APIT - Java recap etc

Dr. Simon Rogers

14/11/2014

Contents

Overview

- Programming
- Objects, interfaces etc
- Immutable objects
- Call by reference / value
- Final
- Testing
- Documenting
- Tools

Programming

Your experience

- How many people had their first experience of programming in S1?
- What other languages have you used?
- What programming tools have you used?
- How many of you know what the following things are:
    - Objects?
    - Functions?
    - Stacks? Queues? Linked lists? Arrays?
    - Regular Expressions?

High v low -level languages

- Computers follow instructions in *machine code*
    - binary… quite hard for humans to read

- not *that* long ago, humans had to program computers like this (some academics in SoCS will remember…)
    - Machine code is *low level*
- At the other extreme, *high level* languages are eas(y,ier) for humans to read
    - Java is a fairly *high level* language

Compiled v Interpreted v Java

- Computers run programs in Machine Code
    - Low level language. Not human readable
- Some languages require programs to be *compiled* into Machine Code
    - e.g. C++
- Some languages are interpreted line by line as they are run
    - e.g. Matlab
- Java is a bit different
    - It is compiled into Bytecode
    - Bytecode is run on the Java Virtual Machine
    - What is a virtual machine?

Compiling and running Java from the command line

- I will do this in class
- In simplest case, it involves two steps:
    - Compiling: `javac MyClass.java`
    - Running: `java MyClass`
- We will see some more complex examples throughout the course

---

Organising projects

- All the programs in this course involve small numbers of classes
- For larger projects, it is important to organise all your files in a standard manner
    - Eclipse does this automatically
- If you want to do it manually, good description is available here

---

Object Orientation

- Java is an *Object Oriented* language
- What are objects?
- Why program with objects?
- Why not program with objects?
- Homework: read this

Classes

- Classes define objects
- `Pet` is a simple class used by `PetTest`
- Classes allow us to neatly combine related attributes and methods

---

Inheritance

- One of the big strengths of OOP is *inheritance*
  - Creating classes that *inherit* everything of another class and add more
  - e.g. `Dog`, `Goldfish`, `PetInheritanceTest`
- In this example we also see `overridden` methods
  - `Dog` and `Goldfish` override the `description` method
  - The loop does not care which subclass the objects belong to.
  - This is very useful in many applications - *polymorphism*

Abstract Classes

- Standard classes can be *instantiated*
  - i.e. we can create objects of their type (e.g. `Pet`, `Dog`, etc)
- Java allows you to define classes that cannot be instantiated: **Abstract classes**
- Abstract classes can only be sub-classed
  - e.g. `AbstractPet`, `Cat` and `AbstractPetTest`
- There is no situation where you would *have* to use an abstract class but many where it's neater
- Note that sub-classes have to implement all abstract methods or be abstract themselves

---

Interfaces

- Interfaces are similar to abstract classes but:
  - Cannot have fields (unless they are `static` and `final`)
- See `InterfacePet`, `Parrot` and `TestInterfacePet`
- Interfaces are like contracts: they just specify the methods a class must implement
  - Note that methods in interfaces are abstract by default
- Note:
  - Classes can only sub-class one class…
  - …but can implement many interfaces

Exercise: measurement with units

You are working in a team building a system to work with GPS data (from e.g. a running watch). Your task is to create the part of the code to deal with distance values, that can be stored in a number of different units (metres, kilometers, miles, etc). Objects with different units have to be able to be compared (for e.g. sorting).

- Can you think of a way of doing this with a single class (e.g. `UnitDistance` that stores the distance as a double and an object representing the unit). You'll need an interface somewhere…
- My solution in `recap/code/UnitDistance`
- Could you solve this with an abstract class instead of an interface? How?

To help understand this exercise, think about what `unit` objects need to **do**. Interfaces define the methods that objects must pefform, i.e. the things they must be able to do. In this example, a unit needs to be able to convert a value into some standard (i.e. for distances, we convert everything to metres) and provide a string for displaying (e.g. m, km, ft).

In my solution there is another example of interfaces: `Comparable`, which you saw in semester 1, that allows, e.g., objects in an array to be sorted.

Some odds and ends

`public`, `private` and `protected`

- Fields/attributes and methods are either public, private, or protected
  - Public: anything can access
  - Private: only objects of this type can access
    - e.g. `provideBone` method in `Dog`
  - Protected: only objects of this type, sub-classes (and other things within the same package)
    - e.g. `name` and `age` in `Pet` and `AbstractPet`
- In general, be as restrictive as possible.

`static`

- Fields and methods can also be declared `static`
- This means that they are accessible without an object being instantiated
- Useful for storing generic methods and constants
- e.g. `MyMath` and `MyMathTest`
  - `areaOfCircle` is used without creating a `MyMath` object
  - Another static thing is used here - what is it?

`static` attributes

- Static attributes within an object are *shared* by all instances
- Change the value in one, and it will change in all of the others…
  - Useful, but not always the neatest solution

Memory in Java

- Most data in Java is stored in Objects
- Objects are stored in an area of memory called the *heap*

    - There is one heap for the whole program
- Each thread has its own stack
    - All programs have at least one thread
    - In your programmes so far, there is one thread

The Stack

- The stack is used for three purposes:
    - Evaluating expressions
    - Storage of local variables (variables in the current *scope*)
    - Management of method calls
- Think of it as a stack of paper.
    - Pieces of paper are put on (pushed), and taken off (popped), the top of the pile
    - LIFO: Last In First Out

The Heap

- The heap is an area of memory used to store objects in Java
- Objects in the heap are accessible from any part of the program that has a local reference to the object
- Threads share a single heap
    - i.e. each thread can access objects in the heap
    - Useful, but causes all of the multi-threading problems we wil see
- In Java, objects are stored in the heap, references to objects are stored in the stack
    - This is very important, and we will come back to it later…

Garbage collection

- Java periodically deletes objects when they are not needed
- An object is not needed if it is *unreachable*
    - i.e. no references to it exist

```java
public class Garbage {
    public static class A {
        B b;
        public A(B b) {
            this.b = b;
        }
    }
    public class B {}
    public static void main(String[] ar
        B b = new B();
        A a = new A(b);
        B b1 = new B();
        B b2 = new B();
        b = null;
        b2 = null;
    }

}
```

Figure 1: Example program

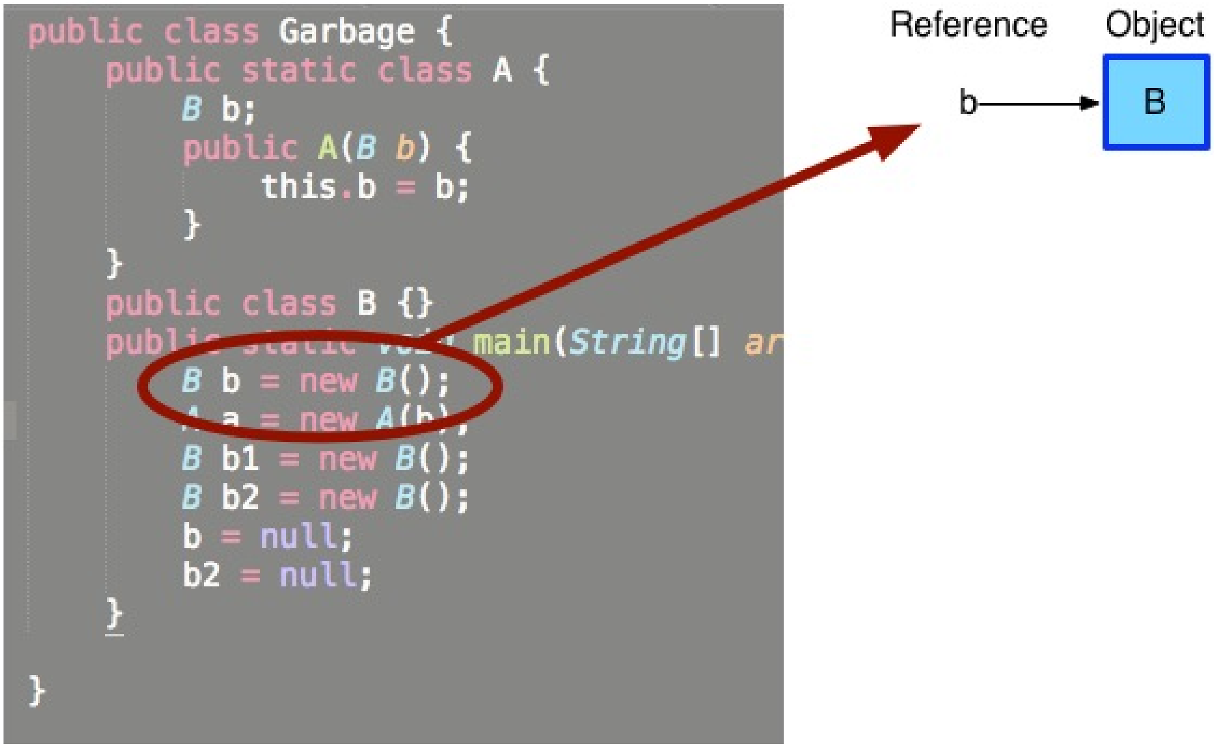

Figure 2: Object (B) and reference (b) created

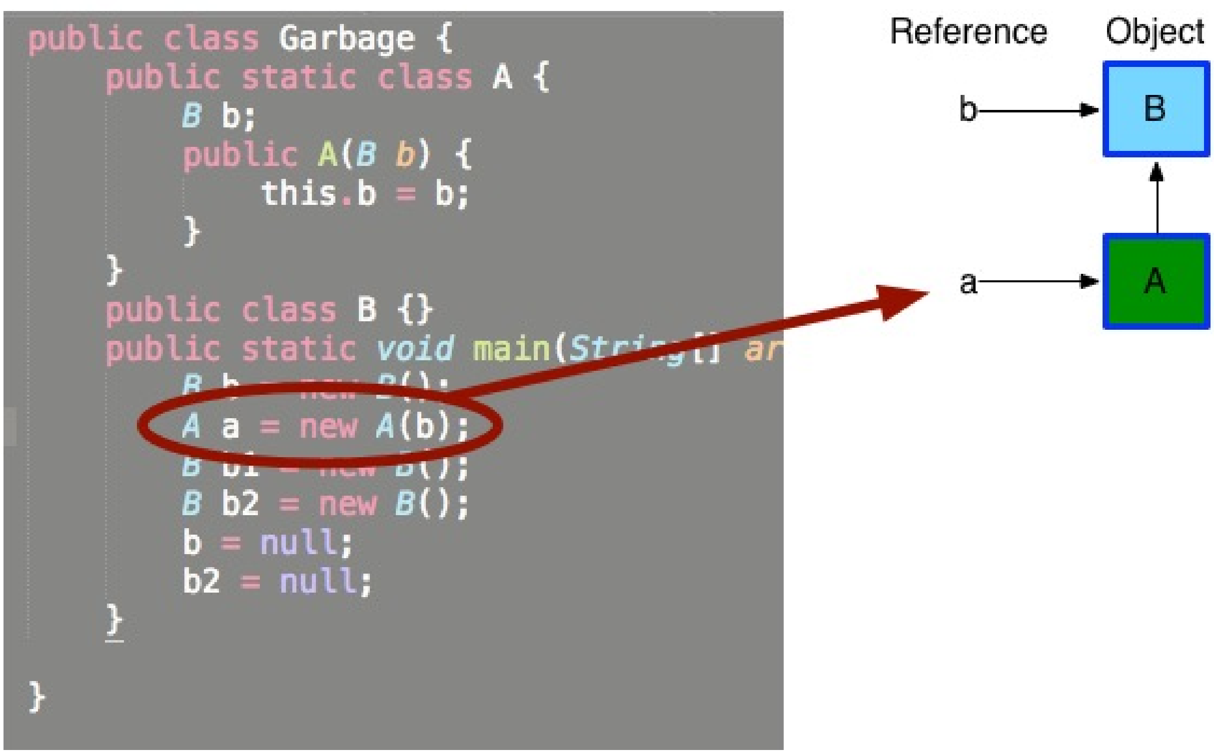

Figure 3: object (A) and reference (a) created. Note that A includes a reference to B)

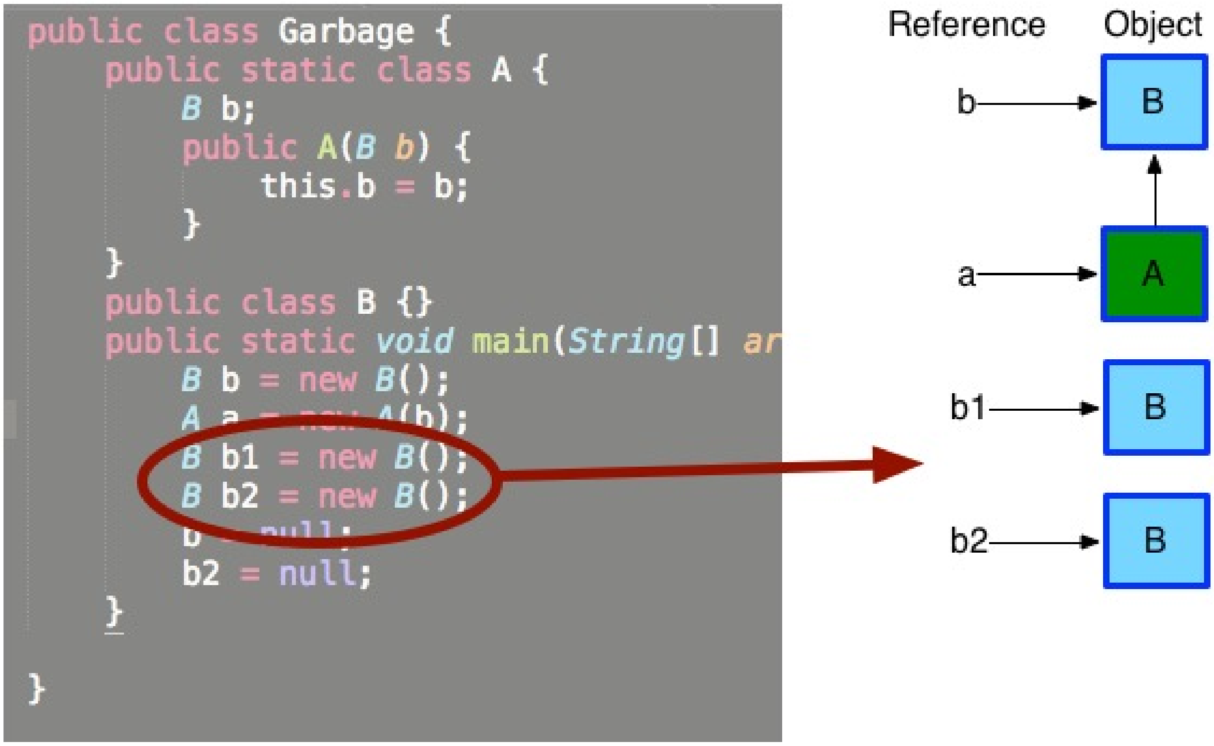

Figure 4: Two more B objects and references created

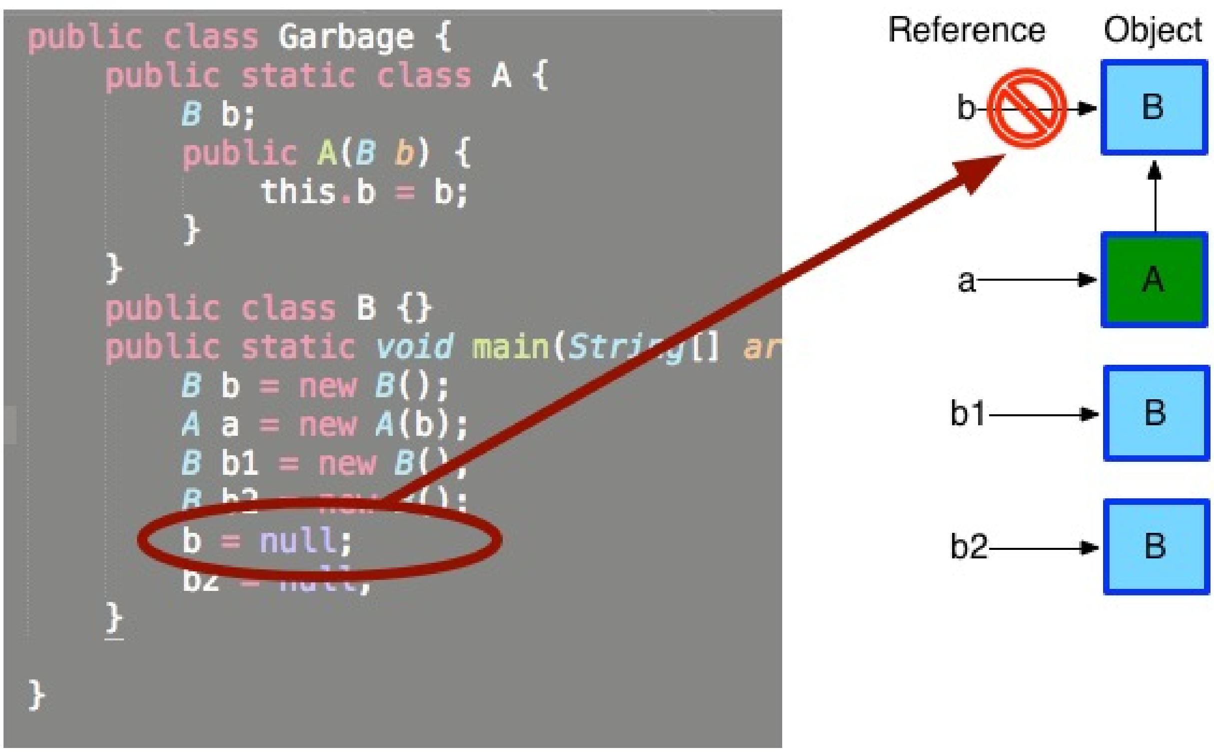

Figure 5: Reference b deleted

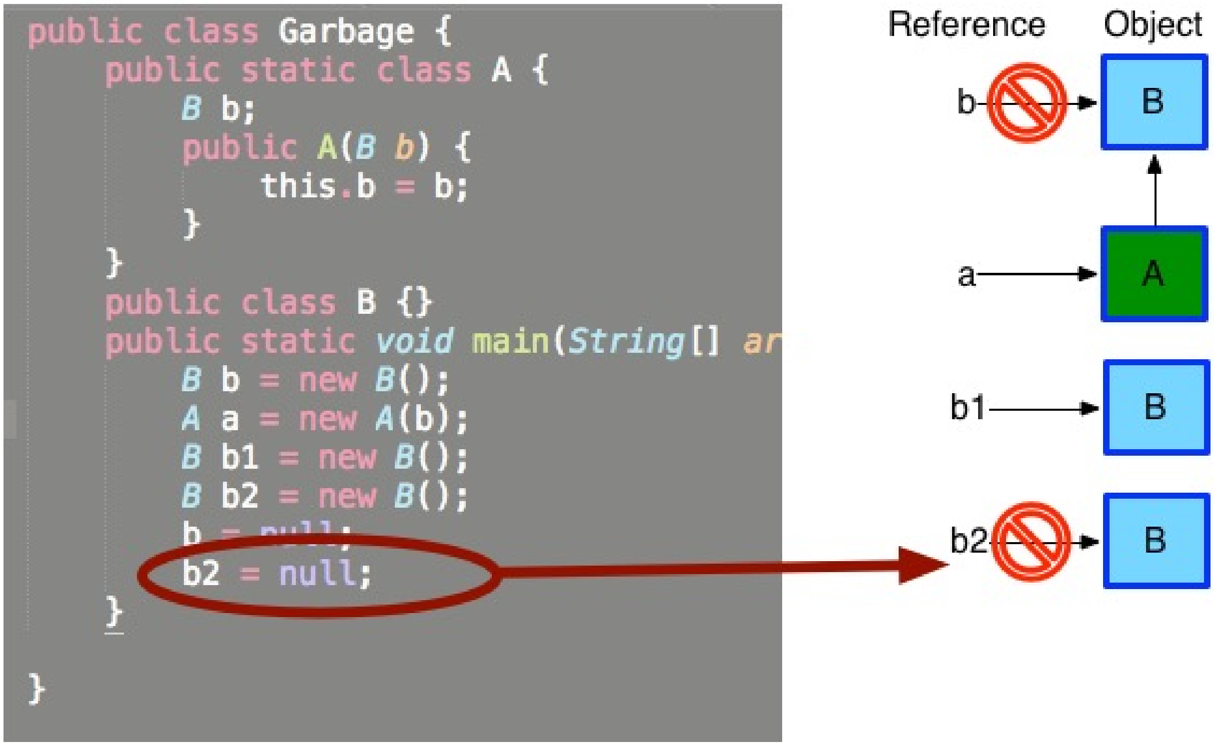

Figure 6: Reference b2 deleted

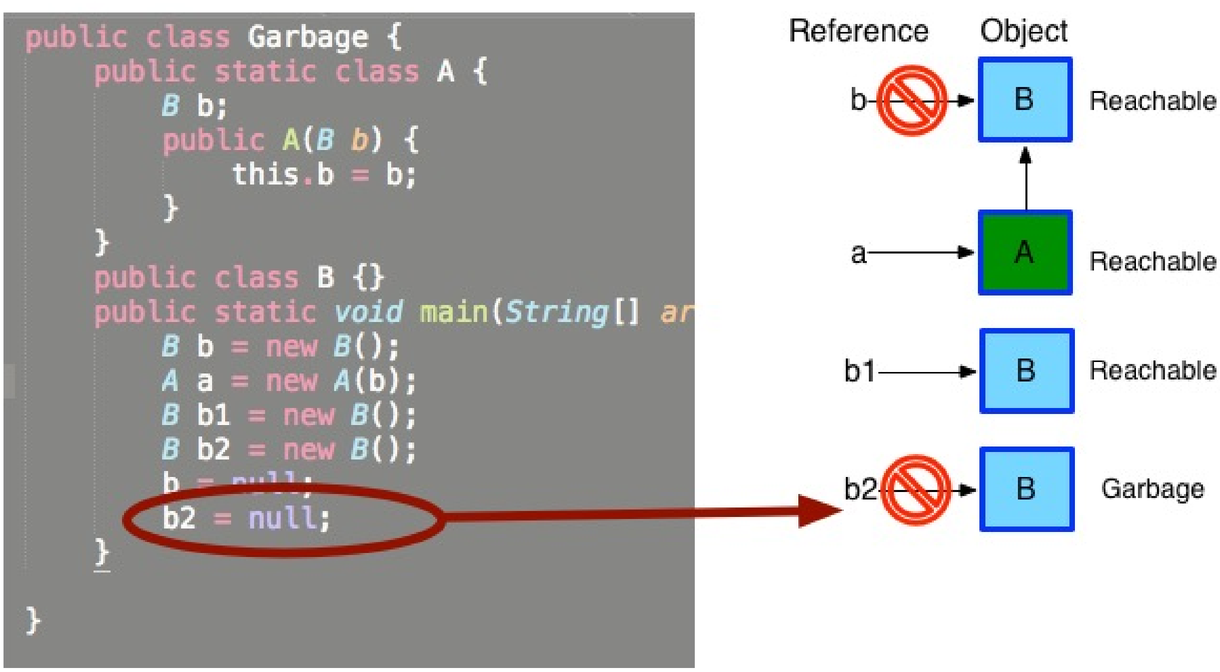

Figure 7: Objects with no reference are garbage. Note that the first B is still referenced from A so isn't garbage even though its original reference has been deleted

Immutable objects

- Some native Java objects are *immutable*
- Once they are created they cannot be changed
  - e.g. String, Double, Float, Integer, etc
- It looks like we can change them?

```
String a = "hello";
a+=" simon";
```

- But, Java is creating a new object and storing the reference in `a`
  - Objects in heap, references in stack…
- See `StringExample`

Call by value and call by reference

- Call by value
  - Value of a variable is passed to a method
  - Changes to the local copy are not reflected in the calling space
- Call by reference (e.g. C++)
  - Object references are passed to method
  - Actual object can be modified

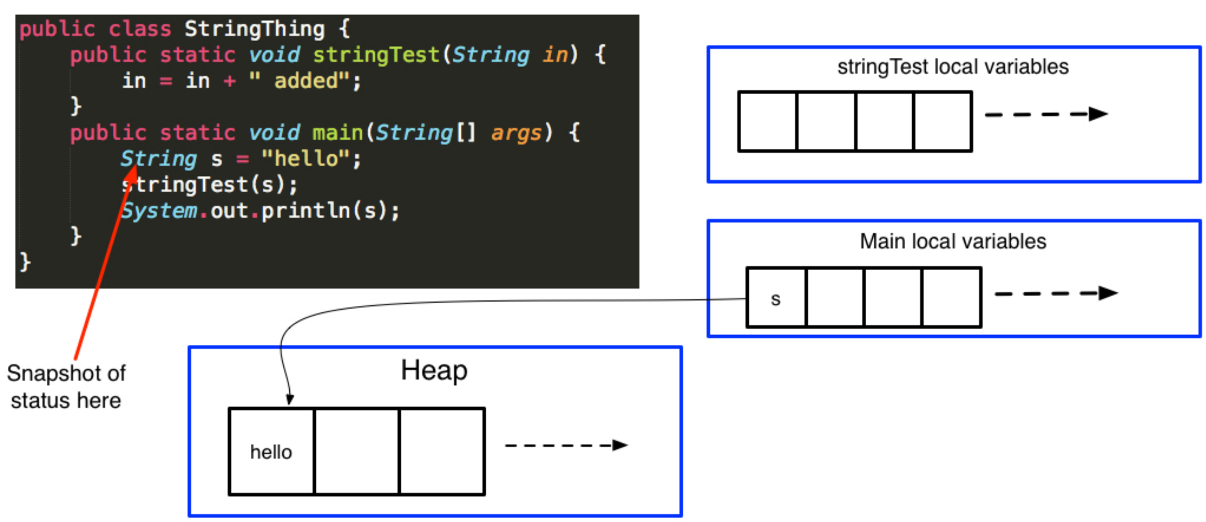

Figure 8:

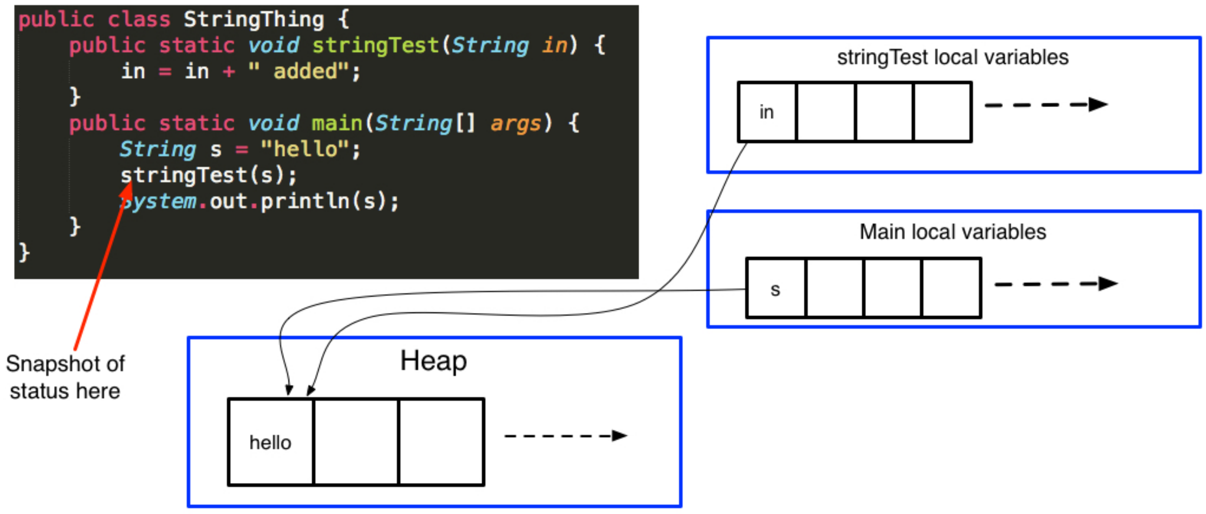

Figure 9:

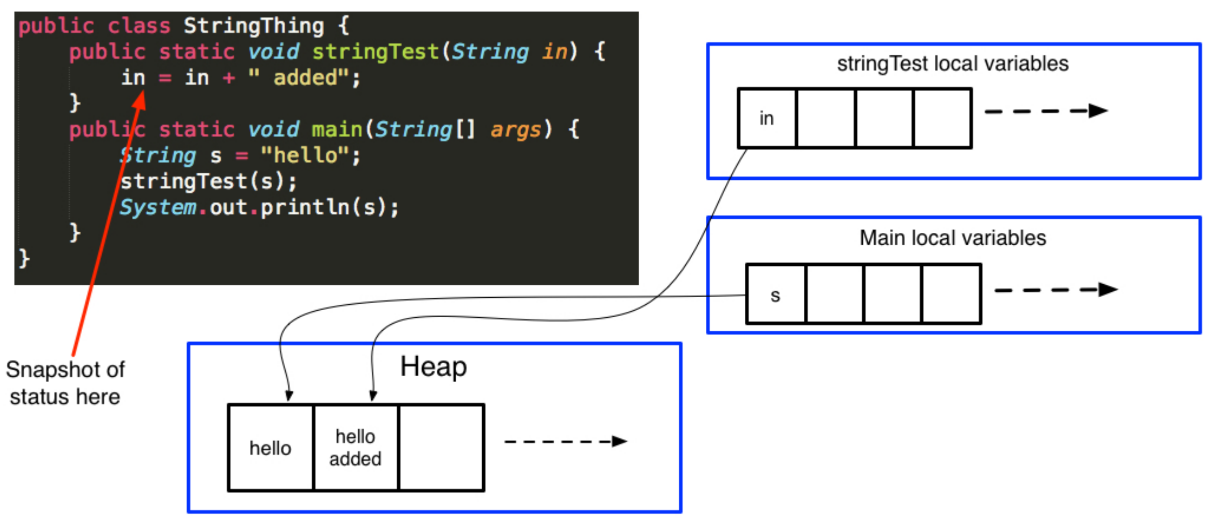

Figure 10:

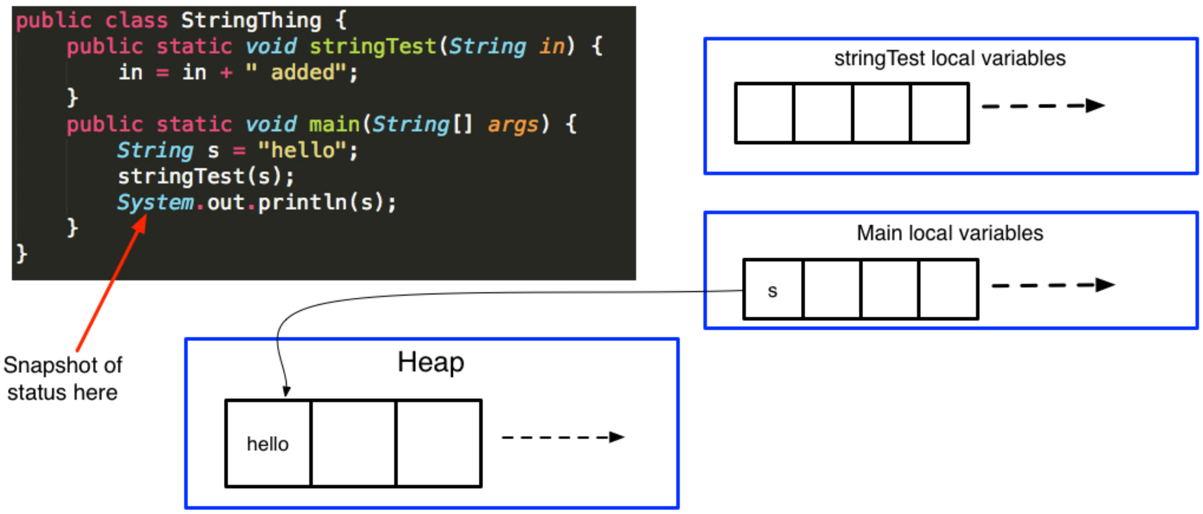

Figure 11:

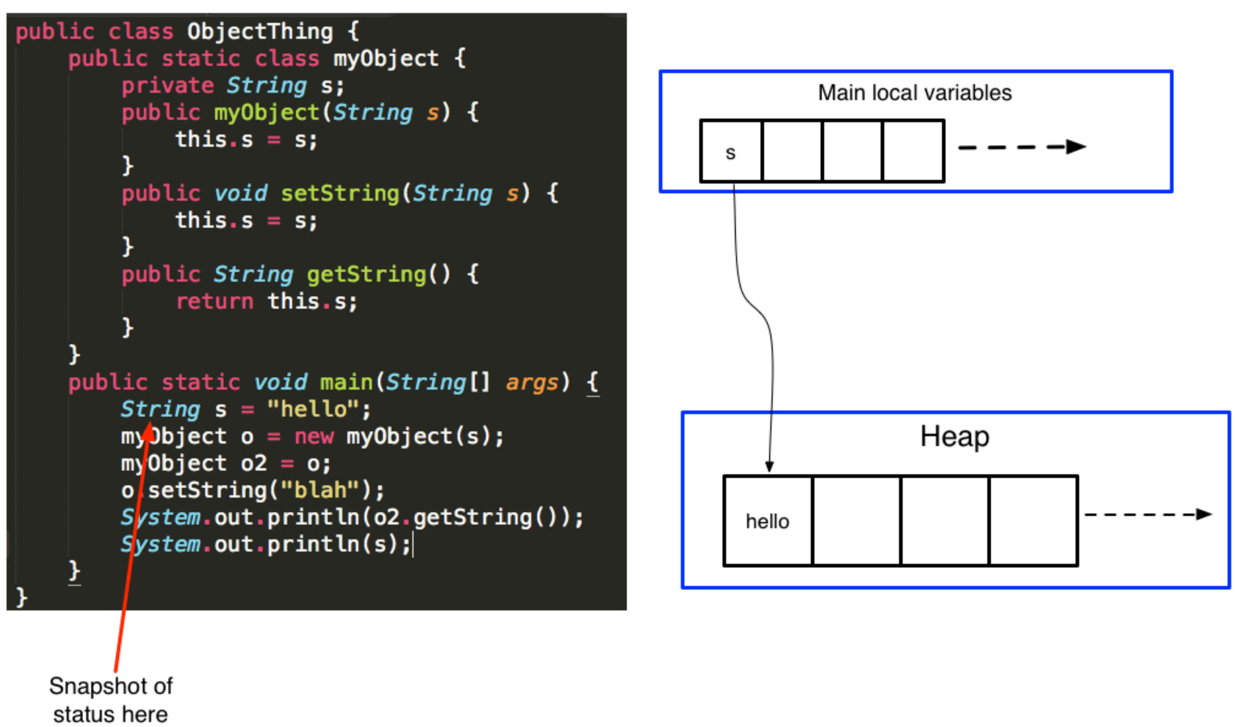

Figure 12:

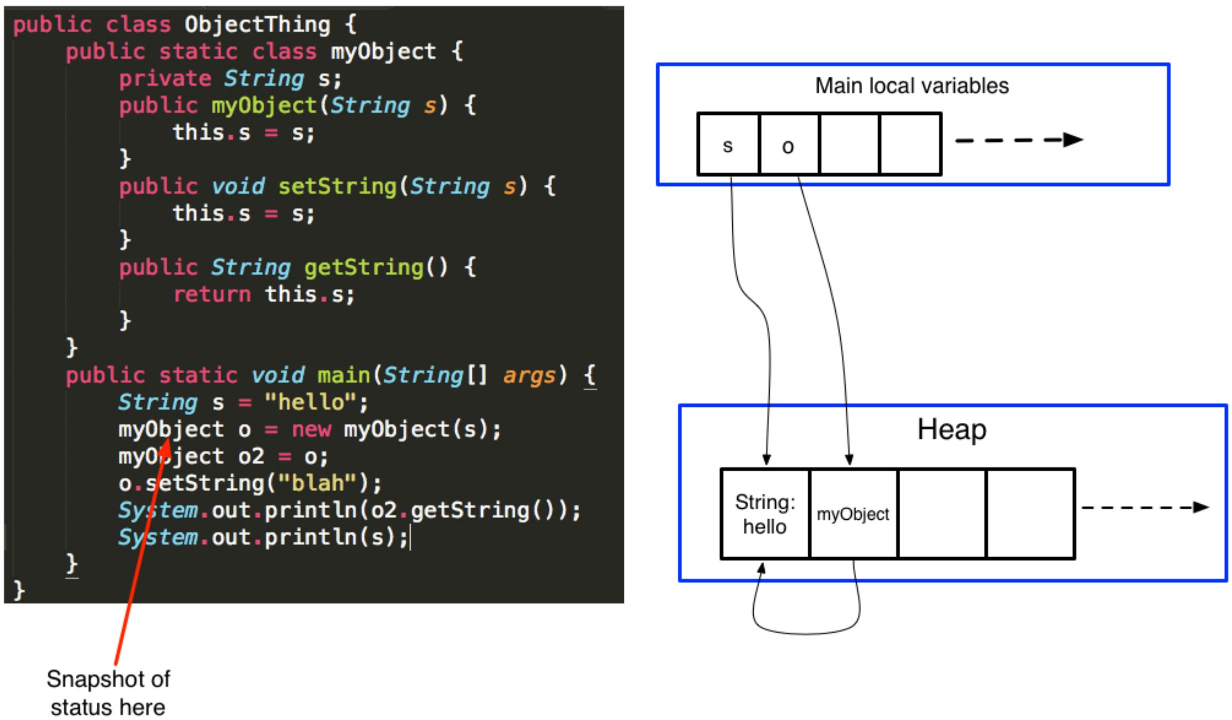

Figure 13:

- In Java, numbers and object references are call by value. Note that there is a difference between:
  - Objects are passed by reference
  - Object references are passed by value
- Objects passed to a method can be modified, but creating new ones will not be reflected in the calling scope (the reference cannot change)
  - `CallExamples`
- Objects are stored in the heap, references to objects are stored in the stack

```
public class CallExamples {
    public static class MyClass {
        int a = 0;
        public MyClass(int a) {
            this.a = a;
        }
        public int getValue() {
            return a;
        }
    }
    public static class DoubleClass {
        Double a = 0.0;
        public DoubleClass(Double a) {
            this.a = a;
        }
```

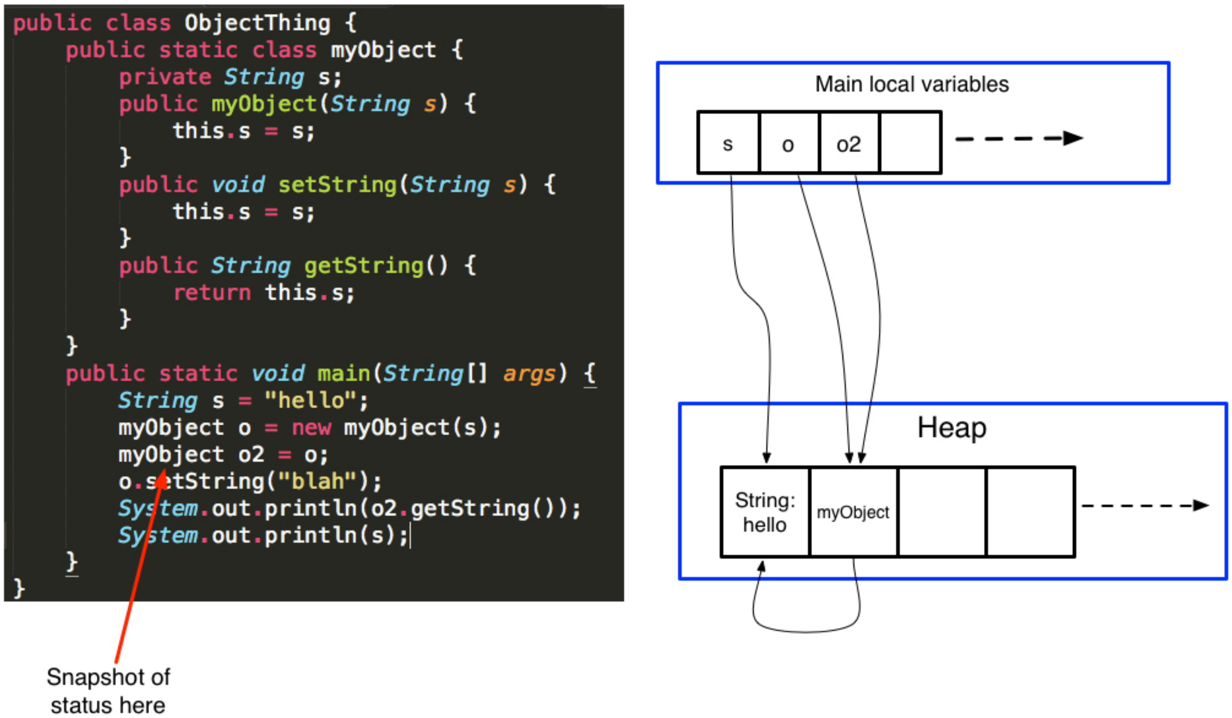

Figure 14:

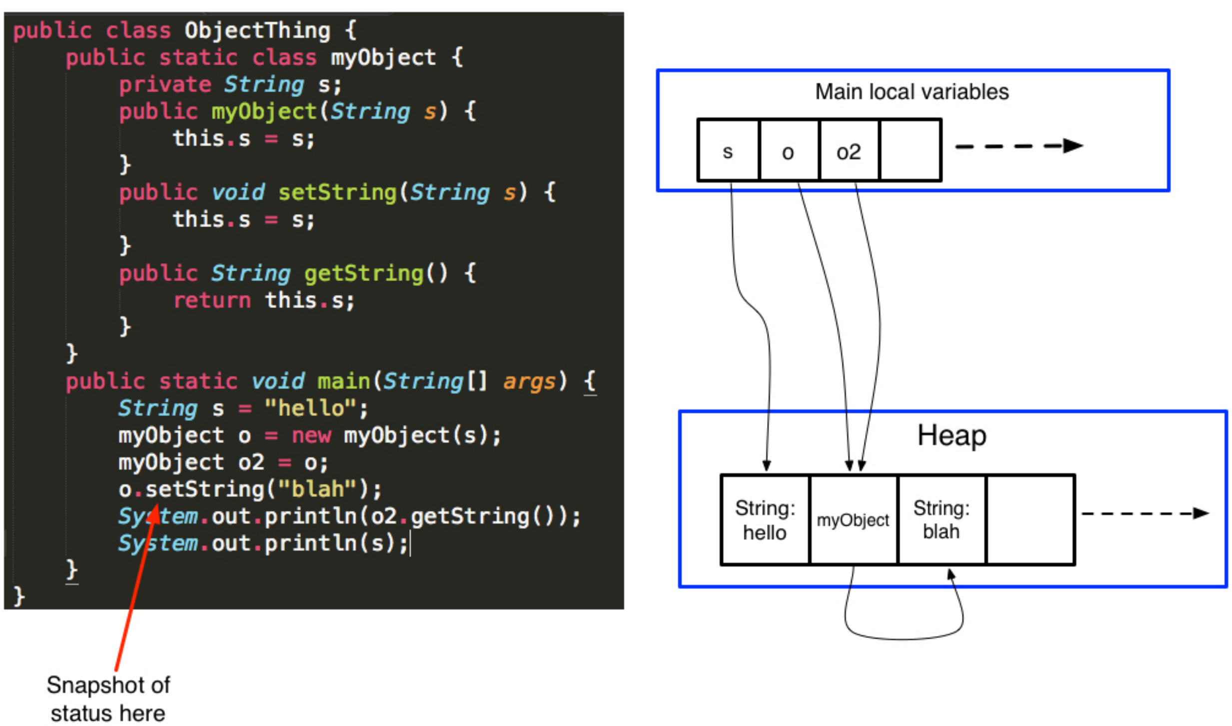

Figure 15:

```java
        public void multiply(Double m) {
            a = a * m;
        }
        public Double getValue() {
            return a;
        }
    }
    private static void aTest(MyClass in) {
        in = new MyClass(5);
    }

    private static void stringTest(String in) {
        in = in + " added";
        System.out.println(in);
    }

    private static void doubleTest(Double in) {
        in = in * 2;
    }
    private static void doubleObjectTest(DoubleClass in) {
        in.multiply(2.0);
    }

    public static void main(String[] args) {
        MyClass m = new MyClass(3);
        aTest(m);
        System.out.println(m.getValue());
        // What value will be displayed?


        String s = "hello";
        stringTest(s);
        System.out.println(s);
        // What will s be?

        Double d = 3.2;
        doubleTest(d);
        System.out.println(d);
        // What will d be?

        DoubleClass d2 = new DoubleClass(3.2);
        doubleObjectTest(d2);
        System.out.println(d2.getValue());
        // What will the value be?
    }
}
```

In the first example, `main` creates an object of type `MyClass` (with value 3). The `aTest()` method is passed the value of the reference to the object (this sentence is important - make sure you understand it!). Inside the `aTest()` method it creates a new object and stores the reference in `in`. So, why don't we see the new object in `main`? It is because Java uses call by value for references and therefore changes to the value are not reflected in the calling scope. When `new` is invoked, the value of `in` changes, but this is the local `in`.

In the second example, we pass the string `"Hello"` to `stringTest`. `stringTest` appends `added` to the String. Why isn't this change seen in `main`? This time it is because String is an immutable type. When it looks like we're changing a String we're actually making a new object. The value of new object reference is stored in the local `in` variable and so isn't seen in `main`.

In the third example, we see the same behaviour, but this time with the immutable Double object instead of a String.

In the final example, we do see the value change reflected in `main`. This is because our `DoubleClass` is mutable. When we pass a `DoubleClass` object around, it's attributes can be changed as long as we never change the value of the reference (by, say, making a new object).

If you're struggling with this, keep in mind what a reference to an object is. Perhaps the best way to think of it is as the address of the bit of memory in which the object is stored. E.g. if you think of memory as a set of pigeon holes, then the reference stores which pigeon hole it is in. When we pass an object reference to a method, it creates its own local copy of the reference and therefore knows where the object is stored, and can change it (assuming it permits changes). When a new object is created, it is put in a new pigeon hole and the local reference is changed (but not the original). So, in the first example above, the object referenced by `m` is left unchanged when `aTest()` creates a new object. The same thing is happening in examples 2 and 3, even though there is no `new` statement - because String and Double are immutable, new objects are made when we try and change them (see Figures below). In the final example, the reference never changes so the method can make changes to the original object.

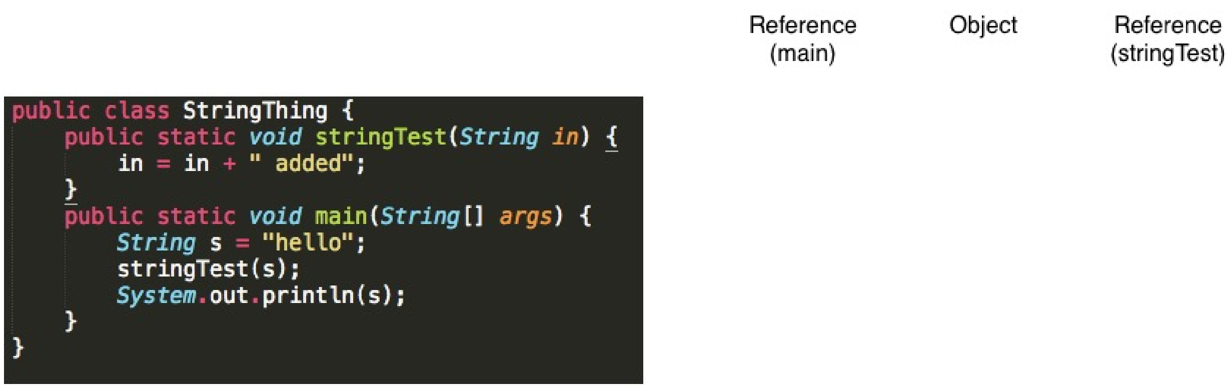

Figure 16: Example program

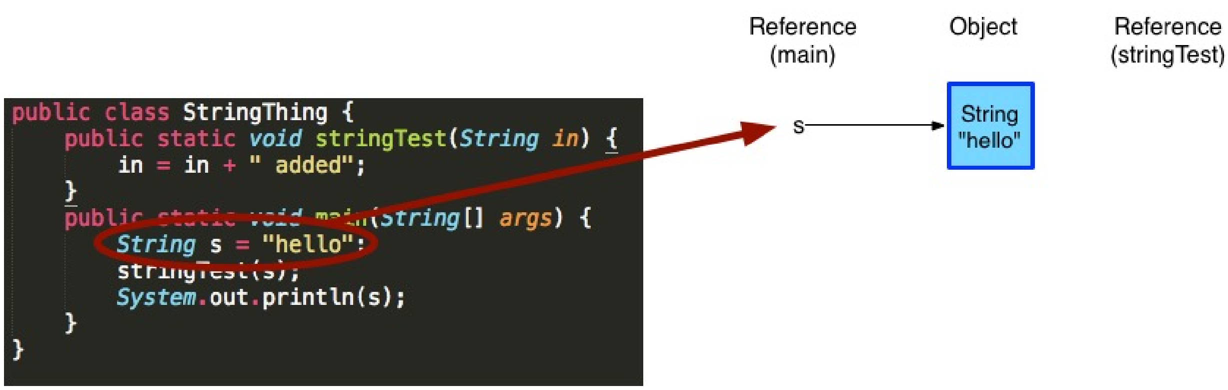

Figure 17: Main makes a String object and a reference (s)

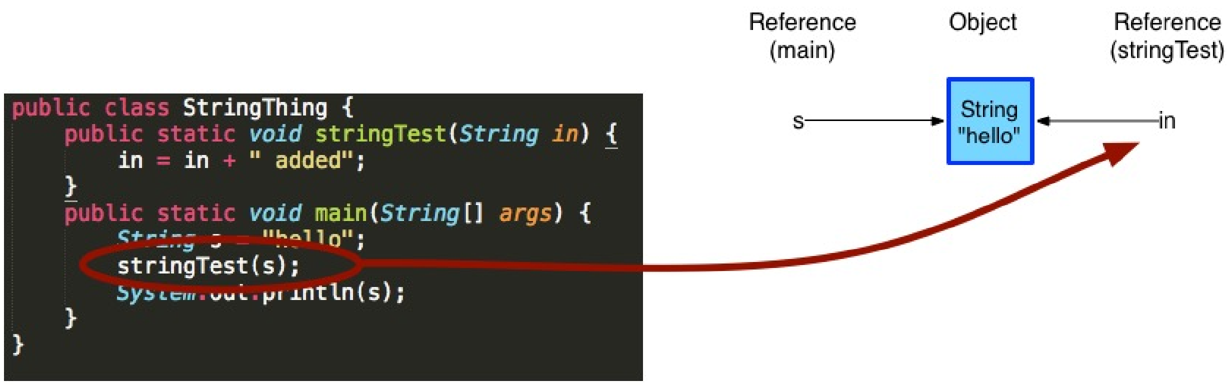

Figure 18: stringTest makes its own reference to the String object (in)

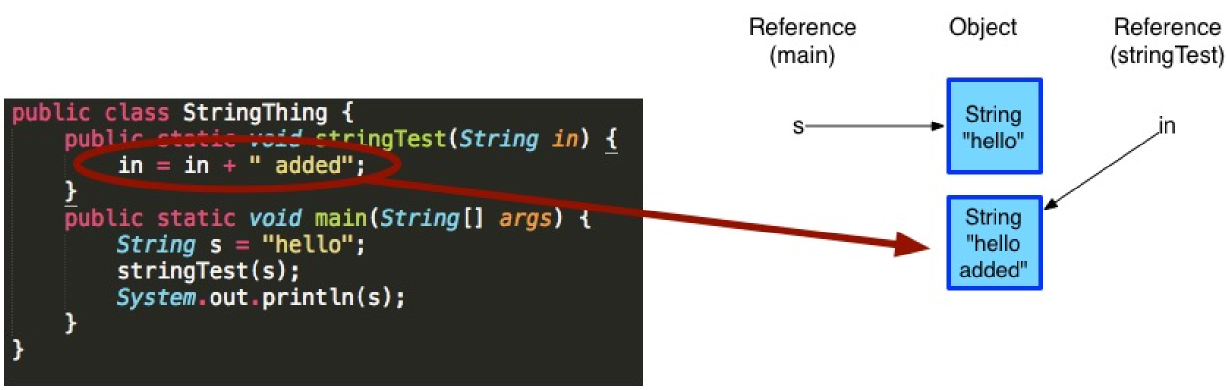

Figure 19: String is an immutable type so when we change it, a new String is made

Mutable objects

- In `StringExample` the main method created a new String object `s+=" simon"`
- The original one remained unchanged
  - This is because `String` is *immutable*
- What about a mutable object?
- `MutableNastiness`
- Returning mutable objects is bad practice
- `MutableNastinessFixed` fixes it by returning a new object

```
public class MutableNastiness {
    public static class MyDouble {
        private Double a;
        public MyDouble(Double a) {
            this.a = a;
```

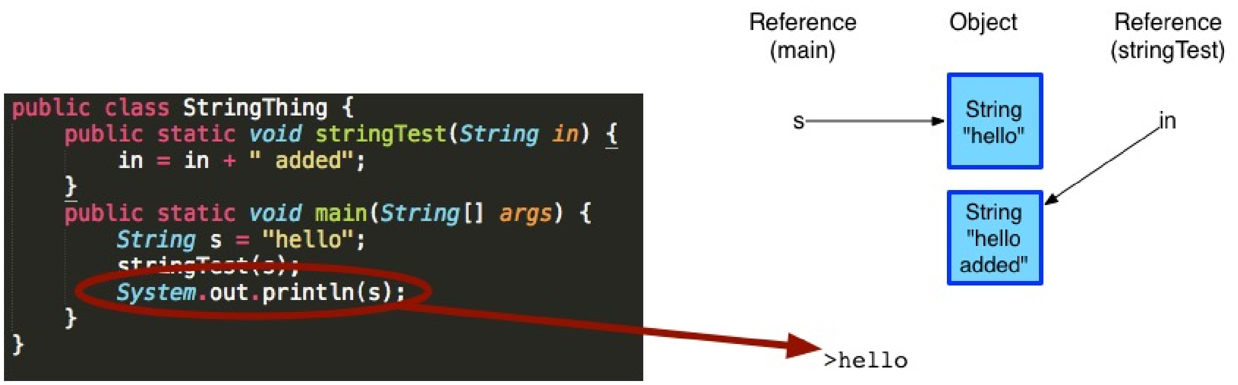

Figure 20: Back in main, s is still a reference to the original object. What happens to the “hello added” string when we return to main?

```
        }
        public void multiply(Double m) {
            a = a * m;
        }
        public Double getValue() {
            return a;
        }
    }
    public static class DoubleWrapper {
        private MyDouble d;
        public DoubleWrapper(Double in) {
            this.d = new MyDouble(in);
        }
        public MyDouble getMyDouble() {
            return d;
        }
        public void multiply(Double m) {
            d.multiply(m);
        }
        public Double getValue() {
            return d.getValue();
        }
    }

    public static void main(String[] args) {
        // Create a Double object
        DoubleWrapper dw = new DoubleWrapper(3.2);
        MyDouble d = dw.getMyDouble();
        System.out.println(d.getValue());
        d.multiply(2.0);
        System.out.println(d.getValue());
        System.out.println(dw.getValue());
    }
}
```

In this example, `DoubleWrapper` is used to hold a `MyDouble` object. When we call `getMyDouble()` it returns the reference to its own `MyDouble` object. So, when we subsequently change that object, we are changing the object within `DoubleWrapper`. This is not good practice as it can result in unpredictable behaviour - someone else changes an object you rely on. To avoid this, try to only ever return immutable objects or new objects. For example, in `MutableNastinessFixed` the `getMyDouble()` method is changed to create a new object. Now, changes to the object returned do not change the original:

```
public MyDouble getMyDouble() {
    return new MyDouble(getValue());
}
```

Final

- It is good practice to make as many things `final` as possible
- Make as many attributes `final` as possible
- Stops other people doing bad things to your code
    - `final` classes can not be sub-classed
    - `final` methods can not be overloaded
    - `final` variables cannot be modified once declared
- `final` is not the same as `immutable`
- `FinalTest` and `FinalTestFixed`

```
public class FinalTest {
    public static final class Person {
        private String name;
        private Integer age;
        public Person(String name,Integer age) {
            this.name = name;
            this.age = age;
        }
        public void setName(String name) {
            this.name = name;
        }
        public void setAge(Integer age) {
            this.age = age;
        }
        public String getName() {
            return name;
        }
        public Integer getAge() {
            return age;
        }
    }
    public static void main(String[] args) {
        Person p = new Person("Ella",1);
        p.setAge(2);
        System.out.println(p.getName() + " is " + p.getAge());
    }
}
```

Here, `Person` is a final class. But this does not mean that its attributes are immutable. You can see this as `main` can invoke methods that change the values of the `name` and `age` attributes. Remember that a class being `final` just means that it cannot be sub-classed (inherited). To make immutable classes, all attributes must be immutable (and no mutable objects must be returned). In this case, we change the attribute declarations to:

```
private final String name;
private final Integer age;
```

With this modification, the code will not even compile.

Some useful Java objects

ArrayList

- Java arrays are of fixed length
- `ArrayList` gives you an object that can handle arrays of any object that change length

```
ArrayList<Integer> a = new ArrayList();
a.add(3);
a.add(5);
System.out.println(a.contains(4)); // Checks is 4 is in a
```

Be careful of arraylists of integers. Various arraylist methods are overloaded to take an object or an integer as the argument (e.g. remove()). This can get very confusing!

HashSet

- Useful way of keeping a set of objects together (not ordered)

```
HashSet<String> h = new HashSet<String>();
h.add("hello");
h.add("simon");
h.add("hello"); // Wont add as already in there
h.contains("hello"); // returns true
h.remove("simon"); // removes this one
```

- Very fast for checking if an item is in the set

HashMap

- Useful way of storing key,value pairs

```
HashMap<String,Double> h = new HashMap<String,Double>();
h.put("banana",3.0);
h.put("apple",2.0);
System.out.println(h.get("apple")); //print 2
h.keySet(); // Returns a set of the keys
```

- Very fast for obtaining items for a particular key

Generics

ArrayList

- What is the `<Double>` for in `ArrayList`?
- It is a `generic`
- i.e. `ArrayList` can work with any type (specified when you create it)
- You can make classes with generics too…

Creating generic objects

```java
public class MyClass<T> {
    private T t;
    public MyClass(T t) {
        this.t = t;
    }
}
```

- In the code above `T` can be any class
- Can also have multiple types in the definition (`<A,B,C,D>`)
- See `Dictionary.java`

```java
import java.util.ArrayList;
// A hacky alternative to HashMaps to demonstrate
// making a class with generics
public class Dictionary<A,B> {
    private ArrayList<A> listA = new ArrayList<A>();
    private ArrayList<B> listB = new ArrayList<B>();
    public void add(A a,B b) {
        listA.add(a);
        listB.add(b);
    }
    public B getDefinition(A a) {
        int index = listA.indexOf(a);
        return listB.get(index);
    }
}
```

```java
public class DictionaryTest {
    public static void main(String[] args) {
        Dictionary<String,Double> d = new Dictionary<String,Double>();
        d.add("apple",3.0);
        d.add("banana",2.5);
        System.out.println(d.getDefinition("banana"));

        // Can also make the reverse!
        Dictionary<Double,String> d2 = new Dictionary<Double,String>();
        d2.add(3.0,"apple");
        d2.add(2.5,"banana");
        System.out.println(d2.getDefinition(3.0));

    }
}
```

Testing

Debugging

- In semester 1 you learnt to use the Eclipse debugger
- There is more to testing than debugging
- In real development projects, many people are wholly devoted to testing
- Black box, white box
- Unit testing…

Unit testing

- Testing individual components (e.g. classes, methods) to see if they are fit for use
- Design a suite of tests that can be run every time objects are changed
- Separates testing from the classes themselves

JUnit

- `JUnit` is a popular Java unit test framework
- A *test class* is created for each normal class
- We can then run `JUnit` and it will automatically perform the tests

`Pointless.java`

```java
public class Pointless {
    public int myInt;
    public Pointless(int n) {
        myInt = n;
    }
    public void increment() {
        myInt++;
    }
    public int getMyInt() {
        return myInt;
    }
}
```

`PointlessTest.java`

```java
import org.junit.Test;
import org.junit.Assert;
```

```java
public class PointlessTest {
    private static final double EPSILON = 1e-12;
    @Test public void testIncrement()
    {
        Pointless p = new Pointless(1);
        p.increment();
        int expected = 2;
        Assert.assertEquals(expected,p.getMyInt(),EPSILON);
    }
}
```

Compiling

- To compile `PointlessTest` we need `JUnit`
  - You can do this in eclipse
  - Or from the command line
- On a mac:

```
javac -cp .:../JUnit/junit-4.12.jar PointlessTest.java
```

- `-cp` sets the *class path*
  - In this case, '.' means current directory and `./JUnit/junit-4.12.jar` is where the `JUnit` `.jar` file is

---

Running

- Again, possible in Eclipse or from the command line
- From command line (mac):

```
java -cp .:../JUnit/junit-4.12.jar:../JUnit/hamcrest-core-1.3.jar PointlessTest
```

- Result:

```
JUnit version 4.12
.
Time: 0.004

OK (1 test)
```

`Pointless2.java`

- We now add a doubling function
- and write a new test case (`PointlessTest2.java`)
- What happens?
- Note: the compile commands start getting a bit tricky - this can be overcome by using a build system (not covered in this course)

Assertions

- `JUnit` testing is done at compile time
- We might also want runtime checks
    - to catch runtime errors (e.g. based on input that is unknown at compile time)
- The naive way is through the use of `if` statements

```java
public class AssertionExample {
    private int myInt;
    public AssertionExample(int n) {
        myInt = n;
    }
    public void decrement(int d) {
        if(d>myInt) {
            // Cannot decrement!
            System.out.println("Can't decrement!!");
        }else {
            myInt = myInt - d;
        }
    }
    public static void main(String[] args) {
        new AssertionExample(5).decrement(10);
    }
}
```

- Assertions are a neater way to achieve this
    - Cause the program to exit if the condition is not met
    - Can be switched on or off at runtime
        - e.g. switch between runtime and debugging

```java
public class AssertionExample2 {
    private int myInt;
    public AssertionExample2(int n) {
        myInt = n;
    }
    public void decrement(int d) {
        assert myInt >= d;
        myInt = myInt - d;
    }
    public static void main(String[] args) {
        new AssertionExample2(5).decrement(10);
    }
}
```

- Running:

```
java -enableassertions AssertionExample2
```

- can also use `-ea`
- Try running with and without
- An alternative is to explicitly throw exceptions but..
    - Takes longer to write

- Exceptions cannot be switched off at runtime (slows things down)

JavaDoc

- It's very important to properly document your code
- Standard comments `//` `/*` are good
- Javadoc is better!
- [This]{http://agile.csc.ncsu.edu/SEMaterials/tutorials/javadoc/} is quite a good tutorial
- See `MyMath` in `JavaDoc` directory
- Compile with `javadoc MyMath.java` and open `index.html`

Things we are not covering here

Testing

- We have only touched upon testing. It's very important! Those of you doing SE will cover it more there.
- Much software engineering is now done in a *test driven* manner.
  - First write test cases and then write code.
  - Stop coding when the test cases are finished.
  - Writing a good set of test cases is hard!

Data structures

- We make use of Java objects (e.g. `ArrayList`) but we don't worry about how Java implements this
- We also don't worry too much about the efficiency of different data structures and algorithms
- Those of you in ADS will do lots of this

Build systems

- Compiling from the command line is fine for simple projects
- But..when you have a more complex project with lots of *dependencies* things get very complex
- Systems exist to help you with this
- Examples:
  - Maven (the current standard for Java)
  - ANT (older but still popular)
- See Tim Storer's ANT guide on Moodle

Software Engineering

- Programming is only a small part of building software
- Engineering large software projects is hard (evidenced by the number of times they end badly)

- Youll get lots of SE in, erm, SE