/BigInteger.html
[20] http://java.sun.com/products/jdk/1.2/docs/api/java/math/BigDecimal.html
[21] http://java.sun.com/products/jdk/1.2/docs/api/java/math/package-tree.html

	By the way can you draw a picture for me for the following situation:
	`String a = "Hello!";`
	`String b = a.toUpperCase();`

I sure can: 	Thanks. It is the same with `substring`, isn't it? I *know* it is. `String` objects are called *immutable* objects.
That's right: `toUpperCase` on a `String` works like `add` on a big number...	...or like `intersection` on a `Rectangle`.

Predicates

`boolean` values, expressions, and `if` statements.

What's this? `x < 15`	A `boolean` expression: an expression whose value is either `true` or `false`.
Can you define `boolean` variables in Java?	Sure, `boolean` is a primitive type in Java.
`boolean b;`	`b` can only hold `true` and `false` values.
What are the primitive types in Java? • `int, byte, short, long` • `double, float` • `boolean` • `char`	There are four kinds: • whole numbers • floating-point numbers • boolean values • characters
We've seen `int`, and `double` values...	...we now take a look at `boolean` values.
How do you read this? `x <= 15`	"x is less than or equal to 15"
How do you write "x is in between 9 and 15"?	First, what value does it have: `true` or `false`?
We don't know yet, it depends on what x is.	So we might as well call it $p(x)$.
Very well; now we can look at it for particular values of x.	$p(3)$ is `false`
$p(20)$ also is `false`	Come to think of it, $p(x)$ is `false` for *many* values.
What's $p(x)$ again?	A statement about x being between 9 and 15.

When is it `true`?	When `x <= 15` and at the same time `x >= 9`
How do you write AND in Java?	`&&`
So $p(x)$ can be written as: `(x >= 9) && (x <= 15)`	`&&` is read as AND
And `||` is read as OR.	While `!` stands for NOT in Java.
I like A201 `!`	I do `!` think `this` is *that* funny...
So `&&`, `||`, and `!` are operators for truth values.	Yes. How do they work?

Let me draw a table.

p	q	p && q	p || q	! q
true	true	true	true	false
true	false	false	true	true
false	true	false	true	
false	false	false	false	
		AND	OR	NOT

So `&&` works in the following way: you graduate if you satisfy both of two requirements.

Otherwise you don't graduate.	`||` is a bit more lenient.
You graduate if you satisfy *at least one* of the two requirements.	And only when none of them has been satisfied you do not graduate.
And `!` is easy.	It is, indeed.
The `boolean` type is called after mathematician George Boole, a pioneer in the study of logic.	Logic is tricky: suppose `a` is a `boolean` value, `true` or `false`. What value does `a || !a` have?
Doesn't it depend on the value of `a`?	No.

Well, then let's look at all possible cases:

a	! a	a || (! a)
true	false	true
false	true	true

Doesn't it look easy now?

Yes, and there weren't even too many cases.

How do you compute:

```
3 + 5 * 2
```

Why are you bringing this up?

Because as you know there is an implicit order of evaluation for arithmetic expressions.

Does a similar set of rules apply to `boolean` expressions?

Yes. In arithmetic, unary minuses are taken into account before we do any multiplications...

...and only after that we may do addtions, if any.

If there are no parentheses, otherwise the parens dictate the order of evaluation.

What rules govern the order of evaluation for `&&`, `||`, and `!` ?

`!` has the highest priority. Then comes `&&`, and the `||` has the lowest priority.

So if you look at

```
a || b && ! c
```

It's evaluated as

```
a || (b && (! c))
```

What is the truth table for

```
!a && !b
```

Let's build it at the same time for

```
!(a || b)
```

a	b	a && b	!(a && b)	!a	!b	!a || !b
true	true	true	false	false	false	false
true	false	false	true	false	true	true
false	true	false	true	true	false	true
false	false	false	true	true	true	true

We have just proved one of DeMorgan's law.

What is the other one?

It's the dual of this:

```
!(a || b)
```

is the same as

```
!a && !b
```

There are many other identities that one can prove.

Perhaps we can do that later, as needed.

Yes, but let me give some examples, in case you get bored and want to practice.

Sure.

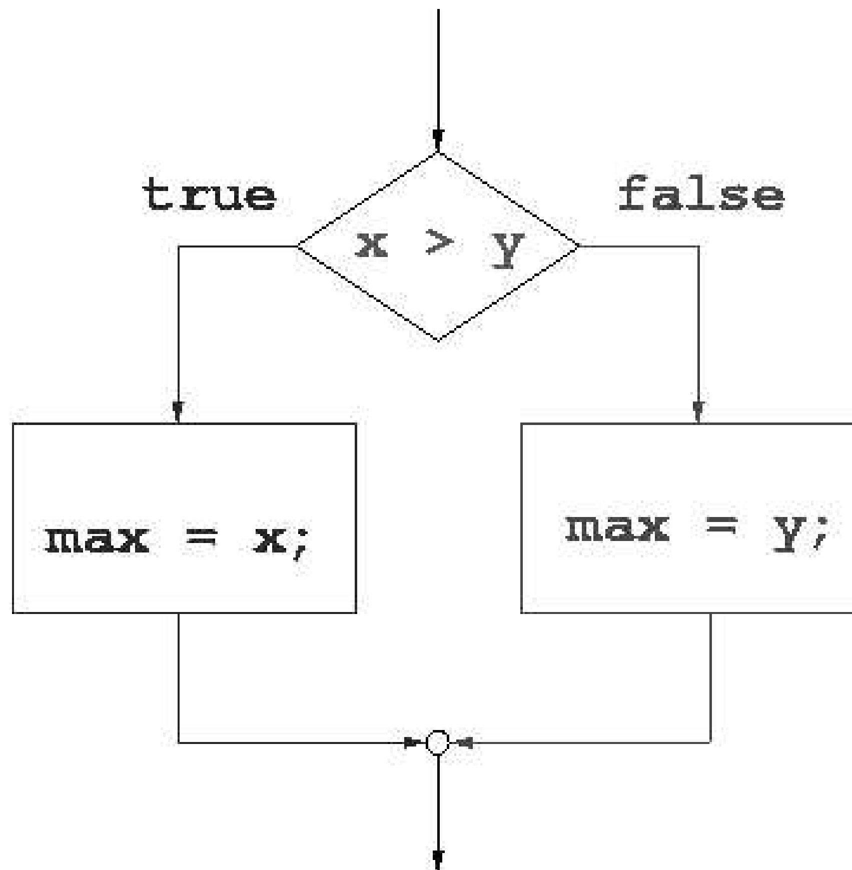
This...

...is the same as this

```
a && (b || c)
```

```
a && b || a && c
```

This...	...is the same as this
`a || true`	`true`
`a && true`	`a`
`a || false`	`a`
`a && false`	`false`
`a == true`	`a`
`a == false`	`! a`
Let's face it: `booleans` can make you dizzy.	Yes, but they are clearly necessary.
For example, the programs we have seen so far are fairly inflexible.	Except for variations in the input they work the same way with every program run.
One of the essential features of nontrivial computer programs is the ability to make decisions...	...and to carry out different actions, depending on the nature of the inputs.
With `booleans` one can program simple and complex decisions.	Learning that will greatly increase our expressive power in Java.
In some of the previous assignments we went to great length to either avoid...	...or fake (or, simulate) decisions by building them into clever formulas.
Being able to make decisions would greatly simplify those programs.	The `if/else` statement is used to implement a decision in a program. The `if/else` statement has three parts: • a *test* (a `boolean` expression), • a *then* branch, and • an *else* alternative.
If the test succeeds, the body of the *then* branch,	...also known as the body of the `if` statement,
...is executed. Here's an example, as a flowchart:	This is from one of the problems of last week.



And in Java:

```
if ( x > y ) { max = x; }
else { max = y; }
```

The condition is red, the body of the `if` statement is blue, while the body of the *else* alternative appears in green. (OK, I know we don't have colors here but I'm sure you know which is which—plus, notice the correct spelling!).

A statement such as

```
max = x;
```

is called a *simple* statement.

A conditional statement, such as:

```
if (x > 0) { y = x; }
else { y = -x; }
```

is called a *compound* statement.

By the way, this last compound statement could be replaced by (as it's equivalent to):

```
y = Math.abs(x);
```

I know, but that's only because it's so simple.

Our programs remain essentially *sequences* of statements, we just allow compound statements, such as `if` statements, in.

But they (at least) become two-dimensional.

Quite often the body of an `if` statement consists of multiple statements that must be executed in sequence whenever the test is succesful.

These statements must be grouped together to form *a block statement* by enclosing them in braces:

- `{` and
- `}`.

Also, while an `if` statement must have a *test* and a *body*, the `else` alternative is optional.

I want to see examples.

Here's one, a bit contrived:

```
if (x < y) {
  temp = x;
  x = y;
  y = temp;
}
```

We assume, of course, that `x`, `y`, and `temp` have been declared, and that `x` and `y`, at least, have been initialized already.

Can you briefly say what the code is doing?

It makes sure that of the two values the larger one is always in `x`.

Very good. What were we saying about braces?

They group statements together.

What if we drop them?

Then the code no longer works as intended.

So what is the syntax of an `if` statement?

The body of an `if` statement (or an `else` alternative) must be *a statement* (just one).

But it can be:

- a *simple* statement
- a *compound* statement (such as another `if` statement), or
- a *block* statement

It's good to get into the habit of using braces (and thus block statements) all the time.

Yes, as we will see when we get to exercises, shortly.

I can hardly wait. But first, let's analyze the `if` statement closer, and look at what makes a *test*.

Its outcome is either `true` or `false`.

In many cases the test compares two values.

Comparison operators such as `<=` (read "less than or equal") are called *relational operators*.

Java has six relational operators.

Java	Math	Description
`>`	$>$	Greater than
`>=`	$\geq$	Greater than or equal
`<`	$<$	Less than
`<=`	$\leq$	Less than or equal
`==`	$=$	Equal
`!=`	$\neq$	Not equal

The == operator is initially confusing to most newcomers to Java.

In Java, the = symbol already has a meaning, namely assignment.

The == denotes equality testing:

```
a = 5; // assign 5 to a
if (a == 3) // tests whether a equals 3
  System.out.println("a is equal to 3");
else
  System.out.println("a is not equal to 3");
```

You will have to remember to use == for equality testing and to use = for assignment.

Floating point numbers have only a limited precision, and calculations can introduce roundoff errors.

That means we need to be careful when we want to test if two floating point quantities are representing the same thing.

Here's an example:

```
double r = Math.sqrt(2);
if (r * r == 2)
  System.out.println(r * r + " == 2");
else
  System.out.println(r * r + " != 2");
```

Unfortunately such roundoff errors are unavoidable.

In most circumstances it does not make a lot of sense to compare floating point numbers exactly.

Instead we should test whether they're *close enough*.

That is, the absolute value of their difference should be less than some threshold.

Mathematically, x and y are close enough if

$$|x - y| \leq \epsilon$$

...for a very small number, ϵ.

Greek letter *epsilon* (ϵ) is commonly used to denote a very small quantity.

It is common to set ϵ to 10^{-14} when comparing `double` numbers.

However, this is not always good enough.

Indeed, if the two numbers are very big, then one can be a roundoff of the other even if their difference is much bigger than 10^{-14}.	To overcome this problem we need to normalize: we divide by the magnitude of the numbers before comparing how close they are.
So x and y are close enough if $$\frac{\lvert x-y\rvert}{\max(\lvert x\rvert,\lvert y\rvert)} \leq \epsilon$$	And to avoid division by zero it is better to test whether $$\lvert x-y\rvert \leq \epsilon \max(\lvert x\rvert,\lvert y\rvert)$$

In Java, this is:

```
Math.abs(x - y) <= EPSILON * Math.max(Math.abs(x), Math.abs(y))
```

OK, I think I understand how I test numbers (integers or floating point) for equality.

What else can we test for equality?	How about `String`s?
To test whether two strings are equal to each other,	...that is, that their contents is the same...
...one must use method `equals`.	Why not use == like for numbers?
`String`s are objects.	And so are `Rectangle`s.
If you compare two object references with the == operator, you test whether the references refer to the same *object*.	That's because you check to see whether the two locations contain the same thing.
The *exact* same thing.	Which is in each case an address, to an actual object.

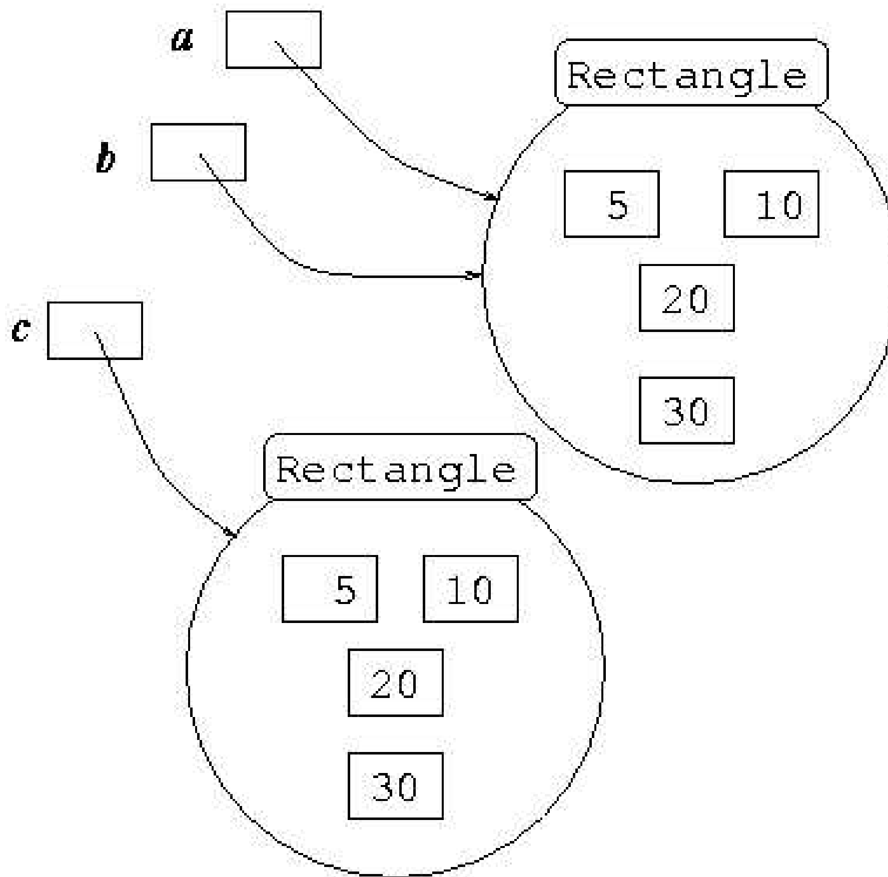
Let's see some examples.

```
Rectangle a = new Rectangle(5, 10, 20, 30);
Rectangle b = a;
Rectangle c = new Rectangle(5, 10, 20, 30);
```

The comparison `a == b` is `true`.

Both object variables refer to the same object.	But the comparison `a == c` is `false`.

The two object variables refer to different objects.



	It does not matter that the objects have identical contents.
You can use the `equals` method to test whether two rectangles have the same contents.	Thus `a.equals(c)` is `true`.
And so is `c.equals(b)` obviously.	Same with `String`s, so we will have to remember to use `equals` for string comparison.
In Java letter case matters. Thus `"harry".equals("HARRY")` evaluates to `false`.	But `"harry".equalsIgnoreCase("HARRY")` evaluates to `true`.
Even if two strings don't have "identical" contents we may still want to know the relationship between them.	The `compareTo` method compares strings in dictionary order.
If `string1.comparesTo(string2) < 0`	...then `string1` comes before `string2` in dictionary order.

If `string1.comparesTo(string2) > 0`	...then `string1` comes after `string2` in dictionary order.
If `string1.comparesTo(string2) == 0`	...then the two strings have identical contents.
You should look this method up in class `String`.	Actually the dictionary ordering used by Java is slightly different from that of a normal dictionary.
Java is case-sensitive and sorts characters by listing numbers first, then uppercase characters, then lowercase characters.	For example `1` comes before `B` which comes before `a`.
And the space character comes before all other characters.	Can we describe the comparison process a little bit in greater detail?
When comparing two strings, corresponding letters are compared until one of the strings ends or the first difference is encountered.	If one of the strings ends, the longer string is considered the later one.
If a character mismatch is found, compare the characters to determine which string comes later in the dictionary sequence.	The process is called *lexicographic* comparison.
That's why `"car"` comes before `"cargo"`,	And `"cathode"` comes after `"cargo"` in lexicographic ordering.
Time for a break.	I sure think so.
And some exercises too.	Yes, but the break *first*, please.
OK, here's what we'll do: we'll put the exercises into the break altogether.	And combine the *useful* with the *necessary*.
The text of the problem is always the same.	I know it already: " *What is the output produced by the following snippets of code when embedded in a complete program.*"

Let's see the snippets.

Snippet 1:

Easy. Draw a diagram.

```
int x = 3;
if (2 > x)
  System.out.print(1);
else
  System.out.print(2);
if (x < 2)
  System.out.println(3);
System.out.print(4);
```

Snippet 2:
 	
`int x = 3;`
`if (x > 5) {`
`  if (x < 10)`
`    System.out.print(1); }`
`else`
`  System.out.print(2);`
`  System.out.print(3);`
	Messy. The curly braces change everything.

What if you take them out?	The diagram changes significantly.
And you have experienced a *dangling* else.	That's right. (Also, your indentation was a bit misleading.)

Snippet 3:

```
int x = 3;
if (x > 0) System.out.print(x + 1);
else if (x > 1) System.out.print(x);
else if (x > 2) System.out.print(x - 1);
else if (x > 3) System.out.print(2 * x);
else System.out.print(x * x);
```

Easy. Diagram it.

Snippet 4 (and last):

```
int x = 3;
if (x > 0) System.out.print(x + 1);
else if (x > 1) System.out.print(x);
else if (x > 2) System.out.print(x - 1);
else if (x > 3) System.out.print(2 * x);
else System.out.print(x * x);
```

Remove the boxed `else`'s.	Who would ever do that in a program?
Nobody. It's for practice.	Messy again. You have to redraw everything.
I agree it's messy, but is it *hard*?	No. Is this the last one?
Yes.	Can we do a *reasonable* example now?

OK, here's `Nineteen` from the first set of problems.

```
/* Solution to problem nineteen in the first problem set. Use
   ConsoleReader from lab notes 2 as explained. The trick here
   (as hinted in the text) is to transform a number for a month in
   a position (index) in the string where the month name is starting,
   all names being made of the same length, and then concatenated
   together in one final string.
 */
public class Nineteen {
    public static void main(String[] args) {
        String monthNames = "January   February  March     " +
                            "April     May       June      " +
                            "July      August    September " +
                            "October   November  December  "    ;
        // open a connection with the keyboard
        ConsoleReader console = new ConsoleReader(System.in);
        // greet the user, and ask for input
        System.out.println("Please enter a month number from 1 to 12.");
        // get month name
        int month = console.readInt();
        // report the name of the month
        System.out.println(
            monthNames.substring("September ".length() * (month-1),
                                 "September ".length() * month));
        // formula uses the length of the longest name
    }
}
```

Here is it with `if` statements:

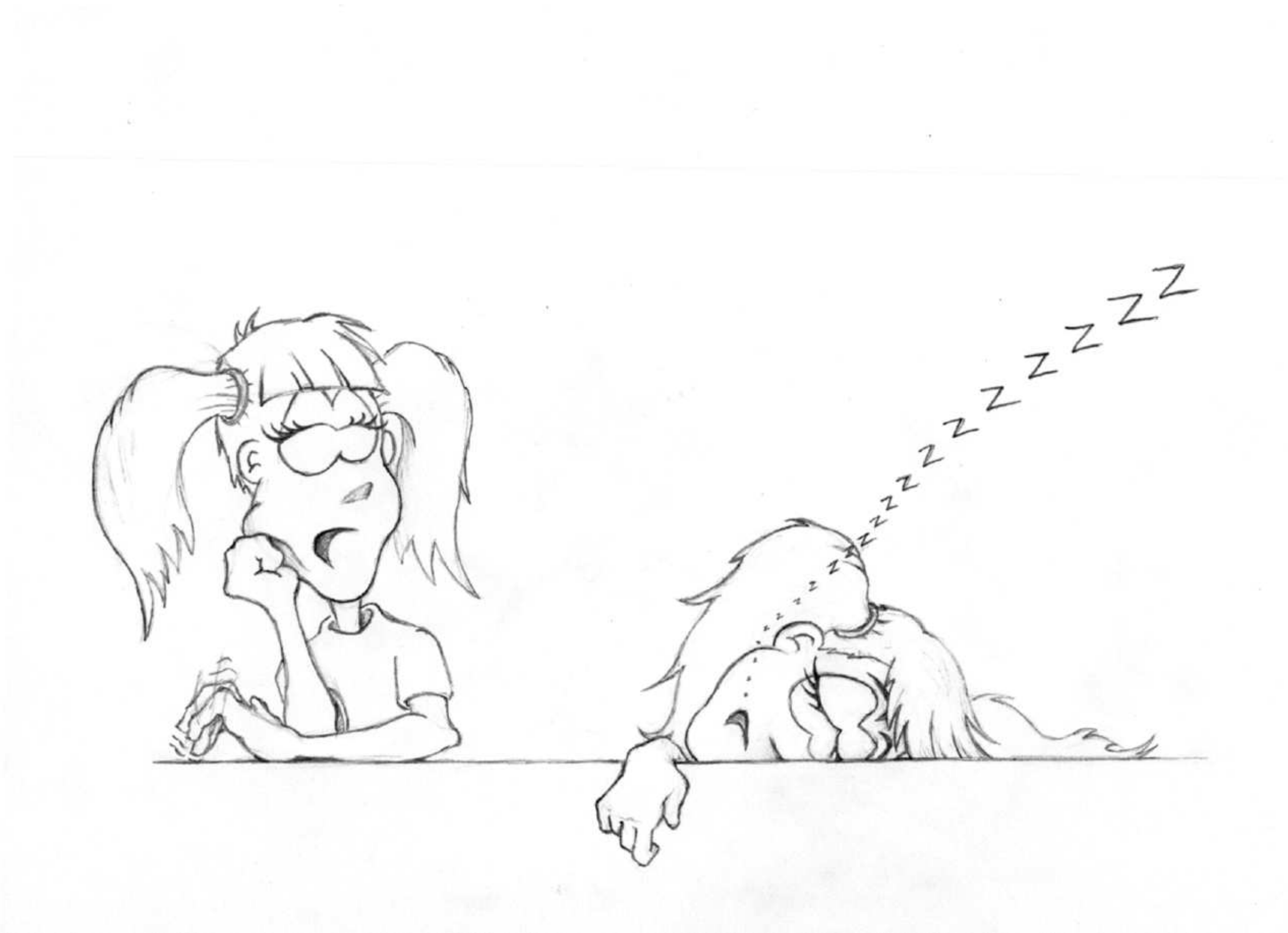
```
public class P19 {
    public static void main(String[] args) {
        ConsoleReader console = new ConsoleReader(System.in);
        System.out.println("Please enter a month number from 1 to 12.");
        int month = console.readInt();
        if      (month == 12) System.out.println("December");
        else if (month == 11) System.out.println("November");
        else if (month == 10) System.out.println("October");
        else if (month ==  9) System.out.println("September");
        else if (month ==  8) System.out.println("August");
        else if (month ==  7) System.out.println("July");
        else if (month ==  6) System.out.println("June");
        else if (month ==  5) System.out.println("May");
        else if (month ==  4) System.out.println("April");
        else if (month ==  3) System.out.println("March");
        else if (month ==  2) System.out.println("February");
        else if (month ==  1) System.out.println("January");
    }
}
```

I thought we agreed to use block statements (with curly braces) for the bodies of `if` statements and `else` alternatives *all the time*.	Yes, but just for once I wanted to keep the code somewhat shorter.
Well, then, just for once, I have two more exercises.	OK, I will remember to put braces from now on, *always*.
Too late.	Show me the first exercise.

Here it is:
 	
`if (false && false || true) {`
`  System.out.print(false);`
`} else {`
`  System.out.print(true);`
`}`
	Can't be `true`!

Snippet 2:
 	
`if (false && (false || true)) {`
`  System.out.print(false);`
`} else {`
`  System.out.print(true);`
`}`
	I can see the difference.

I'm sure you do.	That's probably `true` or `false`.



Classes

Introduction to user-defined types. Classes.

You have now learned about the number and string data types of Java.	Although it is possible to write interesting programs using nothing but numbers and strings, most useful programs need to manipulare data items that are more complex and more closely represent entities in the real world.
Examples of these data items are bank accounts, employee records, graphical shapes, and so on.	The Java language is ideally suited for designing and manipulating such data items, or *objects*.
In Java, you define *classes* that describe the behaviour of these objects. (Classes are *blueprints*).	You will now learn how to define classes that describe objects with very simple behaviour. This will be a very good start.
Let's create a simple class that describes the behaviour of a *bank account*.	Before we can describe what a bank account is in Java we need to be clear (in plain English) what *we* mean by it.
In other words we have to sell it first.	Exactly.
Well, consider what kinds of operations you can carry out with a bank account. You can:...	• deposit money • withdraw money • get the current balance
Sounds like a bank account to me.	In Java these operations are expressed as *method calls*.
So the set of methods that an object of type...	...`BankAccount` (sounds like a good name to me)
...will support, could be...	• `deposit` • `withdraw` • `getBalance`

That's what `BankAccounts` do best!

OK, before we get too euphoric we need to imagine such an object in action.

Objects are *agents*.

Exactly. If they have methods (and most objects do) they have a behaviour, defined by what their methods can do. Now, how do you envision `BankAccounts` in action?

To start with, opening a new bank account should look like this:

```
BankAccount myChecking = new BankAccount();
```

. . . and the initial balance should be zero. How do you put, let's say, $1,000.00 in your account when you create it?

It would be something like this:

```
myChecking.deposit(1000.00);
```

Isn't this very similar to `translate` for `Rectangles`?

It is, indeed, the very same thing: we're translating the balance, in one dimension.

How would you check your current balance?

I'd ask `myChecking` to report the balance, which I could then print:

```
System.out.println(myChecking.getBalance());
```

Isn't this very similar to printing the `length` of a `String`?

It sure is. As for the third method, I just realized I don't even need it. . .

How come? What if you want to withdraw $300.00 from your checking?

I can already express that as:

```
myChecking.deposit(-300.00);
```

Then it won't be too hard though to come up with an extra method. . .

.. that could be called as follows:

```
myChecking.withdraw(300.00);
```

This should make more sense to the user of your class. Are we done now?

I think we are. These three methods form the complete list of what you can do with an object of type `BankAccount`, . . .

. . . at least from the point of view of what we wanted in a bank account. We could certainly add methods to compute interest, etc., and enhance our model (or design), but for now. . .

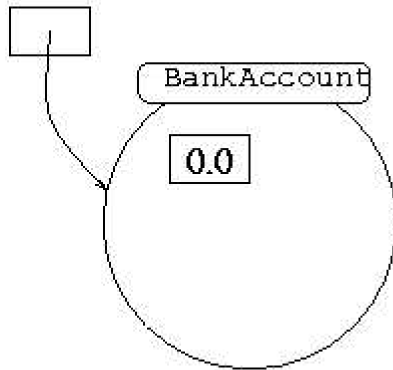
... these three methods are more than enough for what we have in mind with this class.	We want to *implement* it and see it used in a Java program.
Nothing less. For a more complex class, it takes some amount of skill and practice to discover the appropriate behaviour that makes the class useful and is reasonable to implement.	We will learn more about this important aspect when we get to the chapter on "Object oriented design".
The behaviour of our `BankAccount` class is very simple, and we have described it completely. We can now implement it.	Yes, let's go for it.
Although I think it's worth pointing out one thing, before we even write a single line of code of implementation...	What is it?
In our description of the methods we have used objects of type `BankAccount` without knowing (or caring too much to know) about their implementation, which we are only now about to describe.	Yes, that's an important aspect of object-oriented programming. But now, that we are completely clear on how to use objects of the `BankAccount` class, we really need to get started...
... to describe the Java class that implements the described behaviour.	I can get us started and in the following way:

```
public class BankAccount
{ ...
  ...
}
```

What do we put inside?	Methods and data.
I don't understand what you mean by data.	It's what makes tiggers remember where they have bounced last. Do you remember `Rectangles`?
I certainly do. Their diagrams were always containing four slots, in which we were writing the current values of their `x`, `y`, `width`, and `height`.	That's the data that they have, which helps them remember where they are, and how big they are. Could you draw a diagram to illustrate what's happening after we create a new `BankAccount`?
The code would be...	... and the diagram?

```
BankAccount mySavings = new BankAccount();
```

Take a look on the next page.



Very good. What's that in which you put the initial balance of 0 (zero)?

It's a location, like we've seen for `Rectangles`. It looks like a variable, probably of type `double` or of a similar or related type.

Each object must store its current state; for objects of type `BankAccount` the state is the current `balance` of the bank account.

Each object stores the state in one or more *instance variables* .

An instance variable declaration consists of the following three parts:

a) an *access specifier*

...such as `private`

b) the *type* of the variable

...such as `double`

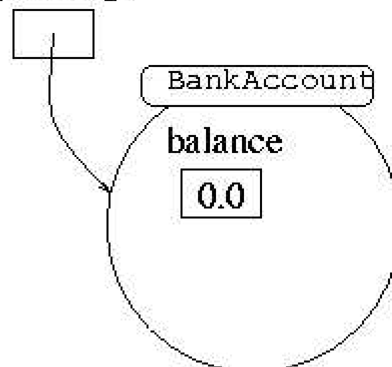
c) the *name* of the variable

...such as `balance`.

So far we have:

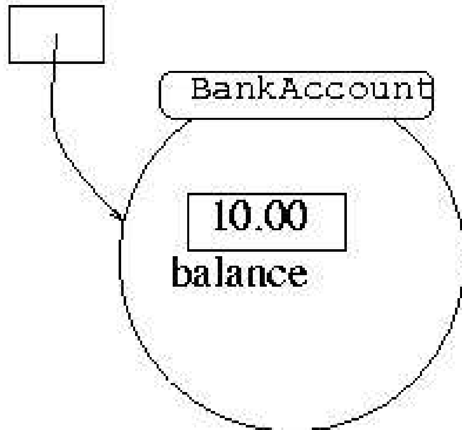
```
public class BankAccount
{ private double balance;
  ...
}
```

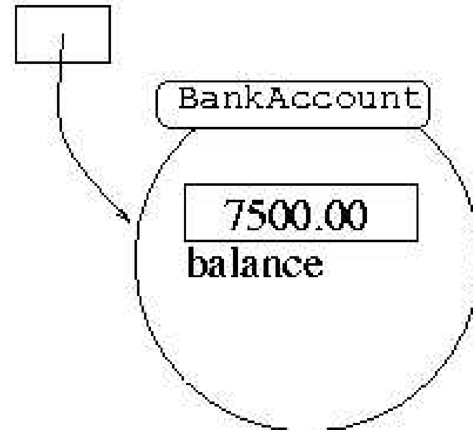
...and the diagram:



The `balance` field is all we need, as far as data goes.

Each object of a class has its own copy of the instance variables.	We've seen that with `Rectangles` and it's the same with `BankAccounts`.





Each object has its own `balance` field.	And in our example they hold different values.
Instance variables are generally declared with the access specifier `private`.	That means that they can be accessed only by methods of the same class. In particular, the `balance` variable can be accessed only by the `deposit`, `withdraw`, and `getBalance` methods.
Let's write these methods in Java.	OK. Let's start by *defining* them.
Yes, let's start by describing their headers.	A method header consists of the following parts:
a) an *access specifier*	...such as `public`
b) the *return type* of the method	...such as `double` or `void`
c) the *name* of the method	...such as `deposit`
d) a list of the *parameters* of the method	...describing the method's initial data. Let's consider each of these parts in detail.
The *access specifier* controls which other methods can call this method.	Most methods should be declared `public`. This way, all other methods in your program can call them. (Ocasionally, it can be useful to have methods that are not so widely callable, but we will look at that later).

The *return type* is the type of the value that the method computes.	For example, the `getBalance` method returns the current account balance, which will be a floating-point number, so its return type is `double`. On the other hand the `deposit` and `withdraw` methods don't return any value. They just update the current balance but don't return it.
To indicate that a method does not return a value, use the special type `void`.	Both the `deposit` and `withdraw` are declared with return type `void`.
The *parameters* are inputs to the method.	The `deposit` and `withdraw` methods each have one parameter: the amount of money to deposit or withdraw.
You need to specify the type of the parameter, such as `double`, and the name for the parameter, such as `amount`.	The `getBalance` method has no parameters. In that case, you still need to supply a pair of parentheses `()` behind the method name.
If a method has more than one parameter, you separate them by commas.	And once you have specified the method header, you must supply the implementation of the method, in a block that is delimited by braces (`{...}`).
Putting all this together we have:	Looks good.

```
public class BankAccount
{ public void deposit(double amount)
  { ...
    ...
  }
  public void withdraw(double amount)
  { ...
    ...
  }
  public double getBalance()
  { ...
    ...
  }
  private double balance;
}
```

We now must provide an implementation for each method of the class.	The implementation of these three methods is straightforward: when some amount of money is deposited or withdrawn, the balance increases or decreases by that amount.

```
public class BankAccount
{ public void deposit(double amount)
  { balance = balance + amount;
  }
  public void withdraw(double amount)
```

```
    { balance = balance - amount;
    }
    public double getBalance()
    { return balance;
    }
    private double balance;
  }
```

The `getBalance` simply *returns* the current balance.	But wait: how can we use `balance` and `amount` without declaring or initializing them, as we said we should be doing *every* single time?
They are not method (local) variables. `amount` is a method parameter. When the method is called the value that will be passed to the method will be placed in a location called `amount` of type `double` that represents the initial data of the method. So `amount`, as a method parameter is like a variable...	...that is, a symbolic name that refers to a memory location...
...and will have been initialized when the method is called.	How about `balance`?
It is also a variable, only not a method variable. It is an instance variable, visible to every instance method (such as `deposit`, `withdraw`, and `getBalance`). For these methods it is a (somewhat) *global* variable (which they *share*).	Instance variables are initialized when the object to which they belong is constructed. In this respect they are really different from method variables.
They are initialized with default values, that depend on their types.	Numbers, for example, are initialized with a value of 0 (zero).
Of course, you can also initialize them in a certain, customized way, if you define *constructors*.	We will talk about constructors tomorrow.
Meanwhile let's see what we mean by `return`.	Well, when you ask someone a question, and you pay for the question to be answered, ...
...you expect an answer, ...	...in `return`.
When a function, that is supposed to provide a value as an answer, has the answer ready...	...(by computing an expression, or by referring to a location where the final result has been stored, ready to be reported)...
...and when it wants to report it, ...	...to whoever called it in the first place...
...it simply states that it's ready to finish and `returns` the value, ...	...after which the function (or method, as it's known in Java) ends.
Interesting. So this is basically *syntax*.	Very much so.

OK, can we see the full program?

Yes, you need to have two classes.

One is the class we designed.

```
public class BankAccount
{ public void deposit(double amount)
  { balance = balance + amount;
  }
  public void withdraw(double amount)
  { balance = balance - amount;
  }
  public double getBalance()
  { return balance;
  }
  private double balance;
}
```

While the other one has the `main`.

```
public class Experiment
{ public static void main(String[] args)
  { BankAccount a = new BankAccount();
    BankAccount b = new BankAccount();
    a.deposit(200);
    b.deposit(300);
    System.out.println(a.getBalance());
    System.out.println(b.getBalance());
    a.withdraw(100);
    b.withdraw(200);
    System.out.println(a.getBalance());
    System.out.println(b.getBalance());
  }
}
```

Place them in the same directory…

…compile and run them, and enjoy!

Not many problems couldn't be *much* harder,

…I think. I agree. By the way, …

"I hope you're not much tired, are you?"

"Nohow. And thank you very much for asking!"

Constructors and Instance Variables

Classes, objects, constructors.

There is only one remaining issue with the `BankAccount`.	We need to define the default constructor.
Why do we need to talk about the default constructor?	Because that's the one we're using now, not having defined any constructor whatsoever yet.
Constructors are not methods, but they are used to create instances of the class (objects).	Many classes have more than one constructor.
The purpose of a constructor is to initialize the instance variables of the object.	The code for a constructor sets the initial state of the object.
When a `BankAccount` object comes into existence it will have an initial state with the current balance being 0 (zero).	So there's just one constructor for the class.
Since it does not take any arguments it is called a no-arg constructor.	Its purpose is to initialize the instance variables of a bank account object when the object is created.
Objects of type `BankAccount` only have one instance variable: their `balance`.	If you do not initialize an instance variable that is a number it will automatically be initialized to 0 (zero) by Java, befor the constructor even comes into play.
In this regard, instance variables act differently than local variables!	By local variables you mean "*variables declared in methods, such as in* `main`", right?
Yes. Those have to be initialized by the programmer before they are used.	It's not the same with instance variables.
Instance variables are set by Java to a default value.	Local variables will not.
No they won't be initialized by Java. Instance variables, however, will be initialized by Java if you don't initialize them (as a programmer).	OK. If the `balance` of a new account will be set to 0 (zero) even before the constructor starts working, then the constructor need not do anything.

Yes. And Java will always provide (by default) a no-arg constructor that doesn't do anything, for every class that you define.	So if you don't define any constructor you will be given one, by default.
Yes, and if you define at least one, those that you define are your constructors.	How do we write a constructor?
They are essentially initialization procedures so they look very similar to methods. They have a header, and a method body.	Their header contains an access specifier, but not a return type.
Their name is always that of the class.	Like methods they have a list of parameters: named locations of a certain type, in which their initial information is placed.
That is, the arguments.	Indeed. Can I see one?

Here's the one that you get by default, if you don't specify any constructor.

It's the one in blue. Note that the body of the constructor is empty.

```
public class BankAccount {
  public BankAccount() {
    // nothing
  }
  public void deposit(double amount) {
    balance = balance + amount;
  }
  public void withdraw(double amount) {
    balance = balance - amount;
  }
  public double getBalance() {
    return balance;
  }
  private double balance;

}
```

Since this is the default constructor that means I get it for free.	Indeed, but you might actually write it anyway to not forget that you can use it.
Can I define a second constructor?	How would you want to use it?
I'd like to create an account with an initial sum of money in it, like this: `BankAccount m = new BankAccount(300.00);`	Yes you can. You will only need to set `balance` to the initial value inside the constructor.

Is that it?

```
public BankAccount(double initial) {
  balance = initial;
}
```

Yes, can you describe it a little?

It is used to create a bank account with an initial balance.	When you call it you need to specify that amount, like you did when you showed me the way you intended to use it.
A constructor *looks* like a method, only the header does not have a return type, and the name of a constructor is the name of the class.	The rest of it is just like a method.
Yes, so I defined a formal parameter `initial` which must be of floating-point type (that is, with a fractional part). I chose `double` for the type of the formal parameter.	So in your previous example this constructor gets called to create a new object, and the initialization steps start by storing `300.00` in a location of type `double` by the name `initial`.
Yes, and in its body I use `initial` to copy its value in `balance`.	Very well. You could have used it in a more involved way, but there was no need for you to do that.
There's a tricky rule in Java about the default no-arg constructor.	We mentioned it above, but in an implicit way.
We can avoid mentioning it here by stating another rule, that is easier to state (and remember).	Always declare <u>all</u> the constructors that you need.
And how's the actual rule?	The default constructor is provided by default when there are no constructors specified. If you specify at least one constructor, the default constructor no longer is provided and if you need it you need to write it explicitly in the class.

I see, so this class definition won't let me create bank accounts with an initial value of 0 (zero)?
 	
`public class BankAccount {`
`  double balance;`
`  public BankAccount(double initial) {`
`    balance = initial;`
`  }`
`}`
	Not directly.

So I can't say `BankAccount m = new BankAccount();`	No, but you can create it this way: `BankAccount m = new BankAccount(0.0);`

Well, can we put the class to work?

Sure, we did that last time, we can do it again. Here's a different test program though:

```
public class BankAccountTest {
  public static void main(String[] args) {
    BankAccount account = new BankAccount(10000);
    final double INTEREST_RATE = 5;
    double interest;
    // compute and add interest for one period
    interest = account.getBalance() * INTEREST_RATE / 100;
    account.deposit(interest);
    System.out.println("Balance after the first year is $" + account.getBalance());
    // add interest again
    interest = account.getBalance() * INTEREST_RATE / 100;
    account.deposit(interest);
    System.out.println("Balance after the second year is $" +
                       account.getBalance());
  }
}
```

And the class is still the same as last time, with the two constructors added,

. . . the no-arg empty constructor and the one that initializes the balance to a certain initial value, that is specified when you call the constructor.

Yes, here it is:

```
public class BankAccount {
  private double balance; // instance variable, the account balance
  public BankAccount() { // the no-arg empty constructor

  }
  public BankAccount(double initial) { // another constructor
    balance = initial;
  }
  public void deposit(double amount) { // instance method deposit
    balance = balance + amount;
  }
  public void withdraw(double amount) { // instance method withdraw
    balance = balance - amount;
  }
  public double getBalance() { // instance method getBalance
    return balance;
  }
}
```

Once again to see this in action I need to copy the code in two files, one for the bank account class and the other one for the bank account test class (that has the main method).

Then you compile them and run the test class.

Can we summarize now?	We sure can.
I have a summary with two short examples.	Let's see them.
Objects are entities that can have memory and specific behaviour. Their memory is represented by variables that they have inside and their behaviour is defined by actions that they know how to perform	... that is, the methods that are associated with those objects.
All objects of the same kind, that have the exact same *structure*, make up a class.	In fact, in programming it's always the other way around: one first defines a class,
which describes how that particular class of objects will look and behave (what methods they have,)	... then one creates as many objects *(of that kind)* as needed
... and lets them loose,	... thus running the program.
To better clarify instance variables and instance methods let's look at two examples.	Each one will resemble a short play (as in a stage representation of an action or a story).
Our *dramatic compositions* will be simple, since we will abstract away all the unwanted details.	The titles of the plays will be: • `Sports`, and • `Babies`.
Better than "*You are old Father William*" already.	OK, let's look at the first one.
A Hoosier basketball fan's simple to describe:	... she cheers, by shouting *'Go Hoosiers!'* when she feels like cheering for the former team of Bob Knight, and that's the end of it.
Write a short program (a play) that presents three Hoosier fans cheering for the IU Hoosiers,	.. each fan cheering once, and in no particular order.
Here's how the program should behave:	The output of the program is in `blue`.

```
tucotuco.cs.indiana.edu% javac Sports.java
tucotuco.cs.indiana.edu% java Sports
Go Hoosiers!
Go Hoosiers!
Go Hoosiers!
tucotuco.cs.indiana.edu%
```

At a basketball game the noise is so loud that you don't know who is cheering and when.	The crowd is anonymous, more or less. Here's the object oriented implementation of this play:

```
public class Sports {
  public static void main(String[] args) {
```

```
    Hoosier a = new Hoosier();
    Hoosier b = new Hoosier();
    Hoosier c = new Hoosier();
    a.cheer();
    b.cheer();
    c.cheer();
  }
}

class Hoosier {
  void cheer() {
    System.out.println("Go Hoosiers!");
  }
}
```

Just a quick question: for all practical purposes *cheering* here essentially means *printing*, right?	Printing, or displaying a big printed note,
. . . which reads (in this case):	"`Go Hoosiers!`".
So we see that a `Hoosier` is an object that knows only one thing: to cheer,	. . . and in only one way.
The objects' behaviour is defined by their methods, and since each object is	. . . an instance of a class the methods themselves are called *instance methods*.
OK. Here's the second play, `Babies`.	We won't have much time for that.

Here's the play:

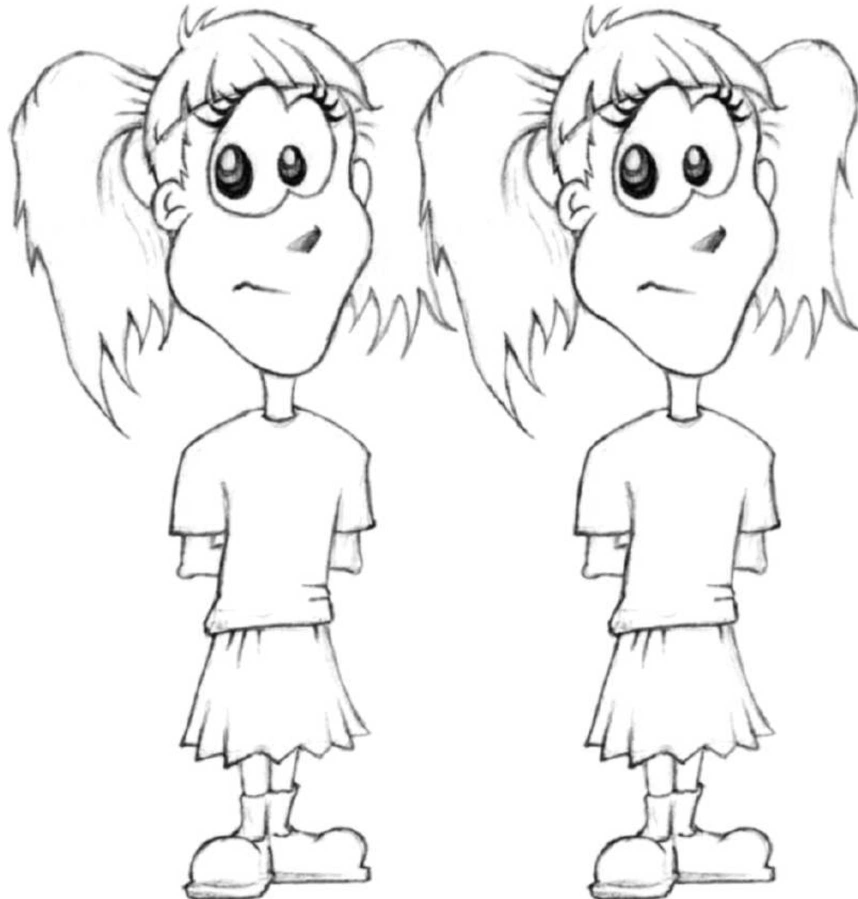
```
tucotuco.cs.indiana.edu% javac Babies.java
tucotuco.cs.indiana.edu% java Babies
Alice: Hello, my name is Alice
Susan: Hello, my name is Susan
Jimmy: Hello, my name is Jimmy
tucotuco.cs.indiana.edu%
```

Looks good to me.

Here's the screenplay and the cast.

```
public class Babies {
  public static void main(String[] args) {
    Baby a = new Baby("Alice");
    Baby b = new Baby("Susan");
    Baby c = new Baby("Jimmy");

    a.talk();
    b.talk();
```

```
    c.talk();

  }
}

class Baby {
  String name;                // instance variable

  Baby(String givenName) {    // constructor
    name = givenName;
  }

  void talk() {               // instance method
    System.out.println(name + ": Hello, my name is " + name);
  }
}
```



Oh, but I think I understand instance variables now. Oh, but I am sure you do.

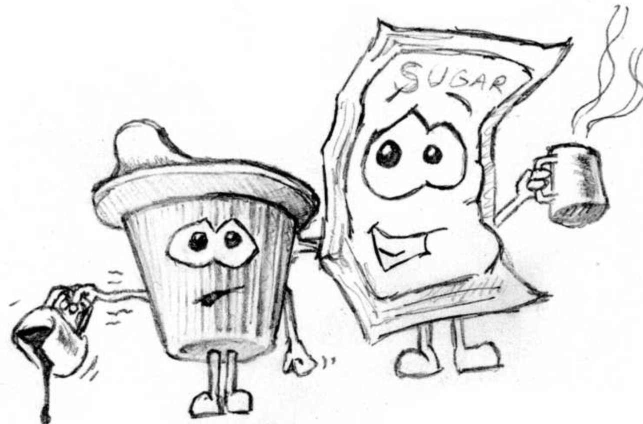
Here's a program:

```
class StrawDispenser {

  final static CAPACITY = 100;

  int balance = StrawDispenser.CAPACITY;

  Straw dispense() {
    this.balance -= 1;
    return new Straw();
  }

  int getBalance() {
    return this.balance;
  }

  void refill() {
    this.balance = StrawDispenser.CAPACITY;
  }
}
```

Admittedly, this looks more like a Bubble Machine (if it *makes* the `Straws` as it goes).

Here's another one:

```
class NumberTwo {
  Sandwich a; // Larry Bird is from Freedom Lick, IN...
  Fries f;
  Drink d;
  NumberTwo(int size, String drink) {
    a = new Sandwich("cheeseburger");
    f = new Fries(size);
    d = new Drink(drink, size);
  }
}
```



Methods

Wrap-up of Classes and Objects material.

Let's go through a set of examples to clarify classes and objects even further.

1. What's this program doing?

   Can you draw a diagram to illustrate what happens when you run it?

   ```
   public class One {
       public static void main(String[] args) {
           Potato p = new Potato();
           Potato q;
           q = new Potato();
           p = q;
       }
   }
   class Potato {

   }
   ```

   How do the `Potato`es get created?

2. What is this next program doing? Can you diagram it? What's new?

   ```
   public class Two {
       public static void main(String[] args) {
           Pair u = new Pair();
           Pair v;
           v = new Pair();
           u.a = 1;
           u.b = 2;
           u.a = u.a + u.b;
           u.b = 1 - u.a;
       }
   }
   class Pair {
       int a;
       int b;
   }
   ```

3. Why does this next program not compile?

   What's wrong with it?

```
public class Three {
    public static void main(String[] args) {
        Pair u = new Pair();
        Pair v = new Pair(1, 2);
    }
}
class Pair {
    int a;
    int b;
}
```

   Can you fix it?

4. What does the next program print and why (or how).

```
public class Four {
    public static void main(String[] args) {
        Calculator m = new Calculator();
        int value = m.fun(3);
        System.out.println(value);
    }
}
class Calculator {
    int fun(int x) {
        int result;
        result = 3 * x + 1;
        return result;
    }
}
```

   Same question if we change the `Calculator` as follows:

```
class Calculator {
    int fun(int x) {
        int result;
        result = g(x) + 1;
        return result;
    }
    int g(int x) {
        int result;
        result = 3 * x;
        return result;
    }
}
```

5. What does the following program print and why (or how)?

```
public class Five {
    public static void main(String[] args) {
        Calculator c = new Calculator();
        int value = c.fun(1) + c.fun(c.fun(2));
        System.out.println(value);
    }
}
class Calculator {
    int fun(int x) {
        int result;
        result = 3 * x + 1;
        return result;
    }
}
```

6. Same question about this one:

```
public class Six {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int value = calc.fun(
                      calc.fun(
                        calc.fun(
                          calc.fun(
                            calc.fun(5)))));
        System.out.println(value);
    }
}
class Calculator {
    int fun(int x) {
        int result;
        if (x % 2 == 0)
          result = x / 2;
        else
          result = 3 * x + 1;
        return result;
    }
}
```

7. What's the output of the following program and why?

```
public class Seven {
    public static void main(String[] args) {
        Oracle a = new Oracle();
        System.out.println(a.odd(5));
        System.out.println(a.odd(6));
        System.out.println(a.odd(7));
        System.out.println(a.odd(8));
        System.out.println(a.odd(9));
    }
```

```
}
class Oracle {
    boolean odd(int n) {
        boolean result;
        if (n % 2 == 0) {
            result = false;
        } else {
            result = true;
        }
        return result;
    }
}
```

8. Let's now review a previous step once again.

   What's the output of this program and why (or how)?

```
public class Eight {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int value = calc.fun(
                      calc.fun(
                        calc.fun(
                          calc.fun(
                            calc.fun(27)))));
        System.out.println(value);
    }
}

class Calculator {
    int fun(int x) {
        int result;
        if (x % 2 == 0)
          result = x / 2;
        else
          result = 3 * x + 1;
        return result;
    }
}
```

9. Now fasten your seat belts and look closely.

   What's the output of this program and why (or how)?

```
public class Nine {
    public static void main(String[] args) {
        Alien x = new Alien();
        int value = x.what(4);
        System.out.println(value);
    }
}
```

```
class Alien {
    int what(int x) {
        int result;
        if (x == 1) {
            result = 1;
        } else {
            result = x + what(x - 1);
        }
        return result;
    }
}
```

10. What's a good name for the method `what`?

11. What's the output of this program?

```
public class Eleven {
    public static void main(String[] args) {
        Alien x = new Alien();
        int value = x.what( 10 );
        System.out.println(value);
    }
}
class Alien {
    int what(int x) {
        int result;
        if (x == 1) {
            result = 1;
        } else {
            result = x + what(x - 1);
        }
        return result;
    }
}
```

12. What's the output of this program and why (or how)?

```
public class Twelve {
    public static void main(String[] args) {
        Alien x = new Alien();
        int value = x.what(10);
        System.out.println(value);
    }
}
class Alien {
    int what(int x) {
        int result;
        if (x == 1) {
            result = 1;
        } else {
```

```
                System.out.println(x);
            result = x + what(x - 1);
        }
        return result;
    }
}
```

13. What's the output of this program and why (or how)?

```
public class Thirteen
    public static void main(String[] args)
        Alien x = new Alien();
        int value = x.what(10);
        System.out.println(value);


class Alien {
    int what(int x) {
        int result;
        if (x == 1) {
            result = 1;
        } else {
            result = x + what(x - 1);
            System.out.println(x);
        }
        return result;
    }
}
```

14. What's the output of this program and why?

```
public class Fourteen {
    public static void main(String[] args) {
        A a = new A();
        a.fun();
        a.fun();
        a.fun();
        System.out.println(a.n);
    }
}
class A {
    int n;
    void fun() {
        n += 1;
    }
}
```

15. What's the output of this program and why?

```
public class Fifteen {
    public static void main(String[] args) {
        Vegetable tomato = new Vegetable();
        tomato.f();
        tomato.f();
        tomato.g();
        System.out.println(tomato.n);
        tomato.g();
        tomato.g();
        tomato.f();
        System.out.println(tomato.n);
    }
}
class Vegetable {
    int n;
    void f() {
        n = n + 1;
    }
    void g() {
        n = n + 1;
    }
}
```

16. What is the output of this program and why?

```
public class Sixteen {
    public static void main(String[] args) {
        Vegetable tomato = new Vegetable();
        Vegetable potato = new Vegetable();
        tomato.fun();
        tomato.fun();
        potato.fun();
        tomato.fun();
        potato.fun();
        potato.fun();
        potato.fun();
        tomato.fun();
    }
}
class Vegetable {
    int n;
    int m;
    void fun() {
        n = n + 1;
        m = m + 1;
        System.out.println("n = " + n + ", m = " + m);
    }
}
```

17. What is the output of this program and why?

```
public class Seventeen {
    public static void main(String[] args) {
        Vegetable tomato = new Vegetable();
        Vegetable potato = new Vegetable();
        tomato.fun();
        tomato.fun();
        potato.fun();
        tomato.fun();
        potato.fun();
        potato.fun();
        potato.fun();
        tomato.fun();
    }
}
class Vegetable {
    int n;
    static int m;
    void fun() {
        n = n + 1;
        m = m + 1;
        System.out.println("n = " + n + ", m = " + m);
    }
}
```

18. Please describe briefly what this picture represents:

instance variable

formal parameter

```
class A {
  int x;
  void fun(int y) {
    int z = 3;
    x = z + y;
  }
}
```

local variable

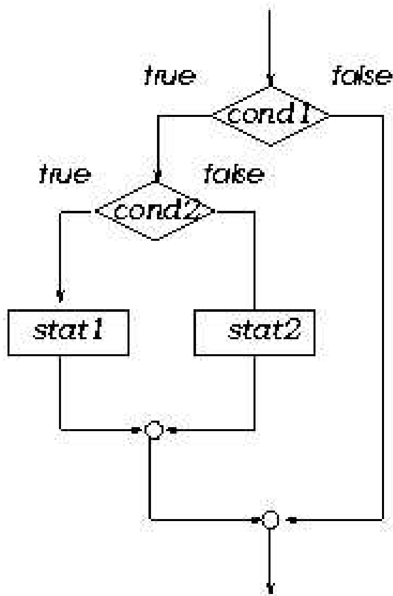
better write it as `this.x`

Define all terms involved, as briefly and completely as you can.

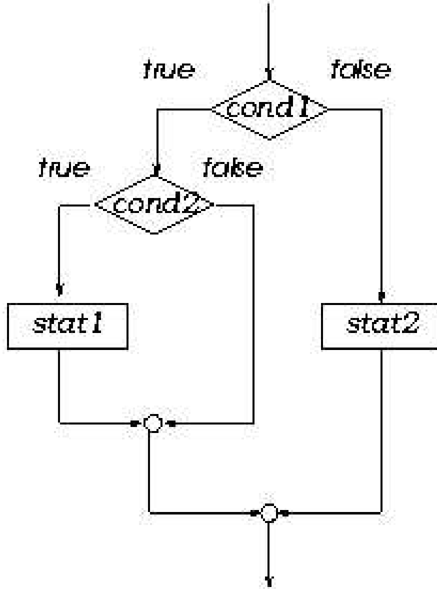
19. Answer the following question: What's `this`?

Decisions

Branches and paths. If statement exercises. (L)oops.

What's the purpose of `if` statements?	It allows us to include *decisions* in our programs.
What do we do with their results?	We can branch our course of action.
How do you code this branching situation? 	Use the space above.
Did you remember to use curly braces?	I try to do this as often as possible, it helps me keep all my branches distinct.
That's good practice.	Indeed, even though sometimes I don't *really* need the brackets.

How would you code this slightly modified situation?



I see there's one *minor* change.

It may look minor on the diagram.

I see... Now I *have to* use brackets.

Yes, there's no way around it.

Otherwise the branches tangle.

In general, in our programs, we have the flowchart clear in mind and we just need to translate it into Java.

But we need to do that with care, as illustrated above, and curly braces are more than just syntactic sugar in Java.

Ocasionally we will have to do the reverse,

... that is, build the diagram out of Java text,

... when we read someone else's code.

Let's see some examples.

OK, here's a bigger one.

I can hardly wait.

Bigger Example One

Assume that `option` is an `int` variable that can only be: `1`, `2` or `3` before each of the following code fragments start executing.

So this is *given.*

Yes. Now the question.

Which of the fragments below set variable `i` to the same value that `option` has?

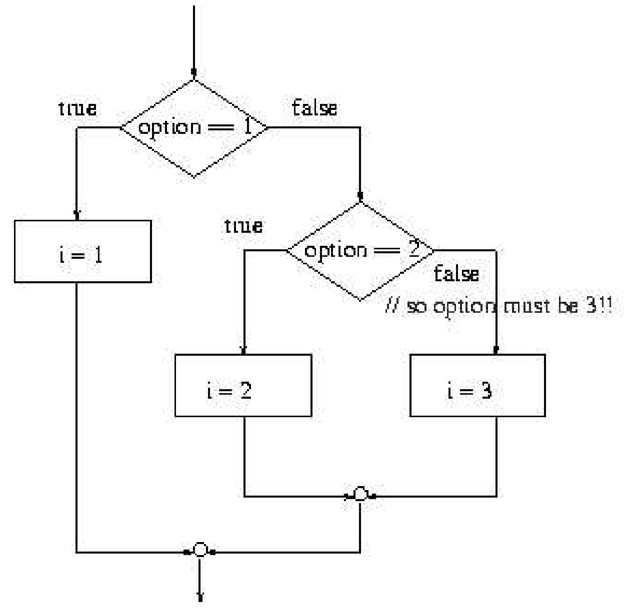
We will look at this kind of problems with the help of diagrams, as announced before.

The (given) code is on the left,	. . . the diagram is on the right, in all examples below.

Code

```
if (option == 1)
  i = 1;
else if (option == 2)
  i = 2;
else
  i = 3;
```

Diagram

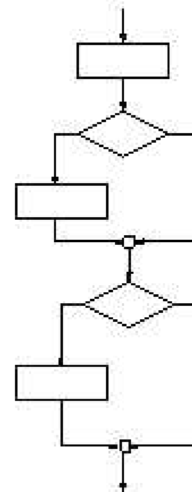


In this particular case it's easy to see that the code sets `i` to the same value as `option`	. . . *given* the assumption about the possible values that `option` can have. We just explore *all* paths.
For the remaining of these notes we only sketch the diagrams here,	. . . and fill them with text in class.

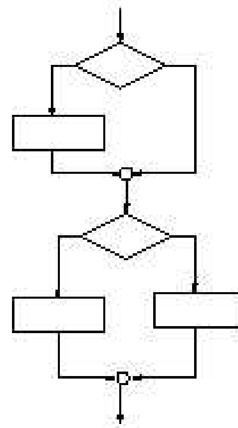
Code

```
i = 1;
if (option >= 2)
  i = i + 1;
if (option == 3)
  i = i + 1;
```

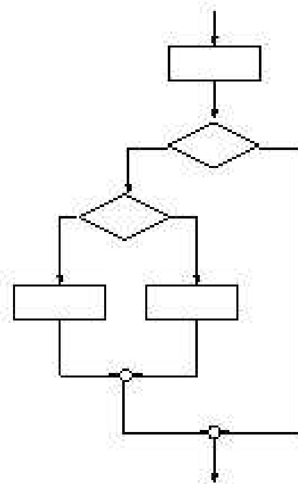
Diagram



Code	Diagram

`if (option == 1)`
`  i = 1;`
`if (option == 2)`
`  i = 2;`
`else`
`  i = 3;`
	

Code	Diagram

`i = 1;`
`if (option > 1)`
`  if (option > 2)`
`    i = 3;`
`  else`
`    i = 2;`
	

Don't forget we still need to provide an answer with each one of the diagrams.

Indeed we need to say whether the code fragments presented are (or not) equivalent to

```
i = option;
```

. . . which is the equivalent code.

If the equivalent code is *that* simple, why do we go through all this trouble, to rephrase a simple assignment statement?

We chose a simple situation to practice *if* statements on it. Don't worry, I have a list of suggested exercises indexed at the end of these notes, below. Those are *real iffy* situations.

Very good. So we keep things simple while we learn the concept, and we use diagrams to reason about these simple, illustrative programs.

Indeed, we rely on diagrams for the time being. With time we won't need them, as you'll be able to see them without drawing them, but for beginners they seem to be more tangible.

Here's the last case:

Code

```
i = 1 if (option == 1);
i = 2 if (option == 2);
i = 3 if (option == 3);
```

Diagram

Incorrect syntax, no diagram.

But, you get a *Picasso*.

Time to move on.

Bigger Example Two

Consider this code fragment

```
if (x > y)
  z = z + 1;
else
  z = z + 2;
```

Which one of the following (code fragments) are equivalent to it? Two code fragments are be considered *equivalent* when they behave in the same way.

Two code fragments are to be considered *equivalent* ...when they behave in the exact same way.

That is they have

- the same output, and
- same sequence of internal states

for identical inputs,

...over their all possible inputs.

Note that the structure of the diagram is as important as what gets written inside the boxes.

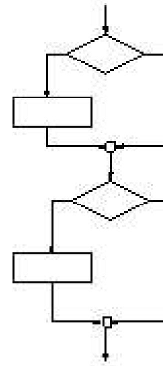
We sketch the diagrams below.

And leave the reasoning to you.

Code

```
if (x > y)
  z = z + 1;
if (x <= y)
  z = z + 2;
```

Diagram



So, what's the answer?

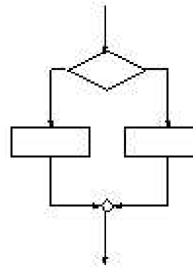
Those paths are not independent, I don't think.

How about this next one, then?

Code

```
if (! (x > y))
  z = z + 2;
else
  z = z + 1;
```

Diagram



This one is *much* easier to think about, no doubt about it. But careful thinking is required when the problem is posed, as it is, with its upside down.

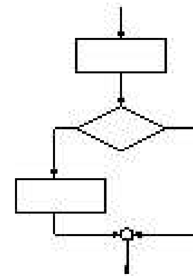
How about this next one, then?

Not harder, but *somewhat* unfamiliar…

Code

```
z = z + 1;
if (x <= y)
  z = z + 1;
```

Diagram



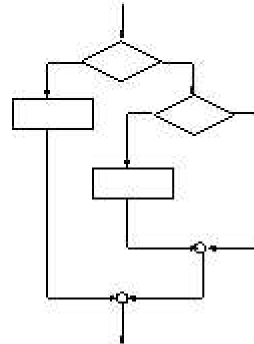
Code has been *factored out.*

How about this next one, then?

Code

```
if (x > y)
  z = z + 1;
else if (x <= y)
  z = z + 2;
```

Diagram



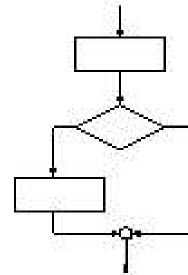
If the last one was *efficient* this one's *redundant*.

How about this next one, then?

Code

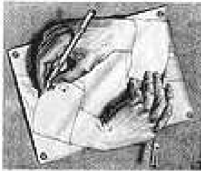
```
z = z + 2;
if (!(x > y))
  z = z - 1;
```

Diagram



The approach looks a bit *contortive* this time.

Like this?



No, that's *circular*, not even *recursive*.

I think it's more like this:

That, my dear friend, is Paul Klee.

With your permission.

Bigger Example Three

Consider the following two program fragments:

Fragment 1

```
if (x == 5)
  x = x + 1;
else
  x = 8;
```

Fragment 2

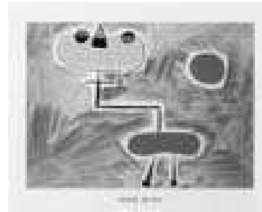
```
if (x == 5)
  x = x + 1;
if (x != 5)
  x = 8;
```

Check all that apply:

- The two fragments are logically equivalent
- Fragment two contains a syntax error.
- If `x` is `6` initially then
  - the value in `x` is `8` after executing fragment one,
  - the value in `x` is `6` after executing fragment two.
- `x` always has the value `8` after executing fragment two.
- `x` has either the value `5` or the value `8` after executing fragment one.

I think this is a bit involved.

Very well, then, here's a hint.



There's more to it than meets the eye.

That much is clear.

Bigger Example Four

Assume that `x` and `y` are integer variables.

Then consider the following nested `if` statement.

```
if (x > 3)
  if (x <= 5)
    y = 1;
  else if (x != 6)
    y = 2;
  else y = 3;
else y = 4;
```

If `y` has the value `2` after executing the above program fragment,

...then what do you know about `x`?

Did you have to draw a diagram?

No, I used a sculpture this time.

Bigger Example Five

Assume that x and y are integer variables,

. . . and consider the code fragment shown below:

```
if (x > 3) {
  if (x <= 5)
    y = 1;
  else if (x != 6)
    y = 2;
} else
  y = 3;
```

I hope you noticed the difference between it and the previous one (*Bigger Example Four*).

You bet.

Now the questions.

Question 1. If x is 1 before the fragment gets executed what's the value of y after the fragment is executed? (Is it possible to give an answer to this question?)

Question 2.
Now erase the curly braces. What value must x have before the fragment gets executed, for y to be 3 at the end of the fragment?

What do you think about these problems?

They make for good practice.

Would you happen to have more?

More? You mean, real problems, programs?

Yes, to practice even further.

I actually do.

Very well, where are they?

Here's a set of warmup problems of the kind you seem to be looking for.

`http://www.cs.indiana.edu/classes/a201/sum2002/notes/ifs.html`

Here are my solutions.

You're pretty quick, aren't you?

http://www.cs.indiana.edu/classes/a201/sum2002/notes/ifsSol.html

Yes, and ready for more.

Then here are some programming problems of the kind you seem to be looking for.

http://www.cs.indiana.edu/classes/a201/sum2002/notes/pIfs.html

And I think you should work them all out.

http://www.cs.indiana.edu/classes/a201/sum2002/notes/pIfsSol.html

Weren't we supposed to start loops today.

Yes, and we can still do it.

Then let's go for it!

What's the motivation?

Remember the investment problem from the first week? It was presented in lecture notes two.

Here's the code in Java, most of it.

```
double balance = 10000;
int year = 0;

year = year + 1;
balance = balance + balance * 0.05;

System.out.println("Year: " + year);
```

What is in red and what is in blue?

The code in blue should be executed only once.

The part in red looks like what one would want to have done repeatedly until the balance becomes big enough, or doubles (or reaches 20,000).

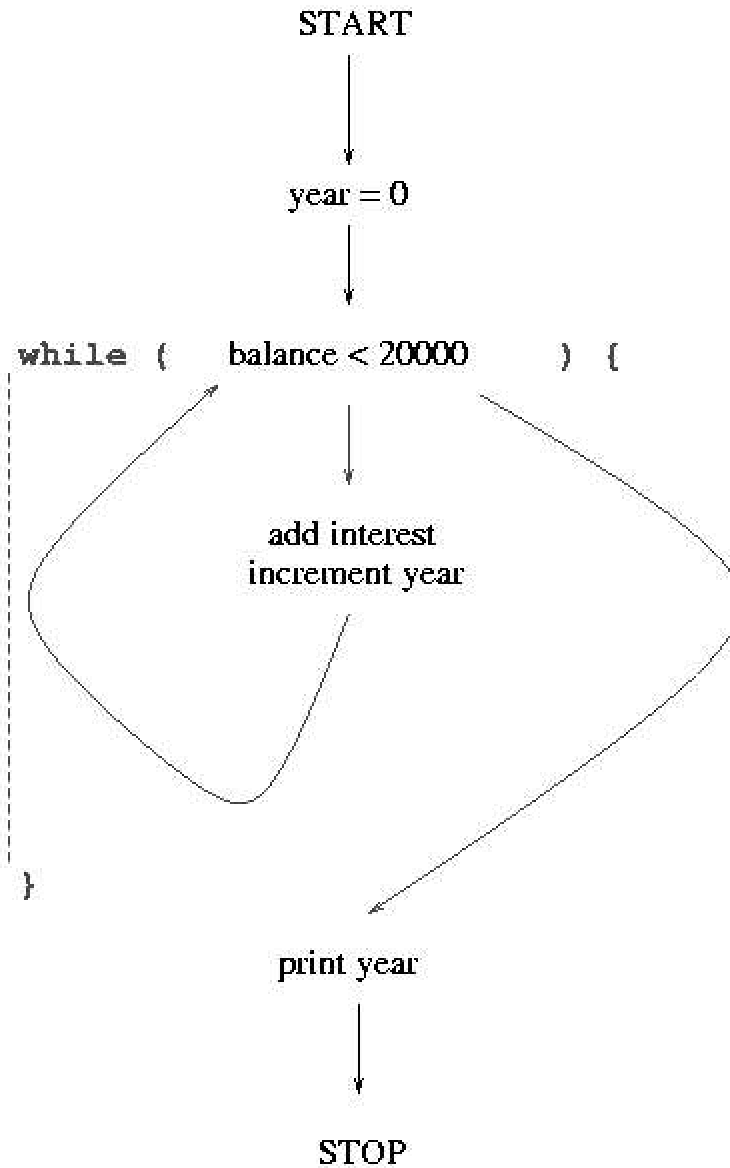
In Java there is a `while` statement which is able to do the iteration for us. Like `if` is a composite statement.

With it one only need specify the red part, with an indication of when to stop. Or, rather, for how long it should go on with the computing of it.

Here's the code:

```
double balance = 10000;
int year = 0;

while (balance < 20000) {
  year = year + 1;
  balance = balance + balance * 0.05;
}

System.out.println("Year: " + year);
```

	Can I draw a flowchart for this?
You sure may.	Here it is:



It takes a `while` to understand all this,	...but once you do it you're *home free.*

```
import java.io.*;

public class Ten {

    public static void main(String[] args) {

        int year;

        ConsoleReader console = new ConsoleReader(System.in);

        System.out.print("Please enter the year then press Enter : ");

        year = console.readInt();

        if (    (((year % 4)  == 0) && ((year % 100) != 0) || (year < 1582))
            ||
                (  year % 400 == 0)
           ) {

            System.out.println("Leap year: " + year);

        } else {

            System.out.println(year + " not a leap year!");

        }
    }
}
```

Loops

What is the purpose of a `while` loop?	To execute a statement while a condition is true.
Let's see some examples.	Here's one that prints the first `n` numbers, where the value of `n` comes from the user:

```
ConsoleReader console = new ConsoleReader(System.in);
int n = console.readInt();
int i = 0;
while (i < n) {
  System.out.println(i);
  i += 1;
}
```

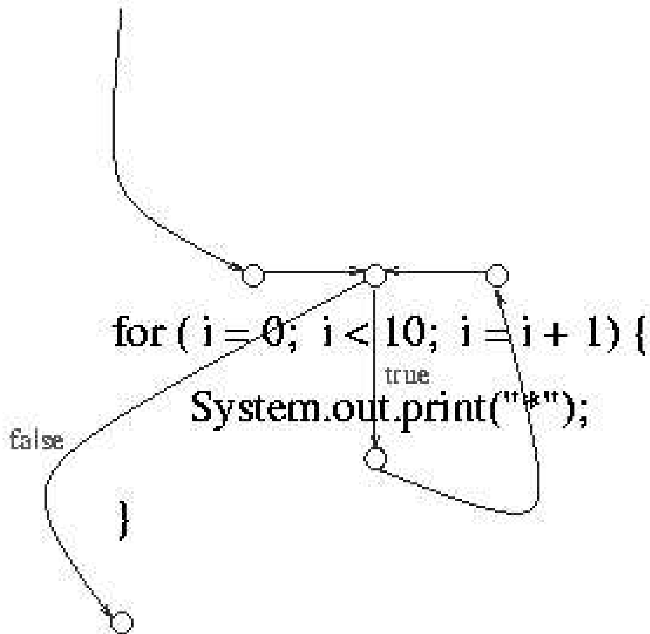
Very good. What do you think of this one?

```
int year = 0;
while (year < 20) {
  double interest = balance * 0.05;
  balance = balance + interest;
}
```

Almost good, except it's an infinite loop.

What did I forget?	The `year` doesn't change.
OK. What is the purpose of a `for` loop?	To do what `while` does, except in a more systematic manner.
In what way?	It clearly distinguishes an initialization step, the condition that needs to be true for the loop to keep going, and what we do from one step to the other.

Is this what you mean?



Yes, this is printing a line of 10 asterisks.

Could you do that with a `while` statement?

Yes, and here's how:

```
i = 0;
while (i < 10) {
  System.out.print("*");
  i = i + 1;
}
```

The `for` and `while` statements are equivalent.

Yes, and this makes for good exercises.

What's the purpose of a `do-while`?

It lets you do the body once, first.

Can we see an example?

First the syntax:

```
do {
  statement
} while ( condition );
```

Very good, now the example.

OK. Here's a code fragment that adds all the numbers that the user types in and then reports the sum of all these numbers.

```
int sum = 0;
do {
```

```
  int number = console.readInt();
  sum = sum + number;
} while (number != 0);
System.out.println(sum);
```

Does this go on for ever? | No. Our *ad-hoc* convention is that the program

```
frilled.cs.indiana.edu%webster ad-hoc
1ad hoc \(')ad-'ha^:k, -'ho^-k; (')a^:d-'ho^-k\ adv
[L, for this]
(1659)
:for the particular end or case at hand without consideration of wider
     application
frilled.cs.indiana.edu%
```

...ends when the user types a value of 0 (zero).

So you use a *sentinel*. | Yes, the value of 0 (zero) acts as a sentinel in this case, guarding the end of our processing.

This is only a convention, right? | Yes, but it works for us.

Can you do this with a `while` or a `for` loop? | Yes, but you'd have to test *first*.

And `do-while` just as well in this case. | Plus, *zero* is such a great sentinel for addition!

How do you break a loop? | Use the `break` statement.

What's `continue` doing? | It just *resumes* the loop from that point.

How often are you likely to use these? | Not often (`break` and `continue`, that is) but in some situations they can come in *real* handy.

OK, let's do an exercise. | Let's see it.

What's this code doing?

```
int i = 0;
while (i < 10)
  i += 1;
```

Just going through the first 10 integers I suppose.

Correct. What's this one do? | `<snicker>`

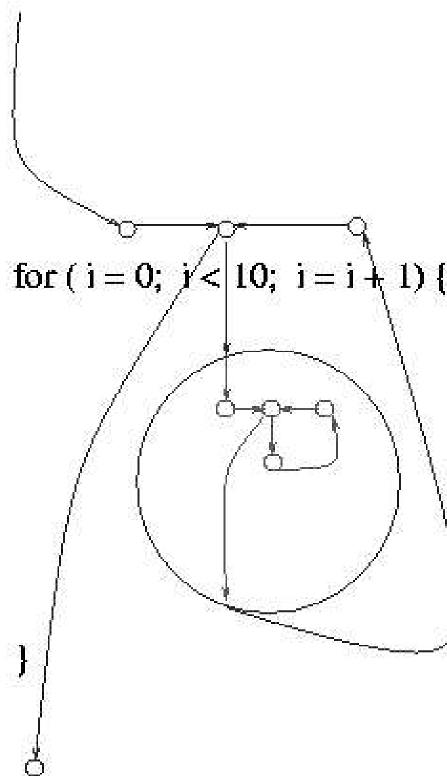
```
int i = 0;
while (i < 10) ;
  i += 1;
```

Doesn't it do the *same* thing?

Don't you see a difference?	Ah, the semicolon – that's an infinite loop now.
Tricky.	Indeed.
You have to be careful.	How do you write 10 asterisks on a line.
Use a `for` loop.	How do you write 10 such lines?

Use a `for` inside a `for`?

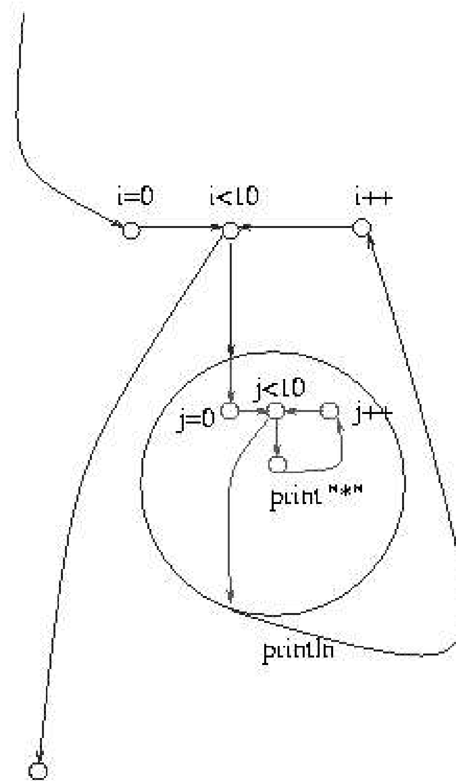
Yes, here's a diagram:



And how does the code look?

```
for (i = 0; i < 10; i++) {
    for (j = 0; j < 10; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

And the diagram again:



So these are *nested* loops.	Yes, nested `for` loops.
Can you draw a square of asterisks of any size?	Yes. Just replace 10 in the code above by the size.
Can you make it hollow?	With no stars inside?
Yes.	I'd have to distinguish between • the border and • the inside, and print • spaces inside and • stars on the border.
Can you do that?	Take a look.

```
public class Square {
    public static void main(String[] args) {
        ConsoleReader console = new ConsoleReader(System.in);
        System.out.print("Enter the size: ");
        int size = console.readInt(), i, j;
        for (i = 0; i < size; i++) {
            for (j = 0; j < size; j++) {
                if (i == 0 || j == 0 ||
                    i == (size - 1) ||
                    j == (size - 1)) {
                    System.out.print("* ");
                } else {
                    System.out.print("  ");
                }
            }
            System.out.println();
        }
    }
}
```

How does it work? Here you go:

```
frilled.cs.indiana.edu%java Square
Enter the size: 4
* * * *
*     *
*     *
* * * *
frilled.cs.indiana.edu%java Square
Enter the size: 8
* * * * * * * *
*             *
*             *
*             *
*             *
*             *
*             *
* * * * * * * *
frilled.cs.indiana.edu%java Square
Enter the size: 10
* * * * * * * * * *
*                 *
*                 *
*                 *
*                 *
*                 *
*                 *
*                 *
*                 *
* * * * * * * * * *
frilled.cs.indiana.edu%
```

Looks good.	Thanks. Lab notes contain *all* the details.
So here's a more appropriate challenge for you.	Oh, no...

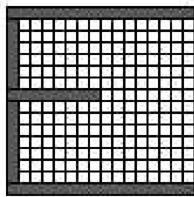
Write a program that produces

- an "E" (uppercase)
- whose size is user-defined.

Can you draw that for me?

There you go:

Three borders and half of a middle line...



And the user inputs the size?

Yes.

Not bad, not bad at all.

```
public class E {
    public static void main(String[] args) {
        ConsoleReader console = new ConsoleReader(System.in);
        System.out.print("Enter the size: ");
        int size = console.readInt(), i, j;
        for (i = 0; i < size; i++) {
            for (j = 0; j < size; j++) {
                if (i == 0 ||
                    i == (size - 1) ||
                    j == 0 ||
                    (i == (size - 1) / 2 && j < (size / 2))
                   ) {
                    System.out.print("* ");
                } else {
                    System.out.print("  ");
                }
            }
            System.out.println();
        }
    }
}
```

Could have been much harder.

I agree.	I know you do.
Yes: `X`, 4…	Well, let's not even get into that now.
Sure, let's wait for the lab.	*I* can hardly wait.

```
frilled.cs.indiana.edu%java E
Enter the size: 10
* * * * * * * * * *
*
*
*
* * * * *
*
*
*
*
* * * * * * * * * *
frilled.cs.indiana.edu%java E
Enter the size: 19
* * * * * * * * * * * * * * * * * * *
*
*
*
*
*
*
*
*
* * * * * * * * *
*
*
*
*
*
*
*
*
* * * * * * * * * * * * * * * * * * *
frilled.cs.indiana.edu%
```

Two Dimensional Patterns

Nested loops, other loops, loops and a half, scalable letters.

Let's practice a bit with `for` loops.

Can you print the numbers from 0 to 9?

Easy:

```
frilled.cs.indiana.edu%cat One.java
class One {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
        }
    }
}
frilled.cs.indiana.edu%javac One.java
frilled.cs.indiana.edu%java One
0
1
2
3
4
5
6
7
8
9
frilled.cs.indiana.edu%
```

Well, what if you want to print the numbers on the *same* line?

Just use `print` in the loop,

```
frilled.cs.indiana.edu%cat One.java
class One {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
```

```
            System.out.print(i);
        }
        System.out.println();
    }
}
frilled.cs.indiana.edu%javac One.java
frilled.cs.indiana.edu%java One
0123456789
frilled.cs.indiana.edu%
```

...and one (empty) `println` outside of it:

Very good.

But don't you want to space them out a bit?

OK, I will print each number in parentheses.

```
frilled.cs.indiana.edu%cat One.java
class One {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            System.out.print(" (" + i + ")");
        }
        System.out.println();
    }
}
frilled.cs.indiana.edu%javac One.java
frilled.cs.indiana.edu%java One
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
frilled.cs.indiana.edu%
```

Looks good. Can you write 10 such lines?

Let me first highlight the code that prints a line.

```
class One {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            System.out.print(" (" + i + ")");
        }
        System.out.println();
    }
}
```

That's the part that you have in blue.

Exactly. Now let's do that 10 times.

Use a `for` loop, with a different index, `j`.

Why j, when I can call it line?	Calling it line would be just fine with me.

```
frilled.cs.indiana.edu%cat One.java
class One {
    public static void main(String[] args) {
        for (int line = 0; line < 10; line++) {
            for (int i = 0; i < 10; i++) {
                System.out.print(" (" + i + ")");
            }
            System.out.println();
        }
    }
}
frilled.cs.indiana.edu%javac One.java
frilled.cs.indiana.edu%java One
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
frilled.cs.indiana.edu%
```

	Although that doesn't influence in the very least the way the program actually works.
Looks good, doesn't it?	Relax. I marked two of the (4)'s in your output with red and blue. Can you tell me what the difference is between them?
They appear on different lines.	Indeed, for the first one line is 0 (zero), while for the second one line has a value of 3 (three).
Let me change the output to include information about this second dimension.	Good idea.

```
frilled.cs.indiana.edu%cat One.java
class One {
    public static void main(String[] args) {
        for (int line = 0; line < 10; line++) {
            for (int i = 0; i < 10; i++) {
                System.out.print(" (" + line + ", " + i + ")");
            }
            System.out.println();
        }
    }
```

```
}
frilled.cs.indiana.edu%javac One.java
frilled.cs.indiana.edu%java One
 (0, 0) (0, 1) (0, 2) (0, 3) (0, 4) (0, 5) (0, 6) (0, 7) (0, 8) (0, 9)
 (1, 0) (1, 1) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (1, 8) (1, 9)
 (2, 0) (2, 1) (2, 2) (2, 3) (2, 4) (2, 5) (2, 6) (2, 7) (2, 8) (2, 9)
 (3, 0) (3, 1) (3, 2) (3, 3) (3, 4) (3, 5) (3, 6) (3, 7) (3, 8) (3, 9)
 (4, 0) (4, 1) (4, 2) (4, 3) (4, 4) (4, 5) (4, 6) (4, 7) (4, 8) (4, 9)
 (5, 0) (5, 1) (5, 2) (5, 3) (5, 4) (5, 5) (5, 6) (5, 7) (5, 8) (5, 9)
 (6, 0) (6, 1) (6, 2) (6, 3) (6, 4) (6, 5) (6, 6) (6, 7) (6, 8) (6, 9)
 (7, 0) (7, 1) (7, 2) (7, 3) (7, 4) (7, 5) (7, 6) (7, 7) (7, 8) (7, 9)
 (8, 0) (8, 1) (8, 2) (8, 3) (8, 4) (8, 5) (8, 6) (8, 7) (8, 8) (8, 9)
 (9, 0) (9, 1) (9, 2) (9, 3) (9, 4) (9, 5) (9, 6) (9, 7) (9, 8) (9, 9)
frilled.cs.indiana.edu%
```

It was an *easy* change.	I see, the part in blue is new.
Now I have 100 cells in the output, and I have a name for each one of them.	Yes, `line` and `i`, as a pair of numbers.
Might as well rename `i` as something more *meaningful*.	Such as `column`.
And let's ask the user to specify the size of the square (number of lines and columns).	Use `ConsoleReader` for that.

```
frilled.cs.indiana.edu%cat One.java
class One {
    public static void main(String[] args) {
        ConsoleReader c = new ConsoleReader(System.in);
        System.out.print("What size? ");
        int size = c.readInt();
        for (int line = 0; line < size; line++) {
            for (int column = 0; column < size; column++) {
                System.out.print(" (" + line + ", " + column + ")");
            }
            System.out.println();
        }
    }
}
frilled.cs.indiana.edu%javac One.java
frilled.cs.indiana.edu%java One
What size? 4
 (0, 0) (0, 1) (0, 2) (0, 3)
 (1, 0) (1, 1) (1, 2) (1, 3)
 (2, 0) (2, 1) (2, 2) (2, 3)
 (3, 0) (3, 1) (3, 2) (3, 3)
frilled.cs.indiana.edu%java One
What size? 6
 (0, 0) (0, 1) (0, 2) (0, 3) (0, 4) (0, 5)
```

```
 (1, 0) (1, 1) (1, 2) (1, 3) (1, 4) (1, 5)
 (2, 0) (2, 1) (2, 2) (2, 3) (2, 4) (2, 5)
 (3, 0) (3, 1) (3, 2) (3, 3) (3, 4) (3, 5)
 (4, 0) (4, 1) (4, 2) (4, 3) (4, 4) (4, 5)
 (5, 0) (5, 1) (5, 2) (5, 3) (5, 4) (5, 5)
frilled.cs.indiana.edu%
```

Looks good, doesn't it?	Relax. What's new is in red and blue, isn't it?
Yes. It looks really good. I have 6 characters for each cell, and everything looks nice and tidy.	Can you highlight the second column?
You mean the set of cells for which `column` has a value of 1 (one)?	You got it.

```
frilled.cs.indiana.edu%cat One.java
class One {
    public static void main(String[] args) {
        ConsoleReader c = new ConsoleReader(System.in);
        System.out.print("What size? ");
        int size = c.readInt();
        for (int line = 0; line < size; line++) {
            for (int column = 0; column < size; column++) {
                if (column == 1) {
                    System.out.print("   **  ");
                } else {
                    System.out.print(" (" + line + ", " + column + ")");
                }
            }
            System.out.println();
        }
    }
}
frilled.cs.indiana.edu%javac One.java
frilled.cs.indiana.edu%java One
What size? 6
 (0, 0)   **   (0, 2) (0, 3) (0, 4) (0, 5)
 (1, 0)   **   (1, 2) (1, 3) (1, 4) (1, 5)
 (2, 0)   **   (2, 2) (2, 3) (2, 4) (2, 5)
 (3, 0)   **   (3, 2) (3, 3) (3, 4) (3, 5)
 (4, 0)   **   (4, 2) (4, 3) (4, 4) (4, 5)
 (5, 0)   **   (5, 2) (5, 3) (5, 4) (5, 5)
frilled.cs.indiana.edu%
```

I can do the first diagonal now.	I know, the change is minor.

```
frilled.cs.indiana.edu%cat One.java
class One {
```

```
    public static void main(String[] args) {
        ConsoleReader c = new ConsoleReader(System.in);
        System.out.print("What size? ");
        int size = c.readInt();
        for (int line = 0; line < size; line++) {
            for (int column = 0; column < size; column++) {
                if (column == line) {
                    System.out.print("   **  ");
                } else {
                    System.out.print(" (" + line + ", " + column + ")");
                }
            }
            System.out.println();
        }
    }
}
frilled.cs.indiana.edu%javac One.java
frilled.cs.indiana.edu%java One
What size? 6
   **   (0, 1) (0, 2) (0, 3) (0, 4) (0, 5)
 (1, 0)   **   (1, 2) (1, 3) (1, 4) (1, 5)
 (2, 0) (2, 1)   **   (2, 3) (2, 4) (2, 5)
 (3, 0) (3, 1) (3, 2)   **   (3, 4) (3, 5)
 (4, 0) (4, 1) (4, 2) (4, 3)   **   (4, 5)
 (5, 0) (5, 1) (5, 2) (5, 3) (5, 4)   **
frilled.cs.indiana.edu%
```

What if you want to see both the last column *and* first diagonal?	I don't know, *you* tell me.
Well, here's what I think: I go through all the cells anyway.	Exactly.
I check their names.	And if you can tell by their name
...that they belong to either the first diagonal *or* to the last column,	...you turn them on.
That's it. This would turn on *all* of the cells that appear on the first diagonal and *all* of those that appear on the last column.	It's just a union of sets.

Easy for you to say that, but here's the program:

```
frilled.cs.indiana.edu%cat One.java
class One {
    public static void main(String[] args) {
        ConsoleReader c = new ConsoleReader(System.in);
        System.out.print("What size? ");
        int size = c.readInt();
        for (int line = 0; line < size; line++) {
```

```
            for (int column = 0; column < size; column++) {
                if (column == line || column == (size - 1)) {
                    System.out.print("   **  ");
                } else {
                    System.out.print(" (" + line + ", " + column + ")");
                }
            }
            System.out.println();
        }
    }
}
frilled.cs.indiana.edu%javac One.java
frilled.cs.indiana.edu%java One
What size? 6
   **   (0, 1) (0, 2) (0, 3) (0, 4)   **
 (1, 0)   **   (1, 2) (1, 3) (1, 4)   **
 (2, 0) (2, 1)   **   (2, 3) (2, 4)   **
 (3, 0) (3, 1) (3, 2)   **   (3, 4)   **
 (4, 0) (4, 1) (4, 2) (4, 3)   **     **
 (5, 0) (5, 1) (5, 2) (5, 3) (5, 4)   **
frilled.cs.indiana.edu%
```

Looks good.

And I was the first to say that.

How can we make this look more like a square?

Maybe change the output a bit.

How about this:

```
frilled.cs.indiana.edu%cat One.java
class One {
    public static void main(String[] args) {
        ConsoleReader c = new ConsoleReader(System.in);
        System.out.print("What size? ");
        int size = c.readInt();
        for (int line = 0; line < size; line++) {
            for (int column = 0; column < size; column++) {
                if (column == line || column == (size - 1)) {
                    System.out.print("* ");
                } else {
                    System.out.print("  ");
                }
            }
            System.out.println();
        }
    }
}
frilled.cs.indiana.edu%javac One.java
frilled.cs.indiana.edu%java One
What size? 8
*             *
```

```
  *           *
    *         *
      *       *
        *     *
          *   *
            * *
              *
frilled.cs.indiana.edu%
```

Did you catch that?

You bet...

Now the names are only implicit.

Only in our program's mind.

We can draw patterns, scalable patterns.

Here's code for a `Z`:

```
frilled.cs.indiana.edu%cat One.java
class One {
    public static void main(String[] args) {
        ConsoleReader c = new ConsoleReader(System.in);
        System.out.print("What size? ");
        int size = c.readInt();
        for (int line = 0; line < size; line++) {
            for (int column = 0; column < size; column++) {
                if (column == (size - 1 - line) ||
                    line   == 0                 ||
                    line   == (size - 1))
                {
                    System.out.print("* ");
                } else {
                    System.out.print("  ");
                }
            }
            System.out.println();
        }
    }
}
frilled.cs.indiana.edu%javac One.java
frilled.cs.indiana.edu%java One
What size? 7
* * * * * * *
          *
        *
      *
    *
  *
* * * * * * *
frilled.cs.indiana.edu%java One
What size? 4
```

```
* * * *
    *
  *
* * * *
frilled.cs.indiana.edu%
```

Can you draw a circle?

A circle?

Sure, why not?

Of course:

```
frilled.cs.indiana.edu%cat One.java
class One {
    public static void main(String[] args) {
        ConsoleReader c = new ConsoleReader(System.in);
        System.out.print("What size? ");
        int size = c.readInt();
        for (int line = 0; line < size; line++) {
            for (int column = 0; column < size; column++) {
                if (
                      Math.abs( (line   - size / 2) * (line   - size / 2) +
                                (column - size / 2) * (column - size / 2) -
                                (size -1) * (size - 1) / 4
                              )   <=   (0.15 * size)
                   )
                {
                    System.out.print("* ");
                } else {
                    System.out.print("  ");
                }
            }
            System.out.println();
        }
    }
}
frilled.cs.indiana.edu%javac One.java
frilled.cs.indiana.edu%java One
What size? 20

              *           *
          *                   *


    *                               *

  *                                   *
```

```
      *                                   *
        *                               *


              *                   *
                  *           *
   frilled.cs.indiana.edu%
```

Hey, now *wait* a minute!

I hope you understand *approximations.*

And the equation of a circle.

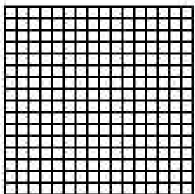
If you don't, don't worry...

Don't worry, be happy!

Well, then it's time for a new lab assignment!

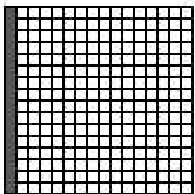
```
public class Patterns {
  public static void main(String[] args) {
    ConsoleReader console = new ConsoleReader(System.in);
    System.out.print("Enter the size: ");
    int size = console.readInt(), i, j;
    for (i = 0; i < size; i++) {
      for (j = 0; j < size; j++) {
        if (false) {
          System.out.print("* ");
        } else {
          System.out.print("  ");
        }
      }
      System.out.println();
    }
  }
}
```

Here's the grid:



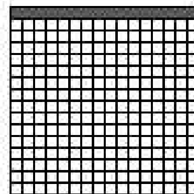
Now, let's look at all the patterns.

Pattern 1:



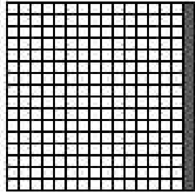
`(j == 0)`

Pattern 2:



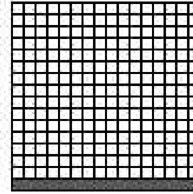
`(i == 0)`

Pattern 3:



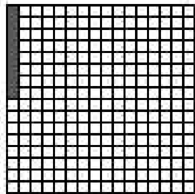
`(j == (size - 1))`

Pattern 4:



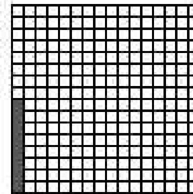
`(i == (size - 1))`

Pattern 5:



`((j == 0) && (i <= size/2))`

Pattern 6:

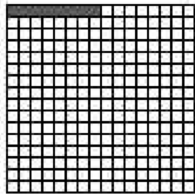


`((j == 0) && (i > size/2))`

If you see that (in places) we're off by 1 (one), ... then you're on the right track.

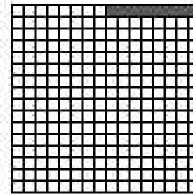
But keep in mind we are interested here in the overall understanding of inequalities that determine (or define) the patterns, ... so we trade (where appropriate) absolute accuracy for a shorter (simpler, yet still reasonably exact) formula.

Pattern 7:

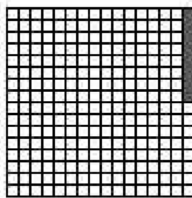


`((i == 0) && (j <= size/2))`

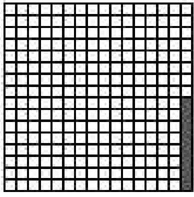
Pattern 8:



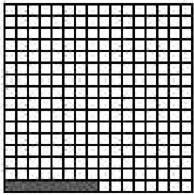
`((i == 0) && (j > size/2))`



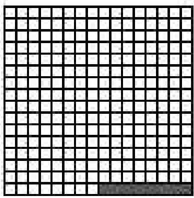
Pattern 9: `((j == (size - 1)) && (i <= size/2))`



Pattern 10: `((j == (size - 1) && (i > size/2))`

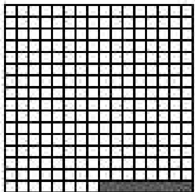


Pattern 11: `((i == (size - 1)) && (j < size/2))`



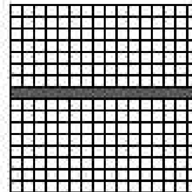
Pattern 12: `((i == (size - 1)) && (j > size/2)`

Pattern 13:



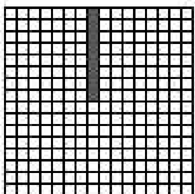
`(j == size/2)`

Pattern 14:

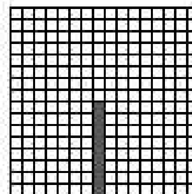


`(i == size/2)`

Pattern 15:

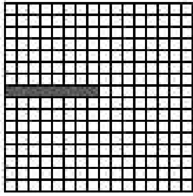


Pattern 16:

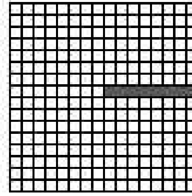


15: `((j == size/2) &&(i <= size/2))`
16: `((j == size/2) &&(i > size/2))`

Pattern 17:

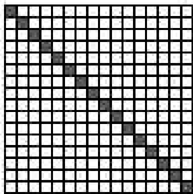


Pattern 18:



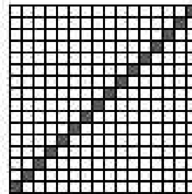
```
17: ((i == size/2) &&(j <= size/2))
18: ((i == size/2) &&(j > size/2))
```

Pattern 19:



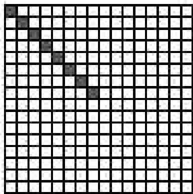
```
(i == j)
```

Pattern 20:

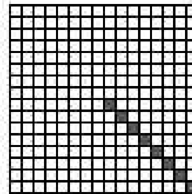


```
(i + j == (size - 1))
```

Pattern 21:



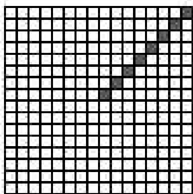
Pattern 22:



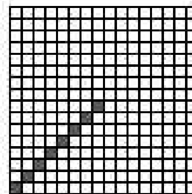
```
21: ((i == j) &&(i <= size/2) && (j <= size/2))
22: ((i == j) &&(i > size/2) && (j > size/2))
```

Can either one of these conditions be simplified at all?

Pattern 23:



Pattern 24:

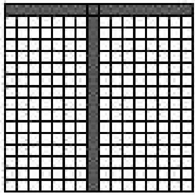


```
23: ((i + j == (size - 1)) &&(i <= size/2) && (j > size/2))
24: ((i + j == (size - 1)) &&(i > size/2) && (j <= size/2))
```

Can either one of these be simplified?

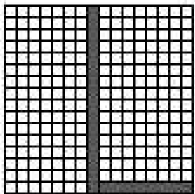
If we have these "atomic" patterns (described above), how we can combine them to obtain more complicated patterns (such as the ones illustrated below):

Uppercase T:



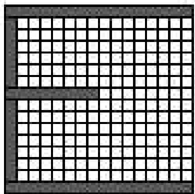
A cell should be turned ON if it appears in the group described by pattern 13 OR if it appears in the group that is described by pattern 2, otherwise the cell is OFF (blank)

Uppercase L:



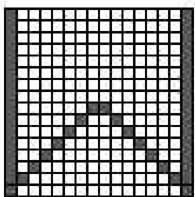
A cell should be turned ON if it appears in the group described by pattern 13 OR if it appears in the group described by pattern 12

Uppercase E:



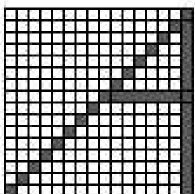
If cell is in pattern 1 OR in pattern 2 OR in pattern 4 or in pattern 17 then the cell should be turned ON otherwise leave the cell blank (print a space)

Uppercase W:



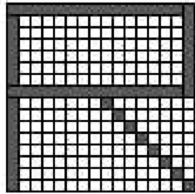
Cell to be turned ON if pattern 1 matches OR if pattern 24 matches OR if pattern 22 matches OR if pattern 3 matches, otherwise leave cell blank

Uppercase A:



(20) OR (3) OR (18)

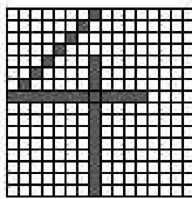
Uppercase R:



(22) || (1) || (14) || (2) || (9)

I am sure you can think of many other patterns: Y, Q (for example). A diamond.

Your task is to write a program that produces a scalable 4 (four):



Did you notice the space on right?	Yes, it's for writing the conditions.
Let me give you another problem, for when you're finished. I will tell you the problem, and show you the solution and you tell me what's wrong about it.	OK.
A bank account starts out with `$10,000`. Interest is compounded at the end of every month at `6` percent per year (`0.5` percent per month). At the beginning of every month, `$500` is withdrawn to meet college expenses after the interest has been credited. After how many years is the account depleted?	OK, I know what you want to say now: suppose the numbers (`$10,000`, `6` percent, `$500`) were user-selectable. Are there any values for which the algorithm we develop does not terminate? If so, make sure it always terminates. Was that it?
Yeah...	Well, I don't see anything *fishy* just yet.

```
frilled.cs.indiana.edu%webster fishy
fishy \'fish-e^-\  fish-i-er; -est
(1547)
1: of or resembling fish esp. in taste or odor
2: creating doubt or suspicion: QUESTIONABLE
frilled.cs.indiana.edu%
```

What do you plan to do about it in your program?

So here's the code, for you to look at:

```
class Interests {
    public static void main(String[] args) {
        ConsoleReader c = new ConsoleReader(System.in);
        System.out.println("Welcome to the financial calculator.");
        System.out.print("What's your initial balance? ");
        double initialBalance = c.readDouble();
        System.out.print("What's the yearly interest? ");
        double yearlyInterest = c.readDouble() / 100;
        System.out.print("How much do you plan to withdraw monthly? ");
        double monthlyStipend = c.readDouble();
        double calculation =
            initialBalance * (1 + yearlyInterest / 12) - monthlyStipend;
        if (calculation >= initialBalance) {
            System.out.println("This will last forever.");
        } else {
            double balance = initialBalance;
            int months = 0;
            while (balance * ( 1 + yearlyInterest / 12) > monthlyStipend) {
                months += 1;
                balance = balance * (1 + yearlyInterest / 12) - monthlyStipend;
            }
            int years = months / 12;
            months = months % 12;
            System.out.println("The account will last " + years +
                               " year(s) and " + months + " month(s).");
            System.out.println("Ending balance will be: " +
                               Math.round(balance * 100) / 100.00
                               + " dollars."
                               );
        }
    }
}
```

Why is this program not calculating the last case correctly?

Also, is this program printing the right (correct) output?

Why or why not?

```
frilled.cs.indiana.edu%cat Why.java
class Why {
  public static void main(String[] args) {
    double sum = 0;
    for (int i = 0; i < 10; i++)
      sum = sum + 0.1;
    System.out.println(sum);
  }
}
frilled.cs.indiana.edu%javac Why.java
frilled.cs.indiana.edu%java Why
0.9999999999999999
frilled.cs.indiana.edu%
```

More Loops

More practice with loops. Loops, tokenizers, and Monte Carlo problems.

Do you like loops?	I don't know: so far, so good.
Now we need to move on.	First a few simple, basic exercises.
Can you explain this?	I sure can.

```
frilled.cs.indiana.edu%cat Two.java
class Two {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            System.out.print(i);
        }
        i = 10;
        System.out.println(i);
    }
}
frilled.cs.indiana.edu%javac Two.java
Two.java:6: Undefined variable: i
        i = 10;
        ^
Two.java:7: Undefined variable: i
        System.out.println(i);
                           ^
2 errors
frilled.cs.indiana.edu%
```

Can you fix it?	I sure can.

```
frilled.cs.indiana.edu%cat Two.java
class Two {
    public static void main(String[] args) {
        int i;
        for (i = 0; i < 10; i++) {
            System.out.print(i);
```

```
        }
        i = 10;
        System.out.println(i);
    }
}
frilled.cs.indiana.edu%javac Two.java
frilled.cs.indiana.edu%java Two
012345678910
frilled.cs.indiana.edu%
```

How do you call that?

I'd say: *scope* of a variable.

```
frilled.cs.indiana.edu%webster scope
scope |'sko^-p| n
[It scopo purpose, goal, fr. Gk skopos; akin to Gk skeptesthai to
     watch, look at -- more at SPY]
(1555)
1: space or opportunity for unhampered motion, activity, or thought
2: INTENTION, OBJECT
3: extent of treatment, activity, or influence
4: range of operation
syn see RANGE

frilled.cs.indiana.edu%
```

Sounds good.

It's the range of operation for that variable.

Take a look at this:

It doesn't compile!

```
class Wow {
    public static void main(String[] args) {
        {
            int i;
            i = 3;
            System.out.println(i);
        }
        System.out.println(i);
    }
}
```

Why?

The curly braces!

Indeed, they are defining the scope.

And they do it in the same way for:

- `classes`
- `if` statements
- `for` loops
- `while` loops

As well as other kinds of loops.

There's just one more, as we will see below.

In any event, having blocks of statements available as standalone entities that can be placed anywhere inside the program can sometimes be a source of confusion for the beggining Java programmer.

Let me see some examples.

Here's one:

That is not a source of *any* confusion!

```
frilled.cs.indiana.edu%cat Wow.java
class Wow {
    public static void main(String[] args) {
        int x = 1;
        if (x > 2) {
            System.out.println("Yes, " + x + " is greater than 2.");
        }
    }
}
frilled.cs.indiana.edu%javac Wow.java
frilled.cs.indiana.edu%java Wow
frilled.cs.indiana.edu%
```

I know, but consider this:

Ah, I see the semicolon!

```
frilled.cs.indiana.edu%cat Wow.java
class Wow {
    public static void main(String[] args) {
        int x = 1;
        if (x > 2) ; {
            System.out.println("Yes, " + x + " is greater than 2.");
        }
    }
}
frilled.cs.indiana.edu%javac Wow.java
frilled.cs.indiana.edu%java Wow
Yes, 1 is greater than 2.
frilled.cs.indiana.edu%
```

I hope you do.

That is an if with an *empty* body.

That's how we got an infinite loop last time.

There's one thing to be learned from this.

Syntax rules.

And people who use it properly, rock!

Let's look at other kind of loops.

OK, here's a program that talks to the user.

```
frilled.cs.indiana.edu%cat Three.java
class Three {
    public static void main(String[] args) {
```

```
            ConsoleReader c = new ConsoleReader(System.in);
            String line;
            do {
                System.out.print("Type something: ");
                line = c.readLine();
                System.out.println("You typed: " + line);
            } while (! line.equals("bye"));
            System.out.println("Good bye!");
        }
    }
    frilled.cs.indiana.edu%javac Three.java
    frilled.cs.indiana.edu%java Three
    Type something: I am here
    You typed: I am here
    Type something: You are there
    You typed: You are there
    Type something: Your name is Echo
    You typed: Your name is Echo
    Type something: Bye
    You typed: Bye
    Type something: bye
    You typed: bye
    Good bye!
    frilled.cs.indiana.edu%
```

Using `while` is just a bit longer.

Yes, since the test comes first.

```
    frilled.cs.indiana.edu%cat Three.java
    class Three {
        public static void main(String[] args) {
            ConsoleReader c = new ConsoleReader(System.in);
            String line;
            System.out.print("Type something: ");
            line = c.readLine();
            System.out.println("You typed: " + line);
            while (! line.equals("bye")) {
                System.out.print("Type something: ");
                line = c.readLine();
                System.out.println("You typed: " + line);
            }
            System.out.println("Good bye!");
        }
    }
    frilled.cs.indiana.edu%javac Three.java
    frilled.cs.indiana.edu%java Three
    Type something: Works the same, doesn't it?
    You typed: Works the same, doesn't it?
    Type something: It does seem so.
    You typed: It does seem so.
    Type something: I am happy.
    You typed: I am happy.
```

```
Type something: BYe
You typed: BYe
Type something: byE
You typed: byE
Type something: bye
You typed: bye
Good bye!
frilled.cs.indiana.edu%
```

In both cases we work with *whole* loops.	Yes, we know we're done either at the beginning or at the end of the loop's body of statements.
Sometimes we'd like to allow for being able to realize we're done halfway through the loop.	And end your processing in mid-loop? Doesn't this coding situation have a specific name?
Yes, it's called the *loop and a half* problem.	Very good. Here's an example: A program that reads lines from a file, then reverses them.

```
frilled.cs.indiana.edu%cat Four.java
class Four {
    public static void main(String[] args) {
        ConsoleReader c = new ConsoleReader(System.in);
        boolean done = false;
        String line;
        while (! done) {
            System.out.print("Echo> ");
            line = c.readLine();
            if (line == null) { // EOF or ctrl-D for that
                done = true;
            } else {
                System.out.println(line);
            }
        }
    }
}
frilled.cs.indiana.edu%javac Four.java
frilled.cs.indiana.edu%java Four
Echo> Hello!
Hello!
Echo> I am here.
I am here.
Echo> How are you?
How are you?
Echo> I am fine, how about you?
I am fine, how about you?
Echo> You don't say...
You don't say...
Echo> I am going to type control-D now
I am going to type control-D now
Echo> Bye!
Bye!
Echo> frilled.cs.indiana.edu%
```

Where's the file?	You can pipe one into your program! (See your friendly TA or ask me for more details).
Alright. So we read lines, one by one. When the line we read is empty there's nothing to be reversed. So the program simply quits the loop.	Oh, I see: and we skip the lower half of the loop.
Indeed, at that point we simply quit it.	How do you do that?
One can do that with `break`, or...	...using a `boolean` variable, and an `if` statement.
We choose the second approach, because one can argue it's a bit more structured.	But the first one is also often used, and it greatly simplifies code on occasion.
So now we have a *basic* echo program.	Yes, but as of right now nothing is being reversed. Wasn't that what we set out to do?
OK, let's make this more exciting, as if through a looking glass.	Yes, s'tel esrever sretcarahc ni sdrow!
Well, yletanutrofnu ew t'nac od taht.	?neht ,elbissop si siht spahrep tuB

That, we can do.

```
frilled.cs.indiana.edu%cat Four.java
class Four {
    public static void main(String[] args) {
        ConsoleReader c = new ConsoleReader(System.in);
        boolean done = false;
        String line;
        while (! done) {
            System.out.print("Echo> ");
            line = c.readLine();
            if (line == null) { // EOF or ctrl-D for that
                done = true;
            } else {
                String rev = "";
                int i;
                for (i = line.length() - 1; i >= 0; i--) {
                    rev += line.charAt(i);
                }
                System.out.println(rev);
            }
        }
    }
}
frilled.cs.indiana.edu%javac Four.java
frilled.cs.indiana.edu%java Four
Echo> Hello!
```

```
!olleH
Echo> I say, that's Spanish.
.hsinapS s'taht ,yas I
Echo> Arabic?
?cibarA
Echo> What's going on, can you please tell me.
 .em llet esaelp uoy nac ,no gniog s'tahW
Echo> Hmmm...
 ... mmmH
Echo> :-)
)-:
Echo> ... ees I
I see...
Echo> .ahctoG
Gotcha.
Echo> frilled.cs.indiana.edu%
```

I like this guy, Gotcha.

Once upon a time there was a Polish carpenter, by the name of Zbigniew Gotcha, who lived in Kracow. You know the story?

No.

I don't either, but I like the way it starts.

Time to wrap up.

Let's do random numbers.

Yes, we kept π for dessert.

Do you like *sherbet*?

Let's program the following simulation:

> Darts are thrown at random points onto the square with corners (1 ,1) and (-1, -1). If the dart lands inside the unit circle, that is, the circle with center (0, 0) and radius 1 it is a hit. Otherwise it is a miss. Run this simulation to determine an experimental value for the fraction of hits in the total number of attempts, multiplied by 4.

Oh, this is *so* easy! Easy as π!

Fine, if it's easy, why don't you do it first?

Relax, here it is:

```
frilled.cs.indiana.edu%cat Pi.java
import java.util.Random;
public class Pi {
    public static void main(String[] args) {
        Random r = new Random();
        double x, y, d;
        int i, count = 0;
        for (i=0; i < 100000 ; i++) {
            x = r.nextDouble() * 2 - 1;
            y = r.nextDouble() * 2 - 1;
            d = Math.sqrt(x * x + y * y);
```

```
            if (d < 1) count++;
        }
        System.out.println("Pi is approximately " + 4.0 * count / i);
    }
}
frilled.cs.indiana.edu%javac Pi.java
frilled.cs.indiana.edu%java Pi
Pi is approximately 3.13648
frilled.cs.indiana.edu%java Pi
Pi is approximately 3.15064
frilled.cs.indiana.edu%java Pi
Pi is approximately 3.14424
frilled.cs.indiana.edu%java Pi
Pi is approximately 3.1422
frilled.cs.indiana.edu%java Pi
Pi is approximately 3.1438
frilled.cs.indiana.edu%
```

Can you explain it?	The probability of a hit is the fraction that the circle represents of the total area.
It is also close to the ratio of the measured frequencies: hits divided by attempts.	So we write the formulas, and the radius simplifies, as it appears on both sides.
But there is a factor of half ($\frac{1}{2}$) which...	...is squared, so it participates as a fourth ($\frac{1}{4}$) in the end, and there you have it.
Pretty good.	I want to do more problems.
I was hoping you would.	I'm in great shape today.
Here's a program that helps with problem 6.1	Where does this number come from?
Just ignore it for now...	What does the program do?

```
import java.util.*;
import java.io.*;
class Six {
    public static void main(String[] args) {
        ConsoleReader console = new ConsoleReader(System.in);
        System.out.print("Hello> ");
        String line = console.readLine();
        while (line != null) { // ^D would do it
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                String token = tokenizer.nextToken();
                System.out.println(token.toUpperCase());
            }
            System.out.print("Hello> ");
            line = console.readLine(); // what if we take this out?
```

```
        }
        System.out.println("End of program.");
    }
}
```

Reads lines, one by one.	When does it end?
When you type Control-D.	Which is EOF (end of file).
Yes, in that case the line is `null`.	What does it do, line by line?
Looks at the tokens, and prints them back, but converted into uppercase.	Not much different from the one about Zbigniew, except this one uses this `StringTokenizer`.
Exactly, that's the main difference.	How does that work?

Take a closer look:

```
StringTokenizer tokenizer = new StringTokenizer(line);
while (tokenizer.hasMoreTokens()) {
  String token = tokenizer.nextToken();
  System.out.println(token.toUpperCase());
}
```

Better look this class up.[a]

[a] http://java.sun.com/products/jdk/1.2/docs/api/java/util/StringTokenizer.html

It's like a machine gun loaded with words.	Or like a stapler, if you don't mind.
A stapler would also be a good analogy.	With staples of variable length.
Glued together by blank spaces.	Staples are tokens.

```
frilled.cs.indiana.edu%webster token
to-ken \'to^--ken n
[ME, fr. OE ta^-cen, ta^-cn sign, token; akin to OHG zeihhan sign, Gk
     deiknynai to show -- more at DICTION]
(bef. 12c)
1: an outward sign or expression <his tears were tokens of his
     grief>
2a: SYMBOL, EMBLEM <a white flag is a token of surrender>;
2b: an instance of a linguistic expression
3: a distinguishing feature: CHARACTERISTIC
[... ]
frilled.cs.indiana.edu%
```

For both problem 6.2 and 6.3 the trick is to *interpret* (read and promptly evaluate) input.	I think I can handle that (ignore the numbers).

You're going to have to read

- a rate,
- then numbers,
- finished by zero...

...then numbers again, ending with EOF.

Which is Control-D (in Unix).

Or some such thing.

I could even end it with a keyword, such as "`quit`" or "`bye`", or some other meaningful word.

Really, how?

Take a look.

```
import java.util.*;
import java.io.*;
     class TwoAndThree {
         public static void main(String[] args) {
             ConsoleReader console = new ConsoleReader(System.in);
             System.out.print("Rate>");
             String line = console.readLine();
             StringTokenizer tokenizer = new StringTokenizer(line);
             double rate = Double.parseDouble(tokenizer.nextToken());
             double amount;
             do {
                 System.out.print("Dollars>");
                 line = console.readLine();
                 tokenizer = new StringTokenizer(line);
                 amount = Double.parseDouble(tokenizer.nextToken());
             } while (amount > 0);
             do {
                 System.out.print("Euros>");
                 line = console.readLine();
                 if (line == null || line.equalsIgnoreCase("quit"))
                     break;
                 tokenizer = new StringTokenizer(line);
                 amount = Double.parseDouble(tokenizer.nextToken());
             } while (true);
             System.out.println("Thank you for using this program.");
         }
     }
```

This should get you started.

Nice, but I see that you're assuming the user will never type more than one number on a line.

So I could get by *without* a tokenizer. I agree, but I used one for the sake of practice.

I have a larger set of problems for next time.	Can't wait. Let's press on with the ones for today. Next one up: problem 6.4, page 263 (Horstmann).

Simulate the wandering of an intoxicated person in a square street grid. For 100 times, have the simulated drunkard randomly pick a direction (east, west, north, south) and move one block in the chosen direction. After the iterations, display the distance that the drunkard has covered. (One might expect that on average the person might not get anywhere because the moves to different directions cancel another out in the long run, but in fact it can be shown that with probability 1 (certainty) the person eventually moves outside any finite region.

There are *many* ways of solving this problem.	Yes, but empathy would not be one of them.
Of course. What I meant was that you need to generate random directions.	And you have more than one way to produce random numbers.
Let's use whatever the book uses.	It's a good book.
I know.	What book?
In addition to that, I would like to use an object oriented approach.	And bring a drunkard into the picture.
Exactly.	A drunkard is like a `BankAccount`.

I was going to say.

Well, here's how I'd get started.

```
import java.util.*;
class Drunkard {
    int x, y;
    Drunkard(int x, int y) {
        this.x = x;
        this.y = y;
    }
    void moveNorth() {
        this.y -= 1;
    }
    void moveEast() {
        this.x += 1;
    }
    void report() {
        System.out.println("Hiccup: " + x + ", " + y);
    }
}
class Four {
    public static void main(String[] args) {
        Random generator = new Random();
```

```
        Drunkard drunkard = new Drunkard(100, 100);
        int direction;
        for (int i = 0; i < 100; i++) {
            direction = Math.abs( generator.nextInt() ) % 4;
            if        (direction == 0) { // N
                drunkard.moveNorth();
            } else if (direction == 1) { // E
                drunkard.moveEast();
            } else if (direction == 2) { // S
                System.out.println("Should move South.");
            } else if (direction == 3) { // W
                System.out.println("Should move West.");
            } else {
                System.out.println("Impossible!");
            }
            drunkard.report();
        }
    }
}
```

Not bad at all.

Of course, one needs to finish it first.

Reminds me of homework assignment two.

Somewhat. Now let's look at 6.5 and 6.6.

Suppose a cannonball is propelled vertically into the air with a starting velocity v_0. Any calculus book will tell us that the position of the ball after t seconds is

$$s(t) = -0.5gt^2 + v_0t$$

where $g = 9.81\frac{m}{s^2}$ is the gravitational force of the earth. No calculus book ever mentions why someone would want to carry out such an obviously dangerous experiment, so we will do it in the safety of the computer. In fact, we will confirm the theorem from calculus by a simulation. In our simulation, we will consider how the ball moves in very short time intervals Δt. In a short time interval the velocity v is nearly constant, and we can compute the distance the ball moves as $\Delta s = v\Delta t$. In our program, we will simply set

```
double deltaT = 0.01;
```

and update the position by

```
s = s + v * deltaT;
```

The velocity changes constantly—in fact it is reduced by the gravitational force of the earth. In a short time interval, $v = -g\Delta t$, and we must keep the velocity updated as

```
v = v - g * deltaT;
```

In the next iteration the new velocity is used to update the distance. Now run the simulation until the cannonball falls back onto earth. Get the initial velocity as an input ($100\frac{m}{s}$ is a good value). Update the position and velocity 100 times per second, but print out the position only every full second. Also print out the the values from the exact formula $s(t) = -0.5gt^2 + v_0 t$ for comparison.

What is the benefit of this kind of simulation when an exact formula is not available? Well, the formula from the calculus book is *not* exact. Actually the gravitational force diminishes the further the cannonball is away from the surface of the earth. This complicates the algebra sufficiently that it is not possible to give an exact formula for the actual motion, but the computer simulation can simply be extended to apply a variable gravitational force. For cannonballs, the calculus book formula is actually good enough, but computers *are* necessary to compute accurate trajectories for higher-flying objects such as ballistic missiles.

Now to complete the picture we need to say that most cannonballs are not shot upright but at an angle. If the starting velocity has magnitude v and the starting angle is α, then the velocity is actually a vector with components $v_x = v\cos\alpha$ and $v_y = v\sin\alpha$. In the x-direction the velocity does not change. In the y-direction the gravitational force takes its toll. Repeat the simulation from the previous exercise, but store the position of the cannonball as a `Point2D` variable. Update the x and y positions separately, and also update the x and y components of the velocity separately. Every full second, plot the location of the cannonball on the graphics display. Repeat until the cannonball has reached the earth again.

This kind of problem is of historical interest. The first computers were designed to carry out just such ballistic calculations, taking into account the diminishing gravity for high-flying projectiles and wind speeds.

Isn't a cannonball like a drunkard?	Yes, they're both tiggers.

They all deal directly with gravitational fields.

Cannonballs require more physics, though.

```
class Cannonball {
    double x;
    double y;
    double vx;
    double vy;
    final double g = 9.81;
    Cannonball(double speed, double angle) {
        x = 0;
        y = 0;
        vx = Math.cos(angle) * speed;
        vy = Math.sin(angle) * speed;
    }
    void move() {
        double deltaT = 0.01;
        x += vx * deltaT;
        y += vy * deltaT;
        vy -= g * deltaT;
    }
    void report() {
        System.out.println("Located at: (" + x + ", " + y + ")");
```

```
        }
        double height() {
            return y;
        }
    }
    class FiveAndSix {
        public static void main(String[] args) {
            Cannonball c = new Cannonball(10, Math.PI / 4);
            for (int i = 0; i < 10 * 100; i++) { // flying 10 seconds
                c.move();
                c.report();
                if (c.height() < 0) {
                    System.out.println("SPLOOOF! The cannonball landed!");
                    break;
                }
            }
        }
    }
```

Homework assignment one!	Almost, only in finer steps.
Could you also compute the highest point that the cannonball gets to?	That would almost be problem 16, wouldn't it?

Problem 16. Write a program that reads a series of floating-point numbers and prints:

- the maximum value
- the minimum value
- the average value

It would.	We'd have to update our notion of a current maximum every time we are looking at a height.
Come to think of it the cannonball could do it.	Intelligent cannonball.
Intelligent, but misguided.	Well, here it is anyway:

```
    class Cannonball {
        double x;
        double y;
        double vx;
        double vy;
        final double g = 9.81;
        double max;
        Cannonball(double speed, double angle) {
            x = 0;
            y = 0;
            vx = Math.cos(angle) * speed;
```

```
        vy = Math.sin(angle) * speed;
        max = 0;
    }
    void move() {
        double deltaT = 0.01;
        x += vx * deltaT;
        y += vy * deltaT;
        vy -= g * deltaT;
        if (y > max) { max = y; }
    }
    void report() {
        System.out.println("Located at: (" + x + ", " + y + ") ");
        System.out.println("     max altitude so far: " + max);
    }
    double height() {
        return y;
    }
}
class Max {
    public static void main(String[] args) {
        Cannonball c = new Cannonball(10, Math.PI / 4);
        for (int i = 0; i < 10 * 100; i++) { // flying 10 seconds
            c.move();
            c.report();
            if (c.height() < 0) {
                System.out.println("The cannonball landed!");
                break;
            }
        }
    }
}
```

You just added three lines?	*Four*, and yes, that was all.
But how often do those get executed?	They're part of the cannonball's movement.
Let's move on to problem 6.7.	This one is easy.

The Fibonacci sequence is defined by the following rule. The first two values in the sequence are 1 and 1. Every subsequent value is the sum of the two values preceding it. Write a program that prompts the user for `n` and prints the `nth` value in the Fibonacci sequence.

Hint: this problem is easy.

Yes, from two values we compute a third. And we keep doing this over and over again.	Now only this new value and the most recent of the two it was computed from should be kept.
And these two values are then added to compute a new value.	And the whole process is repeated, as follows:

```
for (int i = 3; i <= n; i++) {
  fNew = fOld + fOlder;
  fOlder = fOld;
  fOld = fNew;
}
```

Yes, that's it.	This reminds me of another problem:
What problem is that?	See if you can figure it for yourself.

```
import java.io.*;
class Mistery {
    public static void main(String[] args) {
        ConsoleReader console = new ConsoleReader(System.in);
        System.out.print("Pass the salt please: ");
        double a = console.readDouble();
        System.out.print("And the butter: ");
        int n = console.readInt();
        double xold, xnew;
        xold = ...  ;
        do {
          xnew = xold - (Math.pow(xold, n) - a) / (n * Math.pow(xold, n - 1));
          xold = xnew;
          System.out.println("     " + (Math.pow(xnew, n) - a));
        } while (Math.abs(Math.pow(xnew, n) - a) > 0.001);
        System.out.println(xnew);
        System.out.println(Math.pow(xnew, n) + " " + a);
        System.out.println("Thank you!");
    }
}
```

P6.14 perhaps? But what's the ...?	It doesn't matter, if it has a value.
For this problem, at least.	Problems 6.9, 6.10, and 6.15 are easy, although for 10 and 15 we need to wait until we look at applets to do any graphics.
Let's do number 10.	OK, but I won't draw circles, I'll just *create* them.

Drunkards, circles, cannonbals: they're all the same.

Especially drunkards. Here's the code:

```
import java.util.*;
class Ten {
    public static void main(String[] args) {
        System.out.print("Enter number of circles: ");
        ConsoleReader console = new ConsoleReader(System.in);
        int n = console.readInt();
        System.out.println("Generating " + n + " circles.");
```

```
        Random generator = new Random();
        for (int i = 0; i < n; i++) {
            int x = 100 + Math.abs(generator.nextInt()) % 200;
            int y = 100 + Math.abs(generator.nextInt()) % 200;
            int r = 10 + Math.abs(generator.nextInt()) % 40;
            Circle e = new Circle(x, y, r);
            System.out.println(e); // looks better when you draw it
        }
        System.exit(0);
    }
}
class Circle {
  double xCenter, yCenter, radius;
  Circle (double x, double y, double r) {
    xCenter = x;
    yCenter = y;
    radius = r;
  }
  public String toString() {
    return "I am a circle at: ("     +
           xCenter + ", " + yCenter  +
           ") with a radius of "     +
           radius;
  }
}
```

Why is 6.15 easy?	Because it's like 10, and in addition you have complete information about the position of the squares. You'll see, later.
Why is 6.15 hard?	Because you should come up with a formula for the position of a square given its row and column (or line and column).
This way you can use a `for` loop for the lines,	. . . and another one for the columns.
One, inside each other.	Like for the patterns we developed last week.
And why is 6.9 easy?	Because you just have to compute two sums, and take a square root at the end.
Why is 6.9 hard?	6.9 is not hard, but you have to read the problem carefully and use the right formula.
Which is the last one (second of two).	The last one for today anyway.
We'll do a lot more next time.	As always, I can hardly wait.

Appendix Special (Selected Problems and Their Numbers).

Here's another approach to producing random numbers.

Say we need random numbers between a, and b, where a is less than b. Let y be such a random number. Then we can calculate how far into the segment y is, as a percentage.

```
double p = (y - a) / (b - a); // [1]
```

Since y is random that means p can be anywhere between 0 and 1.

Well, such numbers can be generated with `Math.random()`.

So, let's turn [1] on its head, to obtain y as a function of p.

Then, we have:

```
y = p * (b - a) + a;
```

Now replace p by `Math.random()`:

```
y = (b - a) * Math.random() + a;
```

Same as *stretching* followed by a *translation*. End of story.

Problem 6.14. Write a program that asks the user for an integer and then prints out all its factors. For example, when the user enters 150, the program should print: `2 3 5 5`.

Problem 6.9. Write a program that reads a set of floating-point data values from the input. When the end of file is reached, print out the count of the values, the average, and the standard deviation. The average of a data set $x_{i=1\ldots n}$ is $\mu = \sum x_i/n$. The standard deviation is

$$S = \sqrt{\frac{\sum (x_i - \mu)^2}{n-1}}$$

However that formula is not suitable for our task. by the time you have computed the mean, the individual x_i's are long gone. Until you know how to save these values, use the numerically less stable formula

$$S = \sqrt{\frac{\sum {x_i}^2 - \frac{1}{n}(\sum x_i)^2}{n-1}}$$

You can compute this quantity by keeping track of the count, the sum, and the sum of squares as you process the input values.

Problem 6.10. Write a graphical applet (?!—see if you get around that) which prompts a user to enter a number n and then draws n circles with random center and random radius.

Problem 6.15. Write a program that prompts the user for an integer and then prints out all prime numbers up to that integer. For example, when the user enters 20, the program should print: `2 3 5 7 11 13 17 19`. Recall that a number is a prime number if it is not divisible by any number except 1 and itself.

Computer Games

In which Tigger has guests from another tale.

Have you ever played *Nim*?	No. But do *you* play *croquet*?
I'd love to but I don't have the time.	Well, how do you play Nim?
Allow me to describe it.	Very well, but please be *very* clear, my dear. Yes, try to be *very* clear.
Nim is a well-known game with a number of variants. We will consider the following variant, which has an interesting winning strategy.	Two players alternately take marbles from a pile. In each move, a player chooses how many marbles to take, then removes the marbles.
The player must take • at least one but • at most half of the marbles.	Then the other player takes a turn.
You already said that.	I thought you were sleeping.
This is very provoking...	Sorry, dear. *Ahem.* The player who takes the last marble loses.
You will write a program in which a computer plays agains a human opponent.	Generate a random number between 10 and 100 to denote the initial size of the pile.
Generate a random integer between 0 and 1 to decide whether the computer or the human takes the first turn. Then start the game.	This variant of the game has an interesting winning strategy. Careful thinking will reveal that whoever moves first can win.

How?	Take off enough marbles to make the pile a power of two minus one, that is 1, 3, 7, 15, 31, or 63.
I see.	One could program the computer to always play in what could be called *smart* mode.
That would not be too much fun for the user.	One could also program the computer to always perform a random legal move.
That's what we'll do, as it seems a bit more fair.	OK, let's get started.
First of all, what do we need?	No pile of marbles, no Nim.
Let's bring one in.	How does it look?
What do you think of this?	Looks like a good start.

```
class PileOfMarbles {
   int height;
  PileOfMarbles (int height) {
    this.height = height;
  }
  int report() {
    return this.height;
  }
}
```

A pile of marbles is like a bank account.	Whoever withdraws the last cent loses.
It has a balance (`height`).	And a get-balance (`report`) method.
We also need a withdraw, don't we?	Yes, let's call it `move`.
What do you think of this?	You have to be careful when withdrawing.

```
void move(int number) {
  System.out.println("***Removing " + number + " marbles from the pile.");
  this.height -= number;
  System.out.println("Pile of marbles is now: " + this.report());
}
```

So how do you check that?	How about this condition?

```
if (number <= 0 || ((number > height / 2) && (number != 1))) {
  // then it's a bad move
} else {
  // it is a good move
}
```

Can I look at this again?	You mean the check for a *bad* move?
Yes.	Here it is (a *bad* move).

```
if (number <= 0 || ((number > height / 2) && (number != 1))) {
  System.out.println("***Bad move: you lose.");
  System.exit(0);
} else {
```

Or, just the condition, again:

```
(number <= 0 || ((number > height / 2) && (number != 1)))
```

Either zero marbles or less, ...	...or more than half, and not the last.
I think this last part is rather tricky.	Using de Morgan's law we can reformulate this to represent a good move. Perhaps that would help.
Yes, let's write down its negation.	Here it is (a *good* move):

```
(number > 0 && ((number <= height / 2) || (number == 1)))
```

To me this looks better, easier to understand.	It does seem that way to me too.
Even if this is no easier to read than the first version you now have a choice, an option.	I quite agree with that.
So we can finish `report` now.	Yes. Let's make the pile of marble responsible for announcing the end of the game too.
Then it needs to know who moved.	I think I can accommodate that.

```
void move(int number, String user) {
  System.out.println("***Removing " + number +
                     " marbles from the pile for: " + user);
  if (number <= 0 || ((number > height / 2) && (number != 1))) {
    System.out.println("***Bad move for " + user + ". " +
                       user + " loses.");
    System.exit(0);
  } else {
    this.height -= number;
    if (this.height == 0) {
      System.out.println("***End of game. " + user + " loses.");
      System.exit(0);
    }
  }
  System.out.println("Pile of marbles is now: " + this.report());
}
```

That's pretty much *it*, isn't it?	Yes. Now we need to set up the game.
We need a `while` loop.	We need a loop, like we did in `Echo`, yes.
Here's my suggestion.	Looks like you finished it altogether now.

```
int height = (int)(Math.random() * 90 + 10);
PileOfMarbles pile = new PileOfMarbles(height);
System.out.println("Game starts with a pile of height: "
                   + pile.report());
int number, currentHeight;
while (true) {
    System.out.println("*** Computer moves.");
    System.out.println("Pile of marbles of height: " + pile.report());
    currentHeight = pile.report();
    if (currentHeight == 1) {
      number = 1;
    } else {
      number = (int)(Math.random() * (currentHeight / 2)) + 1;
    }
    System.out.println("Computer chooses to remove: " +
                       number + " marbles.");
    pile.move(number, "Computer");
    System.out.println("--------------------------");
    System.out.println("*** Now " + user + " has to move.");
    System.out.println("Pile of marbles of height: " + pile.report());
    System.out.print(user +
                     ", please enter number of marbles you want to take: ");
    number = console.readInt();

    pile.move(number, user);

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

}
```

Yes, here's the whole thing:

```
class Nim {
    public static void main(String[] args) {
        ConsoleReader console = new ConsoleReader(System.in);

        System.out.println("Hello, and welcome to the game of Nim!");
        System.out.print("What is your name: ");
        String user = console.readLine();

        int height = (int)(Math.random() * 90 + 10);
        PileOfMarbles pile = new PileOfMarbles(height);
```

```
        System.out.println("Game starts with a pile of height: "
                           + pile.report());

        int number, currentHeight;

        while (true) {
            System.out.println("*** Computer moves.");

            System.out.println("Pile of marbles of height: " + pile.report());

            currentHeight = pile.report();

            if (currentHeight == 1) {
                number = 1;
            } else {
                number = (int)(Math.random() * (currentHeight / 2)) + 1;
            }

            System.out.println("Computer chooses to remove: " +
                               number + " marbles.");

            pile.move(number, "Computer");

            System.out.println("--------------------------");
            System.out.println("*** Now " + user + " has to move.");
            System.out.println("Pile of marbles of height: " +
                               pile.report());
            System.out.print(user +
                 ", please enter number of marbles you want to take: ");

            number = console.readInt();

            pile.move(number, user);

            System.out.println("--------------------------");
        }
    }
}
class PileOfMarbles {
    int height;
    PileOfMarbles (int height) {
        this.height = height;
    }
    int report() {
        return this.height;
    }
    void move(int number, String user) {
        System.out.println("***Removing " + number +
                           " marbles from the pile for: " + user);
        if (number <= 0 || ((number > height / 2) && (number != 1))) {
```

```
                System.out.println("***Bad move for " + user + ". " + user + " loses.");
                System.exit(0);
            } else {
                this.height -= number;
                if (this.height == 0) {
                    System.out.println("***End of game. " + user + " loses.");
                    System.exit(0);
                }
            }
            System.out.println("Pile of marbles is now: " + this.report());
        }
    }
```

Don't forget your `ConsoleReader`!	I'm ready. Let's play!

The King looks like Humphrey Bogart in Casablanca.

Have you seen Casablanca?	Shh—we're playing Nim now!

```
frilled.cs.indiana.edu%javac Nim.java
frilled.cs.indiana.edu%java Nim
Hello, and welcome to the game of Nim!
What is your name: Mary-Ann
Game starts with a pile of height: 86
*** Computer moves.
Pile of marbles of height: 86
Computer chooses to remove: 23 marbles.
***Removing 23 marbles from the pile for: Computer
Pile of marbles is now: 63
--------------------------
*** Now Mary-Ann has to move.
Pile of marbles of height: 63
Mary-Ann, please enter number of marbles you want to take: 30
***Removing 30 marbles from the pile for: Mary-Ann
Pile of marbles is now: 33
--------------------------
*** Computer moves.
Pile of marbles of height: 33
Computer chooses to remove: 1 marbles.
***Removing 1 marbles from the pile for: Computer
Pile of marbles is now: 32
--------------------------
*** Now Mary-Ann has to move.
Pile of marbles of height: 32
Mary-Ann, please enter number of marbles you want to take: 16
***Removing 16 marbles from the pile for: Mary-Ann
Pile of marbles is now: 16
--------------------------
*** Computer moves.
Pile of marbles of height: 16
Computer chooses to remove: 7 marbles.
***Removing 7 marbles from the pile for: Computer
Pile of marbles is now: 9
--------------------------
*** Now Mary-Ann has to move.
Pile of marbles of height: 9
Mary-Ann, please enter number of marbles you want to take: 2
***Removing 2 marbles from the pile for: Mary-Ann
Pile of marbles is now: 7
--------------------------
*** Computer moves.
Pile of marbles of height: 7
Computer chooses to remove: 1 marbles.
***Removing 1 marbles from the pile for: Computer
Pile of marbles is now: 6
--------------------------
*** Now Mary-Ann has to move.
Pile of marbles of height: 6
```

```
Mary-Ann, please enter number of marbles you want to take: 3
***Removing 3 marbles from the pile for: Mary-Ann
Pile of marbles is now: 3
--------------------------
*** Computer moves.
Pile of marbles of height: 3
Computer chooses to remove: 1 marbles.
***Removing 1 marbles from the pile for: Computer
Pile of marbles is now: 2
--------------------------
*** Now Mary-Ann has to move.
Pile of marbles of height: 2
Mary-Ann, please enter number of marbles you want to take: 1
***Removing 1 marbles from the pile for: Mary-Ann
Pile of marbles is now: 1
--------------------------
*** Computer moves.
Pile of marbles of height: 1
Computer chooses to remove: 1 marbles.
***Removing 1 marbles from the pile for: Computer
***End of game. Computer loses.
frilled.cs.indiana.edu%java Nim
Hello, and welcome to the game of Nim!
What is your name: Queen
Game starts with a pile of height: 77
*** Computer moves.
Pile of marbles of height: 77
Computer chooses to remove: 8 marbles.
***Removing 8 marbles from the pile for: Computer
Pile of marbles is now: 69
--------------------------
*** Now Queen has to move.
Pile of marbles of height: 69
Queen, please enter number of marbles you want to take: 33
***Removing 33 marbles from the pile for: Queen
Pile of marbles is now: 36
--------------------------
*** Computer moves.
Pile of marbles of height: 36
Computer chooses to remove: 11 marbles.
***Removing 11 marbles from the pile for: Computer
Pile of marbles is now: 25
--------------------------
*** Now Queen has to move.
Pile of marbles of height: 25
Queen, please enter number of marbles you want to take: 12
***Removing 12 marbles from the pile for: Queen
Pile of marbles is now: 13
--------------------------
*** Computer moves.
```

```
Pile of marbles of height: 13
Computer chooses to remove: 5 marbles.
***Removing 5 marbles from the pile for: Computer
Pile of marbles is now: 8
--------------------------
*** Now Queen has to move.
Pile of marbles of height: 8
Queen, please enter number of marbles you want to take: 1
***Removing 1 marbles from the pile for: Queen
Pile of marbles is now: 7
--------------------------
*** Computer moves.
Pile of marbles of height: 7
Computer chooses to remove: 2 marbles.
***Removing 2 marbles from the pile for: Computer
Pile of marbles is now: 5
--------------------------
*** Now Queen has to move.
Pile of marbles of height: 5
Queen, please enter number of marbles you want to take: 2
***Removing 2 marbles from the pile for: Queen
Pile of marbles is now: 3
--------------------------
*** Computer moves.
Pile of marbles of height: 3
Computer chooses to remove: 1 marbles.
***Removing 1 marbles from the pile for: Computer
Pile of marbles is now: 2
--------------------------
*** Now Queen has to move.
Pile of marbles of height: 2
Queen, please enter number of marbles you want to take: 1
***Removing 1 marbles from the pile for: Queen
Pile of marbles is now: 1
--------------------------
*** Computer moves.
Pile of marbles of height: 1
Computer chooses to remove: 1 marbles.
***Removing 1 marbles from the pile for: Computer
***End of game. Computer loses.
```

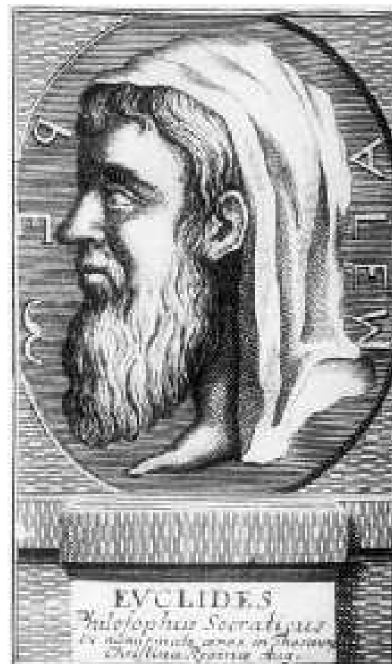
Oh, Nim is *so* easy. I like Nim.

```
frilled.cs.indiana.edu%java Nim
Hello, and welcome to the game of Nim!
What is your name: Mary-Ann
Game starts with a pile of height: 11
*** Computer moves.
Pile of marbles of height: 11
Computer chooses to remove: 4 marbles.
***Removing 4 marbles from the pile for: Computer
Pile of marbles is now: 7
-------------------------
*** Now Mary-Ann has to move.
Pile of marbles of height: 7
Mary-Ann, please enter number of marbles you want to take: 6
***Removing 6 marbles from the pile for: Mary-Ann
***Bad move for Mary-Ann. Mary-Ann loses.
frilled.cs.indiana.edu%
```

"Why, Mary Ann, what are you doing out here?"

"Run home this moment, and fetch me a pair of gloves and a fan! Quick, now!"

Designing Fractions



Designing `Fractions`.

You mentioned De Morgan's name yesterday.	Augustus De Morgan (1806-1871), indeed.

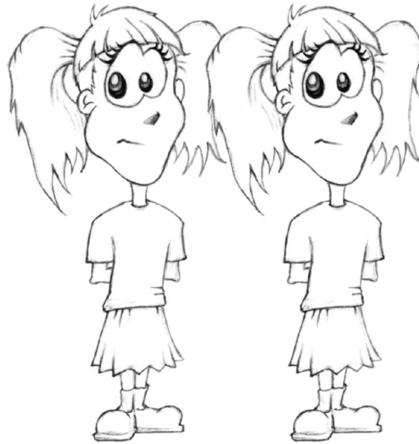
`http://www-groups.dcs.st-andrews.ac.uk/~history/Mathematicians/De_Morgan.html`

De Morgan was the one that, in 1838, defined and introduced the term *mathematical induction*, thus putting a process that had been used without clarity on a rigorous basis.	I don't know if he ever played *Nim*, but here's how he was described by some of his colleagues: "*A dry dogmatic pedant I fear is Mr De Morgan, notwithstanding his unquestioned ability.*"
In 1866 he was a co-founder of the London Mathematical Society and became its first president.	De Morgan was never a Fellow of the Royal Society as he refused to let his name be put forward. He also refused an honorary degree from the University of Edinburgh.
He recognised the purely symbolic nature of algebra and he was aware of the existence of algebras other than ordinary algebra.	He introduced De Morgan's laws and his greatest contribution is as a reformer of mathematical logic.
Very interesting. What other mathematicians were involved in the lecture notes of yesterday?	Charles Lutwidge Dodgson (1832-1898).

`http://www-groups.dcs.st-andrews.ac.uk/~history/Mathematicians/Dodgson.html`

Charles Dodgson is known especially for *Alice's adventures in wonderland* (1865) and *Through the looking glass* (1872), children's books that are also distinguished as satire and as examples of verbal wit.	He invented his pen name of Lewis Carroll by anglicizing the translation of his first two names into the Latin '*Carolus Lodovicus*'.
As a mathematician, Dodgson was conservative.	He was the author of a fair number of mathematics books, for instance *A syllabus of plane algebraical geometry* (1860).

None of his math books have proved of enduring importance except for *Euclid and his modern rivals* (1879) which is of historical interest.	Didn't he write *"The Hunting of the Snark"*?
You bet he did.	As a logician, he was more interested in logic as a game than as an instrument for testing reason.
"I know what you're thinking about," said Tweedledum: "but it isn't so, nohow."	"Contrariwise," continued Tweedledee, "if it was so, it might be; and if it were so, it would be: but as it isn't, it ain't. That's logic."
Here's a picture of him.	Yes, he's the one in the middle.



He contributed in *Jabberwocky*, the word *chortle* (a word that combines *snort* and *chuckle*) to the English language.

```
frilled.cs.indiana.edu%webster chortle
chor-tle vb  chor-tled; chor-tling
[blend of chuckle and snort]
vi
(1872)
1: to sing or chant exultantly <he chortled in his joy --Lewis
     Carroll>
2: to laugh or chuckle esp. in satisfaction or exultation
~ vt :to say or sing with a chortling intonation
-- chortle n
-- chor-tler n
frilled.cs.indiana.edu%
```

Yet *momeraths* and *brillig* didn't quite make it.

Today we're going to implement `Fractions`. And in the process mention Euclid (325-265).

http://www-groups.dcs.st-andrews.ac.uk/~history/Mathematicians/Euclid.html

Have you noticed the numbers? Yes, that was a *long* time ago!

Euclid's most famous work is his treatise on mathematics *The Elements*. The book was a compilation of knowledge that became the centre of mathematical teaching for 2000 years.

The *Elements* is divided into 13 books. Books one to six deal with plane geometry.

Books seven to nine deal with *number theory*. In particular book seven is a self-contained introduction to number theory and contains the Euclidean algorithm for finding the greatest common divisor of two numbers.

Which we will use today. Very good.

A fraction is of course the ratio of two integers:

- a numerator and
- a denominator

We will define a class `Fraction`, which will supply the necessary operations on fractions.

There are many ways in which we could do this. Here's a summary, to be augmented with a more detailed explanation in class.

Let's look first at the new part. Of which there are two parts, as well.

```
class Euclid {
    static int gcd(int a, int b) {
        a = Math.abs(a);
        b = Math.abs(b);
        if (a == 0) return b; // 0 is error value
        if (b == 0) return a;
        int temp;
        while (b > 0) {
            temp = a % b;
            a = b;
            b = temp;
        }
        return a;
    } // there are other ways too...
}
```

We need to get the hang of it, first.

Then, when we become comfortable using it, we need to become sure it always works right. Proof is the bottom line for everyone.

That's from Paul Simon, isn't it?	Yes, but it applies here.
Both goals may take a long time.	But when you're done you can write `Fraction`.
Like this.	That's a lot.

```
class Fraction {
    private int numerator;
    private int denominator;
    public Fraction(int num, int den) {
        int divisor;
        if (den == 0) {
            System.out.println(" Fraction with denominator zero!");
            System.exit(1);
        }
        if (num == 0) { numerator = 0; denominator = 1; }
        else {
            if (den < 0) {
                num *= -1;
                den *= -1;
            }
            if ((divisor = Euclid.gcd(num, den)) != 1) {
                num /= divisor;
                den /= divisor;
            }
            numerator = num;
            denominator = den;
        }
    }
    public String toString() {
        String fraction;
        if (denominator == 1) { fraction = numerator + ""; }
        else { fraction = numerator + "/" + denominator; }
        if (denominator * numerator < 0) {
            return "(" + fraction + ")";
        } else {
            return fraction;
        }
    }
    public boolean isZero() {
        return (denominator == 1 && numerator == 0);
    }
    public boolean isInt() {
        return (denominator == 1);
    }
    public boolean equals(Fraction other) {
        return (numerator == other.numerator && denominator == other.denominator);
    }
    public boolean greaterThan(Fraction other) {
        return (numerator * other.denominator >
```

```
                denominator * other.numerator);
    }
    public Fraction minus(Fraction other) {
        return new Fraction(
                     numerator * other.denominator - other.numerator * denominator,
                     denominator * other.denominator
                   );
    }
    public Fraction plus(Fraction other) {
        return new Fraction(
                     numerator * other.denominator + other.numerator * denominator,
                     denominator * other.denominator
                   );
    }
    public Fraction times(Fraction other) {
        return new Fraction(numerator * other.numerator, denominator * other.denominator);
    }
    public Fraction divideBy(Fraction other) {
        return new Fraction(numerator * other.denominator, denominator * other.numerator);
    }
    public static void main(String[] args) {
        Fraction f = new Fraction(6, 9);
        Fraction g = new Fraction(-4, 6);
        System.out.println("Test of operations: ");
        System.out.println("  Add: " + f + " + " + g + " = " + f.plus(g));
        System.out.println("  Sub: " + f + " - " + g + " = " + f.minus(g));
        System.out.println("  Mul: " + f + " * " + g + " = " + f.times(g));
        System.out.println("  Div: " + f + " / " + g + " = " + f.divideBy(g));
        System.out.println("Test of predicates: ");
        System.out.print("  1. Does " + f + " equal " + g + "? ");
        System.out.println(" The answer is: " + f.equals(g));
        Fraction h = new Fraction(8, -2);
        System.out.print("  2. Is " + h + " an integer? ");
        System.out.println("The answer is: " + h.isInt());
        Fraction i, j;
        i = (f.minus(g)).times(f.plus(g));
        j = f.times(f).minus(g.times(g));
        System.out.print("  3. Does " + i + " equal " + j + "? ");
        System.out.println("The answer is: " + i.equals(j));
        System.out.print("  4. Is 5/8 greater than 2/3? The answer is: ");
        System.out.println((new Fraction(5, 8)).greaterThan(new Fraction (2, 3)));
    }
}
```

There are two parts to it.	First, the blueprint.
Then, the `main`.	Both important.

When you're done you can improve it.

```
boolean equals(Fraction other) {
    return (this.minus(other)).isZero();
}
```

That's a different equals.

Here's a different greaterThan.

```
boolean greaterThan(Fraction other) {
    return (this.minus(other)).isPositive();
}
```

That wouldn't work just yet.

I know, you need another predicate.	Can I write it?
Sure, what's its signature?	`boolean isPositive()` is it's signature.

It's a simple one.
`boolean isPositive() {`
`    return numerator * denominator > 0;`
`}`
	I agree.

This was a long example.	Long, but useful.
And interesting.	If you say so...

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

Milestones

More about methods.

We've reached an important milestone now.	Although you may not realize it now,
...it will become apparent in about a week.	Meanwhile I think it's reasonably safe to say that we're *halfway* through now.
I think so. Diagrams like the ones presented in the previous sets of lecture notes are important.	They can help you understand what's happening inside a program when it's compiled and run.
You should not expect to use diagrams all the time, always, for each and every program.	Their purpose is mostly to help you understand the concepts, the basics, by providing a very detailed picture, as if under a *microscope*.
Once you understand those, you're all set.	And you can think Java without drawing diagrams. You'd be manipulating them in your mind, almost without knowing it.
Today in class we'll touch on lecture notes of Wednesday and then start talking (again) about methods.	We'll finish them quickly, although there will be one important new thing we will touch on.
Recursion.	The saying goes that "*to understand recursion you first need to understand recursion*".
That's just a humorous saying.	Recursion, in fact, is easy, and thoroughly useful.
Once you have a fixed point.	(But we'll get to that shortly.)
Not to mention that we have already seen it.	Yes, it was that method in lab five.
That method, `what`'s its name?	Well, `what` *was* its name, but we changed it.
That we did.	Now let's start the chapter on methods.

Methods.	What about them?
All about them.	You have already implemented several simple methods and are familiar with the basic concepts.
Let's go over parameters, return values, and variable scope in a more systematic fashion.	We will also review some of the more technical issues, such as `static` methods and variables.

When we implement a method we define the parameters of the method.

Here's an example:

```
public class BankAccount {
  ...
  public void deposit(double amount) {
    ...
  }
  ...
}
```

The `deposit` method has two parameters: one is explicit, called `amount`, and has type `double`.	We expect to receive a value in it to be deposited in the account.
The other one is implicit, and can be referred to by the the keyword `this`.	What we mean by that is that inside an instance method (like `deposit`) the keyword `this` will always refer to the host object.
So if it's always available and always in the same way, the `this` reference is not even mentioned in the list of parameters.	But from any instance method we can use the keyword `this` to refer to the object that contains the method, always.
Therefore `amount` is called a *formal* parameter for the method `deposit`.	When we want to deposit some money we need to know two things:
a) the account's name, and	b) the amount of money
...and we need to actually invoke the method,	...in order to deposit the money.

```
// somewhere in main (or another method)
myChecking.deposit(allowance - 200);
```

In this example `myChecking` is an object of type `BankAccount`...	...and `allowance` is probably a `double`.

Both (are names, and the names) should be declared and initialized before we use them.	When `deposit` starts its work, the value of the `allowance - 200` expression becomes the • *actual parameter* or • *argument* to the method...
...and will be known by the name of `amount`	...while `deposit` is running (working on it).
When the method *returns* (or ends) the formal parameter variables are abandoned (they're disposable) and their values are lost.	The entire process is like a phone call: you call `myChecking`'s `deposit` method and you give it some input: the `amount` that you want to deposit.
Once it has that it immediately starts working for you and you stay on the line, waiting for it to tell you: I am done, and your *transaction is completed.*	You can't write your checks before that.
When it's done it says so, before you hang up.	Sometimes a method also *returns* a value before the end of conversation, but not `deposit`.
`deposit` only says when it's done, without returning anything. It is declared as `void`.	`void` is its return *type.*
Yes: it does not return *anything* to its caller.	Hence: `void`.
It only does what it is supposed to do, and then it says: *"I'm done."* And `balance` has changed.	One could also call this *returning* except it's not as in "*returning a value*", but rather...
...more like in "*returning from a trip*".	A trip to the bank.
Explicit parameter variables (the formals) are no different from other variables.	You can modify them during the execution of a method: but unless you have a good reason for that, that is considered bad style.

Let's now consider a more complicated example:

```
public class BankAccount {
  ...
  public void transfer(BankAccount other, double amount) {
    withdraw(amount);
    other.deposit(amount);
  }
  ...
}
```

This method can be used to transfer money from one account to another.

Here's how we can use it:

```
momsSavings.transfer(myChecking, allowance);
```

I have one question before we go further though...

Yes, what is it?

Weren't we supposed to write

```
this.withdraw(amount)
```

in the definition of the `transfer` method?

Yes, since we decided to always put an object reference in front of any instance variable name, so please make the correction in your notes.

OK, I've updated mine.

But why does it work though?

Because it defaults to it, anyway.

But I think that using `this` makes the code more uniform and explicit.

And I think so too.

How many formal parameters does this new function (method) have?

Two of them are explicit: `other` and `amount`.

The first one is a `BankAccount`.

The second one is a `double`.

And in addition to that the method will be able to access the object to which it belongs using `this`.

Yes, `this` is always available in an instance method and means: *"the object that contains this method"*, or *"this method's host"*.

What happens when the method is invoked?

When the method is invoked the reference to `myChecking` is copied into the method's `other` formal parameter...

...and `allowance` will be copied into `amount`.

Note that both the object references and the numbers are *copied* into the method.

After the method exits, the two bank account balances have changed.

The method was able to change the accounts because it received copies of the object *references*.

Of course, the contents of the `allowance` variable was not changed.

In Java no method can modify the contents of a *number* variable that is passed as a parameter.

Or the contents of any other variable of *primitive* type, for that matter.

Yes: parameters are always passed by value.

What names should we give to the parameters?

You can give them any names you want.

As a rule, choose explicit names for parameters that have specific roles. Choose simple names for those that are completely generic.

Your goal is to make the reader understand the purpose of the parameter without having to read the method's description.

The compiler takes the types of the method parameters and return values very seriously.

Very very very seriously.

It is an error to call a method by passing it a value of incompatible type,

...or to use the method in a context that is not compatible with its return type (if any).

Java is a *strongly typed* language.

That, it is.

The compiler automatically converts from `int` to `double` and from ordinary classes to superclasses (as we will see when we talk about inheritance).

But it does not convert when there is a possibility of information loss, as we have seen when we discussed *casting*,

...and does not convert between numbers and strings and objects.

This is a useful feature, because it lets the compiler find programming errors before they create havoc when the program runs.

A method that accesses an object and returns some information about it, without changing the object, is called an *accessor* method.

Such as `getBalance`.

In contrast, a method that modifies the state of an object is called a *mutator* method.

`deposit` and `withdraw` are mutator methods.

You can call an accessor method

...as many times as you like.

If that's all you do, you will always get the same answer, and it does not change the state of the object.

Some classes have been designed such that objects of that kind have only accessor methods and no mutators at all.

Such classes are called *immutable*.

An example is the `String` class.

Once a `String` has been constructed, its contents never change.

For example, the `substring` method does not remove characters from the original string.

Instead it constructs a *new* string that contains the substring characters.

Here's another example of an *accessor* method that simultaneously looks at two objects:

```
public class BankAccount {
  public boolean equals(BankAccount other) {
    return (this.getBalance() == other.getBalance());
  }
}
```

It makes use of two other accessors (or, rather, the same accessor invoked on two different objects) and compares the values that they return, to come up with an answer.

And the answer that it returns is the truth value that comes out of (and describes) the comparison.

So we could use it as follows:
`if (account1.equals(account2)) {`
`  // they have the same balance...`
`} else {`
`  // they do not have the same balance...`
`}`
	Very good.

In general the expectation is that accessor methods do not modify any parameters, ...	... and that *mutator* methods do not modify any parameters beyond `this`.
This ideal situation is not always the case.	Like the `transfer` method discussed before.
It changed its `this`, ...	... while also updating the `other` account.
Such a method is said to have a *side effect*.	A side effect of a method is any kind of *observable behaviour* outside the object.
In an ideal world, all methods would be accessors.	They would simply return an answer without changing any value at all.
In fact, programs that are written in so-called functional programming languages, such as Scheme or ML, come close to this ideal.	`Scheme` is the best! Java is also good.
In an object oriented programming language, we use objects to remember state changes.	Therefore, a method that just changes the state of its implicit parameter is certainly acceptable.
A method that does anything else is said to have a side effect.	While side-effects cannot be completely eliminated, they can be the cause of surprises and problems and should be minimized.
Sometimes you write methods that don't belong to any particular object.	Such a method is called a `static` (or *class*) method and needs to be declared as `static`.
In contrast, methods such as `getBalance`, `withdraw`, and `deposit` in the preceding sections are often called *instance* methods,	... because they operate on particular instances of an object. There's one `getBalance` for each `BankAccount` (object) that gets created.
Have we seen any `static` methods?	`Math.sqrt` is a static method.
And every application must have a `static` method where processing begins, called	... `main`.

Correct.

Here's another example, that involves only numbers:

```
class NumericMethods {
  public static boolean approxEqual (double x, double y) {
    final double EPSILON = 1E-14;
    double xymax = Math.max(Math.abs(x), Math.abs(y));
    return Math.abs(x - y) <= EPSILON * xymax;
  }
  // more numeric methods could come here...
}
```

This method encapsulates computation that involves no objects at all, only numbers (and `booleans`), hence only primitive types.

To call (or *use*) a static method you need to supply the name of the class, for example:

```
double r = Math.sqrt(2);
if (NumericMethods.approxEqual(r * r, 2))
  System.out.println("Math.sqrt(2) is approx. 2");
```

...same as we do with `Math.sqrt`.

Now it should be clear to you why the `main` method is a `static` method.

...when the program starts there may not be any objects at all.

Therefore the *first* method to be called in a program must be a static method.

Good enough.

To summarize our knowledge about static methods we can say that...

...a static method is a method that does not belong to any object, and that has only explicit parameters. (No `this`!)

Let's look at some examples now.

What does the following example illustrate?

```
public class Example {
  public static void addOneToIt (int number) {
    System.out.println(number);
    number = number + 1;
    System.out.println(number);
  }
  public static void main(String[] args) {
    int value = 3;
    System.out.println(value);
    Example.addOneToIt(value);
    System.out.println(value);
  }
}
```

Let's walk through the method call.

When the call is made the parameter value is set to the same value as the argument.

The value is copied.	Changes to it are not seen outside.
That's all there is to it.	Easy.
Is there a moral to it?	In Java method parameters are *copied* into the parameter variables when a method starts.
Computer scientists call this call mechanism "call by value", and we mentioned it in lab 2.	As you have just seen there are some limitations to the "call by value" mechanism.
It is not possible to implement methods that modify the contents of number variables.	Other programming languages support an alternate mechanism, called "by reference".
This involves passing only the address to where the number variable is stored.	This is what happens when you pass an object as an actual parameter.
Let's see an example.	Oh, boy. I like examples best.

```
class NumberHolder {
  int value = 1;
}
class Example {
  public static void main(String[] args) {
    NumberHolder n = new NumberHolder();
    System.out.println(n.value);
    Example.addOneToIt(n);
    System.out.println(n.value);
  }
  public static void addOneToIt (NumberHolder n) {
    n.value = n.value + 1;
  }
}
```

But we've seen this before, haven't we?

Yes, when we discussed copying of variables.	Primitive types are copied *by value*, while reference types are copied *by reference*.
Good enough.	References though are still passed *by value*.
Understood. Can we see an example?	Oh boy – that's what I like best.

```
class Pair {
  double x;
  double y;
  Pair(double x, double y) {
    this.x = x;
    this.y = y;
  }
  void report() {
```

```
    System.out.println("Hello! I'm at: (" + x + ", " + y + ")");
  }
}
class Testing {
  public static void main(String[] args) {
    Pair a = new Pair(100, 0);
    Pair b = new Pair(0, 100);
    a.report();
    b.report();
    Testing.swap(a, b);
    a.report();
    b.report();
  }
  static void swap(Pair a, Pair b) {
    Pair temp = a;
    a = b;
    b = temp;
  }
}
```

I like it.

Easy and understandable. But it still gives you a level of indirection.

Yes. You can, at least in principle, get *inside* those `Pairs`.

Let's summarize: a Java method can update an object's state using the reference to it, but it cannot change the contents of a reference any more it can change a variable of primitive type.

This shows that object references are passed by value in Java, although we can safely say that

. . . objects themselves are passed *by reference.*

Except that the reference itself is *copied.*

Copied, yes – but pointing to the same thing that the original one was.

Fair enough.

The distinction is clear now.

A method that has a return type other than `void` must return a value, by executing a statement of the form:

```
return <expression>;
```

Been there, done that.

Yes, but let's see if we can come up with something new.

Well, for one thing, you can return the value of any expression.

You don't need to store the result in a variable and then return the variable.

When a `return` is processed, the method exits *immediately.*

This is convenient for handling exceptional cases in the beginning.

Oh, yes, here's an example:

```
public static int fibo (int n) {
  if (n == 1)
    return 1;
  else if (n == 2)
    return 1;
  else {
    int fOlder = 1;
    int fOld = 1;
    int result = fOld + fOlder;
    for (int i = 3; i <= n; i++) {
      result = fOld + fOlder;
      fOlder = fOld;
      fOld = result;
    }
    return result;
  }
}
```

These are Fibonacci numbers!

http://www-groups.dcs.st-andrews.ac.uk/~history/Mathematicians/Fibonacci.html

---

Or rather, the method that computes them.

Picky, picky, picky! What can I say.

Can *you* give me an example?

Sure, how about `add`, below.

```
class Fraction {
  int num;
  int den;
  Fraction(int a, int b) {
    this.num = a;
    this.den = b;
  }
  public String toString() {
    return " (" + num + "/" + den + ") ";
  }
  Fraction add(Fraction other) {
    return new Fraction(this.num * other.den + this.den * other.num,
                        this.den * other.den);
  }
  public static void main(String[] args) {
    Fraction a = new Fraction(1, 3);
    Fraction b = new Fraction(2, 3);
    System.out.println(a.toString());
    System.out.println(b);
    System.out.println(a.add(b));
  }
}
```

That's a good example, and so is...

...`toString`. But I like `add` better.

It is important that every branch of a method return a value, that is, a method cannot end without returning a value (if its return type is other than `void`).

Also, a method whose return type is not `void` always needs to return a value. Oh, you just said that! Nevermind, although reinforcement is good.

If the method contains several `if/else` branches make sure that each one of the branches returns an adequate value.

At the end of every possible path through a non-`void` method there should be a `return` statement, returning the value of an expression of compatible type.

For example is this right?

It is not.

```
public static int fibo (int n) {
  if (n <= 0)
    System.out.println("Incorrect argument!");
  else if (n == 1)
    return 1;
  else if (n == 2)
    return 1;
  else {
    int fOlder = 1;
    int fOld = 1;
    int result = fOld + fOlder;
    for (int i = 3; i <= n; i++) {
      result = fOld + fOlder;
      fOlder = fOld;
      fOld = result;
    }
    return result;
  }
}
```

It is not, because if the argument is negative we don't return anything.

What should we return, then?

I don't know, what do you think of this one?

```
return Math.round( (Math.pow((1 + Math.sqrt(5))/ 2, n) -
                    Math.pow((1 - Math.sqrt(5))/ 2, n) ) / Math.sqrt(5) );
```

Ha, that was a good one!

Or we should throw an `Exception`.

Yes, but about those perhaps some other time...

We have now encountered the four kinds of variables that Java supports.

1. Instance variables
2. Static variables
3. Local variables
4. Parameter variables

The *lifetime* of a variable defines when the variable is created and how long it stays around.	When an object is constructed, all its instance variable are created.
As long as the object is around its instance variables will also be there, inside the object.	A static variable is created when its class is first loaded, and it lives as long as the class.
A local variable is created when the program enters the statement that defines it.	It stays *alive* until the block that encloses the variable definition is exited.

Here's an example:

```
public void withdraw (double amount) {
  if (amount <= balance) {
    double newBalance = balance - amount;
    // local variable newBalance created and initialized
    balance = newBalance;
  } // end of lifetime of local variable newBalance
}
```

If you tried to print `newBalance` right before the end of the method you'd get an error.

Yes, and the reason is: it's known only in the *then* branch of the `if` statement.	Inside the inner pair of curly braces.
Can you say that again?	Inside the inner pair of curly braces, *only*.
Very good.	Good to remember.
Finally, when a method is called, its parameter variables are created.	They stay alive until the method returns to the caller. They're disposable. Every time a new set is used. Fresh. New scratch paper, as in `what`.
Next, let us summarize what we know about the *initialization* of these four types of variables.	Instance variables and static variables are automatically initialized with a default value...
... which is • `0` for numbers (and `chars`), • `false` for `boolean` and • `null` for objects (ref. types),	Yes. So instance variables and static variables are automatically initialized with a default value unless you specify another initial value.
So constructors are not *essential.*	They're *hygienic* instead: convenient and clean.
Parameter variables are initialized with copies of the actual parameters.	That's when the method gets called.
Local variables are not initialized by default.	For local variables you must supply an initial value, and the compiler complains if you try to use a local variable that you never initialized.

The *scope* of a variable is that part of a program that can access it.	The part of the program in which you can access it, the variable, is the *scope* of the variable, yes.
OK. As you know, instance and static variables are usually declared as `private`, and you can access them only in the methods of their class.	I see... Scope answers the question: can I see it?
The scope of a local variable extends from the point of its definition to the end of the enclosing block.	The scope of a parameter variable is the entire body of its method.
Now let's look a bit closer to a few situations.	We're going to go through a few examples.

It sometimes happen that the same variable name is used in two methods:

```
public static double area(Rectangle rect) {
  double r = rect.getWidth() * rect.getHeight();
  return r;
}
public static void main(String[] args) {
  Rectangle r = new Rectangle(5, 10, 20, 30);
  double a = area(r);
  ...
}
```

These variables (the two `r`'s) are independent of each other.

You can have variables with the same name `r` in different methods,	...just as you can have different motels with the same name (let's say, `"Super 8"`) in different cities.
In this situation the scopes of the two variables are disjoint.	Problems arise, however, if you have two or more variable names with overlapping scope.
Like when you have two Kroger's in the same city.	Almost, but not exactly. In Java this situation is called *shadowing*.
There are rules in the language that tell you which one of the variables you will be referring to if you use the ambiguous name.	Can we see some examples?

Certainly.

```
class Employee {
  String name;
  Employee (String name) {
    this.name = name;
    // this is mandatory not just good style here!!
  }
}
```

	The parameter, which is like a local variable, shadows the instance variable.
The Java language specifies that when there is a conflict between a local variable name and an instance variable name the local variable wins out.	This sounds pretty arbitrary but there is actually a good reason.
You can still refer to the instance variable using `this`	Which you should do *anyway*.
Do you have any questions?	No, but I have something close to that.
An example!	You bet.

Consider this:

```
class Puzzle {
  public static void main(String[] args) {
    Puzzle p = new Puzzle();
    System.out.println("Final result: " + p.fun(6));
  }

  int fun(int n) {
    int result;
    if (n == 0) return 0;
    else {
      // [1]
      result = n + fun(n - 1);
      // [2]
      return result;
    }
  }
}
```

Neat. What do we do with it?	Well, what's the program computing?
This is our old friend `what`.	Yes, `what`'s its name.
But notice the new name of the method.	I know, I know, this is a lot of `fun`.
Well, isn't it?	I could make it real fun, you know.
How.	I could show you the real power of recursion.
I'd like to see that.	Consider the Tower of Hanoi problem.
I keep hearing about this problem...	Yes. Let me state it here briefly.

This is a neat little puzzle invented by the French mathematician Edouard Lucas[22].

We are given a tower of eight disks, initially stacked in decreasing size on one of the three pegs. The objective is to transfer the entire tower to one of the other pegs, moving only one disk at a time and never moving a larger disk onto a smaller one.

Let's solve the problem in general.	Base case first: one disk is easy.
Now for all other cases by induction.	Assume we can solve the problem for n-1 disks.
Then the general case becomes easy.	Place the top n-1 disks on middle peg first.
Then move the largest disk.	Then bring the n-1 disks back on top of it.
Can I see the program?	Here it is:

```
frilled.cs.indiana.edu%cat Hanoi.java
class Hanoi {
    public static void main(String[] args) {
        int size = 4; // number of disks
        move(size, "source", "middle", "target");
    }
    static void move(int height, String peg1, // from
                                 String peg2, // using
                                 String peg3  // to
                    ) {
        if (height == 1) {
            System.out.println("Move disk from " + peg1 + " to " + peg3);
        } else {
            move(height-1, peg1, peg3, peg2);
            System.out.println("Move disk from " + peg1 + " to " + peg3);
            move(height-1, peg2, peg1, peg3);
        }
    }
}
frilled.cs.indiana.edu%javac Hanoi.java
frilled.cs.indiana.edu%java Hanoi
Move disk from source to middle
Move disk from source to target
Move disk from middle to target
Move disk from source to middle
Move disk from target to source
Move disk from target to middle
Move disk from source to middle
Move disk from source to target
Move disk from middle to target
```

[22] http://www-groups.dcs.st-andrews.ac.uk/~history/Mathematicians/Lucas.html

```
Move disk from middle to source
Move disk from target to source
Move disk from middle to target
Move disk from source to middle
Move disk from source to target
Move disk from middle to target
frilled.cs.indiana.edu%
```

Now *that* was a lot of fun!

I sure think so.

Java Arrays

Part One.

Suppose that you want to write a program...	I have a feeling of *deja vu*.
...that reads a set of prices offered by 10 (*ten*) vendors for a particular product, and *then* prints them, marking the lowest one.	Sounds interesting.
Of course, you need to read in *all* data items first, before you can start printing them.	You can't print them as you read them, can you?
No, your program has to wait until the *last* of the ten prices has been entered,	...and then print all the items.
Exactly.	If I could be sure that it's *always* the case that there are exactly ten data items, ...
...then you could store the prices in *ten* variables: `data1`, `data2`, ...`data10`.	Hey, that was *my* idea!
But such a sequence of variables is not very practical to use.	It *isn't*?
Well, what if you had *a hundred* data items?	Ugh...
Or *a thousand*? You would have to write quite a bit of code...	Or what if the number of vendors is *unknown*,
...to be specified by the user of your program at run-time.	*Then* what do we do?
Wouldn't it be nice if you could call the *entire* set of prices by just one *name*...	...such as `price`
...denoting the entire sequence?	It would, but only if we could easily get to the individual elements of the sequence, like this: $\texttt{price}_1$, $\texttt{price}_2$, ..., $\texttt{price}_n$,

... where `n` could be even specified by the user, at run-time (when the program is run).	Boy, *that* would be nice!
That would be a better way of storing such a sequence of data items, wouldn't it?	Yes, it *would* be.
Fortunately Java has a construct that is designed just for such a circumstance.	The *array* construct.
An *array* is a collection of data items of the same type.	Every element of the collection can be accessed separately.
Here's how you define an array of ten floating-point numbers: `double[] price = new double[10];`	That was a mouthful. Can we take it apart?
Yes, let's do it in stages.	In Java, arrays are objects.
We'll get to that in a second.	Essentially we want to define a variable with the name of `price`.
Exactly, but this variable is of type *array* of `doubles`.	We use the square brackes (`[]`) to denote *array*.
So an *array* of `doubles` is declared as `double[]`	And to declare a variable `price` of this type you need to say: `double[] price;`
I see you put the type in blue, and the name of the variable in red.	Entirely correct.
Now you have a variable (or an array name) but there is no array as of yet.	Have we seen this before?
How about `Rectangle a;`	Same thing.
A variable `a` is defined, that could store references to a `Rectangle` object,	... but there's no actual `Rectangle` as of yet.
I could create one like that: `a = new Rectangle(5, 10, 15, 20);`	Can you do the same with the an *array*?
Yes, in the following way: `price = new double[10];`	And do we have the *array* now?

Yes, the call `new double[10]` creates the actual array of 10 numbers.	Every element of the collection can be accessed separately.
When an array is first created all values are initialized with `0`	... for an *array* of numbers such as `int[]` or `double[]`,
... `false` for a `boolean` array,	... or `null` for an array of objects.
You mean you could create an array of `Rectangles` too?	Of course, how many `Rectangles` do you anticipate you might later need?
How about also `10`?	Then it looks almost the same:

```
Rectangle boxes = new Rectangle[10];
```

Very good.	All ten slots in array `boxes` are currently `null`.
Indeed.	To get some values into the array you need to specify which slot in the array you want to use.
That is done with the `[]` operator.	It must follow the name of the array and enclose an integer-valued expression
... called an *index* or a *subscript*.	Now you need to remember a thing about `Strings`.
Why `Strings`?	Because it is the same for arrays, and very important here.
What is that?	The first element in an array has index `0`.
Just like the first character in a `String s` is accessed with `s.charAt(0)` (we've discussed this before).	So the actual elements in my `price` array will be identified as: $\texttt{price}_0$, $\texttt{price}_1$, ..., $\texttt{price}_{n-1}$ then.
Yes, and `n` is `10`	... so the subscripts go from `0` to `9`.
Also, Java syntax for $\texttt{price}_i$	... is really `price[i]`,
... where `i` is an integer-valued expression.	In this case, an `int` variable.
You can't use `long`, can you?	Only if you *cast* it to `int`.
`int` works for me.	The index in an array reference has a major restriction.
What is that?	Trying to access a slot that does not exist in the array is an error.

So `price[10]` would be such an error?

Yes, it would be, in our case.

Good. Now that we have reviewed all this let's start on the program.

```
double lowest = price[0];
for (int i = 1; i < price.length; i++)
  if (price[i] < lowest)
    lowest = price[i];
```

Here's how you could find out the *lowest* price in an array of prices:

What's that: `price.length`?

You've seen something similar with `String`s.

Note that there are *no parentheses* following `length` and so we can tell about `length` this:

. . . it is an *instance variable* of the *array* object,

. . . not a method.

Oh, man, this is nifty!

However, you cannot assign a new value to this instance variable.

In other words, `length` is a final public instance variable.

This is quite an anomaly.

Normally, Java programmers use a method to inquire about the properties of an object.

You just have to remember to omit the parentheses in *this* case.

Using `length` is a much better idea than using a number such as `10`,

---

. . . even if you know that the array has ten elements.

Note that `i` is a legal index for an array `a` if `0 <= i` and `i < a.length`

Therefore the `for` loop

```
for (int i = 0; i < a.length; i++)
   do something with a[i]
```

. . . is extremely common for visiting all elements in an array.

Can we write the program now?

Yes, let's do that:

```
class Price {
    public static void main(String[] args) {
        double[] price = new double[10];
        ConsoleReader c = new ConsoleReader(System.in);
        System.out.println("Please specify the ten prices:");
        for (int i = 0; i < 10; i++) {
            System.out.print(i + "> ");
            price[i] = c.readDouble();
            // the user presses enter!
        }
        System.out.println("Thank you!");
        double lowest = price[0];
```

```
        for (int i = 0; i < price.length; i++) {
            if (price[i] < lowest) {
                lowest = price[i];
            }
        }
        System.out.println("*** Lowest price computed.");
        System.out.println("Here are the prices: ");
        for (int i = 0; i < price.length; i++) {
            System.out.print("Price " + (i + 1) + ": ");
            System.out.print(price[i]);
            if (lowest == price[i]) }
                System.out.println(" *** lowest price");
            } else {
                System.out.println();
            }
        }
    }
}
```

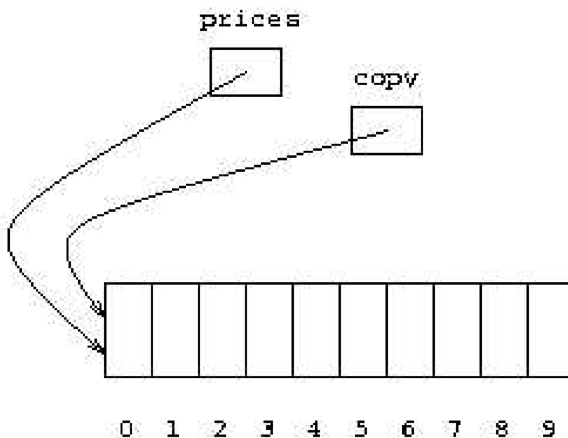
This program illustrates the points discussed thus far.	I could improve on it, and in many ways.
We'll do that in a few minutes.	May I, at least, show you a sample run?
Certainly.	Here it is:

```
frilled.cs.indiana.edu%java Price
Please specify the ten prices:
0> 3
1> 4
2> 5
3> 6
4> 3
5> 4
6> 5
7> 6
8> 3
9> 4
Thank you!
*** Lowest price computed.
Here are the prices:
Price 1: 3.0 *** lowest price
Price 2: 4.0
Price 3: 5.0
Price 4: 6.0
Price 5: 3.0 *** lowest price
Price 6: 4.0
Price 7: 5.0
Price 8: 6.0
```

```
Price 9: 3.0 *** lowest price
Price 10: 4.0
frilled.cs.indiana.edu%
```

Looks good.	I thought so too.
How do you copy an array?	How do you copy a `Rectangle`?
Array variables work just like object variables.	They hold a *reference* to the actual array.

If you copy the reference,
`double[] prices = new double[10];`
`// ...  fill array`
`double[] copy;`
`copy = prices;`
	... you get another reference to the same array.

Both `prices` and `copy`	... point to the exact same thing.
If you were to change `copy[i]`	... you would see the change in `prices[i]`
... and that's because both `copy` and `prices`	... are different names for one and the same *array*.
Here's a picture: 	I like that.
If you want to make a true copy of an array, you must make a new array	... of the same length as the original,

... and copy over all values.
`double copy[] = new double[prices.length];`
`for (int i = 0; i < prices.length; i++)`
`  copy[i] = prices[i];`
	Like this?

Yes. You can specify the size of the array through any integer-valued expression,	...so `prices.length` works just fine.
Instead of the `for` loop you can also use the `System.arrayCopy` method.	It will be my pleasure to look it up in the API.

`http://java.sun.com/products/jdk/1.2/docs/api/overview-tree.html`

Writing the `for` loop should also be *pleasurable*.	Plus I need to practice.
How do you initialize an *array*?	Like we did above.
Indeed, we can allocate it, then fill each entry.	What if we know the elements at the time we write the program?
Then there's an easier way.	You can list all the elements that you want to include in the array,
...enclosed in braces,	...and separated by commas.

```
int[] primes = { 2, 3, 5, 7, 11 };
```

Can we do it in two steps?	Try it.

```
int[] primes;
primes = { 2, 3, 5, 7, 11 };
```

Can you also tell me why?	What was the error message?
Then the answer is: no.	And we need to remember that *array constants can be used only in initializers.*
Sounds good so far.	Now a challenge.
What is it?	Could you improve `Price.java` to behave in the following way:

```
frilled.cs.indiana.edu%java Price
How many prices?
3
Please enter the 3 prices.
Enter1> 3.45
Enter2> 1.20
Enter3> 6.34
Thank you!
*** Lowest price computed.
Here are the prices:
Price 1: 3.45
Price 2: 1.2 *** lowest price
```

```
Price 3: 6.34
frilled.cs.indiana.edu%
```

I could try.

Here's the solution, just in case.

```
class Price {
    public static void main(String[] args) {
        ConsoleReader c = new ConsoleReader(System.in);
        System.out.println("How many prices?");
        int size = c.readInt(); }
        double[] price = new double[size];
        System.out.println("Please enter the " + size + " prices.");
        for (int i = 0; i < size; i++) {
            System.out.print("Enter" + (i + 1) + "> ");
            price[i] = c.readDouble();
            // the user presses enter!
        }
        System.out.println("Thank you!");
        double lowest = price[0];
        for (int i = 0; i < price.length; i++) {
            if (price[i] < lowest) {
                lowest = price[i];
            }
        }
        System.out.println("*** Lowest price computed.");
        System.out.println("Here are the prices: ");
        for (int i = 0; i < price.length; i++) {
            System.out.print("Price " + (i + 1) + ": ");
            System.out.print(price[i]);
            if (lowest == price[i]) {
                System.out.println(" *** lowest price");
            } else {
                System.out.println();
            }
        }
    }
}
```

I have a feeling of *deja vu.*

And I was the first to say *that.*

Java Arrays

Part Two: Partially filled arrays.
Array parameters and return values.
Simple array algorithms.

Have we ever seen an array of `Strings`?

Apparently `main` receives one as a parameter.

Really? Where do those `Strings` come from?

From the command line.

Can you give me an example?

That's what I like best:

```
frilled.cs.indiana.edu%java One one two three
Hello! You have 3 arguments on the command line.
Arg 0: one
Arg 1: two
Arg 2: three
Thank you!
frilled.cs.indiana.edu%
```

OK. Now how do you write this program?

Here's how:

```
class One {
    public static void main(String[] args) {
        System.out.println("Hello! You have " + args.length +
                           " arguments on the command line.");
        for (int i = 0; i < args.length; i++) {
            System.out.println("Arg " + i + ": " + args[i]);
        }
        System.out.println("Thank you!");
    }
}
```

Looks good.

It usually does.

Now let's go back to our price check program.

We have improved on it by asking the user to set the size first.

Yes, but I don't think it's reasonable to ask the user to count the items for us before entering them.

After all, this is exactly the kind of work that the user expects the computer to do.

Unfortunately we now run into a problem.

Yes, we need to set the size of the array before we know how many elements we need.

But notice how passing the command line arguments to `main` makes that transparent to you, as a user.

Yes, we need to find a solution for our price check program too.

In Java once an array size is set, it cannot be changed.

Other programming languages have smarter arrays that can grow on demand,

... and Java also has a `Vector` class that can overcome this problem.

Unfortunately the `Vector` class is not as easy to use as an array.

We will discuss `Vector`s before too long.

To solve this problem, you can sometimes make an array that is guaranteed to be larger than the largest possible number of entries,

... and then *partially fill it.*

For example you can decide that the user will never need more than 1000 data points.

Then allocate an array of size 1000.

Then keep a companion variable that tells how many elements in the array are actually used.

It is an excellent idea *always* to name this companion variable by adding the suffix `Size` to the name of the array.

Here's the program so far.

```
class Two {
    public static void main(String[] args) {
        final int DATA_LENGTH = 1000;
        double[] price = new double[DATA_LENGTH];
        int priceSize = 0; /* first available index,
               also representing number of elements
               being stored in the array already. */
    }
}
```

Now `price.length` is the *capacity* of the array `price`

... and `priceSize` is the *current size* of the array.

Notice how starting with indexing at 0 gives us an alternative semantics for the index.

The alternative semantics is that there are exactly `i` elements in the array stored before the array element `price[i]`.

For any `i`, index of `price`.

That is, for any `i >= 0` and `i < price.length`

Keep adding elements in the array, incrementing the size variable each time.

This way, `priceSize` always contains the correct element count as well as the next available index in the array.

Two meanings in one variable.	When inspecting the array elements, though
...you must be careful to stop at `priceSize`,	...not at `price.length`
Also be careful not to overfill the array.	Insert elements only if there is still room for them!

Here's what I have so far:

```
class Two {
    public static void main(String[] args) {
        final int DATA_LENGTH = 1000;
        double[] price = new double[DATA_LENGTH];
        int priceSize = 0;
        ConsoleReader console = new ConsoleReader(System.in);
        System.out.println("Hello, please start entering prices.");
        while (true) {
            if (priceSize < price.length) {
                double data = console.readDouble();
                price[priceSize] = data;
                priceSize += 1;
                System.out.println("New element entered: " + data);
                for (int i = 0; i < priceSize; i++) {
                    System.out.println("** " + i + ": " + price[i]);
                }
            } else {
                System.out.println("Sorry, ran out of memory!");
                break;
            }
        }
    }
}
```

And how does it work? Here's how:

```
frilled.cs.indiana.edu%javac Two.java
frilled.cs.indiana.edu%java Two
Hello, please start entering prices.
3.45
New element entered: 3.45
** 0: 3.45
7.12
New element entered: 7.12
** 0: 3.45
** 1: 7.12
0.34
New element entered: 0.34
** 0: 3.45
** 1: 7.12
** 2: 0.34
5.00
```

```
New element entered: 5.0
** 0: 3.45
** 1: 7.12
** 2: 0.34
** 3: 5.0
^Cfrilled.cs.indiana.edu%
```

What happens if the array fills up?	Then, there are two approaches you can take.
The simple way out is to refuse additional entries.	That's what we have done above.
I have seen that.	But, of course, refusing to accept all input is often unreasonable.
Users routinely use software on larger data sets than the original developers ever dreamt of.	In Java, there is another approach to cope with data sets whose size cannot be estimated in advance.
When you run out of (allocated) space in an array	... you can create a new, larger array.
Can you also create a new *smaller* array?	Yes, but that's the second case.
If you want to *trim* it.	Let's get back to the array *overflow* case.
When you run out of allocated space in an array	... you can create a new, larger array;
copy all elements into the new array;	and then attach the new array to the old array variable.
An array that grows on demand is often called a *dynamic array* .	If you find that growing an array on demand is too tedious you can use vectors.(We'll get to that in next week's lectures).
We now have all the pieces together to implement the program.	Here it is:

```
class Two {
    public static void main(String[] args) {
        final int DATA_LENGTH = 1000;
        double[] price = new double[DATA_LENGTH];
        int priceSize = 0;
        ConsoleReader console = new ConsoleReader(System.in);
        System.out.println("Hello, please start entering prices.");
        while (true) {
            if (priceSize < price.length) {
                double data = console.readDouble();
                price[priceSize] = data;
                priceSize += 1;
                System.out.println("New element entered: " + data);
                for (int i = 0; i < priceSize; i++) {
```

```
                    System.out.println("** " + i + ": " + price[i] +
                                       " (" + price.length + ")");
                }
            } else {
                double[] newData = new double[2 * price.length];
                for (int i = 0; i < price.length; i++) {
                    newData[i] = price[i];
                }
                price = newData;
            }
        }
    }
}
```

I think you should study this program carefully.	Yes, it's a bit tricky.
The loop executes once every time	...except when the storage limit is reached it is executed one more time, quickly, to reallocate the array, then waits for user input.
How can we experience the reallocation of the array	...without having to set `DATA_LENGTH` to some small value?
Read it from the command line.	Very good.

```
class Two {
    public static void main(String[] args) {
        final int DATA_LENGTH = Integer.parseInt(args[0]);
        double[] price = new double[DATA_LENGTH];
        int priceSize = 0;
        ConsoleReader console = new ConsoleReader(System.in);
        System.out.println("Hello, please start entering prices.");
        while (true) {
            if (priceSize < price.length) {
                double data = console.readDouble();
                price[priceSize] = data;
                priceSize += 1;
                System.out.println("New element entered: " + data);
                for (int i = 0; i < priceSize; i++) {
                    System.out.println("** " + i + ": " + price[i] +
                                       " (" + price.length + ")");
                }
            } else {
                double[] newData = new double[2 * price.length];
                for (int i = 0; i < price.length; i++) {
                    newData[i] = price[i];
                }
                price = newData;
            }
        }
    }
```

```
}
```

Of course, this program is for testing, not for distribution.	Indeed. Let's now talk about trimming.
That's the easier case.	Yes. We just create a new smaller array with size `priceSize`
...then copy all the elements into the new array.	Then attach the new array to the old array variable. This way we keep the data and its size in just one place.
The *array* itself.	But we assume the array won't change after that.
Methods often have array parameters.	Such as `main`, for example.
This method computes the average of an array of floating point numbers.	To visit each element of the array `data`, the method needs to determine the length of `data`.

```
public static double average(double[] data) {
  if (data.length == 0) return 0;
  double sum = 0;
  for (int i = 0; i < data.length; i++)
    sum += data[i];
  return sum / data.length;
}
```

It inspects all elements, with index starting at 0	...and going up to, but not including, `data.length`.
Note that this method is *read-only*. It strives to be that way.	If changes were made to the array the caller would see that.
How come?	You pass arrays as if you're passing `Rectangles`.
Or any other object for that matter.	The invoked method simply receives a copy of the array's address.
Or reference.	When an array is passed to to a method, the array parameter
...`double[] data` in our case,	...contains a copy of the *reference* to the argument array.
The process is *identical* to that of copying array variables	...which we have discussed yesterday.
Or, to that of passing `Rectangles` as parameters to objects.	Indeed.
Because an array parameter is just another reference to the array,	.. a method can actually modify the entries of any array you give to it.

A method can also *return* an array.

This is useful if a method computes a result that consists of a collection of values

...of the same type. Here's an example: a method

...that returns a random data set, perhaps to test a chart-plotting program.

```
public static int[] randomData(int length, int n) {
  Random generator = new Random();
  int[] data = new int[length];
  for (int i = 0; i < data.length; i++)
    data[i] = generator.nextInt(n);
  return data;
}
```

We will discuss several very common

...and very important

...array algoritms. More complex algorithms are based on what we do now.

We will also look at sorting before too long.

Meanwhile let's look how we find a value (also known as *searching*).

Here's an example: suppose we want to find the first price that is lower than 1000 dollars.

```
int i = 0;
boolean found = false;
while (i < prices.length && !found) {
  if (prices[i] <= 1000)
    found = true;
  else
    i += 1;
}
if (found) {
  System.out.println("Item " + i + " is the first.");
} else {
  System.out.println("Not found.");
}
```

Note that the loop may fail to find an answer, namely if all prices are above $1,000.

At the end of the loop though, either `found` is true, in which case `prices[i]` is the first price that satisfies our requirements,

or `i` is `prices.length`, which means that you searched the entire list without finding a match.

So you have to give up smoking.

Or buy a lighter. Note though that you should *not* increment `i` if you had a match –

...if you want to have the correct value of `i` after exiting the loop.

Next comes *counting*.

Suppose you want to find out

...how many prices are below $1,000.

TMTOWTDI, but here's one:

```
double[] prices;
double targetPrice = 1000;
// ...  initialize the array
```

```
int count = 0;
for (int i = 0; i < prices.length; i++) {
  if (prices[i] <= targetPrice)
    count += 1;
}
System.out.println(count + " prices under $1,000.");
```

Yes. Now you don't stop on the first match (if any)	... but keep going to the end of the list,
... counting how many entries do match.	How do we *remove* an element?
There's more than one way to do it.	I know, but what cases do you have in mind?
If the elements of the array are not in any particular order,	... simply overwrite the element to be removed with the *last* element of the array.
Unfortunately, an array cannot be shrunk to get rid of the last element.	In this case, you can use the technique of a partially filled array together with a companion variable.
I don't like this method.	Neither do I.
The situation is more complex if the order of the elements matters.	Then you *must* move all the elements
... beyond the element to be removed	... by one slot.
Then trim the array.	Let's implement that:

```
class Three {
    public static void main(String[] args) {
        int[] price = new int[Integer.parseInt(args[0])];
        for (int i = 0; i < price.length; i++)
            price[i] = i + 1;
        show(price);
        price = removeElementAt(price, 3);
        show(price);
        price = removeElementAt(price, 5);
        show(price);
    }
    public static int[] removeElementAt(int[] a, int index) {
        System.out.println("--> Attempting to remove element at index "
                           + index + " in the array.");
        if (a.length > 0) {
            int[] copy = new int[a.length - 1];
            for (int i = 0; i < index; i++)
                copy[i] = a[i];
            for (int i = index; i < a.length - 1; i++)
                copy[i] = a[i + 1];
```

```
                return copy;
            } else {
                System.out.println("Sorry, array is empty.");
                return a;
            }

        }
        public static void show(int[] a) }
            for (int i = 0; i < a.length; i++)
                System.out.println("** " + i + ": " + a[i]);
        }
}
```

How does this work?

There you go:

```
frilled.cs.indiana.edu%java Three 10
** 0: 1
** 1: 2
** 2: 3
** 3: 4
** 4: 5
** 5: 6
** 6: 7
** 7: 8
** 8: 9
** 9: 10
--> Attempting to remove element at index 3 in the array.
** 0: 1
** 1: 2
** 2: 3
** 3: 5
** 4: 6
** 5: 7
** 6: 8
** 7: 9
** 8: 10
--> Attempting to remove element at index 5 in the array.
** 0: 1
** 1: 2
** 2: 3
** 3: 5
** 4: 6
** 5: 8
** 6: 9
** 7: 10
frilled.cs.indiana.edu%
```

I think there's a lot we can learn from this program.

I think so too; plus, *inserting* an element in an array is done in the same way.

You're right.

Tomorrow we'll start on `Vectors`.

Among other things.	Can I give you a small challenge?
Yes.	What's a good name for this method?

```
public static void fun(int[] a) {
  for (int i = 0; i < a.length - 1; i++)
    for (int j = i + 1; j < a.length; j++)
      if (a[i] < a[j]) {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
      }
}
```

You mean *second best* name...	Yes, that's what I mean.
I'll have to think about it.	Great. See you tomorrow.

The Bald Soprano

Inheritance and the class extension mechanism.

You don't realize it, but you're constantly enjoying the benefits of science.	For example, when you turn on the the radio, you take it for granted that music will come out;
But do you ever stop to think that this miracle would not be possible without the work of scientists?	That's right: there are tiny scientists inside that radio, playing instruments.
A similar principle is used in automatic bank-teller machines, which is why they frequently say: "*Sorry, out of service.*"	They're too embarassed to say: "*Sorry, tiny scientist going to the bathroom.*"
Speaking of banks and ATMs, let's use the `BankAccount` class to study the class extension mechanism, or inheritance (in Java).	*Inheritance* is a mechanism for enhancing existing, working classes.
If you • need to implement a new class, and • a class representing a more general concept is already available,	...then the new class can inherit from the existing class. For example, suppose you need to define a class `SavingsAccount` to model an account that pays a fixed interest rate on deposits.
You already have a class `BankAccount`	...and a savings account is a special case of a `BankAccount`.
Ask *any* tiny scientist!	So, in this case, it makes sense to use the language construct of inheritance.

Here is the syntax for the class definition:

```
class SavingsAccount extends BankAccount {
  <new methods>
  <new instance variables>
}
```

And the *set union of features* kicks in.

Exactly.	In the `SavingsAccount` class definition you specify only *new* methods and instance variables.
All methods and instance variables of the `BankAccount` class are *automatically inherited* by the `SavingsAccount` class.	I see... Concatenation of blueprints (almost).
The more general class that forms the basis for inheritance is called the *superclass*.	That would be `BankAccount`.
The more specialized class that inherits from the superclass is called the *subclass*.	Here, this is `SavingsAccount`.
In Java, every class that does not specifically extend another class,	...extends the class `Object`.
Whoa!... that explains everything!	Yes. It's been a well kept secret until now.
The `Object` class has a small number of methods that make sense for all objects,	.. such as the `toString` method that you can use to obtain a string that describes the state of an object, any object.

I remember the other day we had this code.

```
class Vehicle {
  String owner;
  Vehicle (String owner) {
    this.owner = owner;
  }
  public String toString() {
    return "I belong to: " + this.owner;
  }
  public static void main(String[] args) {
    Vehicle a = new Vehicle("Michael Jordan");
    System.out.println(a.toString());
  }
}
```

And we asked two questions about it.

First off, if you run it now, no mistery.	Yes. All's *copacetic* now. But do this:

```
class Vehicle {
  String owner;
  Vehicle (String owner) {
    this.owner = owner;
  }
  public String toString() {
    return "I belong to: " + this.owner;
  }
  public static void main(String[] args) {
```

```
    Vehicle a = new Vehicle("Michael Jordan");
    System.out.println(a          );
  }
}
```

Yes, that's the minor mistery.	The `toString` is invoked by default.
Very good.	Now do this:

```
class Vehicle {
  String owner;
  Vehicle (String owner) {
    this.owner = owner;
  }
  public static void main(String[] args) {
    Vehicle a = new Vehicle("Michael Jordan");
    System.out.println(a.toString());
  }
}
```

Yes, that's the major mistery.	A `toString` is already there from `Object`.
Doesn't work *that* well, but it's there.	And `Object` is responsible for providing it.
That's called *inheritance*.	It's a side-effect of the class extension mechanism (for efficiency of expression).
When you extend a set of features provide the name of the set your starting from (`extends`) followed by the list of features you're adding.	One important reason for inheritance is *code reuse*. By inheriting from an existing class, you do not have to replicate the effort that went into designing and perfecting that class. For example ...
... when implementing the `SavingsAccount` class, you can rely on the `withdraw`, `deposit` and `getBalance` methods of the `BankAccount` class without touching them.	Let us see how our savings account objects are different from `BankAccount` objects. But before that let me just say this again: For example, ...
... when implementing the `SavingsAccount` class, you can rely on the	• `withdraw`, • `deposit` and • `getBalance` methods of the `BankAccount` class
... without touching them.	OK. Let us now see how our savings account objects are different from `BankAccount` objects.
We will set an • interest rate in the constructor,	... and then we need a • method to apply that interest periodically.

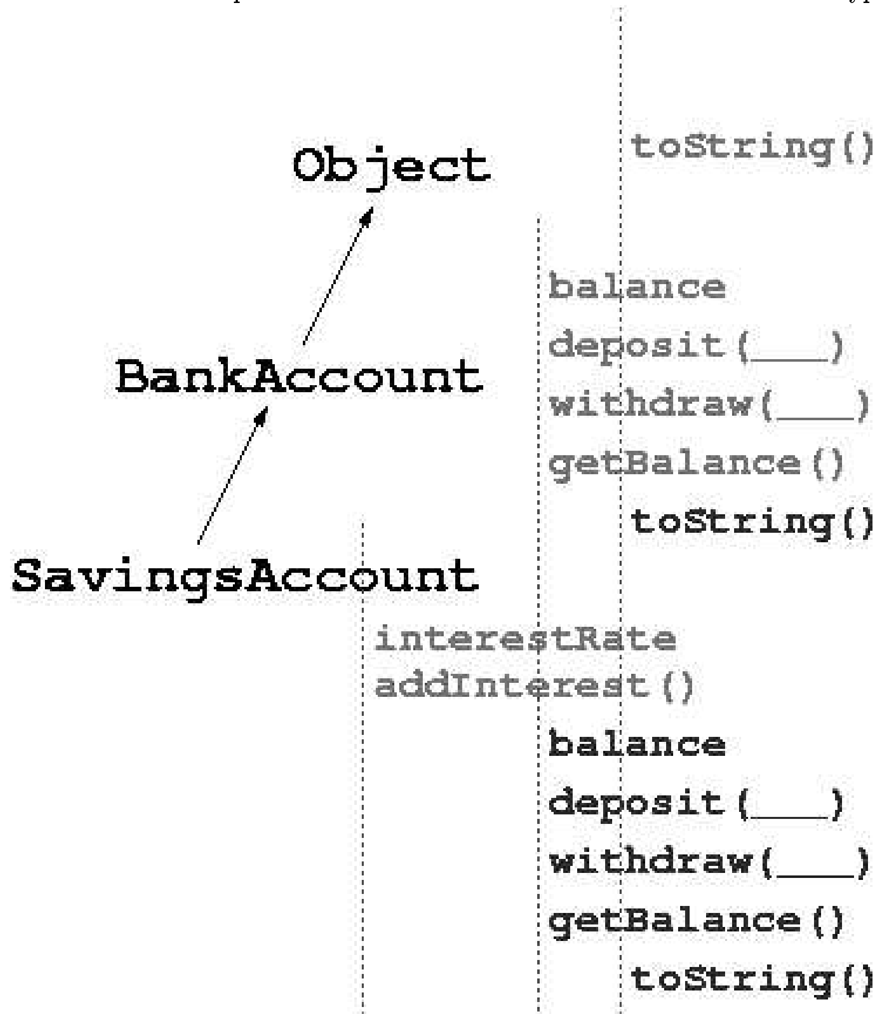
That is, in addition to the three methods that can be applied to *every* `BankAccount`...	...we now have an additional method, `addInterest` which will only work for the new type of `SavingsAccounts`.
These new methods and instance variables must be defined in the subclass.	Here's the definition:

```
public class SavingsAccount extends BankAccount {
  double interestRate;
  public SavingsAccount (double rate) {
    this.interestRate = rate;
  }
  public void addInterest() {
    double interest;
    interest = this.getBalance() * this.interestRate / 100;
    this.deposit(interest);
  }
}
```

Given this definition, what is the structure of a `SavingsAccount` object (as far as fields go)?	It inherits the `balance` instance variable from the `BankAccount` superclass, and it gains one additional instance variable.
Which is... `interestRate`.	Exactly.
It's nice to be able to develop things in *stages*.	Yes, we will have a longer example later.
Next we need to implement the new `public void addInterest()` instance method.	We have in fact already implemented it, but we pretend not to have done, just to discuss it.
This method computes the interest due on the current balance,	...and then deposits that interest to the account.
Note how the `addInterest()` method calls the `getBalance()` and `deposit()` methods of the superclass (`BankAccount`).	Hot ziggity, you're *right*!

```
public class SavingsAccount extends BankAccount {
  double interestRate;
  public SavingsAccount (double rate) {
    this.interestRate = rate;
  }
  public void addInterest() {
    double interest;
    interest = this.getBalance() * this.interestRate / 100;
    this.deposit(interest);
  }
}
```

	Thanks for emphasizing, I would have missed it.
You're *very* welcome.	
Let's now draw a picture to illustrate...	...what each type of object has and why:



Boy, that looks good!	I sure think so.
The class `SavingsAccount` extends the class `BankAccount` (which extends `Object`...	...by default). A `SavingsAccount` object is a special case of `BankAccount`, just as a `BankAccount` is a special kind of `Object`.
A special case has *more* features.	When I define a variable `collegeFund` of type `SavingsAccount` how do you anticipate using it?
Here are all possible uses:	Very good.

```
collegeFund.deposit(___);
collegeFund.withdraw(___);
collegeFund.getBalance();
collegeFund.addInterest();
```

When I define a variable `anAccount` of type `BankAccount` how do you anticipate using it?	Here are all possible uses:
	`anAccount.deposit(___);`
	`anAccount.withdraw(___);`
	`anAccount.getBalance();`
	One *less*.

Very good.	Get ready for a subtle question now.
Hit me.	OK, here (it) goes:
	Can you store a reference to a `SavingsAccount` object into an object variable of type `BankAccount`?
Like this? `BankAccount b = new SavingsAccount(10);`	Yes. Can you do that?
You would never utilize `b` fully,	... but perhaps you don't *need* that.
So the answer is: *yes*.	You *can* store the reference to a `SavingsAccount` object into an object variable of type `BankAccount`
And the reason is that you're just saying:	"*I will not need the extra features that the class* `SavingsAccount` *is defining,...*
I will just attempt to work with the features defined in `BankAccount` *– that's all I need in this particular case*"	*Almost* like casting a `double` to an `int`.
Almost but not *exactly*.	Yes, because the new feature *is* (still) there.
OK. Can you do the opposite? `SavingsAccount a = new BankAccount(___);`	You mean *this*?
Yes.	The answer is: *no*.

Why?	Well, what's the intended use of `a`?
I'd say: if I try to compute the added interest the object won't have an adequate instance variable, nor the capability to do that. That is,	`a.addInterest()` does not make sense.
So we can't let that happen.	And the compiler will complain.
Why doesn't it complain in the previous situation?	Because in that situation we are only giving up on some *amenities*,
...which is fine with the compiler for as long as it's fine with us,	...whereas here we might ask for the *impossible*,
.. which the compiler *can't* accept,	...even if it's fine with us.
Therefore, `a.addInterest()` is what it wants to guard us against.	So, going back to our original example,

```
SavingsAccount              [collegeFund] = new SavingsAccount(10);
[BankAccount] anAccount = [collegeFund];
[Object]      anObject  = [collegeFund];
```

...these are all OK.

Now the three object references stored in `collegeFund`, `anAccount`, and `anObject`,	...all refer to the same object, of type `SavingsAccount`, that much is clear.
However, the object variable `anAccount` knows less than the full story about the object to which it refers. (The object only knows the truth).	Because `anAccount` is a variable of type [`BankAccount`], you can use it to refer to the [`deposit`] and [`withdraw`] methods used to change the `balance` of the actual [`SavingsAccount`] object.
You can't use the `addInterest` method, though.	It is not a method of the `BankAccount` superclass.
You can't see it when you decide to ignore it.	Exactly. And, of course, the variable `anObject` knows even less.
You can't even apply the `deposit` method to it.	`deposit` is not a method of the `Object` class.
Why would anyone *want* to know less about an object and store a reference to it in a variable of the superclass's type?	For generality and uniformity.

Have any example?	I have two of them.
Let's see the first one.	Consider the `transfer` method which transfers money from one account into another.

```
void transfer (BankAccount other, double amount) {
  this.withdraw(amount);
  other.deposit(amount);
}
```

You can use this method to transfer money	... from one `BankAccount` to another,
... and you can *also* use the method to transfer money into a `SavingsAccount`.	The `transfer` method expects a reference to a `BankAccount`, which it will use to `deposit`.
Any `SavingsAccount` object can do that too, so it can be passed as the first explicit argument to `transfer`.	The `transfer` method doesn't actually know (or care, for that matter) that, in this case, `other` refers to an actual `SavingsAccount`.
It knows only that `other` is a `BankAccount`,	... that is, that it can • `deposit` • `withdraw`, and • `getBalance`
... and it doesn't need to know anything else.	Precisely.
I liked your first example.	Thank you. I liked it too.
What's the second example?	It involves arrays, but we need to discuss inheritance hierarchies first.
Very good, let's do that.	Occasionally, it happens that you convert an object to a superclass reference then, later, you need to convert it back.
Suppose you captured a reference to a savings account in a variable of type `Object`: `Object myObj = new SavingsAccount(10);`	A variable reference is like a pair of binoculars.
Or a pair of *blinkers*	... of the type that's used on skittish racehorses.
If you put it on,	... you can only see what it lets you see.
Much later, if you want to • add interest or • deposit to the account,	... you can do that, *with care*.

The *object* still has *all* the features,	...you just need to put the right pair of binoculars on to see them.
That's called *casting*.	As long as you are absolutely sure that `myObj` really refers to a `SavingsAccount` object,
...you can use the *cast* notation to convert it back, like this: `SavingsAccount x = (SavingsAccount) myObj;`	What if you're sure but *wrong*?
If you are wrong, and the object doesn't actually refer to a savings account, your program will throw an exception, and terminate.	You will see examples of casting soon now.
In real world, we often categorize concepts into *hierarchies*. Hierarchies are frequently represented as trees,	...with the most general concepts at the root of the hierarchy, and the more specialized ones towards the branches.
I think I get that.	Let's see an example in Java.
Suppose that we have more than just one extension to `BankAccount`.	Consider a `CheckingAccount` class that describes accounts with no interest,
...gives you a small number of free transactions per month,	...and charges a transaction fee for each additional transaction.
I can visualize that.	Good. All accounts have something in common.
They are all bank accounts with a balance and the ability to deposit money,	...and (within limits) to withdraw money.
This leads us to the following inheritance hierarchy:	A simple, very basic class hierarchy.

BankAccount

SavingsAccount **CheckingAccount**

	Now suppose that you have 100 bank account objects, and half of them are checking accounts and the other half are...
...savings accounts? That's plausible.	Can you keep them *all* in an array?
Only if the array is declared as having an interest in (being concerned with describing) only their most general, common features.	

Like this:

```
BankAccount[] a = new BankAccount[100];
```

This strategy, in its most general form, is used by the `Vector` class.

Which makes use of arrays of `Object`s.

To store something in a `Vector` it must be of type `Object`. I mean: *that's* it!

So you can't store an `int`?

Not directly.

But you *can* store a `Rectangle`

If you do, it will get stored as an `Object`.

And when you retrieve it,

```
Vector v = new Vector();
v.addElement(new Rectangle(_,_,_,_));
(v.elementAt(0)).translate(_,_);
```

...it comes back as an `Object` and not as a `Rectangle`. So you will need to cast the reference to a `Rectangle` or it won't work.

```
Vector v = new Vector();
v.addElement(new Rectangle(_,_,_,_));
((Rectangle)(v.elementAt(0))).translate(_,_);
```

Whatever you want to do with it as a `Rectangle`, you need to *cast* it to a `Rectangle` from the `Object` that it comes back as.

Not casting it, will give you an error first time you try to use it as a `Rectangle`.

In most situations of this kind, though, it is better to play it safe and test whether a cast will succeed, before carrying out the cast.

For that purpose one can use the `instanceof` operator.

That's right, it tests whether an object belongs to a particular class.

For example, when retrieving one of our accounts from the array we could take the appropriate type of action depending on the type of account:

```
if (anObject instanceOf SavingsAccount) {
  // ... do savings account type of work
} else if (anObject instanceOf CheckingAccount) {
  // ... do checking account type of work
} else {
  // ... in which case the tiny scientist reports an error
}
```

Is that all there is to it?

Almost.

Can you give me a complete summary?

Yes. Complete for all practical purposes.

Let's start.

OK.

We first defined class `Point`. A `Point` has a position (`x, y`).

```
class Point {
  int x;
  int y;
}
```

These are the features of any `Point` object:

- an `x` coordinate, and
- a `y` coordinate

together defining the *position* of any `Point`. A `Pixel` is a `Point` with `Color`. In Java this is easy to write:

```
class Pixel extends Point {
  Color23 c;
}
```

The features of a `Pixel` are three:

- an `x` coordinate (which is an `int`)
- a `y` coordinate (which is an `int`)
- a `Color`, call it *color*

This set of features is the union between

- the features of a `Point` and
- the one new feature that class `Pixel` is defining

in other words:

$$\text{Point} = \{x, y\}$$

$$\text{Point} = \text{Point} \cup \{\text{color}\}$$

That is, the resulting blueprint (for `Pixel`) is a putting together of the two descriptions. That's fine, but set union means that names should be kept distinct. (We'll come back to this in a second.) How do we use `Point` and `Pixel`? Nothing unusual. We use `new` and expect the blueprints to define the resulting structures.

```
// somewhere in a method...
Point a = new Point();
Pixel b = new Pixel();
a.x = 2;
a.y = -10;
b.x = 3;
b.y = 24;
b.color = Color.blue;
```

[23] http://java.sun.com/products/jdk/1.2/docs/api/java/awt/Color.htmlColor

Notice that the `Color` class is defined in the `java.awt` package. Next we asked: have you ever seen a `Horse`? The answer was: yes. Can you describe a `Horse`? The answer was: that's actually quite complicated. OK, so fortunately we know what we're talking about:

```
class Horse {
  // lots of features ...
}
```

Next we asked: have you ever seen a `Unicorn`? The answer was: no. Can you describe a `Unicorn` though? Everybody said: yes, that's easy.

Here's a picture to get the idea:
So a definition is almost *immediate*.

```
class Unicorn extends Horse {
  Horn h;
}
```

And we had the following situation:

- everybody had seen horses, but nobody felt it was easy to describe them
- nobody had seen unicorns, but everybody thought it was easy to describe them

That's because we factored out the `Horse`. Now we said: let's write a play with horses and unicorns. In our play we could have:

```
Horse   h = new Horse();
Unicorn u = new Unicorn();
```

Both h and u are special kinds of binoculars. If you put them on you should see the features that their type is defining. So `h.mane` and `u.mane` make sense. So does `u.horn` but `h.horn` doesn't. For this reason it's not adequate to say:

```
Unicorn g = new Horse();
```

It is OK, however, to ignore some features, for the sake of being more general. From our description it follows that all `Unicorns` are `Horses`. That's called *polymorphism*. So writing something like this is acceptable:

```
Horse z = new Unicorn();
```

You can never access the `Horn` with `z` but sometimes you don't even need that. We could come up with the following similar example:

```
class Shape {
  // two coordinates
}
class Circle extends Shape {
  // add a radius
}
```

```
class Rectangle extends Shape {
  // add a width and a height
}
class Triangle extends Shape {
  // add two other points relative to location
}
```

Why is this useful? If you want to create an array that can store

- `Circles`,
- `Triangles`, and
- `Rectangles`,

the only thing you can do that is by relying on their generality as `Shapes`.

```
Shape p[] = new Shape[100];
```

I've got room for 100 such shapes (circles, triangles, or rectangles). There's no other way around it, as far as arrays are concerned. Now we want to explore the *name collision* problem. Consider this example:

```
class Horse {
  void neigh() { System.out.println("Horse: Howdy!"); }
}
class Unicorn extends Horse {
  void neigh() { System.out.println("Unicorn: Bonjour! "); }
}
```

`Unicorn` is listing a feature: `neigh`. If `Horse` had not had it already listed things would've been easy. But `Horses` already know how to `neigh`. They say: "Howdy!". So `Unicorns` *redefine* the feature by saying "Hello!" in French. (`Unicorns` are from Paris, `TX`)? That's called *overriding.* The mechanism is that no matter how you look at a `Unicorn`,

- as the `Unicorn` that it *is*, or
- as the `Horse` that it *is*

you are guaranteed to obtain the French greeting out of it. Here's the proof:

```
frilled.cs.indiana.edu%cat Ionesco.java
class Horse {
    void neigh() { System.out.println("I am a Horse: Howdy!"); }
}
class Unicorn extends Horse {
    void neigh() { System.out.println("I am a Unicorn: Bonjour! "); }
}
class Ionesco {
    public static void main(String[] args) {
        Unicorn a = new Unicorn();
        Horse   b = new Unicorn();
        a.neigh();
        b.neigh();
        Horse c = new Horse();
```

```
        c.neigh();
    }
}
frilled.cs.indiana.edu%javac Ionesco.java
frilled.cs.indiana.edu%java Ionesco
I am a Unicorn: Bonjour!
I am a Unicorn: Bonjour!
I am a Horse: Howdy!
frilled.cs.indiana.edu%
```

That's all we need to know before we go into applets.

```
class Vector {
  Object[] localStorage = new Object[0];
  int size() {
    return localStorage.length;
  }
  void addElement(Object obj) {
    int currentLength = this.size();
    Object[] aux = new Object[currentLength + 1];
    aux[currentLength] = obj;
    for (int i = 0; i < currentLength; i++) {
      aux[i] = localStorage[i]; // transfer elements
    }
    localStorage = aux;
  }
  public String toString() {
    String returnString = "Vector of size " + size() + ": (";
    for (int i = 0; i < localStorage.length; i++)
      returnString += " " + localStorage}[i];  // delay printing...
    return returnString + ")";
  }
  public static void main(String[] args) {
    Vector v = new Vector();
    System.out.println(v); // convenient printing
    v.addElement(new Integer(2));
    System.out.println(v);
    v.addElement(new Integer(4));
    System.out.println(v);
    v.addElement(new Integer(6));
    System.out.println(v);
  }
}
```

Utilities

Vectors, Hashtables, Leftovers.

First here's a Selection Sort that we have developed in class.

```
class One {
    static void sort(int[] a) {
        for (int start = 0; start < a.length - 1; start++) {
            for (int j = start; j < a.length; j++) {
                if (a[start] > a[j]) { // sorting in ascending order
                    int temp = a[start];
                    a[start] = a[j];
                    a[j] = temp;
                }
            }
        }
    }
    public static void main(String[] args) {
        int[] numbers = new int[args.length];
        for (int i = 0; i < numbers.length; i++) {
            numbers[i] = Integer.parseInt(args[i]);
        }
        System.out.println("Here's the initial array: ");
        One.show(numbers);
        System.out.println("Let me sort it in ascending order... ");
        One.sort(numbers);
        System.out.println("...  Done\nHere it is sorted: ");
        One.show(numbers);
    }
    static void show(int[] a) {
        for (int i = 0; i < a.length; i++) {
            System.out.print(a[i] + " ");
        }
        System.out.println();
    }
}
```

Here's how it runs:

```
frilled.cs.indiana.edu%javac One.java
frilled.cs.indiana.edu%java One 3 4 1 2 6 7
Here's the initial array:
3 4 1 2 6 7
Let me sort it in ascending order...
... Done
Here it is sorted:
1 2 3 4 6 7
frilled.cs.indiana.edu%
```

Here's a second method of sorting: Bubble Sort.

```
class Three {
    static void sort(int[] a) {
        boolean done;
        do {
            done = true;
            for (int i = 0; i < a.length - 1; i++) {
                if (a[i] > a[i + 1]) { // sort in ascending order
                    int temp = a[i];
                    a[i] = a[i + 1];
                    a[i + 1] = temp;
                    done = false;
                }
            }
        } while (! done);
    }
    static void main(String[] args) {
        int[] numbers = new int[args.length];
        for (int i = 0; i < numbers.length; i++) {
            numbers[i] = Integer.parseInt(args[i]);
        }
        System.out.println("Here's the initial array: ");
        Three.show(numbers);
        System.out.println("Let me sort it in ascending order... ");
        Three.sort(numbers);
        System.out.println("...  Done\nHere it is sorted: ");
        Three.show(numbers);
    }
    static void show(int[] a) {
        for (int i = 0; i < a.length; i++) {
            System.out.print(a[i] + " ");
        }
        System.out.println();
    }
}
```

It runs just like the other one, but the mechanism is different.

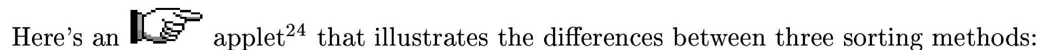
The outer `do-while` loop is normally added at the end.

For this reason I find the following method much more intuitive.

```
class Four {
    static void sort(int[] a) {
        boolean done = true;
        for (int i = 0; i < a.length - 1; i++) {
            if (a[i] > a[i + 1]) { // sort in ascending order
                int temp = a[i];
                a[i] = a[i + 1];
                a[i + 1] = temp;
                done = false;
            }
        }
        if (! done)
            sort(a);
    }

    static void main(String[] args) {
        int[] numbers = new int[args.length];
        for (int i = 0; i < numbers.length; i++) {
            numbers[i] = Integer.parseInt(args[i]);
        }
        System.out.println("Here's the initial array: ");
        One.show(numbers);
        System.out.println("Let me sort it in ascending order... ");
        One.sort(numbers);
        System.out.println("...  Done\nHere it is sorted: ");
        One.show(numbers);
    }
    static void show(int[] a) {
        for (int i = 0; i < a.length; i++) {
            System.out.print(a[i] + " ");
        }
        System.out.println();
    }
}
```

Yes, recursion, in some cases, amounts to just a loop.

One could improve on the Bubble Sort technique above.



Here's an applet[24] that illustrates the differences between three sorting methods:

- quick sort (not selection sort)
- regular bubble sort
- improved bubble sorting

The applet, I believe, is very exciting.

Let's move on.

[24] http://www.cs.indiana.edu/classes/a348-dger/lectures/tsort/example1.html

Last time we talked about: (a) inheritance and (b) the class extension mechanism. These are more or less the same topic. We can illustrate them both by this example:

```
class Horse {
    int numberOfLegs;
    void fun() {
        System.out.println("I am a Horse.");
    }
}
class Unicorn extends Horse {
    void fun() {
        System.out.println("I am a Unicorn. Like a horse, but with a horn.");
    }
}
class Experiment {
    public static void main(String[] args) {
        Horse      a = new Horse();
        System.out.println("A horse has " + a.numberOfLegs + " legs.");
        Horse      b = new Unicorn();
        Unicorn    c = new Unicorn();
        System.out.println("A unicorn has " + c.numberOfLegs + " legs.");
        // Unicorn d = new Horse(); // Not allowed.
        System.out.print("First test:  ");
        a.fun(); // a horse, of course
        System.out.print("Second test: ");
        c.fun(); // a unicorn, of course
        System.out.print("Third test:  ");
        b.fun(); // what will this be?
    }
}
```

Here also the example promised on developing things in *stages*:

```
frilled.cs.indiana.edu%cat Stages.java
class One {
    int add(int n, int m) {
        if (m == 0) return n;
        else return add(n+1, m-1);
    }
}
class Two extends One {
    int mul(int n, int m) {
        if (m == 1) return n;
        else return add(n, mul(n, m-1));
    }
}
class Three extends Two {
    int pow(int n, int m) {
        if (m == 0) return 1;
        else return mul(n, pow(n, m-1));
    }
```

```
}
class Calculator {
    public static void main(String[] args) {
        Three calc = new Three();
        int n = 3, m = 5;
        System.out.println(n + " + " + m + " = " + calc.add(n, m));
        System.out.println(n + " * " + m + " = " + calc.mul(n, m));
        System.out.println(n + " ^ " + m + " = " + calc.pow(n, m));
    }
}
frilled.cs.indiana.edu%javac Stages.java
frilled.cs.indiana.edu%java Calculator
3 + 5 = 8
3 * 5 = 15
3 ^ 5 = 243
frilled.cs.indiana.edu%
```

Here's a different, somewhat similar, example on interfaces.

```
frilled.cs.indiana.edu%cat Example.java
interface Multiplier {
    int mul(int n, int m);
}
class Alpha implements Multiplier {
    public int mul(int n, int m) {
        return n * m;
    }
}
class Beta implements Multiplier {
    public int mul(int n, int m) {
        int result = 0;
        for (int i = 0; i < m; i++)
            result += n;
        return result;
    }
}
class Gamma implements Multiplier {
    public int mul(int n, int m) {
        if (m == 1) return n;
        else return n + mul(n, m-1);
    }
}
class Example {
    public static void main(String[] args) {
        Alpha a = new Alpha();
        Beta b = new Beta();
        Gamma g = new Gamma();
        int n = 5, m = 3;
        System.out.println(n + " * " + m + " = " + a.mul(n,m) + " (by Alpha)");
        System.out.println(n + " * " + m + " = " + b.mul(n,m) + " (by Beta )");
        System.out.println(n + " * " + m + " = " + g.mul(n,m) + " (by Gamma)");
```

```
    }
}
frilled.cs.indiana.edu%javac Example.java
frilled.cs.indiana.edu%java Example
5 * 3 = 15 (by Alpha)
5 * 3 = 15 (by Beta )
5 * 3 = 15 (by Gamma)
frilled.cs.indiana.edu%
```

It often happens that you want to store collections that have a two-dimensional layout.

Such an arrangement, consists of rows and columns of values, and is called (not surprisingly) a *two-dimensional array.*

. . . or *matrix* (sometimes).

When constructing a two-dimensional array, you specify how many rows and columns you need.

If you want `10` rows and `8` columns:

```
int[][] matrix = new int[10][8];
```

A `10 x 8` matrix of `int`s.

To access a particular element in the array

```
matrix[3][4] = 5;
```

. . . specify two subscripts, in separate brackets.

In Java, two-dimensional arrays are stored as arrays of arrays.

And three dimensional arrays are stored as arrays of two-dimensional arrays.

That means arrays of arrays of arrays.

Yes, arrays of (arrays of arrays).

And `n`-dimensional arrays will be stored as arrays of (`n-1`)-dimensional arrays.

What a recursive definition!

We won't use more than two-dimensions.

Let's take a closer look at them though: because a two-dimensional array is really an array of the row arrays,

. . . the number of rows is:

```
int nrows = matrix.length;
```

And the number or columns?

Because a two-dimensional array is really an array of the row arrays,

. . . the number of elements in each row *can* vary.

But it doesn't have to.

If it doesn't, the number of columns is the same as the length of the first row:

```
int ncols = matrix[0].length;
```

Can I see an example of a two-dimensional array that has rows of various lengths?

```
int[][] b = { {1},
              {2, 3},
              {4, 5, 6},
              {7, 8, 9, 10}
            };
```

Yes, you can quickly create one with an array initializer:

Can you do the same thing without it?	Sure, I thought you'd ask:

```
int[][] b = new int[4][];
int count = 1;
for (int i = 0; i < 4; i++) {
  b[i] = new int[i + 1];
  for (int j = 0; j <= i; j++) {
    b[i][j] = count;
    count += 1;
  }
}
```

I see you have to work harder.	Yes, first you allocate space to hold four rows.
You indicate that you will manually set each row by leaving the second array index empty.	Then you need to allocate (and fill) each row separately.
You can access each array element as `b[i][j]`,	. . . but you must be careful that `j` is less than `b[i].length` as illustrated below:

```
class Testing {
    public static void main(String[] args) {
        int[][] b = new int[4][];
        int count = 1;
        for (int i = 0; i < 4; i++) {
            b[i] = new int[i + 1];
            for (int j = 0; j <= i; j++) {
                b[i][j] = count;
                count += 1;
            }
        }
        for (int i = 0; i < b.length; i++) {
            for (int j = 0; j < b[i].length; j++) {
                System.out.print(b[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Naturally, such ”ragged” arrays are not very common.	Except when you need them.
Then they become very natural.	And they're no longer ”ragged”.
Exactly, because they fit that problem	. . . perfectly.

Events

Applets, Events, and Event Handling.

Today in class I would like to discuss this code:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class Game extends Applet {
    int i = 0;
    public void paint(Graphics g) {
        this.i = this.i + 1;
        System.out.println("Paint called: " + i);
    }
    public void init() {
        Umpire ump = new Umpire();
        this.addMouseMotionListener(ump);
    }
}
class Umpire implements MouseMotionListener { // wearing the uniform...
    public void mouseDragged(MouseEvent e) {
        System.out.println("Ha! You're dragging the mouse.");
    }
    public void mouseMoved(MouseEvent e) {
        System.out.print  ( "Mouse seen being moved at: (");
        System.out.println( e.getX() + ", " + e.getY() + ")");
    }
}
```

We'd like to understand this code very well.

When that is done we move on to this code:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class Example extends /*NoFlicker*/Applet {
```

```
    Circle[] circles;

    public void init() {

        circles = new Circle[100];

        for (int i = 0; i < circles.length; i++) {
            circles[i] =
                new Circle(

                            (int) (Math.random() * (280 - 20) + 20), // x
                            (int) (Math.random() * (280 - 20) + 20), // y
                            (int) (Math.random() * ( 20 - 10) + 10), // radius

                            new Color( (float)Math.random(),
                                       (float)Math.random(),
                                       (float)Math.random()
                                       )

                                );

        } // end of for

        Broker peterPan = new Broker(this);

        addMouseListener(peterPan);
        addMouseMotionListener(peterPan);

    } // end of init

    public void paint(Graphics g) {
        for (int i = 0; i < circles.length; i++) {
            // System.out.println(circles[i]);
            circles[i].draw(g);
        }
    }

}
class Broker implements MouseMotionListener,
                        MouseListener {

    Example customer;
    Circle[] a;
    Circle c;

    Broker(Example someone) {
        this.customer = someone;
        this.a = this.customer.circles;
        this.c = null;
    }
```

```
    public void mouseDragged(MouseEvent e) {
        int x = e.getX(), y = e.getY();
        if (c != null) {
            c.move(x, y);
            customer.repaint();
        }
    }
    public void mouseMoved(MouseEvent e) { }

    public void mouseClicked(MouseEvent e) { }
    public void mousePressed(MouseEvent e) {

        int x = e.getX(), y = e.getY();
        System.out.println("Mouse pressed at (" +
                           x + ", " + y +")");
        for (int i = a.length - 1; i >= 0; i--) {
            if (a[i].contains(x, y)) {
                c = a[i];
                break;
            }
        }

        System.out.println("Currently on... " + c);

    }
    public void mouseReleased(MouseEvent e) { c = null; }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
}
class Circle {
    int x; int y; // center, not top-left corner
    int radius;
    Color c;

    public void move(int x, int y) {
        this.x = x; this.y = y;
    }

    Circle(int x, int y, int r, Color c) {
        this.x = x;
        this.y = y;
        this.radius = r;
        this.c = c;
    }

    public boolean contains(int x, int y) {

        double dX = this.x - x,
            dY = this.y - y;
```

```
        if (Math.sqrt(dX * dX + dY * dY) <= this.radius)
            return true;
        else
            return false;

    }

    public String toString() {
        return "Circle at (" + this.x + ", " + this.y + ") " +
            " size " + this.radius + ", color " + this.c;
    }

    public void draw(Graphics b) {

        b.setColor(this.c);
        b.fillOval(this.x - this.radius,
                   this.y - this.radius,
                   2 * this.radius,
                   2 * this.radius);
        b.setColor(Color.black);
        b.drawOval(this.x - this.radius,
                   this.y - this.radius,
                   2 * this.radius,
                   2 * this.radius);

    }

}

<html>
  <head>
    <title>Circles</title>
  </head>
  <body bgcolor=white>
    <applet code="Example.class" width=300 height=300>
    </applet>
  </body>
</html>
```

Question: Why the flicker?

The answer is: because of `update()`. (We inherit this method, and it is called by `repaint()`). Every time it's called it clears the screen and then calls `paint()`. To obtain flicker-free screen updates (which is essentially what we are trying to do here, as in any animation) we will need to override `update()` such that it creates the new image somewhere in the background (that is, in memory) and then updates the screen in one fell swoop. It is easy to implement this as an application of inheritance. Here's the new definition of `update()`: turn to page 31 in this book for the definition of the class `NoFlickerApplet`. Now your program needs to extend this class instead of `Applet` directly.

Abstract Classes

Review and Tutorial: Inheritance and Abstract Classes.

When you define a class, Java guarantees that the class's constructor method is called whenever an instance of that class is created. It also guarantees that the constructor is called when an instance of any subclass is created. In order to guarantee this second point, Java must ensure that every constructor method calls its superclass constructor method. If the first statement in a constructor is not an explicit call to a constructor of the superclass with the `super` keyword, then Java implicitly inserts the call `super()` – that is, it calls the superclass constructor with no arguments. If the superclass does not have a constructor that takes no arguments, this causes a compilation error. There is one exception to the rule that Java invokes `super()` implicitly if you do not do so explicitly. If the first line of a constructor, `C1`, uses the `this()` syntax to invoke another constructor, `C2`, of the class, Java relies on `C2` to invoke the superclass constructor, and does not insert a call to `super()` into `C1`. Of course, if `C2` itself uses `this()` to invoke a third constructor, `C2` does not call `super()` either, but somewhere along the chain, a constructor either explicitly or implicitly invokes the superclass constructor, which is what is required. What this all means is that constructor calls are "chained" – any time an object is created, a sequence of constructor methods are invoked, from subclass to superclass on up to `Object` at the root of the class hierarchy. Because a superclass constructor is always invoked as the first statement of its subclass constructor, the body of the `Object` constructor always runs first, followed by the body of its subclass, and on down the class hierarchy to the class that is being instantiated.

There is one missing piece in the description of constructor chaining above. If a constructor does not invoke a superclass constructor, Java does so implicitly. But what if a class is declared without any constructor at all? In this case, Java implicitly adds a constructor to the class. This default constructor does nothing but invoke the superclass constructor. Note that if the superclass did not declare a no-argument constructor, then this automatically inserted default constructor would cause a compilation error. If a class does not define a no-argument constructor, then all of its subclasses must define constructors that explicitly invoke the superclass constructor with the necessary arguments. It can be confusing when Java implicitly calls a constructor or inserts a constructor definition into a class – something is happening that does not appear in your code! Therefore, it is good coding style, whenever you rely on an implicit superclass constructor call or on a default constructor, to insert a comment noting this fact. Your comments might look like those in the following example:

```
class A {
  int i;
  public A() {
    // Implicit call to super() here
    i = 3;
  }
}
```

```
class B extends A {
  // Default constructor: public B() { super(); }
}
```

If a class does not declare any constructor, it is given a `public` constructor by default. Classes that do not want to be publically instantiated, should declare a `protected` constructor to prevent the insertion of this `public` constructor. Classes that never want to be instantiated at all (in one particular, specific way,) should define that particular constructor `private`. And here's the last part of this overview.

1. An `abstract` method has no body, only a signature followed by a semicolon.

   For example:

   ```
   abstract double area();
   ```

2. Any class with an `abstract` method is automatically `abstract` itself, and must be declared as such. So we need the blue keyword below, it just <u>has to</u> be there:

   ```
   abstract class Shape {
     abstract double area();
   }
   ```

3. A class may be declared `abstract` even if it has no `abstract` methods.

   This prevents it from being instantiated.

   For example:

   ```
   oldschool.cs.indiana.edu%ls -l
   total 1
   -rw-------   1 dgerman       134 Jul 16 12:28 Example.java
   oldschool.cs.indiana.edu%cat Example.java
   abstract class Shape {
     double area() { return -1; }
     public static void main(String[] args) {
       Shape a = new Shape();
     }
   }
   oldschool.cs.indiana.edu%javac Example.java
   Example.java:4: class Shape is an abstract class. It can't be instantiated.
       Shape a = new Shape();
                 ^
   1 error
   oldschool.cs.indiana.edu%
   ```

4. An `abstract` class cannot be instantiated.

   This could be seen in the example above.

   It can however have a `main` method with no problem:

```
oldschool.cs.indiana.edu%ls -l
total 1
-rw-------   1 dgerman       172 Jul 16 12:33 Example.java
oldschool.cs.indiana.edu%cat Example.java
abstract class Shape {
  double area() { return -1; }
  public static void main(String[] args) {
    System.out.println("Hello!");
    // Shape a = new Shape();
  }
}
oldschool.cs.indiana.edu%javac Example.java
oldschool.cs.indiana.edu%java Shape
Hello!
oldschool.cs.indiana.edu%
```

5. A subclass of an `abstract` class can be instantiated if it overrides each of the `abstract` methods of its superclass and provides an implementation (i.e., a method body) for all of them.

   Here's an example that does that and notice the inherited variables too.

```
oldschool.cs.indiana.edu%ls -l
total 1
-rw-------   1 dgerman       511 Jul 16 12:40 Example.java
oldschool.cs.indiana.edu%cat Example.java
abstract class Shape {
  int x, y;
  abstract double area();
}

class Circle extends Shape {
  int radius;
  double area() { return 2 * Math.PI * radius * radius; }
  double manhattanDistanceToOrigin() {
    return Math.abs(x) + Math.abs(y);
  }
}

class Tester {
  public static void main(String[] args) {
    Circle c = new Circle();
    c.radius = 10;
    c.x = -3; c.y = 5;
    System.out.println("Area is " + c.area() +
      " and the distance is " + c.manhattanDistanceToOrigin());
  }
}
oldschool.cs.indiana.edu%javac Example.java
oldschool.cs.indiana.edu%java Tester
Area is 628.3185307179587 and the distance is 8.0
oldschool.cs.indiana.edu%
```

6. If a subclass of an `abstract` class does not implement all of the `abstract` methods it inherits, that subclass is itself `abstract`.

   To see this in the code above remove the definition of `area()` in class `Circle` and recompile:

```
oldschool.cs.indiana.edu%ls -l
total 1
-rw-------   1 dgerman       577 Jul 16 12:44 Example.java
oldschool.cs.indiana.edu%cat Example.java
abstract class Shape {
  int x, y;
  abstract double area();
}

class Circle extends Shape {
  int radius;
  /*** let's take area() out, see if it still compiles:
  double area() { return 2 * Math.PI * radius * radius;
  ****/
  double manhattanDistanceToOrigin() {
    return Math.abs(x) + Math.abs(y);
  }
}

class Tester {
  public static void main(String[] args) {
    Circle c = new Circle();
    c.radius = 10;
    c.x = -3; c.y = 5;
    System.out.println("Area is " + c.area() +
      " and the distance is " + c.manhattanDistanceToOrigin());
  }
}
oldschool.cs.indiana.edu%javac Example.java
Example.java:6: class Circle must be declared abstract.
It does not define double area() from class Shape.
class Circle extends Shape {
      ^
Example.java:18: class Circle is an abstract class. It can't be instantiated.
    Circle c = new Circle();
               ^
2 errors
oldschool.cs.indiana.edu%
```

7. It doesn't compile, and for two reasons, not just one.
8. But the two reasons are closely related.
9. I hope you enjoyed this tutorial.
10. I strongly hope the information presented in it was quite manageable.
11. Please let me know if you have any questions.

Threads

Individual execution paths (with or without sharing).

A thread is a single sequential flow of control within a process. A single process can have multiple concurrently executing threads. For example, a process may have a thread reading input from the user, while at the same time another thread is updating a database containing the user's account balance, while at the same time a third thread is updating the display with the latest stock quotes. Such a process is called a multithreaded process; the program from which this process executes is called a multithreaded program. The `Thread` class is used to represent a thread, with methods to control the execution state of a thread. To create a new thread of execution, you first declare a new class that is a subclass of `Thread` and override the `run()` method with code you want executed in this thread. You then create an instance of this subclass, followed by a call to the `start()` method (which really is, because of inheritance, `Thread.start()`). That method will execute the `run()` method defined by this subclass. You can achieve the same effect by having the class directly implement the `Runnable` interface. Each thread has a priority that is used by the Java runtime in scheduling threads for execution.

There are four kinds of threads programming:
(a) unrelated threads[25],
(b) related (but unsynchronized) threads[26] ,
(c) mutually-exclusive threads[27],
(d) communicating and mutually-exclusive threads [28]

[25]The simplest threads program involves threads of control that do different things and don't interact with each other.

[26]This level of complexity has threaded code to partition a problem, solving it by having multiple threads work on different pieces of the same data structure. The threads don't interact with each other. Here, threads of control do work that is sent to them, but don't work on shared data, so they don't need to access it in a synchronized way. An example of this would be spawning a new thread for each socket connection that comes in.

[27]Where threads start to interact with each other, life becomes a little more complicated. In particular we use threads which need to work on the same pieces of the same data structure. These threads need to take steps to stay out of each others' way so that they don't each simultaneously modify the same piece of data leaving an uncertain result. Staying out of each other's way is known as mutual exclusion. A race condition occurs when two or more threads update the same value simultaneously. To avoid data races, follow this simple rule: whenever two threads access the same data, they must use mutual exclusion. You can optimize slightly, by allowing multiple readers at one instant. In Java, thread mutual exclusion is built on data objects. Every `Object` in the system has its own mutex semaphore (strictly speaking this is only allocated if it is being used), so any object in the system can be used as the "turnstile" or "thread serializer" for threads. You use the synchronized keyword and explicitly or implicitly provide an object, any object, to synchronize on. The runtime system will take over and apply the code to ensure that, at most, one thread has locked that specific object at any given instant. The synchronized keyword can be applied to (A) a class, (B) a method, or (C) a block of code. In each case, the mutex (MUTual EXclusion) lock of the name object is acquired, then the code is executed, then the lock is released. If the lock is already held by the another thread, then the thread that wants to acquire the lock is suspended until the lock is released.

[28]Here's where things become downright complicated until you get familiar with the protocol. The hardest kind of threads programming is where the threads need to pass data back and forth to each other. This is precisely the case with our Penguin in the beginning of the book and in class we will carefully go over the four cases, and all the relevant details.

Contents

Prelude (Preliminaries)	3
Problems and Pain	13
Getting Started (with Feathers)	17
Your First Java Program	21
Algorithms	35
Simple Programs (Truly Basic Java)	41
`ConsoleReader`	50
Types and I/O	51
Reference vs. Primitive Types	63
Syntax	65
Predicates	75
Classes	89
Constructors and Instance Variables	97
Methods	105
Decisions	113
Loops	125
Two-Dimensional Patterns	133
More Loops	149
(Computer) Games	167
Designing Fractions	177
Milestones	183
Java Arrays (Part One)	199
Java Arrays (Part Two)	207
The Bald Soprano (Inheritance)	217
Utilities	231
Events	239
Abstract Classes	243
Threads	247
References	