Workshop 13: Objects

This workshop looks at how object-oriented programming works, and how it differs to traditional programming.

Difficulty: Intermediate workshop

Contents

0 Introduction

Object-oriented programming is a different way to represent complex concepts than the traditional programming methods. Today we'll take a look at what objects are using the Java programming language. Then, we'll use objects to create characters for a computerised tabletop role-playing game.

This is an intermediate workshop, that builds upon a familiar knowledge of a text-based programming language like Python.

How to use these booklets

The aim of these booklets is to help you attempt these workshops at home, and to explain concepts in more detail than at the workshop. You don't need to refer to use these booklets during the workshop, but you can if you'd like to.

When you need to make changes to your code, they'll be presented in *listings* like the example below. Some lines may be repeated across multiple listings, so check the line numbers to make sure you're not copying something twice.

```
1 public class MyClass
2 {
3     public static void main(String[] args)
4     {
```

When you need to type a command in the command line interface, they will appear in *listings* like the following example. Copy everything *after* the dollar sign. Lines without dollar signs are example outputs, and do not need to be copied.

```
$ java HelloWorld
Hello, World!
```

Occasionally, a concept will be explained in greater detail in *asides*, like the one below. You can read these as you wish, but they're not required to complete the workshop.

Java byte code

Programming languages are traditionally either compiled or interpreted. Compiled languages like C are turned into architecture-specific machine code. Interpreted languages like Python is loaded by their interpreter and ran line-by-line, which is usually a lot slower.

In an attempt to get the flexibility of interpreted language with most of the performance of compiled languages, Java code is compiled into an efficient 'byte code', which is not specific to any architecture, but is instead run through the Java virtual machine.

Everything else

These booklets were created using LaTeX, an advanced typesetting system used for several sorts of books, academic reports and letters. If you'd like to have a look at using LaTeX, We recommend looking at TeXstudio, which is available on most platforms, and also in the Raspbian repository.

To allow modification and redistribution of these booklets, they are distributed under the CC BY-SA 4.0 License. Latex source documents are available at http://github.com/McrRaspJam/booklet-workshops

If you get stuck, find errors or have feedback about these booklets, please email me at: `jam@jackjkelly.com`

1 Java quick start

In your preferred editor, create a new file and copy the following code. Note that Java is case sensitive.

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

Save this file as **HelloWorld.java**, then open a terminal to the same folder, and type the commands listed below. You should get the following output.

```
$ javac HelloWorld.java
$ java HelloWorld
Hello, World!
```

Now, compare that to the equivalent Python program:

```python
print("Hello, World")
```

Coming from most over languages, Java may seem quite complicated. You'll find out that in Java, we have to use a lot of terms that would be optional or excluded in other languages. Java is a language where seemingly nothing is implied.

println()

println() is the print function for Java. In the Java language we have to state which library or class this function comes from, which in this case is from the `System` class. `out` is the standard output stream, text sent to this stream is printed in the terminal.

We'll explain what some of the other lines are soon. For now, each program we write will have these same lines, so we don't need to worry about them just yet.

1.1 Variables

In Java, variables must be defined with a type. Common types might include:

- `int` – An integer (whole number).
- `double` – A double-precision floating point number i.e. a number with a decimal point.
- `boolean` – A boolean value i.e. true or false.
- `String` – A String i.e text. Strings are a collection of type `char`.

An example of variables being defined, then modified is shown below.

```java
int a;
a = 5;
```

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

What would the values stored in a and b be after this code was run?

1.2 Loops and Conditional Statements

If you've done any programming before, you'll know that we frequently use loops, or **iterative statements** to perform repetitive tasks. The `for` statement is a loop that will operate a set number of times based on what parameters we provide it.

```
for(int i=0; i<10; i++)
{
    // Code to repeat
}
else
```

The above `for` loop would loop it's contents 10 times, starting with `i` at 0, and stopping when `i` reaches 10. `i++` is a shorthand notation for `i = i+1`, so each time the loop ends `i` is increased by 1.

Let's modify our hello world program to print multiple times using this loop. Make the changes in the listing below, then compile (`javac`) and run (`java`) your program as before, and you should now see "Hello, World!" printed 10 times.

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

A **conditional statements** like `if else` are used to selectively run code based on a condition. Let's use an `if else` statement to change our program a little. We'll change the `println()` to print out `i`, the current count number. Then, when we reach a specific number, we'll print out a bit of text after the number.

```
        for(int i=0; i<10; i++)
        {
            System.out.println(i);

            if (i == 5)
                System.out.println("My favourite number!");
        }
```

Now compile and run the program again, you should see something like

```
...
4
5
```

```
My favourite number!
6
...
```

Optional: A standard basic programming test is FizzBuzz. Use loops and conditions to recreate the game.

1.3 Methods

Sequential Execution

A traditional computer program runs one line at a time from top to bottom. In a language like Java, the *main method* is the thing that runs from start to end, and the program ends when the main method is complete. We use method calls to 'extend' the main method by jumping to additional code.

Java also has **methods**, often called **functions** or **procedures** in other languages, used in much the same way.

Let's write a method that prints out a square number. In our current program, we'll define a new method like the following listing. We also need to make a call to the function from `main()`. When you run your program again, it should now print 'Hello, World!' 10 times.

```java
    private static void printHelloWorld()
    {
        System.out.println("Hello, World!");
    }

    public static void main(String[] args)
    {
        for(int i=0; i<10; i++)
        {
            printHelloWorld();
        }
    }
```

Parameters are passed in the same way as many other languages, within the parentheses with a type deceleration. Functions can also **return** a value, but the method deceleration must be changed to say what type it returns.

Currently the function is `void`, meaning it returns nothing. To return a number, we need to change `void` to `int`. Let's make the function return the square of the parameter input, then `main()` will print that number.

```java
    private static int squareOf(int number)
    {
        int squareNumber = number * number;
        return squareNumber;
    }

    public static void main(String[] args)
    {
        for(int i=0; i<1000; i++)
        {
            System.out.println(squareOf(i));
        }
    }
```

2 Object-oriented Design

When compared with Object-oriented programming, the more traditional form of programming that we have been using is usually referred to as **Procedural programming** due to its use of procedure (method) calls.

One problem that arises with procedural programming is the difficulty of representing more complex things in code. **Primitive** types like `int` and `float` can store simple values, but how do we store a complex concept like a user account, or perhaps a product in an store inventory system? That product may a multitude of attributes, such as a name, price, a position within the store, and a barcode.

Object-orientated design was designed to be a solution to this problem. For today, we are interested in the following core properties of objects:

- Objects contain their own **variables**, which are properties of the concept they are representing.
- Objects contain their own **procedures**. This allows us to write methods that may do things like interact with other objects or automatically generate their variables.
- Objects can be **instantiated**. Each instance of an object has it's own unique set of variables.

We'll see what these things mean as we code our own objects.

2.1 Example: A Vector Object

In mathematics, a **vector** is a quantity consisting of a **direction** and a **magnitude**. A vector describes how to get from one point in space to another, but does not in itself have a position. You can think of a vector as an 'arrow', the arrow points from one point to another, but if you pick up the arrow and place it somewhere else, it is still the same vector.

In computing, a vector is usually stored as a coordinate pair (x, y and z for 3D vectors), and the other properties, like magnitude can be calculated from these. To find the magnitude we calculate $\sqrt{x^2 + y^2 + z^2}$.

A basic vector object, with a coordinate pair and magnitude method might look like the following listing. Note the set of variables, as well as a method that can be called for each object.

```java
import java.lang.*;

public class Vector3
{
    public double x;
    public double y;
    public double z;

    public double magnitude()
    {
        // Magnitude is the
        // square root of x^2 + y^2 + z^2
        double sum = Math.pow(x, 2) + Math.pow(y, 2) + Math.pow(z, 2);

        return Math.sqrt(sum);
    }
}
```

This would be used in a program as shown below. We'll explain some of this notation with next example.

```
Vector3 testVector = new Vector3();
testVector.x = 3;
testVector.y = 4;
testVector.z = 0;
System.out.println(testVector.magnitude());
```

2.2 Creating a Pet Object

Creating a class

Time for you to have a go. We're going to use objects to create a `Pet` type for a simple program.

An object is an **instance** of a **class**, so we'll start by creating the class for our pet type. Create a new file called `Pet.java`, then type the same class definition as we had in our hello world program.

```
public class Pet
{

}
```

Naming our pet

We said before that objects have their own set of variables. For each `Pet`, we will store the `name` of that pet, as well as its `species`. Add variable decelerations to the `Pet` class, as shown below.

```
public class Pet
{
    public String name;
    public String species;
}
```

Note that the variables are prefixed by `public`. We'll explain why this is the case shortly. We've create a fairly spartan class, but we can already begin using it in a program to represent pets.

Create another class called `PetProgram.java`, this class will contain a main method, and will be where we write the program that uses `Pet` objects.

```java
public class PetProgram
{
    public static void main (String[] args)
    {
    }
}
```

Our program will create a pet, and give it a name and species. Within your main method, create a `Pet` variable as shown below.

```java
        //Create a pet
        Pet pet1 = new Pet();
```

the `new` keyword means the program will make an **instance** of the `Pet` class. Each instance of a class is an **object**, and has its own copy of the variables from the `Pet` class. we access an object's variables using the notation `object.variable` . To set the `name` and `species` of our pet, we do the following:

```java
        pet1.name = "Doug"
        pet1.species = "Dog"
```

We'll also create a second pet, so we may compare two objects at once. Then, to test our objects are working as expected, we'll print the names of our pets by accessing each object's `name` variable.

```java

        //Create a second pet
        Pet pet2 = new Pet();
        pet2.name = "Polly"
        pet2.species = "Parrot"

        //print out the pet names
        System.out.println("My two pets are called " + pet1.name + " and "
            + pet2.name);
    }
```

Let's compile and run our program. When working with multiple classes, compile the class that contains `main()` method, and any classes that are mentioned in the program will also be compiled.

```
$ javac PetProgram.java
$ java PetProgram
My two pets are called Doug and Polly
```

Feeding our pets treats

Let's implement a behaviour for our pets. We'll cast aside realistic animal behaviour, and have all species behave the same way to make the problem a little simpler.

When we give our pet a treat, we'll get them to do one of two things.

1. Eat the treat immediately.
2. Store the treat to eat later.

We'll store the number of treats the pet has stored, and create a method to reward the pet with a treat.

Start by adding treats as an integer variable. Then write a function to add to this variable.

```java
public class Pet
{
    public String name;
    public String species;

    private int treats;

    public void giveTreat()
    {
        treats = treats + 1;
    }
}
```

We'll decide whether the pet will eat its treat using a virtual coin flip, which we can do by generating a random `boolean`.

```java
    public void giveTreat()
    {
        Random rand = new Random();
        boolean eatnow = rand.nextBoolean();

        //Store treat if not ate.
        if (eatnow == false)
            treats = treats + 1;
    }
```

You'll notice that in the listing above, `treats` is set to private. This means that only code within this class can access that variable. calling `pet.treats` in `PetProgram` would produce an error. I've done this because the pet should decide which treats get ate, and being able to set `treats` directly would disallow the pet the chance to eat its treats.

Access Modifiers

The most common access modifiers are `public` and `private` variables and methods that do not specify their access modifier are given a default modifier that for the purposes of this workshop behaves the same as `public`.

Access modifiers are useful for preventing incorrect modification of variables or use of methods.

We can currently change `treats` by calling `giveTreat()`. To print the current number from our main method, we'll need to provide an **accessor method**, this will take a `private` variable and allow us to read it through a `public` method.

```java

    public int getNumberOfTreats()
    {
        return treats;
    }
```

Now, let’s return to `PetProgram` and feed one of our pets some treats.

```java
public class PetProgram
{
    public static void main (String[] args)
    {
        //Create a pet
        Pet pet1 = new Pet();
        pet1.name = "Doug"
        pet1.species = "Dog"

        //feed the pet some treats
        pet1.giveTreat();
        pet1.giveTreat();
        pet1.giveTreat();

        //how many treats did the pet store?
        System.out.println(pet1.name + " has " + pet1.getNumberOfTreats() +
             " treats.");
    }
}
```

Because our program now has a random element, you’ll get a different result each time you run your program.

```
$ javac PetProgram.java
$ java PetProgram
Doug has 2 treats.
$ java PetProgram
Doug has 1 treats.
$ java PetProgram
Doug has 3 treats.
```

Optional: Modify `giveTreat()` so that instead of calling it 3 times, we can instead do `giveTreat(3);`

Optional: Think of some other properties that you could add, and what type of variable they should be (some might be better represented by their own type). Add one of those to your `Pet` class, along with a method to do something with that variable.

3 A Simple Text-based RPG Encounter

We'll be recreating a simple gameplay mechanic for a fantasy RPG game at this point in the workshop. We will create a **combat encounter** between the **player**, a Wizard, and a **monster**, a magically reanimated zombie.

We'll be starting with the simplest possible implementation, then gradually iterate upon it to make the game more interesting to play. We'll be making too many small changes to give full instructions here, so instead, the starting point of the program will be listed here, and an outline of potential changes will be provided.

Character.java

```java
public class Character
{
    public String name;
    public int health;
    public int damage;  //attack damage

    public void attackCharacter(Character opponent)
    {
        opponent.health -= this.damage;
    }
}
```

CombatProgram.java

```java
public class CombatProgram
{
    public static void main (String[] args)
    {
        //Create the player
        Character player = new Character();
        player.name = "Wizard";
        player.health = 100;
        player.damage = 25;

        //Create the enemy
        Character enemy = new Character();
        enemy.name = "Zombie";
        enemy.health = 75;
        enemy.damage = 25;

        //Player attacks enemy
        player.attackCharacter(enemy);
    }
}
```

Changes to make

1. Currently, the player makes an attack but nothing is printed. Modify `attackCharacter` to describe each attack.

2. Create a game loop in `CombatProgram`, where the two characters attack each other until one has been defeated.

3. The game has the same result each time it is played. Change the attack method so that damage dealt is based on a virtual dice roll.
4. Give the player a choice between ranged and close-quarter attacks. the two attacks have different damage potential. Zombies can only use close-quarter attacks.
5. Add a defence calculation to each attack.
6. Change the program so that a continuous stream on enemies can be fought. You will probably want to move the Game Loop into a separate method.

Congratulations if you manage to implement all of those! You can keep thinking of changes to make from here. Perhaps your wizard should have an amount of Mana to spend on attacks? Perhaps the player should be able to heal somehow?

4 What we didn't cover

We didn't have time to cover everything about objects in this one workshop, so here are a few extra important features to look at, if you want to explore object-oriented programming further.

Constructor methods

We can use constructor methods to set variables when we first create an object. The constructor is called when you use the keyword `new`. When we called `new` in our programs before, a default constructor was used which does not set any variables.A constructor method is identified by a method with the deceleration `public <Class Name>()`. A typical constructor for the vector class from earlier would be

```

    public Vector3(double xIn, double yIn, double zIn)
    {
        x = xIn;
        y = yIn;
        z = zIn;
    }
```

for the same use case as before, we would now use the following code;

```
Vector3 testVector = new Vector3(3, 4, 0);
System.out.println(testVector.magnitude());
```

The static context

Methods and variables can be prefixed with the keyword `static`. When a method or variable is `static`, it is not copied to each object instance. Instead, the variable/method is *shared* between all instances of the class.

Static members can not be accessed through an instance, and vice versa. If Vector3 had static variables and methods, we would access them like so:

```
//To access static things
Vector3.staticVariable += 1;
Vector3.staticMethod();

//this will NOT work
testVector.staticVariable +=1;
testVector.staticMethod();

//and NEITHER will this
vector3.x += 1;
vector3.magnitude();
```

Inheritance

An important feature of Object-oriented design, sub-classes can inherit properties of other classes.

For example, if we wanted to create a wild animal in our Pet program, the wild animal wouldn't have a name. We could make an Animal class, with just the name variable (and perhaps treats), then pet would be a subclass of animal, in which would just be the name variable.

```
public class Animal
{
    public String species;
    ...
}

public class Pet extends Animal
{
    //Pet's also have a species variable,
    //it is inherited from Animal
    public String name;
    ...
}
```

A Completed Code Listings

A.1 Hello, World!

HelloWorld.java

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

HelloWorld.java (with loop)

```java
public class HelloWorld
{
    public static void main(String[] args)
    {
        for(int i=0; i<10; i++)
        {
            System.out.println("Hello, World!");
        }
    }
}
```

HelloWorld.java (counting program)

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

            if (i == 5)
                System.out.println("My favourite number!");
        }
    }
}
```

A.2 Vector

Vector3.java

```java
import java.lang.*;

public class Vector3
{
    public double x;
    public double y;
    public double z;

    public double magnitude()
    {
        // Magnitude is the
        // square root of x^2 + y^2 + z^2
        double sum = Math.pow(x, 2) + Math.pow(y, 2) + Math.pow(z, 2);

        return Math.sqrt(sum);
    }

    public Vector3(double xIn, double yIn, double zIn)
    {
        x = xIn;
        y = yIn;
        z = zIn;
    }
}
```

A.3 Pet

Pet.java

```java
public class Pet
{
    public String name;
    public String species;

    private int treats;

    public void giveTreat()
    {
        Random rand = new Random();
        boolean eatnow = rand.nextBoolean();

        //Store treat if not ate.
        if (eatnow == false)
            treats = treats + 1;
    }

    public int getNumberOfTreats()
    {
        return treats;
    }
}
```

PetProgram.java

```java
public class PetProgram
{
    public static void main (String[] args)
    {
        //Create a pet
        Pet pet1 = new Pet();
        pet1.name = "Doug"
        pet1.species = "Dog"

        //feed the pet some treats
        pet1.giveTreat();
        pet1.giveTreat();
        pet1.giveTreat();

        //how many treats did the pet store?
        System.out.println(pet1.name + " has " + pet1.getNumberOfTreats() +
             " treats.");
    }
}
```