Sallybagelspasta

1. Classes: Rubber Stamps

Here's a rubber stamp in Java:

```
class Student {

}
```

This looks more like this:

The stamp is the class (the blueprint, the vision). What the stamp leaves behind through the act of stamping are objects (instances). They're kind of the same, all of them. In Java:

```
class Student { // see picture previous page (kiss-kiss stamps)

}

class Stamping {
  public static void main(String[] args) {
    Student a, b, c;
    a = new Student();
    b = new Student();
    c = new Student();
  }
}
```

2. Instance Variables: Nametags

Let's make the blueprint a bit more complex, like a girl with nametag:

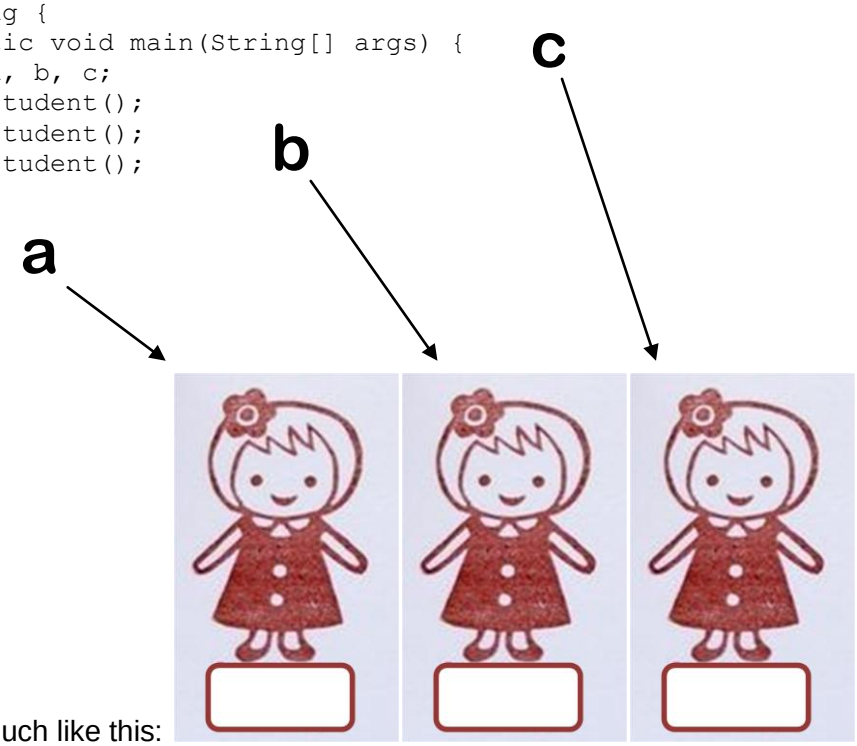
This adds structure and has the following expression in Java:

```
class Student {
  String name;
}
```

If we use this class (blueprint, vision) in Java:

```
class Stamping {
  public static void main(String[] args) {
    Student a, b, c;
    a = new Student();
    b = new Student();
    c = new Student();
  }
}
```

The result looks very much like this:



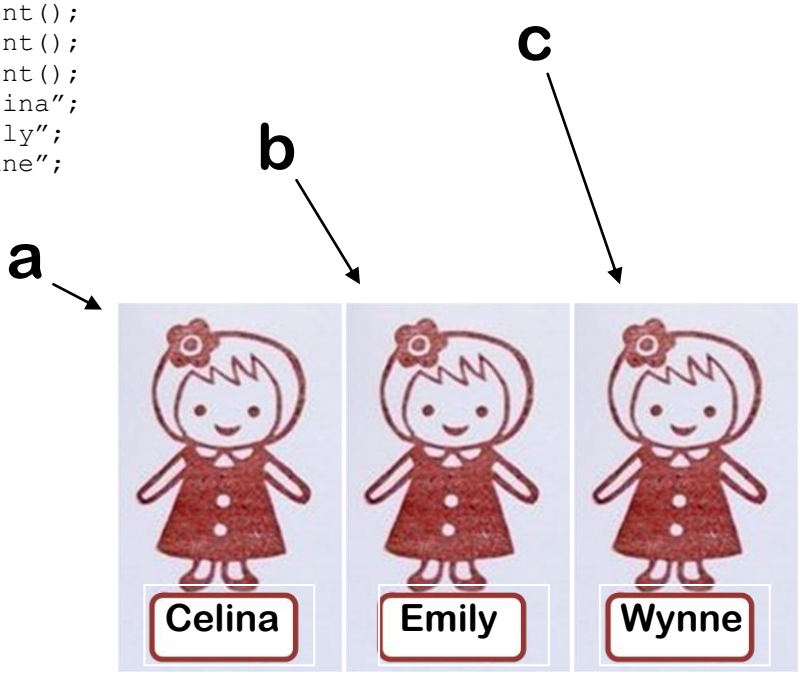
We get three students each with room to write down specific names (e.g., Celina, Emily, etc.)

There is a difference between what we call them (a, b, c) and what they report as their name(s).

In Java the process of using a blueprint to stamp a few characters into existence looks like this:

```
class Stamping {
  public static void main(String[] args) {
    Student a, b, c;
    a = new Student();
    b = new Student();
    c = new Student();
    a.name = "Celina";
    b.name = "Emily";
    c.name = "Wynne";
  }
}
```

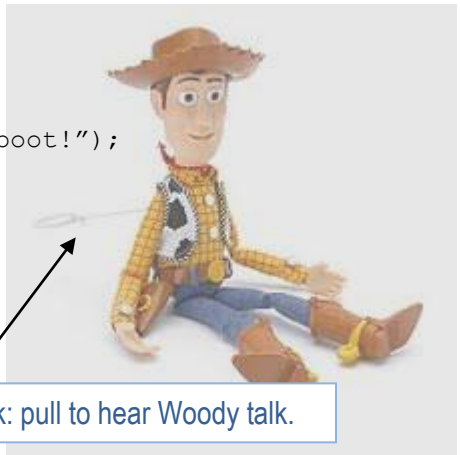
Which looks like:



3. Instance Methods: "There's a snake in my boot!"

But it's even more magical than this. Every time we stamp with (instantiate) a class what we get is a full-blown entity that can even act according to what we taught it. For example:

```
class Student {
  String name;
  void talk() {
    System.out.println("There's a snake in my boot!");
  }
}
```



The string in its back: pull to hear Woody talk.

The class now contains an instance method called `talk()`.

For this the model is Woody from the movie Toy Story.

If you pull the string in the back, it `talk()`s:

But any time you pull its string Woody would say the exact same thing.

If you had two or three of such toys you wouldn't be able to tell which one is `talk()`ing.

If we make a change to the blueprint Woody would be able to report its instance data:

```
class Student {
  String owner;
  void talk() {
    System.out.println("Hello I belong to" + owner);
  }
}
```

So once you write a name on its boot it prints that name in its report.

Now go and watch this on YouTube:

- http://www.youtube.com/watch?v=XPN2duFwDxw&feature=related
- http://www.youtube.com/watch?v=8Hz7m2_VzQM&feature=related

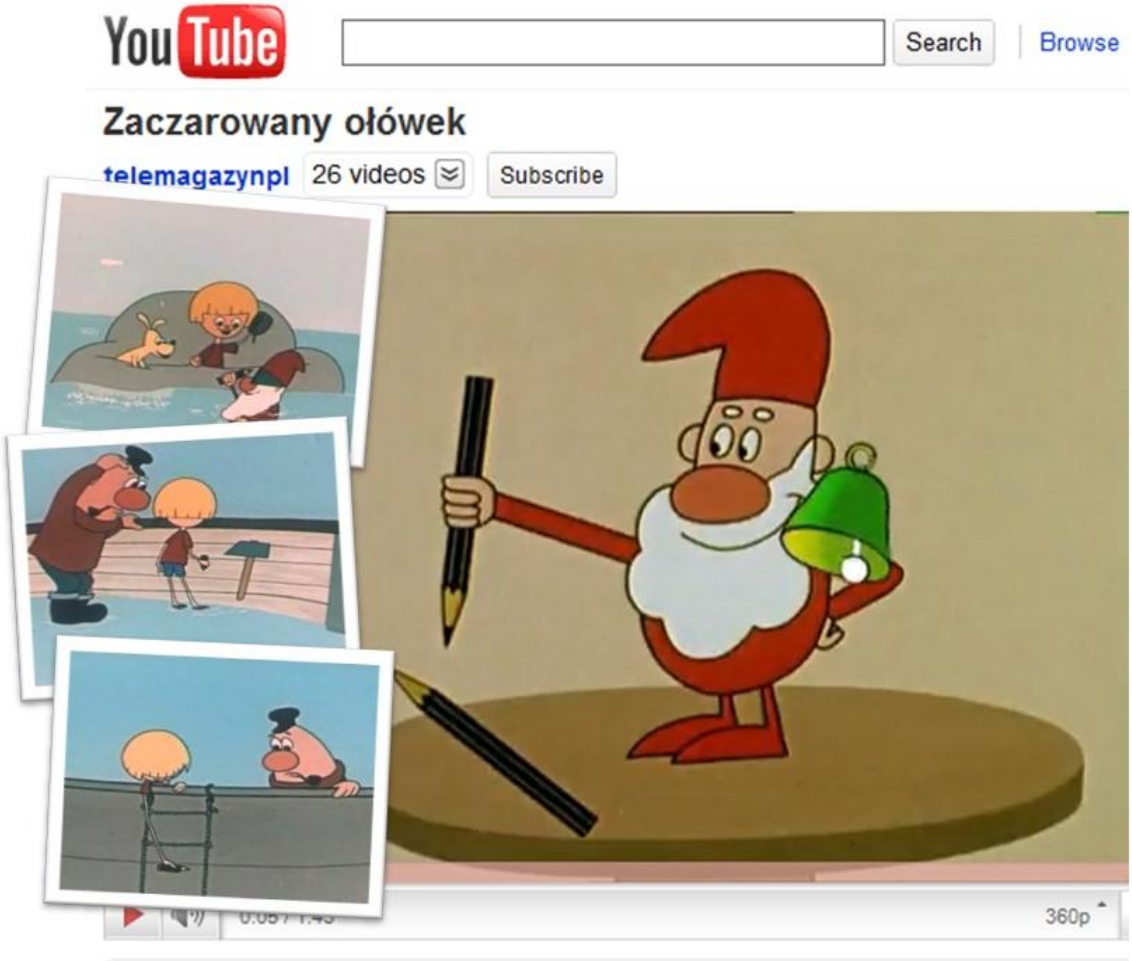
There are many more episodes available online but these two are good to watch for what we need. Here's some background information on the series: it is called Zaczarowany ołówek in original (Enchanted Pencil) and is a Polish cartoon from 1964-1976 made by Se-ma-for. The serial had no dialogues. It tells a story of a boy named Piotr and his dog, aided by an enchanted pencil, which can materialize anything they draw. The boy and his dog usually get in trouble in the first two minutes of the ten minute episode and then an elf/dwarf comes and gives them a magic pencil. Drawing with the pencil is like invoking the `new` operator on any class you may have in mind: as a result you get an instance of that object right away in front of you.

The first 26 episodes, are by and large independent (they have no linking story) although the ones that followed (5-6) were centered around Piotr's quest to save a shipwrecked refugee. There were 31 episodes total, with the episodes 27-31 being remade into a movie in 1991.

Credits (courtesy of Wikipedia):

Art: Karol Baraniecki
Script: Adam Ochocki
Music: Zbigniew Czernelecki and Waldemar Kazanecki

Here's a picture to help you locate it:



Now let's make a change to our class:

```
class Student {
  String name;
  void talk() {
    System.out.println("Hello my name is " + name);
  }
  void eat(String food) {
    name = name + food;
  }
}
```

You are what you eat, remember?

What happened is that the students we create are customizable and they can talk. They report the name we gave them. As a result each one of them sounds differently. They talk when we tell them but they say what is specific to them. The new method allows us to ask them to act on something we give them (a parameter). When we call the method `eat` we give them a `String` (called `food`) which they append to their name(s). If we ask them to talk right after eating something we could see the food they ate showing in their name.

For example:

```
class Student {
  String name;
  void talk() {
    System.out.println("My name is " + name);
  }
  void eat (String food) {
    System.out.println(name + " is eating " + food);
    name = name + food;
  }
}

class Program {
  public static void main(String[] args) {
    Student a, b, c;
    a = new Student();
    b = new Student();
    c = new Student();
    a.name = "Sally";
    b.name = "Aziz";
    c.name = "Adrian";
    a.talk();
    a.talk();
    c.talk();
    a.eat("bagels");
    a.talk();
    b.eat("noodles");
    b.talk();
    a.eat("pasta");
    a.talk();
  }
}
```

Now if we compile the above and run Program we get:

```
-bash-3.2$ javac Program.java
-bash-3.2$ java Program
My name is Sally
My name is Sally
My name is Adrian
Sally is eating bagels
My name is Sallybagels
Aziz is eating noodles
My name is Aziznoodles
Sallybagels is eating pasta
My name is Sallybagelspasta
-bash-3.2$
```

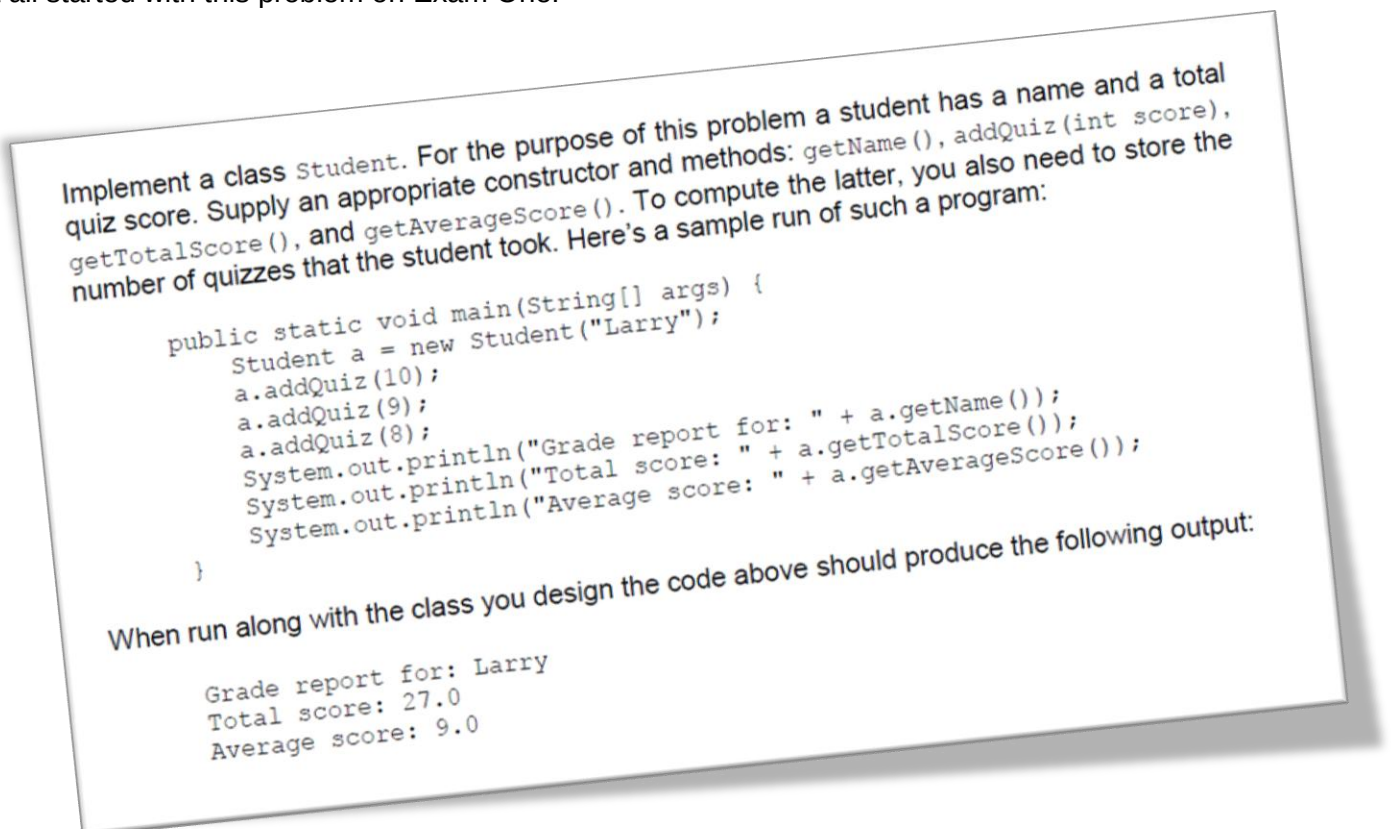
If you want to see such a thing in action go to:

http://www.geeky-gadgets.com/robotic-dog-piggy-bank-17-07-2009/

You will see a `BankAccount` (shaped like a robotic dog) that `eat()`s coins when you deposit them. Watch the video, compare to the above. Now let's remember where it all started.

4. The Exam Problem

It all started with this problem on Exam One:



Implement a class `Student`. For the purpose of this problem a student has a name and a total quiz score. Supply an appropriate constructor and methods: `getName()`, `addQuiz(int score)`, `getTotalScore()`, and `getAverageScore()`. To compute the latter, you also need to store the number of quizzes that the student took. Here's a sample run of such a program:

```
public static void main(String[] args) {
    Student a = new Student("Larry");
    a.addQuiz(10);
    a.addQuiz(9);
    a.addQuiz(8);
    System.out.println("Grade report for: " + a.getName());
    System.out.println("Total score: " + a.getTotalScore());
    System.out.println("Average score: " + a.getAverageScore());
}
```

When run along with the class you design the code above should produce the following output:

```
Grade report for: Larry
Total score: 27.0
Average score: 9.0
```

What if we change the code as follows:

```
class Student {
  String name;
  int points;
  int count;
  void talk[1]() {
    System.out.println("My name is " +
                       name + " (" + points +
                       ", " + count + ") avg: " +
                       (float) points / count);
  }
  void eat (String food) {
    System.out.println(name + " is eating " + food);
    name = name + food;
  }
  void eatGrade (int grade) {
    points = points + grade;
    count = count + 1;
  }
}
```

[1] What this method prints: a concatenation of name, points, count and the calculated average

```
"My name is " + name + " (" + points + ", " + count + ") " avg: " + (float) points / count
```

Then if we run this program:

```
class Program {
  public static void main(String[] args) {
    Student a, b, c;
    a = new Student();
    b = new Student();
    c = new Student();
    a.name = "Sally";
    b.name = "Aziz";
    c.name = "Adrian";
    a.talk();
    a.talk();
    c.talk();
    a.eat("bagels");
    a.talk();
    b.eat("noodles");
    b.talk();
    a.eat("pasta");
    a.talk();
    c.eatGrade(9);
    c.talk();
    c.eatGrade(8);
    c.talk();
    c.eatGrade(10);
    c.talk();
    c.eatGrade(10);
    c.talk();
  }
}
```

The output is:

```
-bash-3.2$ java Program
My name is Sally (0, 0) avg: NaN
My name is Sally (0, 0) avg: NaN
My name is Adrian (0, 0) avg: NaN
Sally is eating bagels
My name is Sallybagels (0, 0) avg: NaN
Aziz is eating noodles
My name is Aziznoodles (0, 0) avg: NaN
Sallybagels is eating pasta
My name is Sallybagelspasta (0, 0) avg: NaN
My name is Adrian (9, 1) avg: 9.0
My name is Adrian (17, 2) avg: 8.5
My name is Adrian (27, 3) avg: 9.0
My name is Adrian (37, 4) avg: 9.25
-bash-3.2$
```

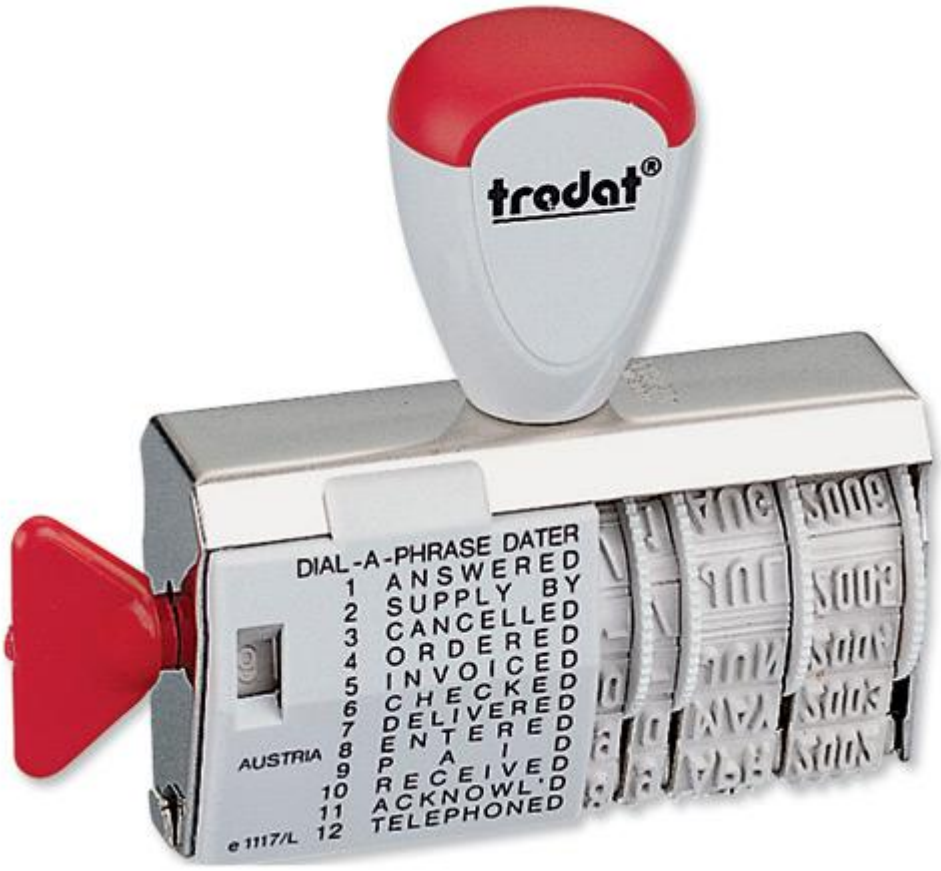
Now Adrian eats grades like Sally eats bagels and pasta.

His name (name, points, total and points/total) keeps changing with what he eats.

Before we look at the solution let's consider the following wrinkle in all of the above.

5. Convenience in Initialization: Constructors

Here's a class:



Any rubber stamp is a class. You can instantiate it by creating objects. Here's an object created from this class:

PAID 14 DEC 2010

A stamp. An object. It's not very versatile or entertaining. It's not Woody, it can't talk() etc. but that is not what it's for. It's a timestamp. It's useful to have it. Like the girl Lilly or the mouse at the beginning of this document it's value and purpose resides in being able to have it and look at it. The class Date defined in java.util implements most of what you see above. Example:

```
class Student {
  public static void main(String[] args) {
    System.out.println(new java.util.Date());
  }
}
```

Compile and run it to see what I mean.

Now I can easily use what we discussed so far to say this is the structure of the class:

```
class Rubberstamp {
  String what;
  int day, year;
  String month;
}
```

Creating an instance of this can be done like before:

```
class Program {
  public static void main(String[] args) {
    Rubberstamp m = new Rubberstamp();
    m.what = "PAID";
    m.day = 14;
    m.year = 2010;
    m.month = "DEC";
  }
}
```

As you can tell this is not exactly what we have in the picture. In the program above we actually implemented something like this:

You get your object but you have to customize it yourself. The stamp we had earlier is such that after you set it you can quickly (and I mean *very fast* by comparison) stamp 100 dates 12/14/10. To do the same with the above you'd have to write (by hand) prohibitively quickly.

So what does the **trodat** stamp have that the *Xstamper* above doesn't?

The answer is: a constructor.

A constructor is an initialization procedure, for the convenience of our users (customers).

Let's see how we code that in Java:

```
class Rubberstamp {
  String what;
  int day, year;
  String month;
  Rubberstamp(String what, int day, String month, int year) {
    this.what = what;
    this.day = day;
    this.month = month;
    this.year = year;
  }
}

class Program {
  public static void main(String[] args) {
    Rubberstamp m = new Rubberstamp("PAID", 14, "DEC", 2010);
  }
}
```

It's more convenient for the user in the real world and in Java.

6. The Solution (Exam Problem)

```
class Student {

  private String name;
  private double quizScore;
  private int quizNumber;

  public Student(String name) {
    this.name = name;
    quizScore = 0;
    quizNumber = 0;
  }

  public String getName() {
    return name;
  }

  public void addQuiz(int score) {
    quizScore += score;
    quizNumber++;
  }

  public double getTotalScore() {
    return quizScore;
  }

  public double getAverageScore() {
    return quizScore / quizNumber;
  }
```

```
  public static void main(String[] args) {
    Student a = new Student("Larry");
    a.addQuiz(10);
    a.addQuiz(9);
    a.addQuiz(8);
    System.out.println("Grade report for: " + a.getName());
    System.out.println("Total score: " + a.getTotalScore());
    System.out.println("Average score: " + a.getAverageScore());
  }
}
```

7. Practice Problem

Try this problem now.

> Design a class of objects called `Car`.
>
> A `Car` has fuel inside.
>
> You can `add` more `fuel` to it.
>
> You can `drive` a `Car`.
>
> When you drive it it burns one gallon of gas every 27 miles.
>
> A `Car` would `report()` its state (the amount of fuel in its tank)

Here's a test program:

```
class Car {
  // this part is your responsibility
}

class Exam {
  public static void main(String[] args) {
    Car a = new Car();
    a.report();
    a.addFuel(3.2);
    a.report();
    a.drive(27);
    a.report();
  }
}
```

It should produce when compiled and run:

```
New Car being created.
Car with 0 gallons of fuel.
3.2 gallons added.
Car with 3.2 gallons of fuel.
Car being driven 27 miles.
Car with 2.2 gallons of fuel.
```

8. Variables and Scope: Buzz in Spanish Mode

Forthcoming. As a preview:

Buzz accessing one of his public methods:
“How dare you open a Space man's helmet on an **uncharted planet**? My eyeballs could have been sucked from of their sockets!” [closes his helmet by pushing button on his right]

Helmet closed.
It was precisely because the method was public that Woody was able to open it (albeit accidentally).

By contrast when they need to reset him they need to access a private method/variable.

The pictures below try to remind you of the context.

9. Questions

1. Does this document help explain what an "instance" or "object" is compared with a "class"? Or does it make it even more confusing? Either way please explain.

2. Does this document help explain what an "instance variable" is? Or does it make it even more confusing? Explain your position.

3. Does the document help explain what an "instance method" is? Or does it make it even more confusing? Why or why not?

4. Does the document clearly explain what a constructor is or not? Explain your position.

Alternatively, if you prefer, consider solving the problem in the next section instead.

10. Apply Your Knowledge

Consider the following code:

```
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;
    }
}

class Exercise {
    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)));

    }
}
```

Can you design a class `Fraction` (or as much as you can of it) such that the code above prints the following output when compiled and run:

```
-bash-3.2$ java Exercise
Test of operations:
  Add: 2/3 + (-2/3) = 0
  Sub: 2/3 - (-2/3) = 4/3
  Mul: 2/3 * (-2/3) = (-4/9)
  Div: 2/3 / (-2/3) = (-1)
Test of predicates:
  1. Does 2/3 equal (-2/3)?  The answer is: false
  2. Is (-4) an integer? The answer is: true
  3. Does 0 equal 0? The answer is: true
  4. Is 5/8 greater than 2/3? The answer is: false
```