1 The instanceof operator

The `instanceof` operator can be used to test whether a reference variable contains an object of a particular type. It returns true or false, depending on whether the object belongs to the class used in the test or not.

The most common use of `instanceof` is

- to test the class type, before attempting a downcast from a base class type variable to a subclass variable:

```java
class Image { }
class ColourImage extends Image { }

ColourImage colImage;
Image i = /* ... some code */
if (i instanceof ColourImage)
    colImage = (ColourImage) i;
```

Example:

```java
class Phone {
    public void connect() {
        System.out.println("Normal Connection");
    }
}
```

2AIT515 - OBJECT ORIENTED SOFTWARE DEVELOPMENT

Lecture 8: The instanceof operator - The final keyword - The Java class hierarchy - Order of object initialisation - The `toString()` method - Enumerated types - Classpath - Defining packages - Inner classes

Dr Dimitris C. Dracopoulos

`email`: d.dracopoulos@westminster.ac.uk
class MobilePhone extends Phone {
    public void connect_bluetooth() {
        System.out.println("Bluetooth Connection");
    }
}

class PhoneManager {
    public Phone getPhoneType(int type) {
        if (type == 1)
            return new MobilePhone();
        else
            return new Phone();
    }
}

public class InstanceofExample {
    public static void main(String[] args) {
        MobilePhone mp;
        PhoneManager man = new PhoneManager();
        Phone p = man.getPhoneType(1);

        //p.connect_bluetooth(); // Error!

        if (p instanceof MobilePhone) {
            mp = (MobilePhone) p;
            mp.connect_bluetooth();
        } else {
            p.connect();
        }
    }
}

2 Final Classes and Methods

The keyword final can be used for the following:

- **final variables**: these are constants, i.e. their value cannot be changed.
- **final classes**: these classes cannot be used as a base for another class. You cannot inherit from a final class.
- **final methods**: these cannot be overridden in a subclass.
Example:

```java
final class Patent {
}

class MyPatent extends Patent { } // Error! Cannot inherit from final class

class Calculator {
    final int increaseByOne(int x) {
        return x+1;
    }
}

class MyCalculator extends Calculator {
    int increaseByOne(int x) { // Error! Cannot override final method
        return x+5;
    }
}

class MyInteger {
    public int i;
}

public class FinalExample {
    public static void main(String[] args) {
        final MyInteger m1 = new MyInteger();
        m1 = new MyInteger(); // Error! m1 is constant
        m1.i = 5; // OK
    }
}
```

As illustrated in the example above, a final object reference variable cannot be changed so as to point to a different variable. However, the contents of the object that the variable is pointing to, can be changed (assuming that the object itself is mutable).

3 The Java class hierarchy

The parent of every Java class (whether user defined or library) is class `Object`. This means that all the methods found in `Object` are inherited by every class.

It should be noted that as Java does not support multiple inheritance (i.e. a class can only inherit from a single class directly), the above implies that `Object` is the direct or indirect parent class of any Java class. For example, in Figure 1 `Object` is an indirect parent class of `ArrayList` (its direct parent class is `AbstractList`).
4 Order of Object Initialisation

The following series of events occur during the creation of an object:

1. Sufficient memory is allocated in the heap to hold the object. (including all instance variables specific to the object and all instance variables of its superclass - and since the superclass inherits all instance fields from its superclass enough memory is allocated to include the data of its super-superclass, etc.). It should be noted that any user specified initialisation values are not assigned to the fields at this point (see Step 4).

2. All instance variables of the object are initialised to their default values, i.e. all field objects to nulls, primitive numerics to zero and booleans to false.

3. The default constructor (i.e. the constructor with no arguments) of the direct superclass of the object is called. The constructor of the superclass will invoke the default constructor of its own superclass and so on, until the constructor of the parent class of all classes java.lang.Object is called.

4. The user specified initialisation values of the instance variables are assigned to them and any initialisation blocks are executed.

5. The actual body of the constructor is executed.
Example:

The following example illustrates the sequential execution of the above steps during the creation of an object of class `TimeTravel`:

```java
class SpaceTravel extends Travel {
    private float distance = 5000;

    SpaceTravel() {
        System.out.println("SpaceTravel() constructor!");
    }
}

class TimeTravel extends SpaceTravel {
    private String timeElapsed = "0 years";

    // initialisation block
    {
        System.out.println("Initialisation block");
    }

    TimeTravel() {
        System.out.println("TimeTravel() constructor!");
    }
}

class Travel {
    Travel() {
        System.out.println("Travel() constructor!");
    }

    public static void main(String[] args) {
        TimeTravel t = new TimeTravel();
    }
}
```

When the example is run it produces the output:

`Travel() constructor!`
`SpaceTravel() constructor!`
`Initialisation block`
`TimeTravel() constructor!`

The sequence of events which take place when the statement `TimeTravel t = new TimeTravel()` is encountered:

1. Memory is allocated in the heap to hold an object of class `TimeTravel`. 
2. The object’s instance variables, i.e. timeElapsed are initialised to their default values, so timeElapsed is assigned the null value.

3. The constructor of TimeTravel is called. Before doing anything else, this calls the default constructor of its superclass SpaceTravel.

4. The instance variable of SpaceTravel distance is initialised to its default value 0.

5. The constructor of SpaceTravel calls its superclass default constructor Travel and the body of that constructor is executed printing the string “Travel() constructor!”.

6. The constructor of Travel returns and the instance variable distance is initialised to 5000.

7. The body of the constructor SpaceTravel is executed, printing the message “SpaceTravel() constructor!”.

8. The constructor of SpaceTravel returns, and the instance variable of TimeTravel timeElapsed is initialised to the string “0 years”.

9. The initialisation block in TimeTravel is executed, printing the message “Initialisation block”.

10. The body of the constructor of TimeTravel is executed, producing the string “TimeTravel() constructor!”.

5 The toString() method

- Class Object implements method toString() so as to return an hexadecimal representation of the memory address of an object.
  This method is inherited by all Java classes (all methods of class Object are inherited by all Java classes).
- Whenever an object needs to be converted to a String, its toString() method is called. For example:
  
  ```java
  Boat b = new Boat();  // assume a defined class Boat
  System.out.print(b);  // b.toString() called: memory address of b is printed
  ```
- Classes can override their toString() method, so as to provide a “textual representation” of their content in a meaningful manner.

Example:

```java
class Boat {
}

class HighSpeedBoat {
  private String name;
}
HighSpeedBoat(String name) {
    this.name = name;
}

public String toString() {
    return name;
}

public class ObjectPrintingTest {
    public static void main(String[] args) {
        Boat b1 = new Boat();
        System.out.println("b1: " + b1);

        HighSpeedBoat b2 = new HighSpeedBoat("Dolphin");
        System.out.println("b2: " + b2);
    }
}

When the above program is run, it displays:

b1: Boat@82ba41
b2: Dolphin

Since class Boat does not override toString() method, the implementation of toString() inherited from Object will be used when attempting to print the b1 object. Class HighSpeedBoat overrides toString(), so as to print the instance field name of the object.

6 Enumerated Types

Versions of Java which are newer that 1.5 (J2SE 5.0 or later) support enumerated types.

In the following example, a variable of type ShirtSize can only take a value among small, medium, large, extra_large.

enum ShirtSize { small, medium, large, extra_large}

- Enumerations are type safe. The compiler will not allow a variable of an “enumerated” type to take a value not defined within the enumeration.

Example:

public class EnumerationExample {
    enum ShirtSize { small, medium, large, extra_large}

    public static void main(String[] args) {

    }
ShirtSize john_size = ShirtSize.medium;
// ShirtSize george_size = 10;  // Error!
System.out.println("John wears shirt size: " + john_size);
}
}

The output of the above code is:

John wears shirt size: medium

7 Specifying the Classpath (finding the path of classes)

Assuming that some user classes have been defined in different directories. How does the java compiler and JVM know where to search for the classes used in a program?

- The -classpath option can be passed to javac and java, indicating that classes can be located in a number of different directories.

- The different directories that javac, java have to search are included after the -classpath option, and they are separated from each other using colon (:) in Unix and semicolon (;) in Windows.

Example:

In the following Unix example, the compiler and the JVM will search for classes, in the current directory (indicated by dot .), in directory /home/w0121345/java_files and in directory /home/w0121345/programs.

javac -classpath ./:home/w0121345/java_files:/home/w0121345/programs MyProgram.java
java -classpath ./:/home/w0121345/java_files:/home/w0121345/programs MyProgram

In the following Windows example, the compiler and the JVM will search for classes, in the current directory (indicated by dot .), in directory C:\home\w0121345\java\_files and in directory D:\home\w0121345\programs.

javac -classpath .;C:\home\w0121345\java_files;D:\home\w0121345\programs MyProgram.java
java -classpath .;C:\home\w0121345\java_files;D:\home\w0121345\programs MyProgram

If no classpath is specified, by default the compiler and JVM will search in the current directory for any classes utilised by a program.
8 Defining the package that a class belongs to

Related classes can be “packaged” together in a single package which is a library of classes.

- A unique name must be chosen for the package name, so that there will be no ambiguity when specifying two different classes with the same name located in two different packages.

- The Java recommendation says, that in order to create a unique package name, the name of the package should consist of the reverse of the internet domain name.
  For example, package uk.ac.wmin.cs.2ait515 could contain all the classes necessary to run programs for module 2ait515 in the CS department of the University of Westminster.

- The name of the package should be specified in the first line of a file, as follows:

```java
package uk.ac.wmin.cs.2ait515;

class A {
    // ...
}

public class MyProgram {
    // ...
    public static void main(String[] args) {
        // details of implementation omitted ...
    }
}
```

// other classes in the file ....

Note that the name of the package becomes part of the name of the class. Therefore to run the above program, the following should be used:

```
java uk.ac.wmin.cs.2ait515.MyProgram
```

The java utility expects that a directory structure identical to the package name is present. Therefore, class MyProgram should be inside directory uk/ac/wmin/cs/2ait515 (for Windows the equivalent directory is uk\ac\wmin\cs\2ait515).

This implies that the -classpath has to be set appropriately, so as to contain the directory that contains the top directory of the directory structure (in the above example, the -classpath should include the directory which contains directory uk).

Importing Classes from a different package

Before a class uses another class which resides from a different package, the latter class has to be imported. This is illustrated in the example below.

File PackageExample.java located in directory p1 contains:
package p1;

import uk.ac.wmin.cs.Satellite;

public class PackageExample {
    public static void main(String[] args) {
        Satellite sat = new Satellite();
        sat.rotate(45.0);
    }
}

File Satellite.java located in directory uk/ac/wmin/cs contains:

package uk.ac.wmin.cs;

public class Satellite {
    public void rotate(double angle) {
        System.out.println("Rotating");
    }
}

To compile the two files:

javac uk/ac/wmin/cs/Satellite.java
javac p1/PackageExample.java

To run the PackageExample code:

java p1.PackageExample

9 Creating jar files

A single zipped file can contain a number of compiled Java classes. Such a file is called a jar file, because the jar utility is used to create it.

The utility preserves the directory structure, i.e. information about directories is included in the jar file, and when the contents of the file are extracted the directory structure is restored.

To create file myclasses.jar which contains all the files and directories under directory uk:

jar cvf myclasses.jar uk

the jar utility responds by showing which files are included in the jar file:

added manifest
adding: uk/(in = 0) (out= 0)(stored 0%)
adding: uk/ac/(in = 0) (out= 0)(stored 0%)
To extract all the files included in a jar file `myclasses.jar`:

```
jar xvf myclasses.jar
```

A class does not have to be extracted from a jar file, before being used in a program. It is sufficient to include the jar file in the classpath of `javac` and `java`. Note that it is not sufficient to include the directory that the jar file is located, but the jar file itself has to be specified explicitly in the classpath.

In the following, program `PackageExample` will search for class `uk.ac.wmin.cs.Satellite` (whose objects are created in `PackageExample`) within the current directory (.) and within the jar file `myclass.jar`:

```
java -classpath .:myclass.jar p1.PackageExample
```

### 10 Inner Classes

Inner classes are classes which are not defined at the top level, but they are either defined:

- inside another class: *inner member classes*.
- inside a block of code: *inner local classes*.

Inner classes are not visible outside the class (or block of code), in which they are defined. Inner member classes have access to the instance fields of the enclosing class.

A typical usage of inner classes are in the creation of event handlers for graphical user interfaces. Additional examples of inner classes include *anonymous inner classes* and *static nested classes*. These are not covered in this module.

**Example:**

```java
public class InnerClassExample {
    int i = 55;

    // inner member class
    class MyUtility {
        int m = i; // access i of enclosing class

        void foo() {
            System.out.println("Calling foo() of MyUtility class");
        }
    }
}
```
public void doWork() {
    // create an instance of the inner member class
    MyUtility u1 = new MyUtility();
    u1.foo();
}

public static void main(String[] args) {
    // inner local class
    class MyCalc {
        int k;
    }

    MyCalc m1 = new MyCalc();
    m1.k = 5;
}