APIT - Java recap etc

Dr. Simon Rogers

14/11/2014

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, Python
- 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?
- Useful link

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 cannot be - they 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?

`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

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

Evaluating expressions

- Consider the expression $2 + 3 * 4$ (i.e. $2 + (3 * 4)$)
- It can be represented by a *syntax tree*

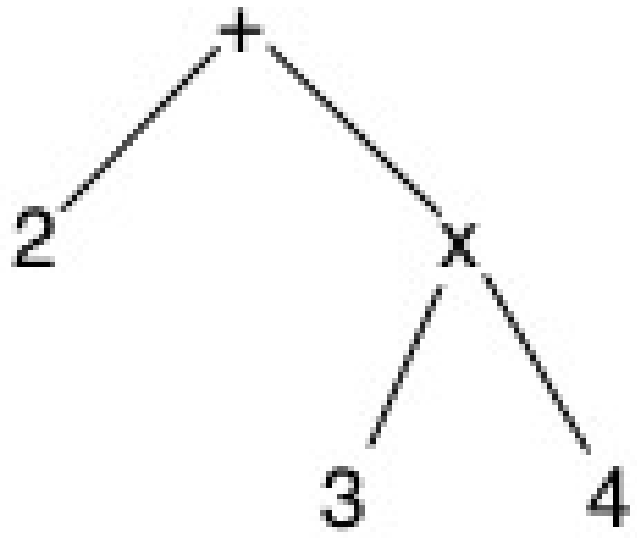


Figure 1:

- Traversing depth-first, left to right we get: `2 3 4 x +`
- This can be evaluated via the stack...

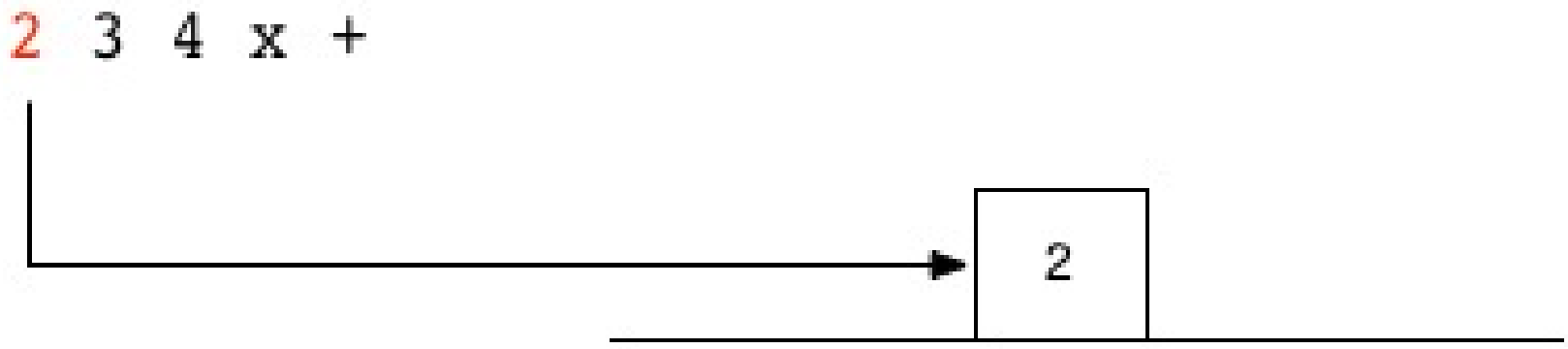


Figure 2: 2 is added to the stack

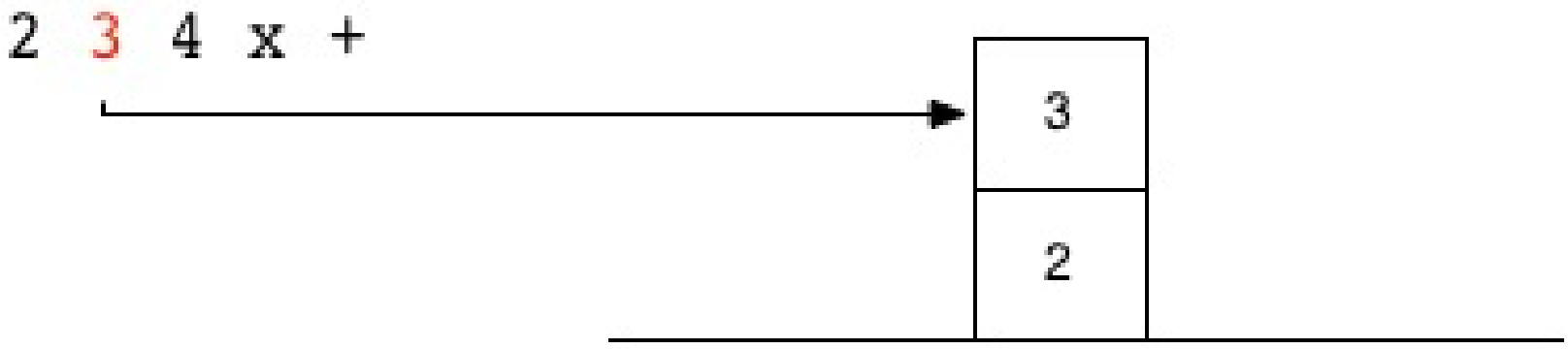


Figure 3: 3 is added to the stack

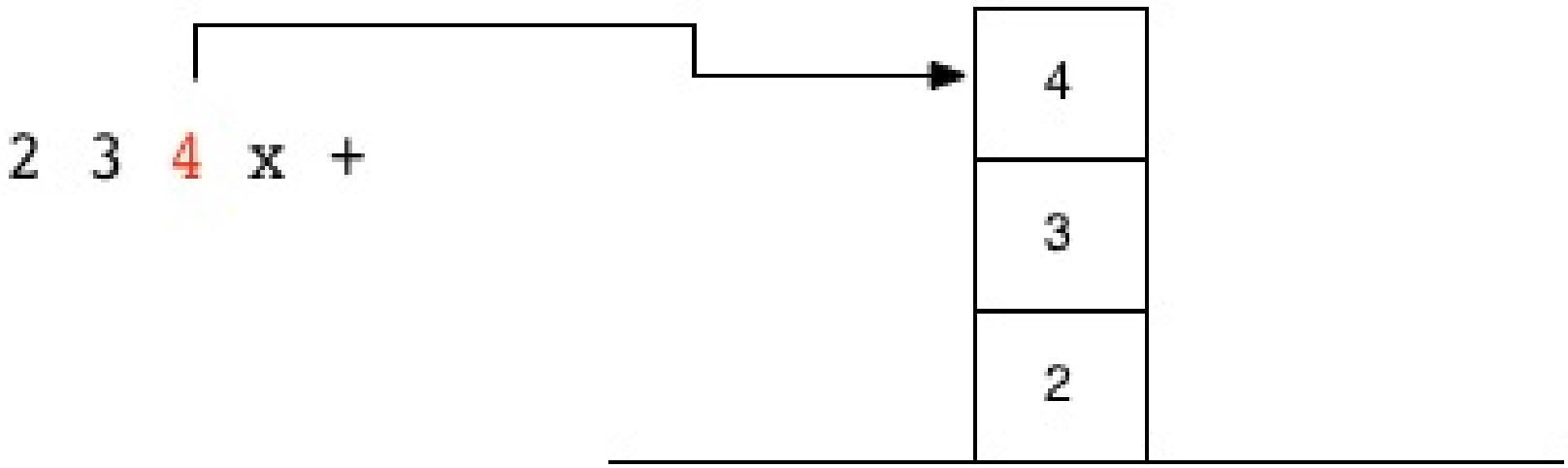


Figure 4: 4 is added to the stack

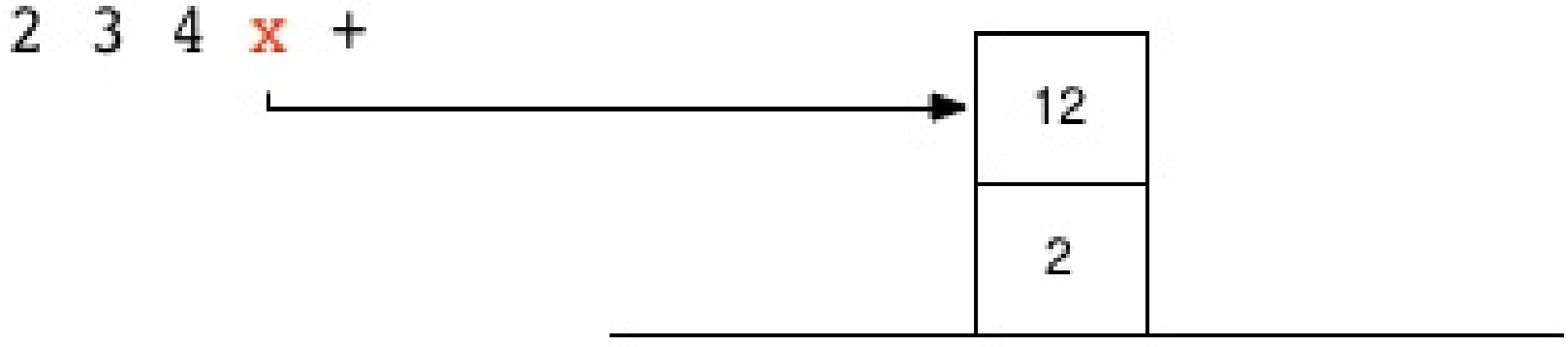


Figure 5: `x` operator pops out top element, multiplies it by new top element

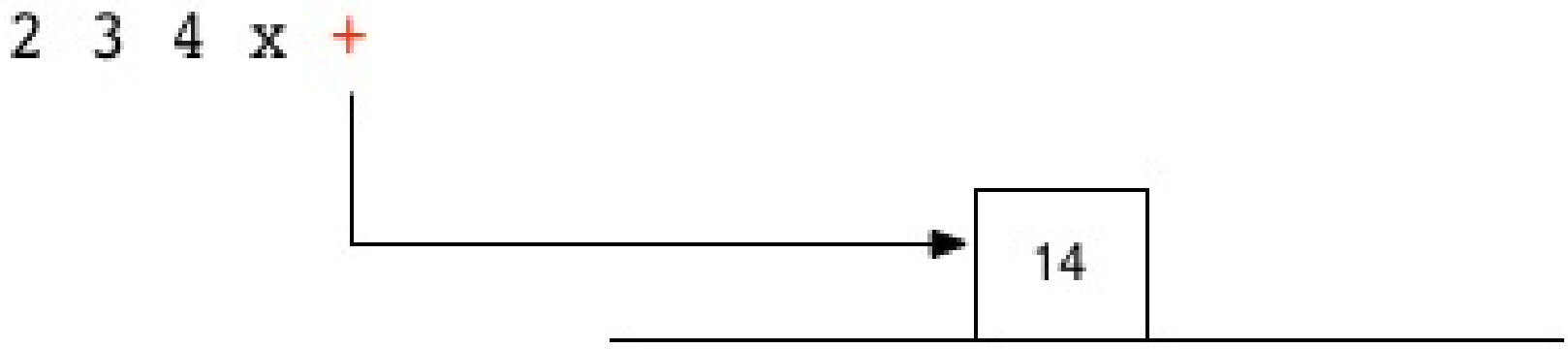


Figure 6: `x` operator pops out top element, adds it to new top element

Aside: Reverse Polish Notation

- This calculation was performed in Reverse Polish Notation
- Operators appear immediately after operands e.g.
  - `32-5+` = `(3-2) + 5`
  - `45*2-` = `(4x5) - 2`
  - `234++` = `(2+3) + 4`
  - `512+4x+3-` = ?
- See this

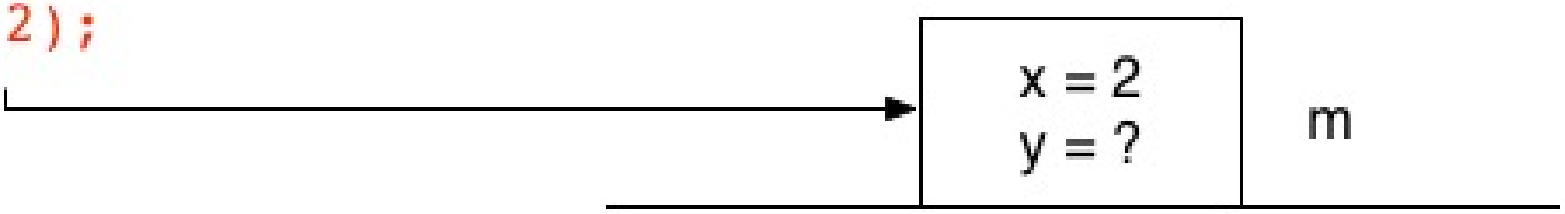
Stacks and methods

- When a method is called, a *stack frame* is created
  - An area at the top of the stack with space for the method to store local variables

```
int m(int x) {
    int y = n(x+1);
    return y;
}

int n(int x) {
    return x+1;
}

m(2);
```



```
int m(int x) {
    int y = n(x+1);
    return y;
}

int n(int x) {
    return x+1;
}

m(2);
```

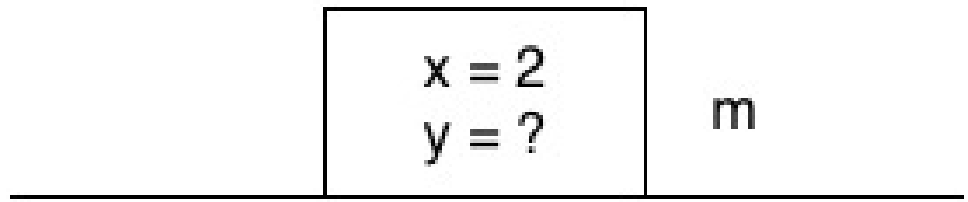


Figure 8:

```
int m(int x) {
    int y = n(x+1);
    return y;
}

int n(int x) {
    return x+1;
}

m(2);
```

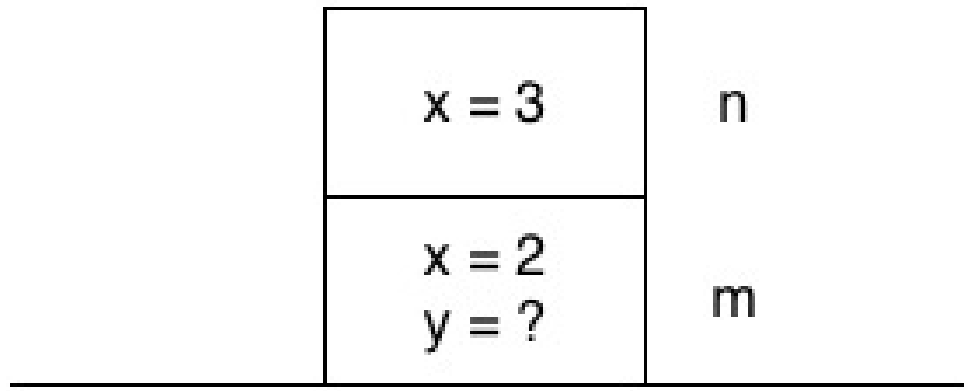


Figure 9:

```
int m(int x) {
    int y = n(x+1);
    return y;
}

int n(int x) {
    return x+1;
}

m(2);
```

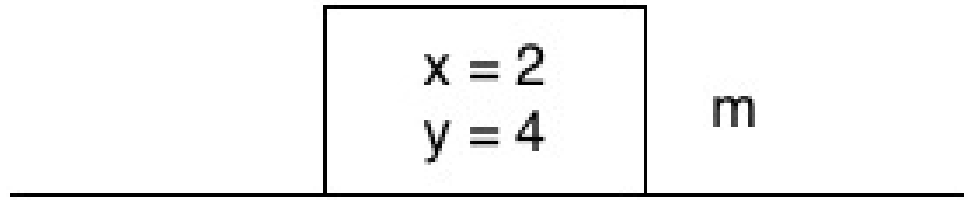


Figure 10:

```
int m(int x) {
    int y = n(x+1);
    return y;
}

int n(int x) {
    return x+1;
}

m(2);
```

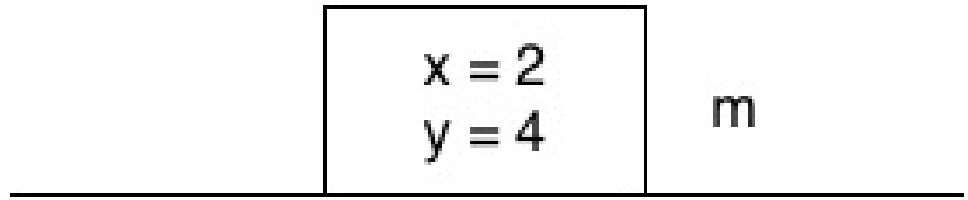


Figure 11:

```
int m(int x) {
    int y = n(x+1);
    return y;
}

int n(int x) {
    return x+1;
}

m(2);
```

4

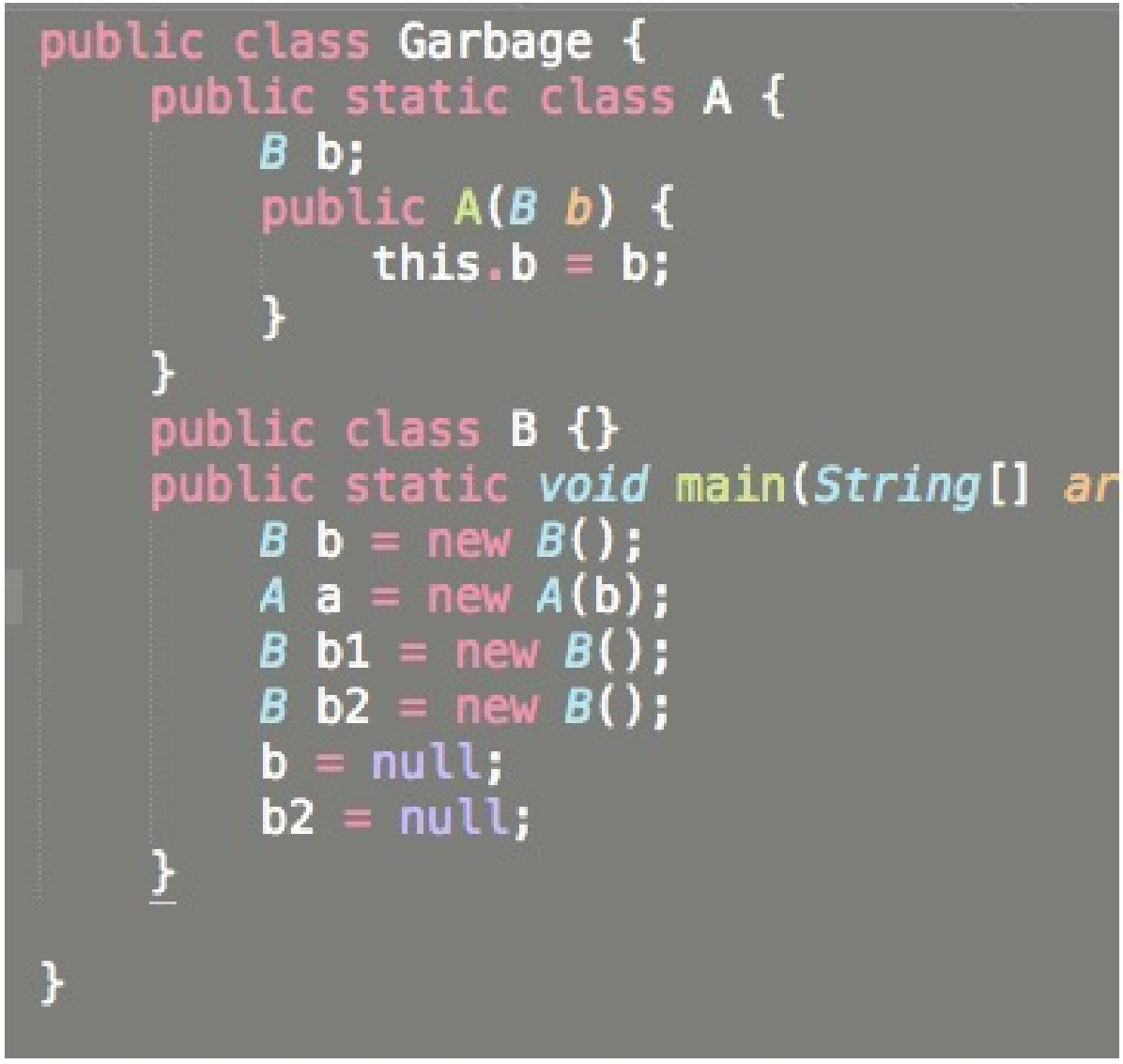
Figure 12:

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'll 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



```
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 13: Example program

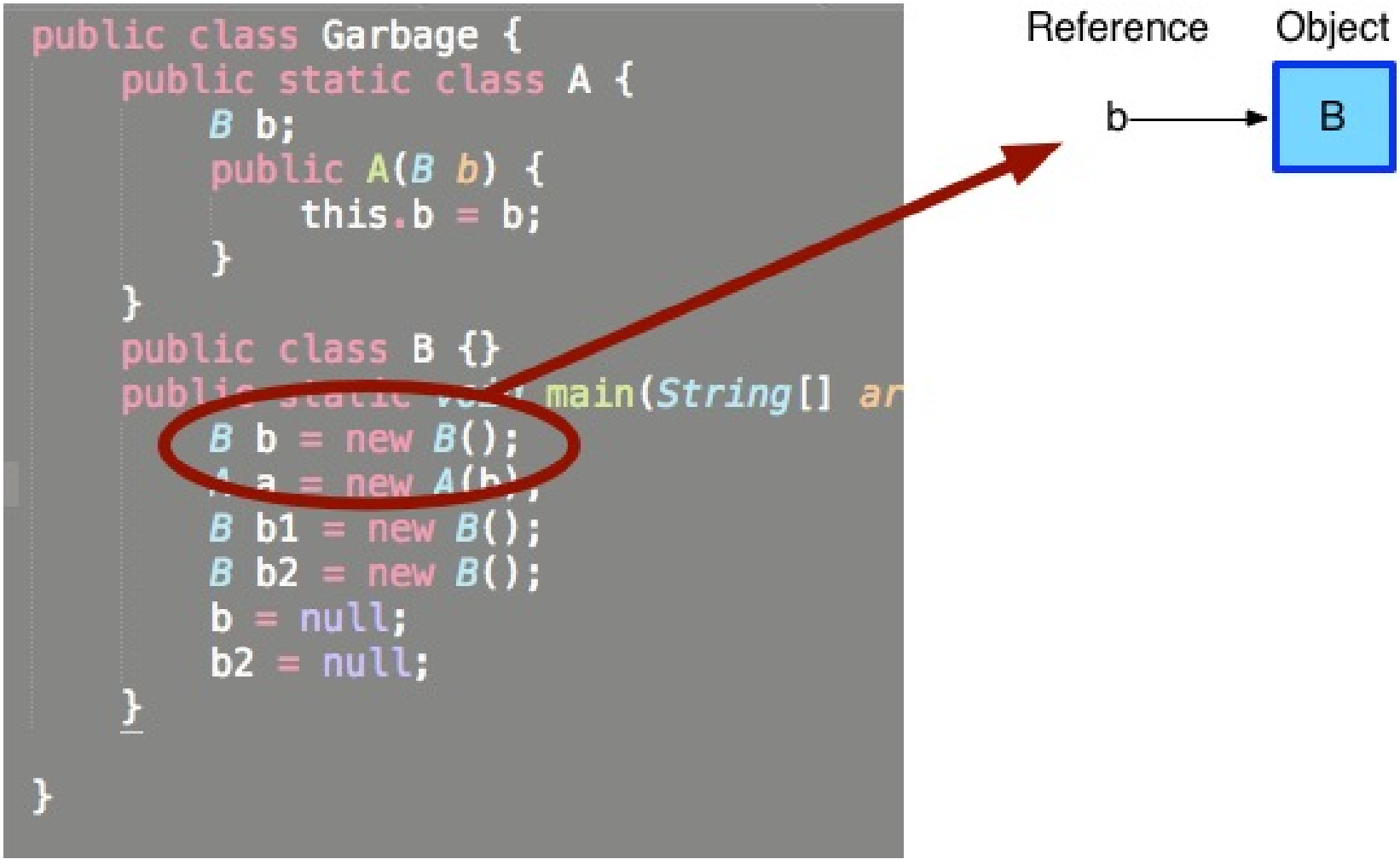


Figure 14: Object (B) and reference (b) created

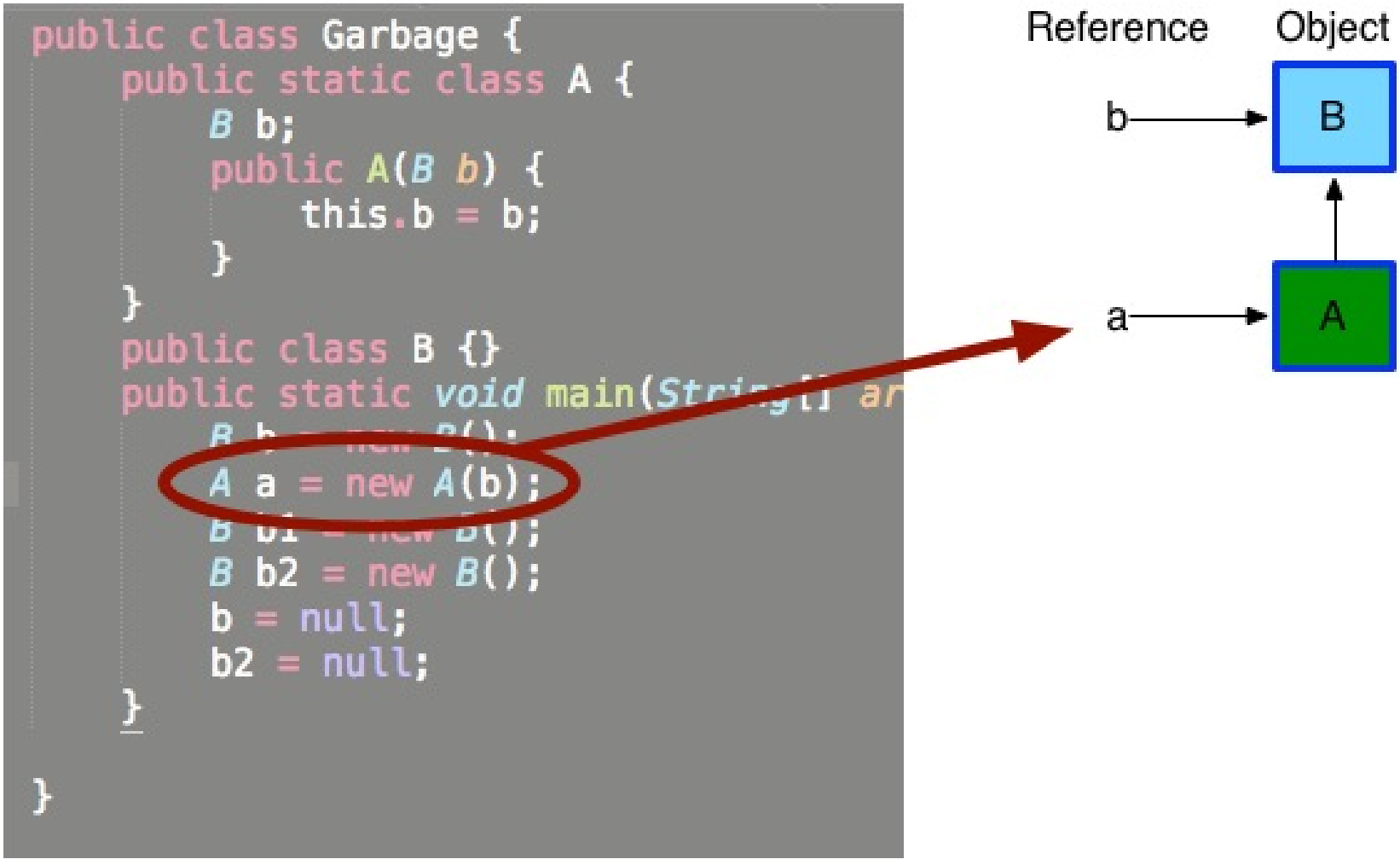


Figure 15: object (A) and reference (a) created. Note that A includes a reference to B)

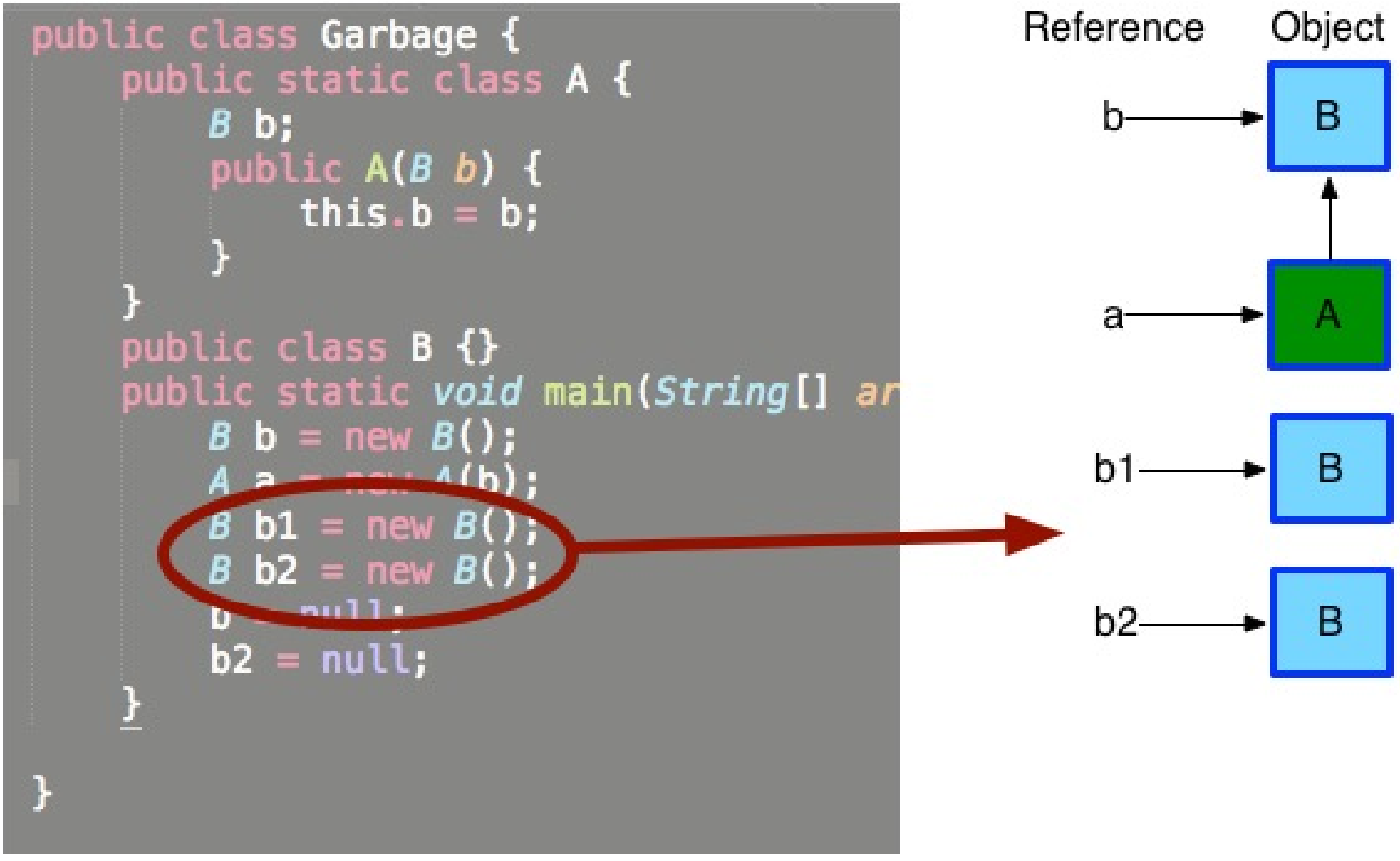


Figure 16: Two more B objects and references created

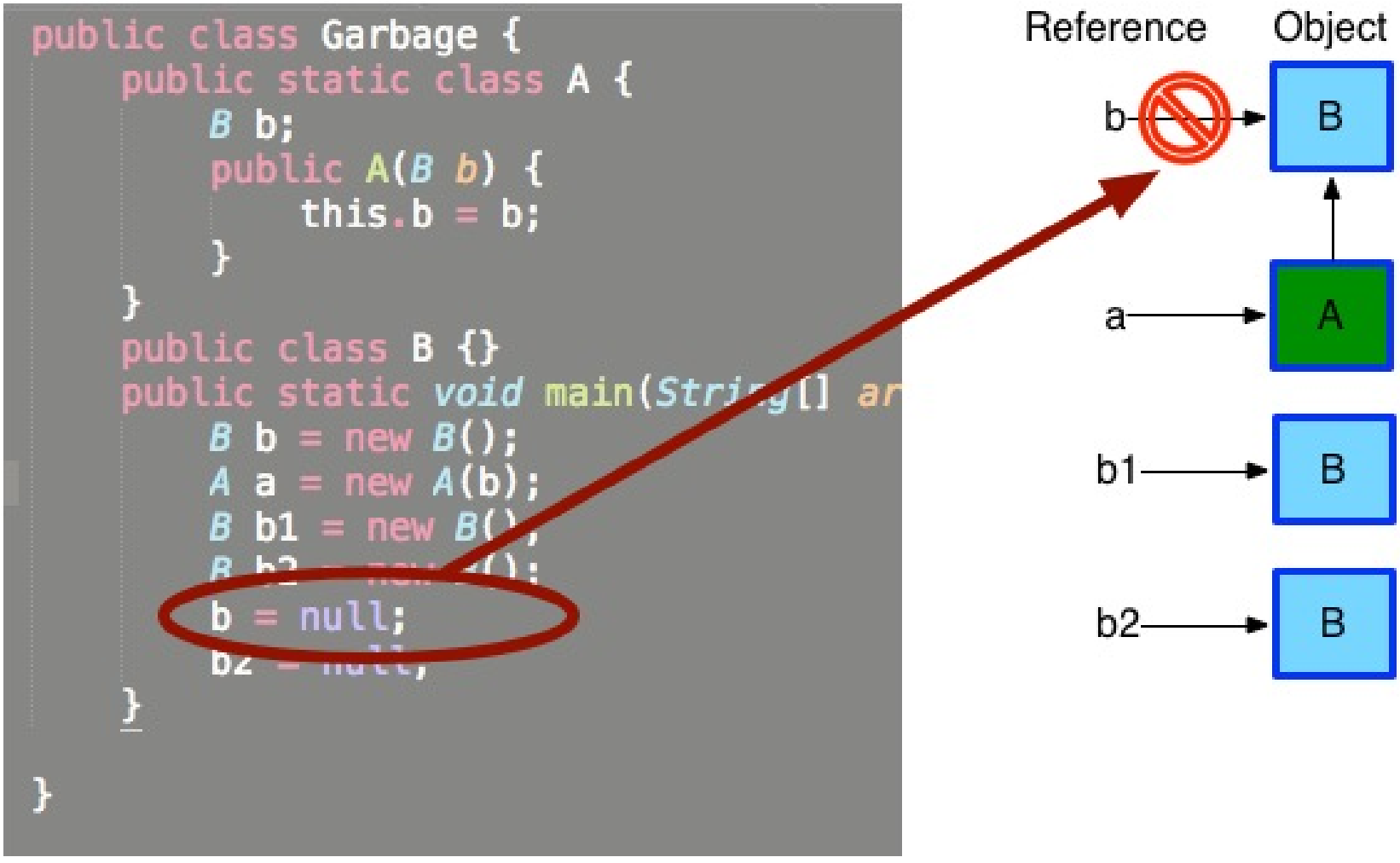


Figure 17: Reference b deleted

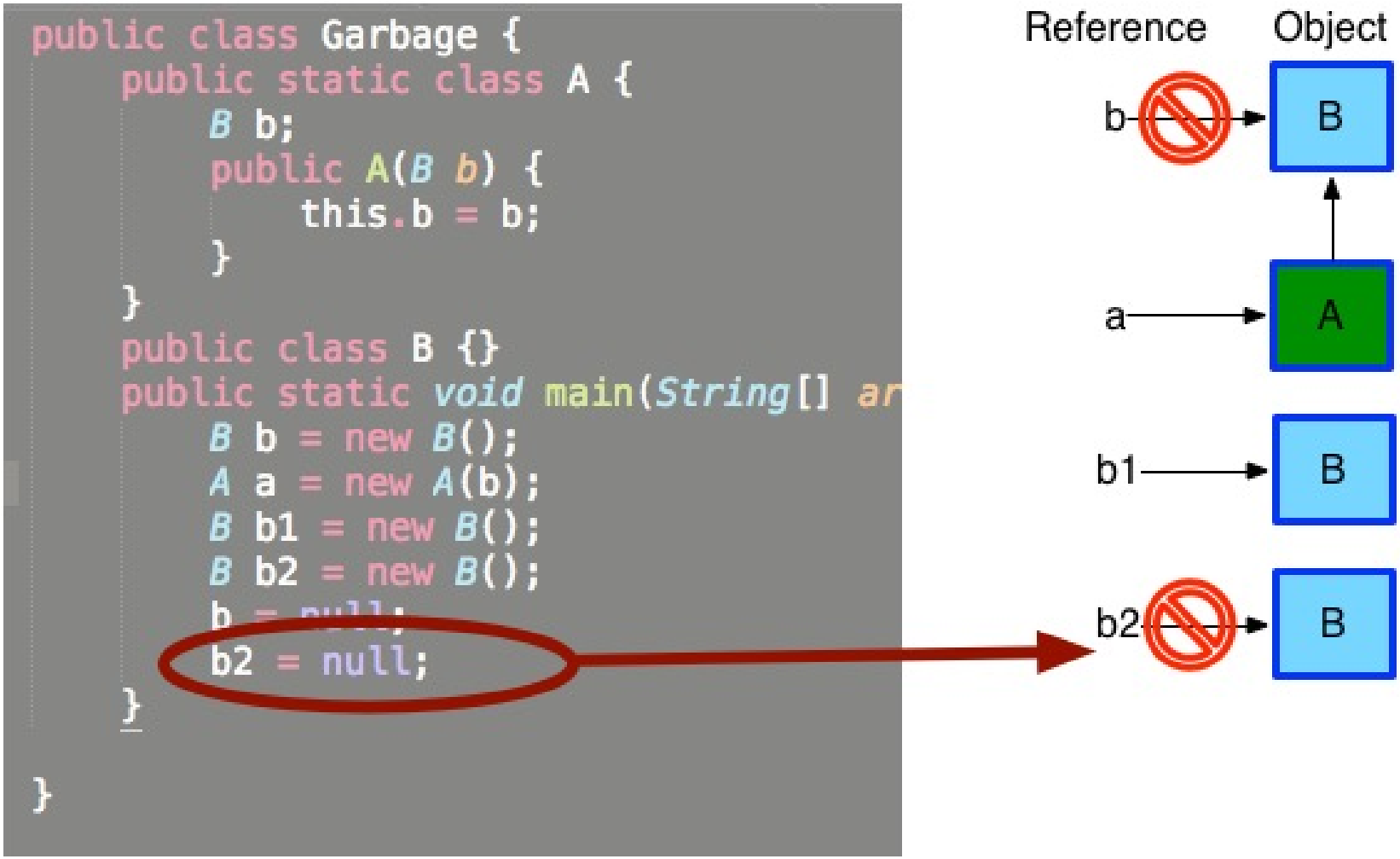


Figure 18: Reference b2 deleted

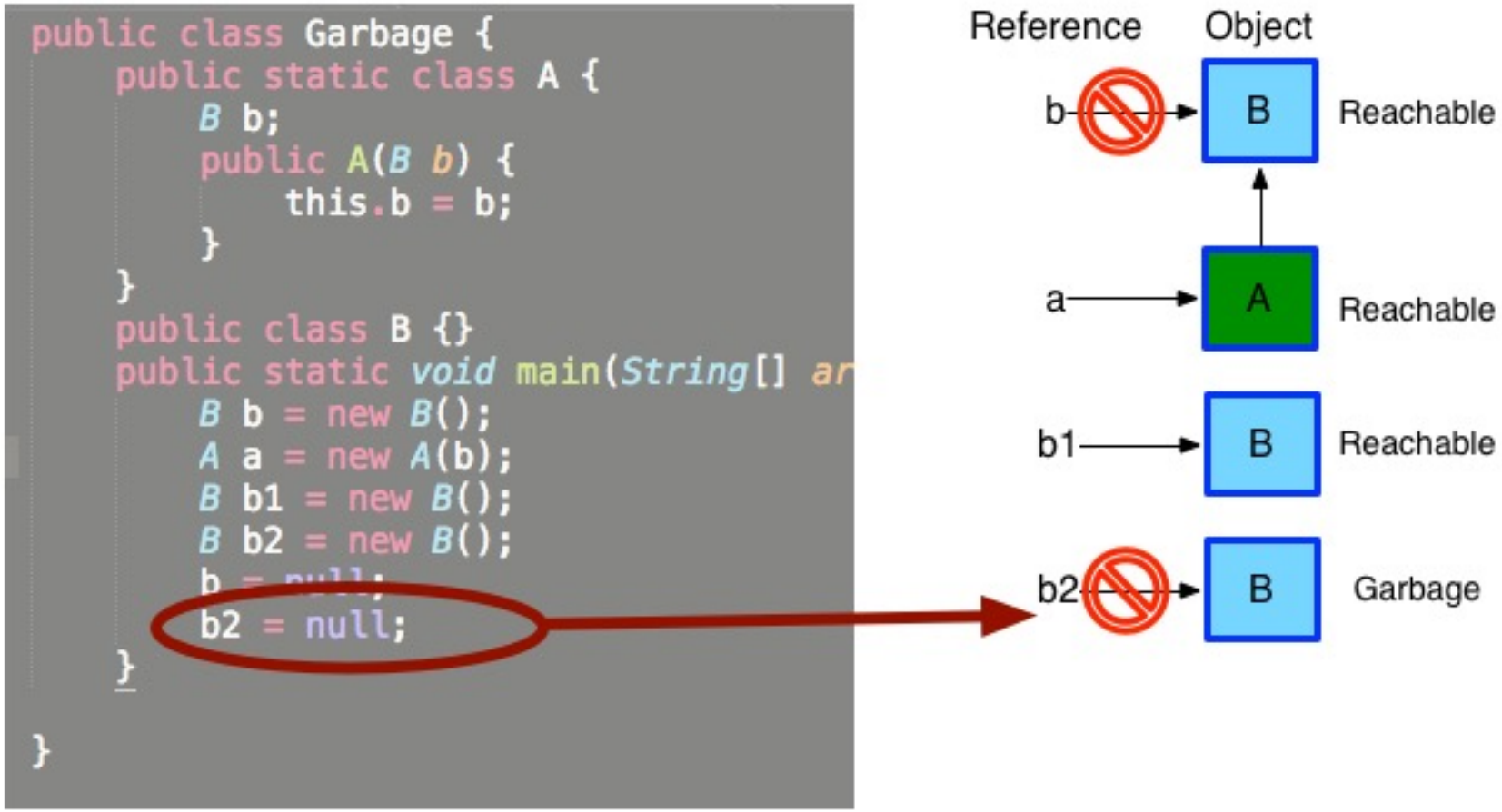


Figure 19: Objects with no reference are garbage. Note that the first B is still referenced from A so isn't garbage even though its original reference has been deleted

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

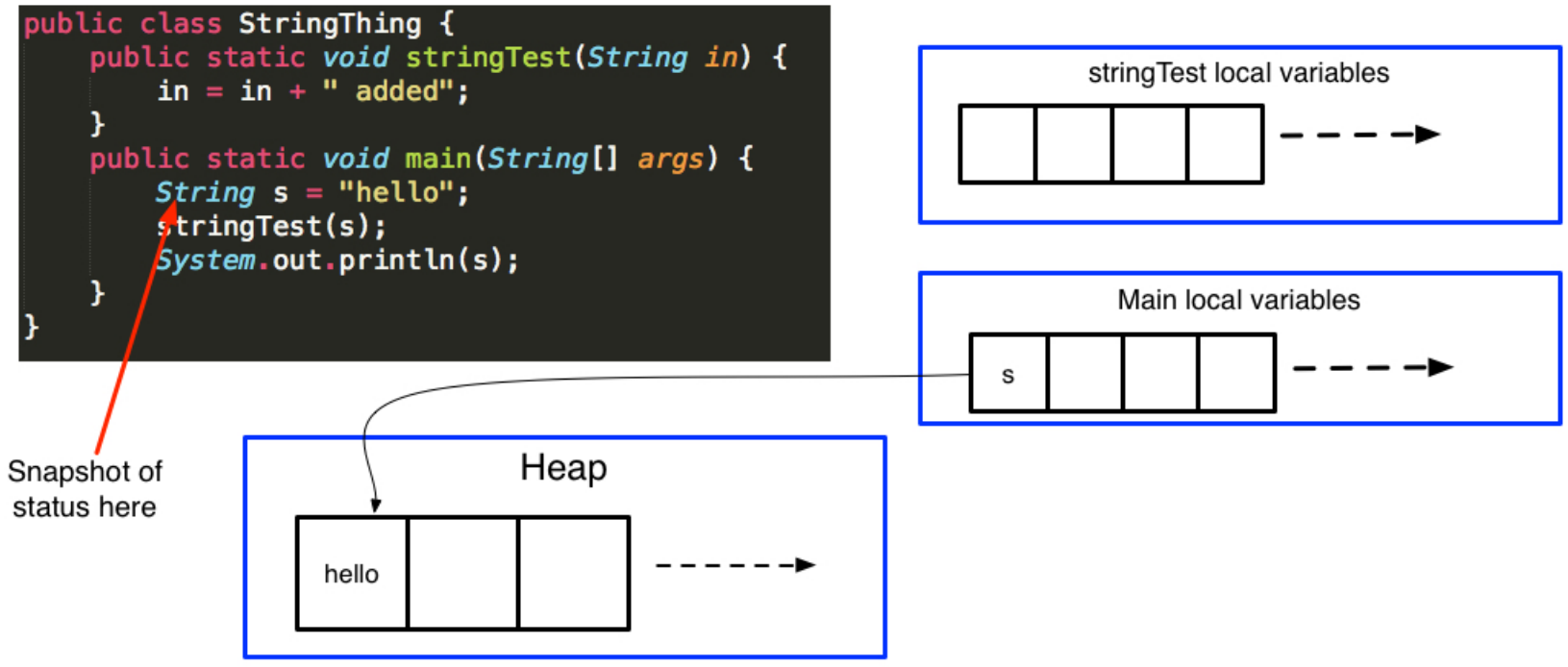

public class StringThing {
public static void stringTest(String in) {
in = in + " added";
}
public static void main(String[] args) {
String s = "hello";
stringTest(s);
System.out.println(s);
}
}
stringTest local variables
Main local variables
s
Heap
hello
Snapshot of
status here

Figure 20:

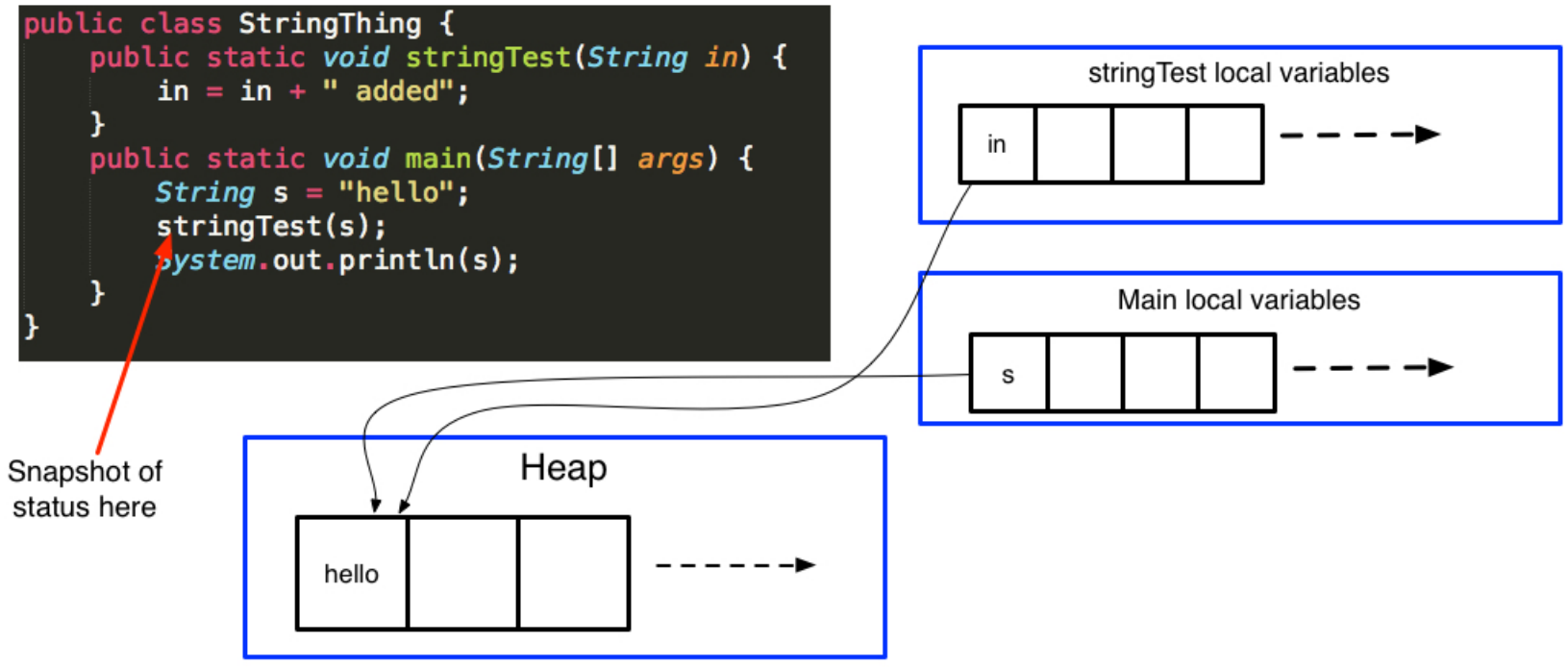

public class StringThing {
public static void stringTest(String in) {
in = in + " added";
}
public static void main(String[] args) {
String s = "hello";
stringTest(s);
System.out.println(s);
}
}
Snapshot of status here
stringTest local variables
in
Main local variables
s
Heap
hello

Figure 21:

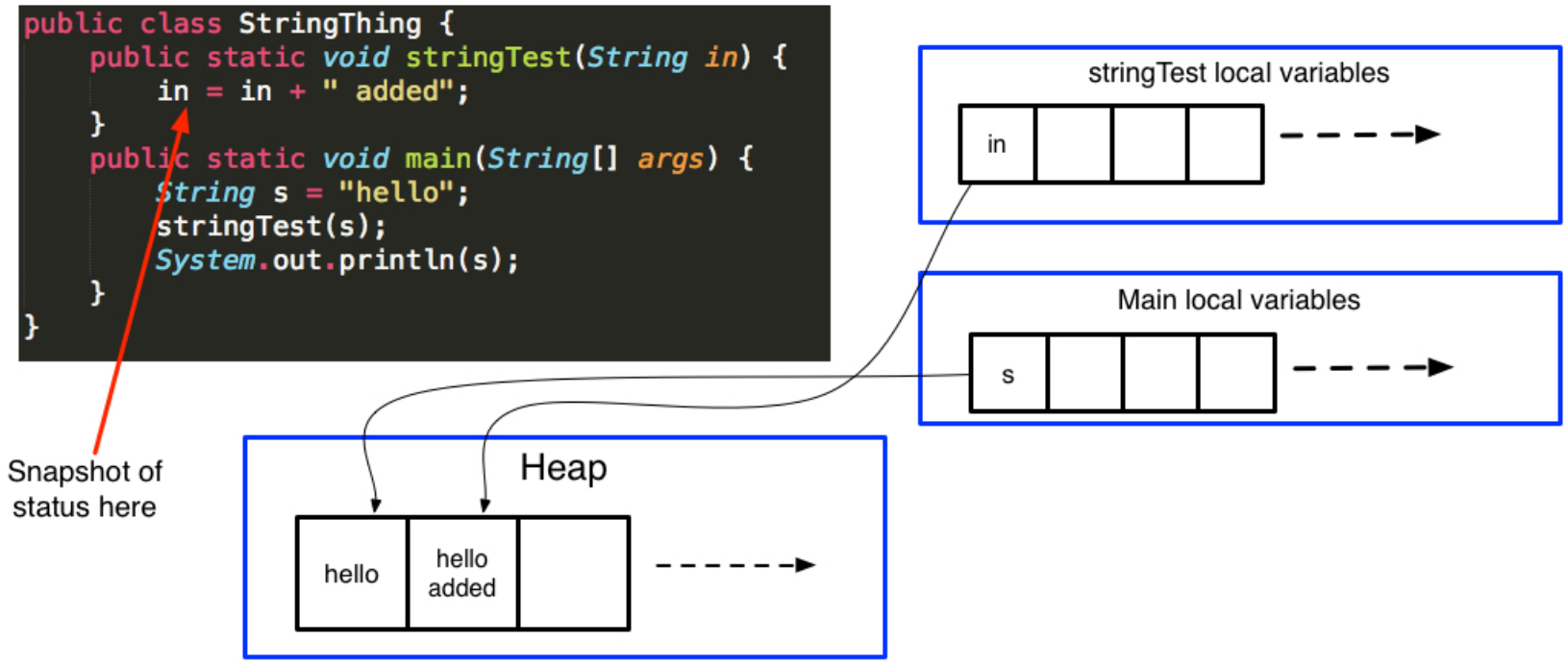

public class StringThing {
public static void stringTest(String in) {
in = in + " added";
}
public static void main(String[] args) {
String s = "hello";
stringTest(s);
System.out.println(s);
}
}
Snapshot of status here
stringTest local variables
in
Main local variables
s
Heap
hello
hello added

Figure 22:

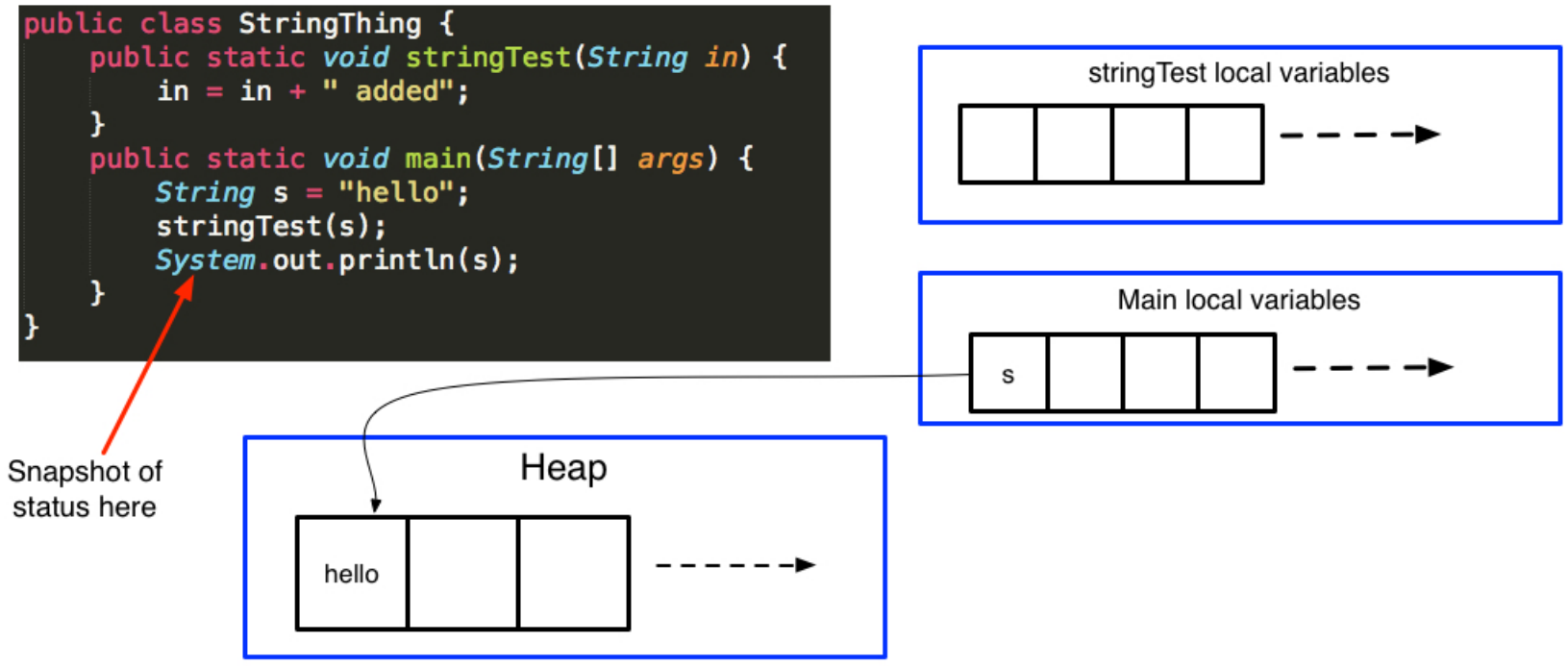

public class StringThing {
public static void stringTest(String in) {
in = in + " added";
}
public static void main(String[] args) {
String s = "hello";
stringTest(s);
System.out.println(s);
}
}
stringTest local variables
Main local variables
s
Heap
hello
Snapshot of
status here

Figure 23:

```
public class ObjectThing {
    public static class myObject {
        private String s;
        public myObject(String s) {
            this.s = s;
        }
        public void setString(String s) {
            this.s = s;
        }
        public String getString() {
            return this.s;
        }
    }
    public static void main(String[] args) {
        String s = "hello";
        myObject o = new myObject(s);
        myObject o2 = o;
        o.setString("blah");
        System.out.println(o2.getString());
        System.out.println(s);
    }
}
```

Snapshot of status here

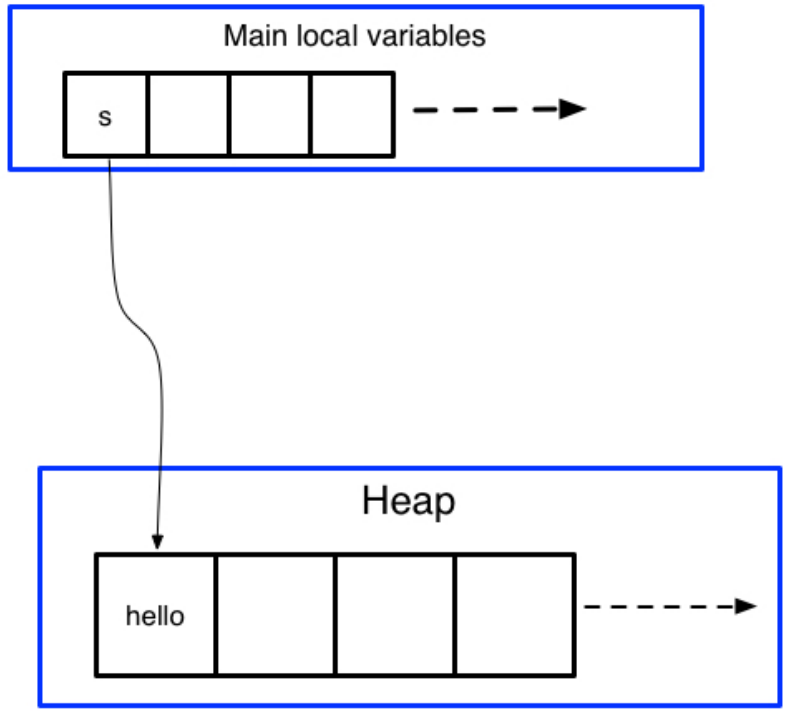


Figure 24:

```
public class ObjectThing {
    public static class myObject {
        private String s;
        public myObject(String s) {
            this.s = s;
        }
        public void setString(String s) {
            this.s = s;
        }
        public String getString() {
            return this.s;
        }
    }
    public static void main(String[] args) {
        String s = "hello";
        myObject o = new myObject(s);
        myObject o2 = o;
        o.setString("blah");
        System.out.println(o2.getString());
        System.out.println(s);
    }
}
```

Snapshot of status here

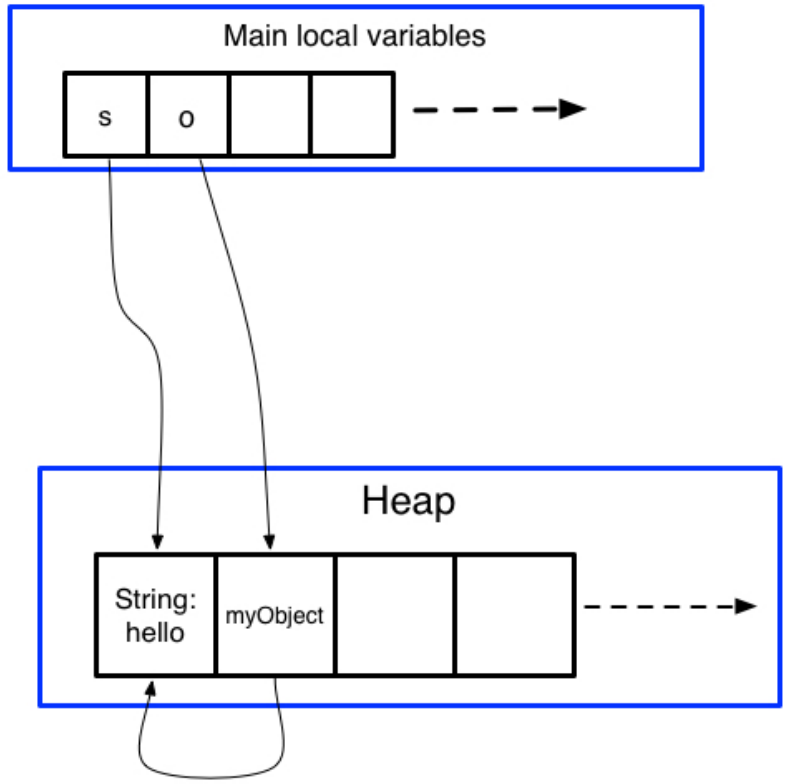


Figure 25:

```
public class ObjectThing {
    public static class myObject {
        private String s;
        public myObject(String s) {
            this.s = s;
        }
        public void setString(String s) {
            this.s = s;
        }
        public String getString() {
            return this.s;
        }
    }
    public static void main(String[] args) {
        String s = "hello";
        myObject o = new myObject(s);
        myObject o2 = o;
        o.setString("blah");
        System.out.println(o2.getString());
        System.out.println(s);
    }
}
```

Snapshot of status here

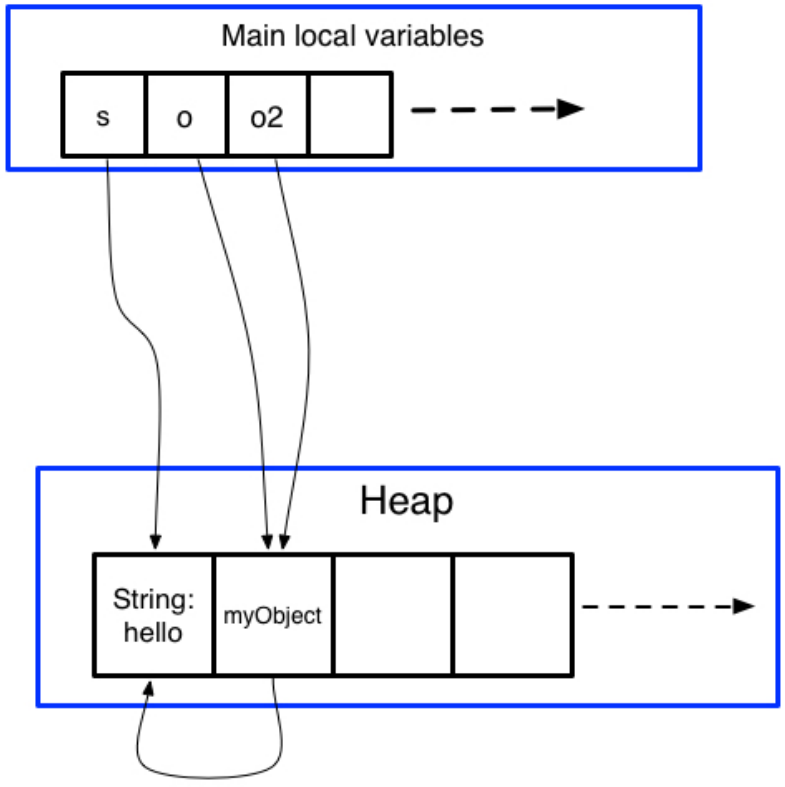


Figure 26:

```
public class ObjectThing {
    public static class myObject {
        private String s;
        public myObject(String s) {
            this.s = s;
        }
        public void setString(String s) {
            this.s = s;
        }
        public String getString() {
            return this.s;
        }
    }
    public static void main(String[] args) {
        String s = "hello";
        myObject o = new myObject(s);
        myObject o2 = o;
        o.setString("blah");
        System.out.println(o2.getString());
        System.out.println(s);
    }
}
```

Snapshot of status here

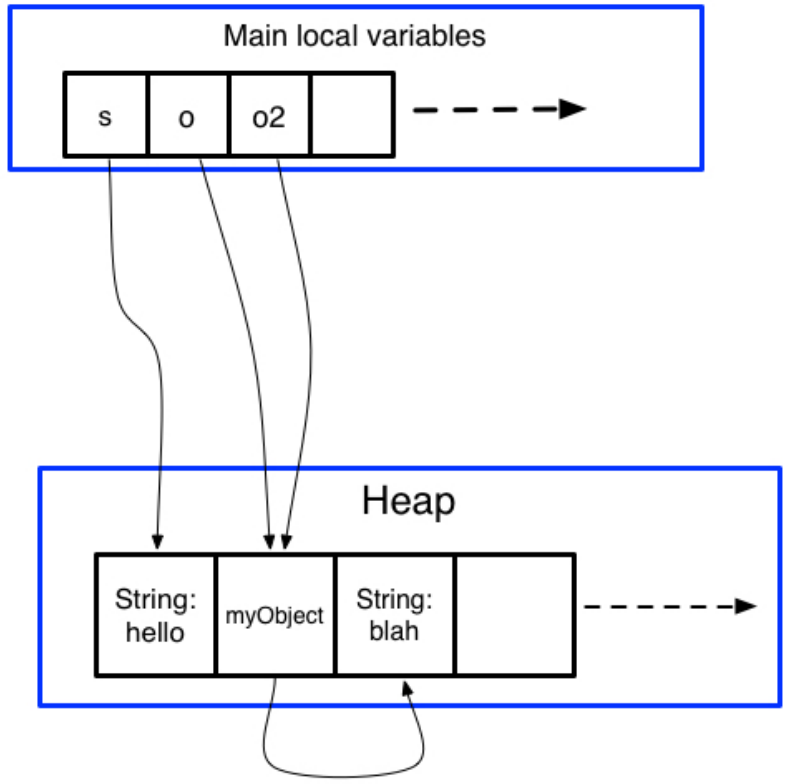


Figure 27:

- 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

Reference (main) | Object | Reference (stringTest)

```
public class StringThing {
    public static void stringTest(String in) {
        in = in + " added";
    }
    public static void main(String[] args) {
        String s = "hello";
        stringTest(s);
        System.out.println(s);
    }
}
```

Figure 28: Example program

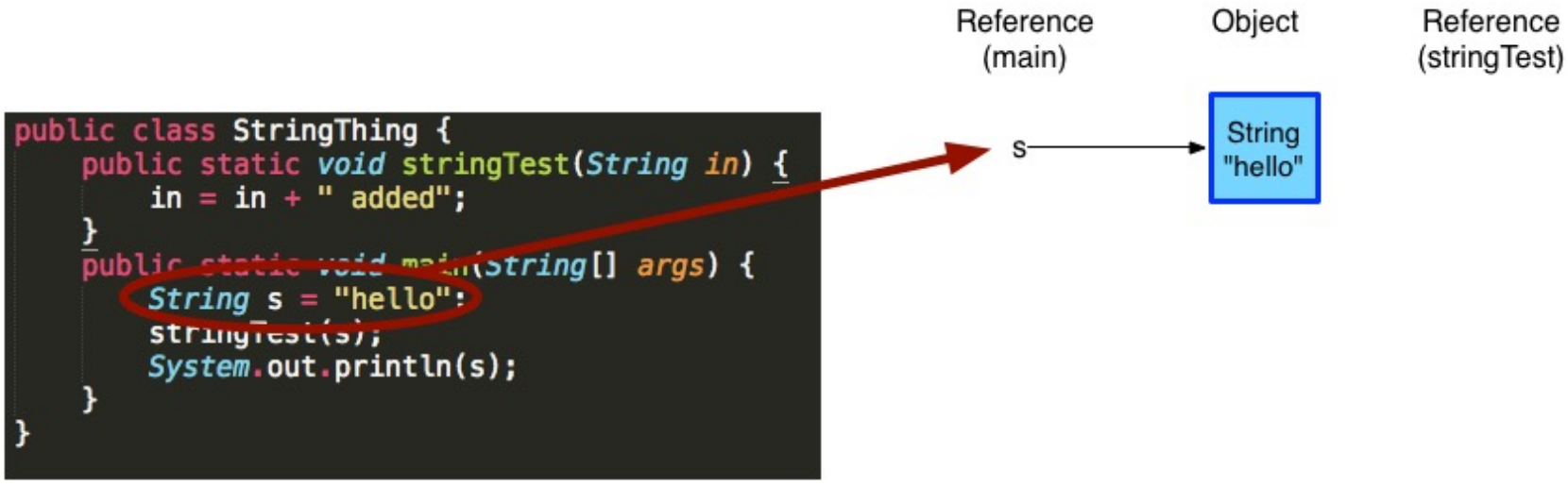


Figure 29: Main makes a String object and a reference (s)

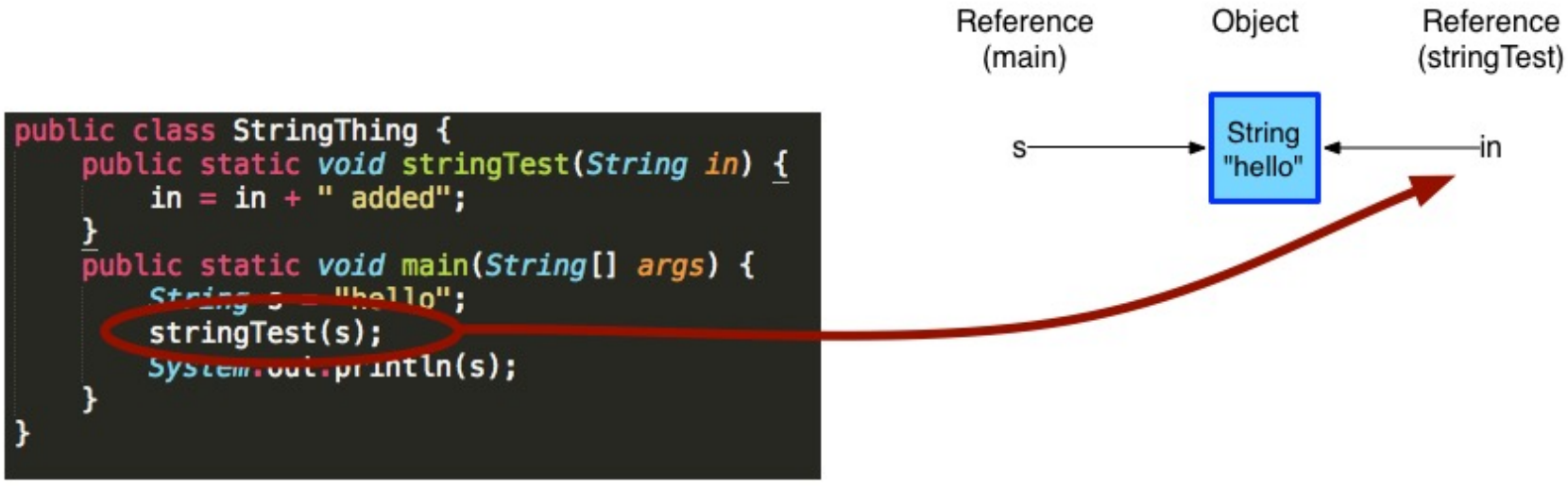


Figure 30: stringTest makes its own reference to the String object (in)

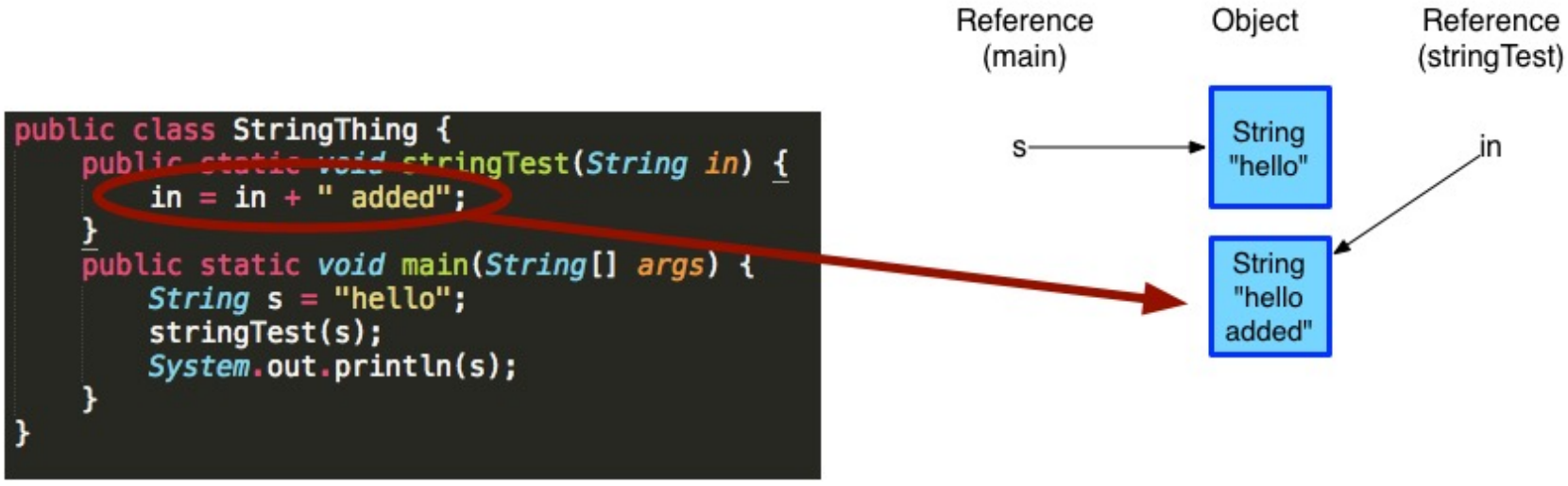


Figure 31: String is an immutable type so when we change it, a new String is made

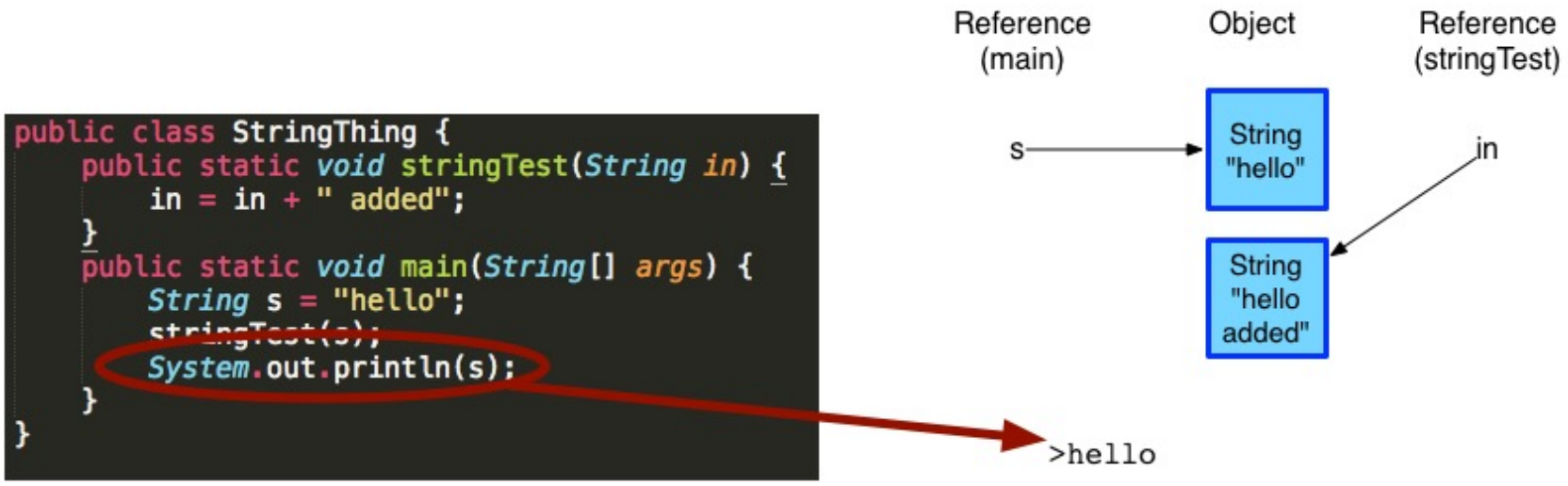


Figure 32: Back in main, s is still a reference to the original object. What happens to the "hello added" string when we return to main?

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

Final

- It is good practice to make as many things `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` does not necessarily mean `immutable`
- `FinalTest` and `FinalTestFixed`

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

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

PointlessTest.java

```
import org.junit.Test;
import org.junit.Assert;
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 - we'll see how to make this easier later in the course through the use of the `ANT` build system.

Assertions

- `JUnit` testing is done at compile time
- We might also want runtime checks
- The naive way is through the use of `if` statements

```
public class AssertionExample {
    private int myInt;
    public AssertionExample(int n) {
        myInt = n;
    }
    public void decrement(int d) {
        if(d>myInt) {
            // Can't 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. runtime or debugging

```
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`

Tools

Version control

- Keeping large projects organised and backed up, particularly if there are multiple authors
- Examples:
  - Git
  - SVN

A short introduction to Git

- Git
  - Version control system
- Github
  - A free server that hosts Git repositories

- Each git repository is a hierarchy of directories
- Git keeps track of how files change within the repository
- Git stores snapshots of the file system, *not* changes to files
- Each snapshot is called a `commit`
- I *strongly* recommend you read the documentation at git-scm.com

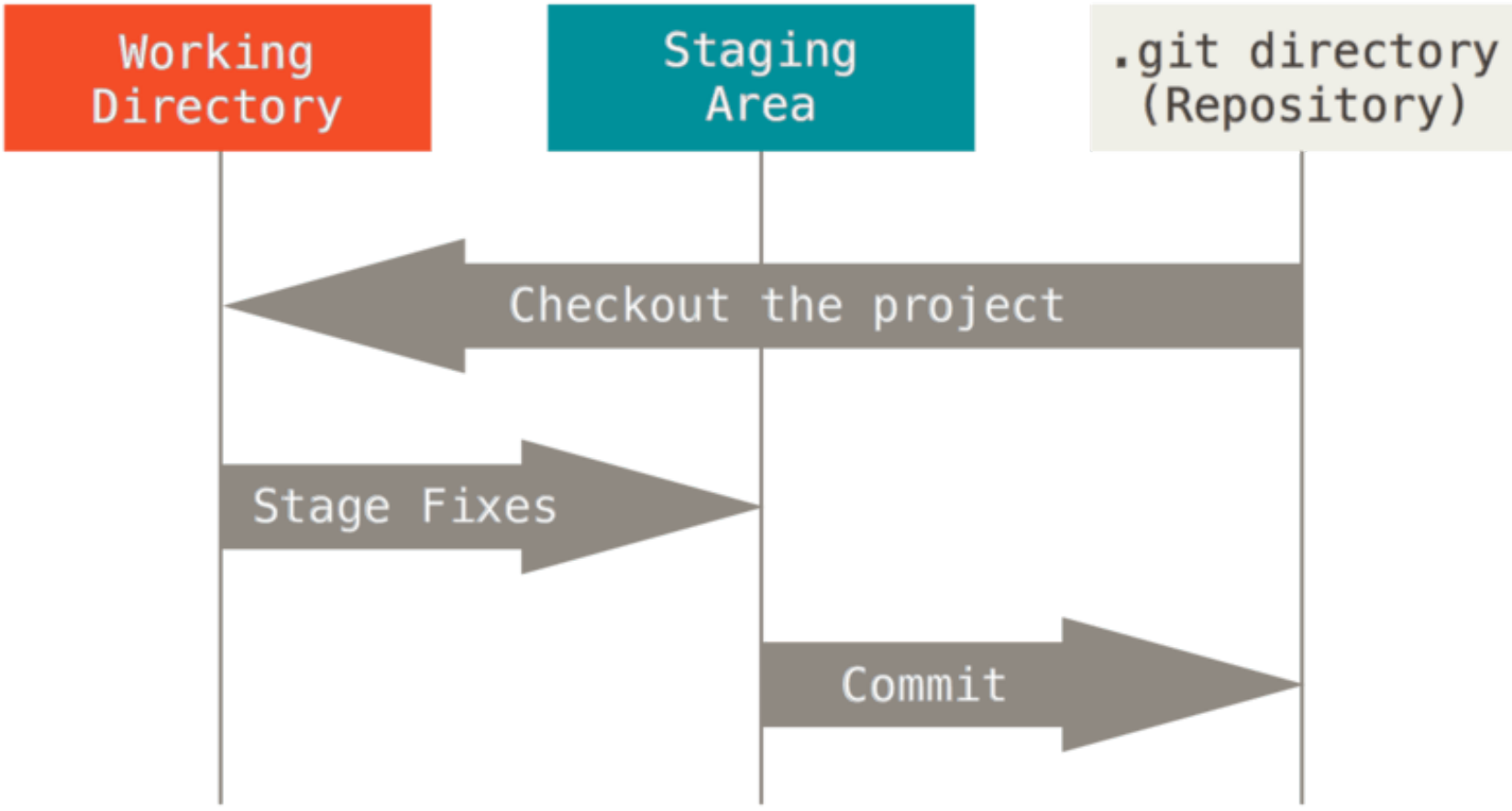


Figure 33: The three components of Git (figure from git-scm.com)

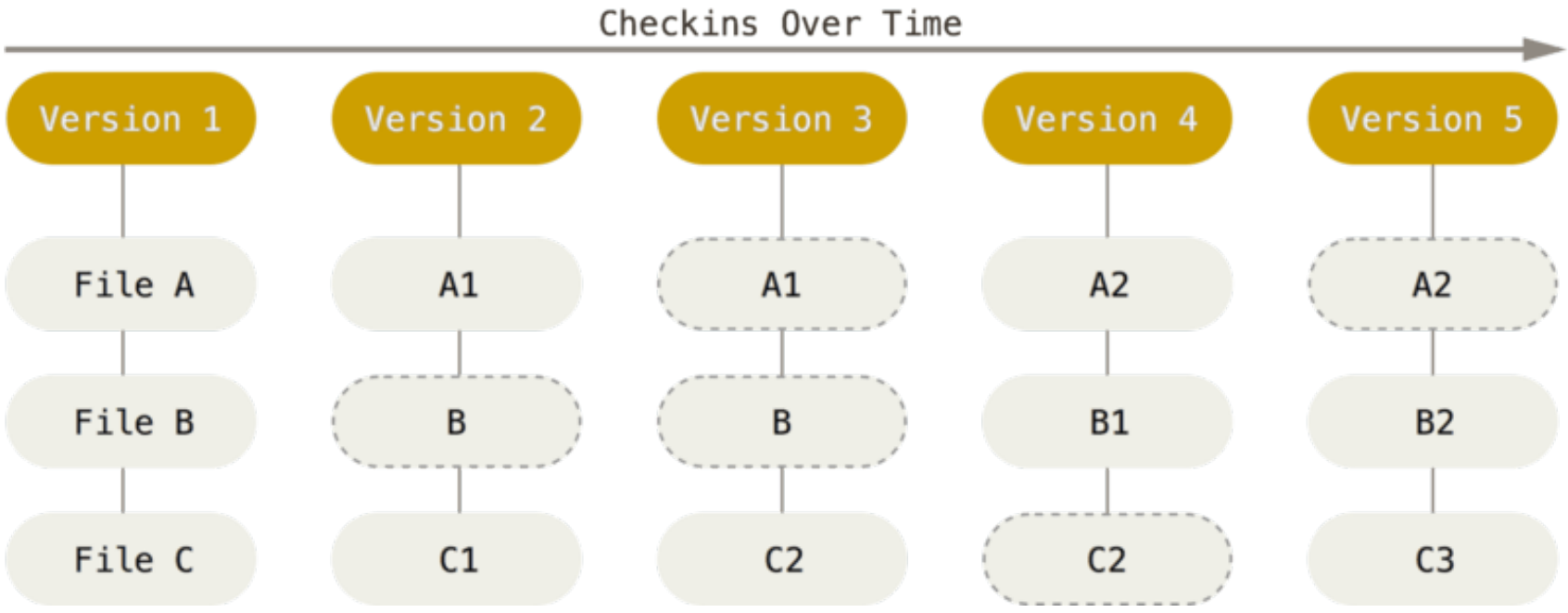


Figure 34: The repository (from git-scm.com)

- *tracked* files are files that are under Git's control
- You have to manually tell it which files to track
  - although it will automatically ignore file types listed in the `.gitignore` file in the project root.
- The Git process:
  - You make some changes (changing files, adding files, removing files)
  - You `add` those changed files to the staging area
  - You `commit` those changes to the local database
- If in doubt, use `git status` to see what's going on
- Example...

Branching

- Git's real power is in the branching functionality
- A branch is a points to a commit (a particular snapshot)
- The Head is a special pointer that points to the current branch
- Each repository has a `master` branch
  - Nothing special about it, it's just the default name for the first branch
- We can create and move between (`checkout`) other branches
- `http://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging`
- Example

Creating a repository

- Two ways:
  - move into the desired root directory of the repository and type `git init` (*try this!*)
  - `Clone` a repository from a server (e.g. Github) (*try this too!*)
  - Note that the repository on the server is just another local Git repository

Collaborating with Git

- Several users clone the same repository
- Changes can be `pulled` down from the server (`git pull origin master`)
  - pulls the master branch from the server and merges it into the current local branch (handling conflicts)
- Local changes can be `pushed` up to the server (`git push origin master`)
- Example...

Build systems

- To compile complex projects that depend on code from different sources
- Examples:
  - Maven (the current standard for Java)
  - ANT (older but still popular)
- See Tim Storer's ANT guide on Moodle