Ian Kenny

October 16, 2017

Software Workshop

Lecture 3

In this lecture

- An introduction to classes and object-oriented programming.

Introduction to classes

When you write a Java program you have to write a *class*. All Java programs must contain at least one class. All of the ones we have seen so far have contained exactly one class. Usually a program will have many classes.

A program will also use many existing classes. Our programs to date have used `String`, and `Scanner`, for example, but many, MANY others are available to use.

We also write our own classes. We can write as many classes as we feel we need for a program.

So, our programs will contain combinations of classes we have written and classes others have written.

Introduction to classes

Why would we write a class? Well, firstly, as just discussed, you **must** write at least one class to write a Java program, and that class must contain a `main` method.

But we often write many classes. The reason we do that is that simply using the classes in the Java API (and others we may be able to download/use) is not enough for our purposes.

If we were writing a survival horror video game, for example, and required zombies in the game, we *could* consult the Oracle Java API documentation but I doubt we'd find a `Zombie` class there. We would need to write our own. Even if someone, somewhere *had* written a `Zombie` class it probably wouldn't meet our needs.

Introduction to classes

Zombies have their own data and operation needs that will not be met by the classes available to us. We can write our own `Zombie` class in order to fulfil our requirements.

Introduction to classes

Classes include the data and operations needed for some particular *type*. When we write our own classes, we consider the data we will need and the operations that we will need to perform on that data and build them into a class.

For the `Zombie` class, for example, we might have data such as `health`, `decompositionRate`, `angerLevel` and operations called `decompose()`, `becomeMoreAngry()`, `biteNearestFace()`.

We can use other classes in the process of creating our class, and in a variety of ways (which we will get to).

A first class: `BasicZombie`

```
public class BasicZombie {

    public float health;
    public float decompositionRate;
    public int angerLevel;
    public String name;
}
```

Listing 1 : BasicZombie.java

What do you notice about this class?

`BasicZombie` class: first attempt

There is no `main` method. That is fine, but it means that this class is not a program.

There are some data items declared in the class. They are not declared in a `main` method. They couldn't be - there isn't one.

The data members, and any operations that we provide (methods), are called *members* of the class.

Like the class itself, they are declared `public` (unlike previous variables we have seen). This means they are accessible outside the class. Let's look at what that means when we create a `Zombie` object.

To create a `BasicZombie` object, we need *another class* to do that.

Using the BasicZombie class

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

        BasicZombie zombie = new BasicZombie();

        zombie.health = 15.0f;
        zombie.decompositionRate = 2.0f;
        zombie.angerLevel = 1;
        zombie.name = "Nigel";
    }
}
```

Listing 2 : ApocalypticWorld.java

What do you notice about this class?

Using the `BasicZombie` class

This class has a `main` method so it is a program. The main method creates a new `BasicZombie` *object* and then accesses the data inside the object.

The members of an object are accessed in Java using the `.` operator (it's called 'dot').

The next program shows that the data members of `BasicZombie` are just like regular variables and can be used in the same way.

Using the BasicZombie class

```
public class ApocalypticWorld2 {
    public static void main(String[] args) {

        BasicZombie zombie = new BasicZombie();

        zombie.health = 15.0f;
        zombie.decompositionRate = 2.0f;
        zombie.angerLevel = 1;
        zombie.name = "Nigel";

        zombie.health = zombie.health + 2.0f;
        zombie.decompositionRate *= 4;

        if (zombie.angerLevel <= 1)
            zombie.angerLevel++;

        System.out.println(zombie.health);
    }
}
```

Listing 3 : ApocalypticWorld2.java

Classes and objects

This point is very important: the **class** `BasicZombie` class is a 'template' or a 'blueprint' for a `BasicZombie`. It simply describes what a `BasicZombie` contains, what it does and how it does it. It is not **itself** a `BasicZombie`. To get an *actual* `BasicZombie` we need to create an **object** of that type.

The way that we create an actual object in Java is to use the keyword `new`. This causes an actual object to be created that is then usable in a program.

Java uses special methods called *constructors* to actually create objects. If we don't provide a constructor for a class ourselves, Java will silently create a very basic one for us. We can provide our own constructors though.

Default Constructor Level1Zombie

```
public class Level1Zombie {

    public float health;
    public float decompositionRate;
    public int angerLevel;
    public String name;

    public Level1Zombie() {
        health = 0;
        decompositionRate = 0;
        angerLevel = 0;
        name = "NOT_SET";
    }
}
```

Listing 4 : Level1Zombie.java

Default Constructor: `Level1Zombie`

The *default constructor* is a constructor that takes no parameters. The usual purpose of the default constructor is to set the data members of the class to default values that are *appropriate for the class*. The purpose of all constructors is to *initialise* an object of the class.

The default constructor silently provided by Java does (which has now been replaced) does the same as the one we just added, except that the Java-provided constructor would simply set the value of *name* to `null`. `null` is a special value which means that an object variable has no current value.

So, essentially, by providing our own default constructor we get more control over how an object is initialised.

All constructors have the same name as the class itself.

Other constructors

We can provide as many constructors as we need to although they must all have different *parameter lists*. For example, we could provide a constructor that initialises the name of the zombie to a *particular* value, and another one that sets some of the other data members of the object.

Which constructor we use to initialise an object depends on what our needs are at the particular point at which we create the object and which information we actually know at that point. When we create a zombie object, for example, we may not *know* what its name is.

More constructors

```
public class Level2Zombie {

    public float health;
    public float decompositionRate;
    public int angerLevel;
    public String name;

    public Level2Zombie() {
        health = 0;
        decompositionRate = 0;
        angerLevel = 0;
        name = "NOT_SET";
    }

    public Level2Zombie(String n) {
        name = n;
        health = 0;
        decompositionRate = 0;
        angerLevel = 0;
    }

    public Level2Zombie(float h, String n) {
        name = n;
        health = h;
        decompositionRate = 0;
        angerLevel = 0;
    }
}
```

Listing 5 : Level2Zombie.java

Using the new constructors

What will the output be?

```
public class ApocalypticWorld3 {
    public static void main(String[] args) {

        // uses the default constructor
        Level2Zombie z1 = new Level2Zombie();

        // uses the constructor that takes a String
        Level2Zombie z2 = new Level2Zombie("Steven");

        // uses the constructor that takes a String and an int
        Level2Zombie z3 = new Level2Zombie(5, "Brian");

        System.out.println(z1.name);
        System.out.println(z2.name);
        System.out.println(z3.name);
    }
}
```

Listing 6 : ApocalypticWorld3.java

Using arrays with classes

If we want a zombie horde, we can put them in an array. Look at this listing. How many zombies will be created?

```
public class ApocalypticWorld4 {
    public static void main(String[] args) {

        final int NUM_ZOMBIES = 200;

        // how many zombies does this create?
        Level2Zombie[] zombieHorde = new
            Level2Zombie[NUM_ZOMBIES];

        // this causes a 'null pointer exception'
        zombieHorde[0].name = "Terry";
    }
}
```

Listing 7 : ApocalypticWorld4.java

Using arrays with classes

Creating the array does *not* create the elements of the array. We need to create those separately. Which constructor is being called?

```
public class ApocalypticWorld5 {
    public static void main(String[] args) {

        final int NUM_ZOMBIES = 200;

        Level2Zombie[] zombieHorde = new
            Level2Zombie[NUM_ZOMBIES];

        // this loop creates the actual zombies
        for (int i = 0; i < NUM_ZOMBIES; i++) {
            zombieHorde[i] = new
                Level2Zombie("UnknownZombie"+(i+1));
        }
    }
}
```

Listing 8 : ApocalypticWorld5.java

A note on the code examples

Since the zombie and other classes need to change many times to illustrate various points and also to keep them brief enough to keep to one slide, they change frequently. That means that some data members and methods will appear in one version but then disappear from the next version, only to reappear again later. I need to do this to explain the different topics. Just take each class as a separate class and don't worry that things seem to disappear!

Methods

Constructors are *methods*. In object-oriented programming, *functions* (or *subroutines*, etc.) are called *methods*. Methods exist usually to perform operations on the data in an object.

We can create methods other than constructors, i.e. methods whose purpose is not simply to initialise an object.

We have already used methods in our programs so far. Methods such as `substring()` and `charAt()` in the `String` class, for example.

Methods

The listing below shows two new methods added to the zombie class (the constructors are not shown in this listing).

```
public class Level3Zombie {

    public float health;
    public float decompositionRate;
    public int angerLevel;
    public String name;

    public void setHealth(float h) {
        health = h;
    }

    public float getHealth() {
        return health;
    }
}
```

Listing 9 : Level3Zombie.java

Method: `setHealth()`

```
public void setHealth(float h) {
    health = h;
}
```

Listing 10 : Level3Zombie.java

This is a *public* method (we will come to that shortly).

Its *return type* is `void`. This means it does not return a value. Methods can return a type (`int`, `float`, `String`, `Zombie`, etc.) or return nothing (`void`).

Its *name* is `setHealth`.

Its parameter list contains one element: a `float` called `h`.

In the method `definition`, it simply sets the value of the `health` data member to the value of `h`.

Method: `setHealth()`

```
public void setHealth(float h) {
    health = h;
}
```

Listing 11 : Level3Zombie.java

The parameter called `h` is what is known as a *local variable*. It is a variable that is local to this method. It cannot be accessed elsewhere. Once the method ends, the variable `h` is destroyed.

Notice that the method has accessed the data member `health`. That is valid because `health` is declared in the class, not in any individual method. We will come to this again shortly.

Method: `getHealth()`

```
public float getHealth() {
    return health;
}
```

Listing 12 : Level3Zombie.java

This is a *public* method (we will come to that shortly).

Its *return type* is `float`. This means it must `return` a `float` to the *calling code* (whichever bit of code called the method).

Its *name* is `getHealth`.

Its parameter list contains no elements. It has no parameters.

In the method `definition`, it simply returns the **value** of the data member `health`.

Method parameters

To recap, some methods have no parameters. They do not require any information *from the calling code* in order to perform whichever operation they are designed to perform. They might use the data members of the class.

Other methods have parameters. We can have any type as a parameter, and we can have as many parameters as we need. It is not a good idea to have too many parameters though as that makes a method hard to use (the caller has to remember which parameters to pass, which types, in which order, etc.).

Methods: scope

```
public class Level4Zombie {

    public float health;
    public float decompositionRate;
    public int angerLevel;
    public String name;

    public void setHealth(float h) {
        health = h;
        veryShortLivedVariable++; // no!
    }

    public float getHealth() {
        int veryShortLivedVariable = 4;
        return health;
    }
}
```

Listing 13 : Level4Zombie.java

Methods: scope

The data member `health` is declared as a member at the class level. This means it is accessible throughout the class, which means in **all** methods and constructors.

The local variable `h` in the method `setHealth()` is local to that method and cannot be accessed anywhere else in the class.

The local variable `veryShortLivedVariable` is local to the method `getHealth()` and cannot be accessed anywhere else in the class.

Note that this is nothing to do with the 'order of declaration'. The class on the next slide is exactly equivalent and has exactly the same problem.

Methods: scope

```
public class Level4aZombie {

    public float health;
    public float decompositionRate;
    public int angerLevel;
    public String name;

    public float getHealth() {
        int veryShortLivedVariable = 4;
        return health;
    }

    public void setHealth(float h) {
        health = h;
        veryShortLivedVariable++; // no!
    }
}
```

Listing 14 : Level4aZombie.java

Using methods

In order to use the methods we have created, we simply use the dot operator, as with data members. What will the output be?

```
public class ApocalypticWorld6 {
    public static void main(String[] args) {

        Level3Zombie z1 = new Level3Zombie();
        Level3Zombie z2 = new Level3Zombie("Steven");

        z1.setHealth(15.5f);
        z2.setHealth(8.6f);

        float h = z1.getHealth();

        System.out.println(h);
        System.out.println(z2.getHealth());
    }
}
```

Listing 15 : ApocalypticWorld6.java

Using methods

When we invoke a method of an object we say we are *calling* the method. In the previous program, the `main` method is calling the methods in the `Level3Zombie` class. Where necessary it passes parameters to the methods in order to provide the information required by the method, or it passes no parameters if none are required.

Which parameters are required by the method, their types and the order in which they must be passed, is specified by the declaration of the method.

Data validation

One of the things we can do in methods is to validate the 'inputs', i.e. the parameters. Consider an application requiring a method called `setAge()` which receives an integer as a parameter. We may not wish anyone using the method to be able to set a negative age so we could prevent that in the `setAge()` method.

In the zombie case, imagine that the zombie's health must be in the range 0 to 10. We can put a simple `if` statement in the `setHealth()` method to check for that. We can simply have a rule that unless the input parameter has a value in the correct range we ignore it.

That should work, right? ...

Data validation

```
public class Level5Zombie {

    public float health;
    public float decompositionRate;
    public int angerLevel;
    public String name;

    public void setHealth(float h) {
        if (h >= 0 && h <= 10)
            health = h;
    }

    public float getHealth() {
        return health;
    }
}
```

Listing 16 : Level5Zombie.java

Using the data validation

```
public class ApocalypticWorld7 {
    public static void main(String[] args) {

        Level6Zombie z1 = new Level6Zombie();

        // OK, nothing will happen here (which is good)
        z1.setHealth(15.5f);

        // Nothing will happen here either
        z1.setHealth(-5);

        // So, this all seems to be working ... but
        z1.health = 123500.2f;

        // What just happened there?!

    }
}
```

Listing 17 : ApocalypticWorld7.java

Data validation

The problem is that we can *still* directly access the `health` member of the zombie classes and thus avoid the data validation. We don't have to use the methods.

A very simple change will fix that. The problem is that the data members of the zombie classes are declared as `public` which means that other classes are able to access them freely. We can restrict access to the class that 'owns' the data members by simply making them `private`.

Using private

```
public class Level6Zombie {

    private float health;
    private float decompositionRate;
    private int angerLevel;
    private String name;

    public void setHealth(float h) {
        if (h >= 0 && h <= 10)
            health = h;
    }

    public float getHealth() {
        return health;
    }
}
```

Listing 18 : Level6Zombie.java

Using private

```
public class ApocalypticWorld8 {
    public static void main(String[] args) {

        Level6Zombie z1 = new Level6Zombie();

        // OK, nothing will happen here (which is good)
        z1.setHealth(15.5f);

        // Nothing will happen here either
        z1.setHealth(-5);

        // Nice try ...
        z1.health = 123500.2f;
    }
}
```

Listing 19 : ApocalypticWorld8.java

Using `private`

Now it won't compile (which is a good thing).

```
$ javac ApocalypticWorld.java
$ java ApocalypticWorld
ApocalypticWorld8.java:13: error: health has private
    access in Level6Zombie
z1.health = 123500.2f;
^
1 error
```

Listing 20 : Terminal commands

Access specifiers

Now, an attempt to access the data members of `Level6Zombie` (for example) from outside of that class will fail because they are declared as `private`. This means that only the class that contains them may access them directly. Other classes will now need to use the `getHealth()` and `setHealth()` methods (in this case) if they wish to access those variables. But now access to them is *controlled*.

All members of a class should be specified as `private` **unless** other classes really need to access them, in which case they can be declared `public`. Methods can also be declared `private`.

One of the advantages of using the access specifier `private` is, just like the example here, we can control access to data and prevent its corruption.

We will return to this topic in the future.

Getters and Setters

The methods shown above (apart from the constructors) are in a class of methods called 'getters and setters'. Such methods get their name from the fact that their purpose is to enable the getting and setting of data members in a class. The very firm convention is to always start such methods with the words 'get' and 'set'.

'get' methods return the value of a data member (or compute a value) and 'set' methods set the value of a data member. It is not universally the case but you can usually tell a 'getter' by the fact that it has no parameters and a 'setter' by the fact that it does.

Other methods

Other methods can exist apart from constructors and getters and setters. Any method that is required to perform some operation relevant to a class can be created. Consider the next listing.

Other methods

```
public class Level7Zombie {

    private float health;
    private float decompositionRate;
    private int angerLevel;
    private String name;

    public void setHealth(float h) {
        if (h >= 0 && h <= 10)
            health = h;
    }

    public float getHealth() {
        return health;
    }

    public void biteNearestFace(Face nearestFace) {
        nearestFace.bite();
    }
}
```

Listing 21 : Level7Zombie.java

What about this version?

Look at this listing. What do you notice?

```
public class Level8Zombie {

    private float health;
    private float decompositionRate;
    private int angerLevel;
    private String name;

    public void setHealth(float health) {
        if (health >= 0 && health <= 10)
            health = health;
    }

    public float getHealth() {
        return health;
    }
}
```

Listing 22 : Level8Zombie.java

What about this version?

What will the output be?

```
public class ApocalypticWorld9 {
    public static void main(String[] args) {

        Level8Zombie z1 = new Level8Zombie();

        z1.setHealth(5.5f);

        System.out.println(z1.getHealth());
    }
}
```

Listing 23 : ApocalypticWorld9.java

What about this version?

It will output `0.0`.

This is because the new (incorrect) version of the method does not actually set the data member as intended. It sets the value of the parameter to ... the value of the parameter! The parameter is a local variable that exists only for the duration of the method. The intention of this method is to change the data member called `health` but it changes the value of the parameter called health (but to the same value!).

It is actually common to have parameter names the same as data member names. It is not an error in itself but its behaviour may not be what was intended. In order to ensure that we are referring to the correct variable we can use the keyword `this`.

this

```
public class Level9Zombie {

    private float health;
    private float decompositionRate;
    private int angerLevel;
    private String name;

    public void setHealth(float health) {
        if (health >= 0 && health <= 10)
            this.health = health;
    }

    public float getHealth() {
        return health;
    }
}
```

Listing 24 : Level9Zombie.java

`this`

The word `this` is a way of referring to the current object. It allows us to specify that the identifier we want is the one that is *inside* the object, not from elsewhere.

Some Java developers believe that one should always use `this` inside an object to make it completely clear which variables, etc. we are referring to.

Method overloading

We saw that with constructors we can have multiple constructors with the same name (by definition, constructors all have the same name) but with different parameter lists. We can do the same with other methods. We can have multiple methods with the same name as long as they have different parameter lists. Why do that? There are perhaps various ways that an operation can be performed based on the information provided. Look at the `String` class. There are a number of `substring()` methods, for example. This is called *method overloading*.

Note that we cannot override a method by simply creating a new method with the same parameters but a different return type.

Consider the next listing.

Method overloading

```
public class Level10Zombie {

    private float health;
    private float decompositionRate;
    private int angerLevel;
    private String name;

    public void getMoreAngry() {
        angerLevel++;
    }

    public void getMoreAngry(int angerIncrease) {
        angerLevel += angerIncrease;
    }
}
```

Listing 25 : Level10Zombie.java

Method overloading

This version has two `getMoreAngry()` methods. Perhaps it is the case that the default increase in anger is 1. Thus, there is a version of the method that takes no parameters and simply increases the anger by 1. Perhaps it is also the case that, in extreme circumstances, a zombie may need to get much more angry quickly. In that case, we could use the other version of the method and add a value greater than 1 to the anger.

Method overloading

```
public class ApocalypticWorld10 {
    public static void main(String[] args) {

        Level11Zombie z1 = new Level11Zombie();

        // increases the anger by 1
        z1.getMoreAngry();

        // increase anger by 10
        z1.getMoreAngry(10);

        // increase anger by 1000!
        z1.getMoreAngry(1000);

        // increases the anger by 1 again
        z1.getMoreAngry();
    }
}
```

Listing 26 : ApocalypticWorld10.java

The `Object` class

The following will be expanded in full detail when we look at *inheritance* and *polymorphism*.

All classes in Java are ultimately derived from the `Object class`. This means that `Object` is the ultimate ancestor of **all** Java classes, including all of the classes we have written so far. We will look in detail at what this means in a later session.

However, for now, what it means is that there are some methods that we can provide in our classes that help all other classes to use them. One of them is the `equals()` method (you've already used the one from `String`) and the other is called `toString()`. We will now create the `toString()` method and leave the `equals()` method for another time (since it requires understanding features that we haven't got to yet).

`toString()`

The purpose of the `toString()` method is to give a `String` representation of an object to allow information about the object to be gathered, printed out, etc. This can be extremely helpful for debugging.

toString()

```
public class Level11Zombie {

    private float health;
    private float decompositionRate;
    private int angerLevel;
    private String name;

    public Level11Zombie() {
        name = "NOT_SET";
    }

    public void setHealth(float h) {
        if (h >= 0 && h <= 10)
            health = h;
    }

    public float getHealth() {
        return health;
    }

    public String toString() {
        return "name = " + name + "\nhealth = " + health +
            "\ndecompositionRate = " + decompositionRate +
            "\nangerLevel = " + angerLevel + "\n";
    }
}
```

Listing 27 : Level11Zombie.java

using toString()

```
public class ApocalypticWorld11 {
    public static void main(String[] args) {

        Level11Zombie z1 = new Level11Zombie();

        System.out.println(z1.toString());
    }
}
```

Listing 28 : ApocalypticWorld11.java

```
$ javac ApocalypticWorld11.java
$ java Apocalyptic11World
name = NOT_SET
health = 0.0
decompositionRate = 0.0
angerLevel = 0
```

Listing 29 : Terminal commands

Class members and object members

Recall that we said before that a class is like a template for an object. The class defines what data an object contains, what methods it has and what those methods do. However, we don't actually have an entity that we can *do* things with until we create an *object* of that class.

Each time we create an object, that object gets its own copies of all of the data members of the class. That needs to be the case because, for example, we need our zombies to have different names, different health values, etc. The data is not *shared*, it is individual to each object.(Methods are dealt with differently but we need not worry about that for now).

Class members and object members

Everything on the previous slide is true, however there are circumstances in which we want a data member or a method to belong to the *class* not the object. We would do this in the case that the data member or method is not specific to any particular object, i.e. does not depend on the data or methods that are a part of that object.

An example in the zombie case might be a constant called MAX_RUN_SPEED. Consider the case in which all zombies have the same maximum running speed. No zombie can run more quickly than the rate expressed by MAX_RUN_SPEED. If this is the case, it is pointless embedding this information in **every** zombie object because it never varies, hence that would be wasteful and slightly misleading.

We declare that a variable or method belongs to a class by simply declaring it as `static`.

static data members

```
public class Level12Zombie {

    private float health;
    private float decompositionRate;
    private int angerLevel;
    private String name;
    private int runSpeed;

    private final static int INITIAL_RUN_SPEED = 2;
    public final static int MAX_RUN_SPEED = 5;

    public Level12Zombie() {
        name = "NOT_SET";
        runSpeed = INITIAL_RUN_SPEED;
    }

    public int getRunSpeed() {
        return runSpeed;
    }

    public void speedUp() {
        runSpeed++;
    }

    public void slowDown() {
        runSpeed--;
    }
}
```

Listing 30 : Level12Zombie.java

`static` data members

`static` members do not belong to any object therefore should not be accessed through an object using the `objectName.memberName` syntax.

`static` members are members of the *class* hence are accessed via the *class name*, but also using the dot operator. See the example on the next slide.

Remember that `final` is used to declare a constant. The value of a data item marked with `final` cannot be changed by the program. Any attempt to change such an item will lead to a compiler error.

using static members

```
public class ApocalypticWorld12 {
    public static void main(String[] args) {

        Level12Zombie z1 = new Level12Zombie();

        z1.speedUp();
        z1.speedUp();

        if (z1.getRunSpeed() > Level12Zombie.MAX_RUN_SPEED)
            z1.slowDown();
    }
}
```

Listing 31 : ApocalypticWorld12.java

`static` methods

Methods can also be declared `static` and the same principle applies. `static` methods do not depend on the specific data members of any object so can be declared as belonging to the class instead.

Consider a method that will simply print to the screen the standard zombie greeting. All zombies greet people in the same way so there's no need for this method to belong to any specific object.

using static methods

```
public class Level13Zombie {

    private float health;
    private float decompositionRate;
    private int angerLevel;
    private String name;

    public Level12Zombie() {
        name = "NOT_SET";
        runSpeed = INITIAL_RUN_SPEED;
    }

    public static void giveStandardZombieGreeting() {
        System.out.println("HhuuuuuuAAAAZZzzzzUUUUUGGGGGGG!");
    }


}
```

Listing 32 : Level13Zombie.java

using static methods

```
public class ApocalypticWorld12 {
    public static void main(String[] args) {

        Level12Zombie z1 = new Level12Zombie();

        Level12Zombie.giveStandardZombieGreeting();

    }
}
```

Listing 33 : ApocalypticWorld13.java

And ... finally

Finally, we can explain `public static void main()`!

`public` means its accessible outside the class.

`static` means it belongs to the class and not to any specific object.

`void` means it doesn't return a value.

And `main` is the name of the method.