Scientific Programming Using Object-Oriented Languages
Module 5a: Handling Objects

Aims of Module 5a:

- Understand how object variables differ from variables of primitive data types.
- Know when an object passed as an argument can be modified and how to prevent this.
- Understand the use of the null reference.
- Know how to check whether two objects are equal.
- Know how to create a deep or shallow copy of an object.
- Understand how to copy objects and test for the equality of two objects.

References: Primitive Variables

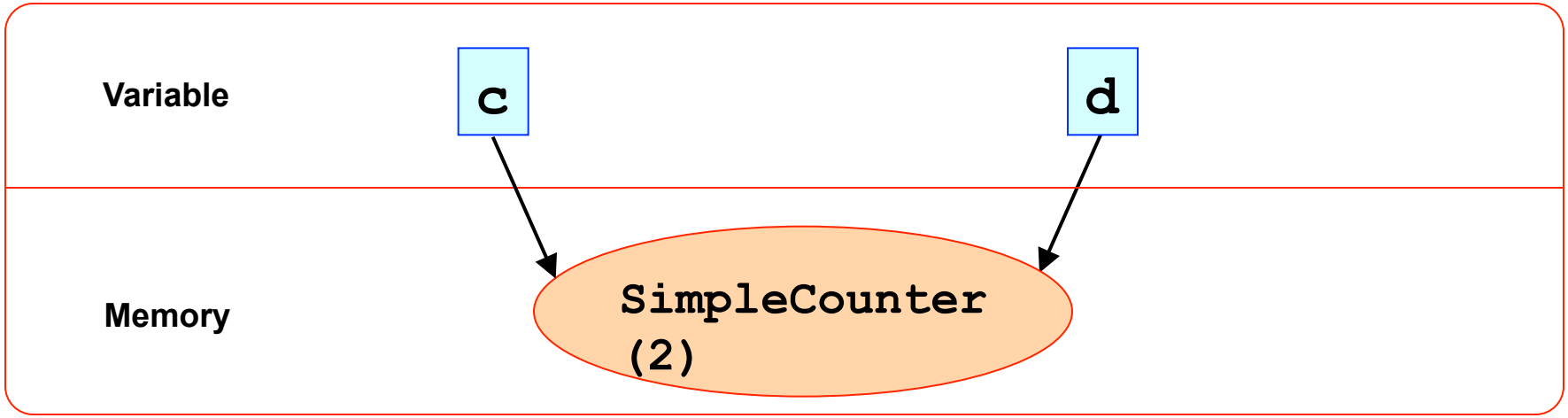
```
int a = 1;
int b = a;
b = 2;
```

What's the end result of doing this ?

References: Object Variables

```
SimpleCounter c = new SimpleCounter(1);
SimpleCounter d = c;
d.setCounter(2);
```

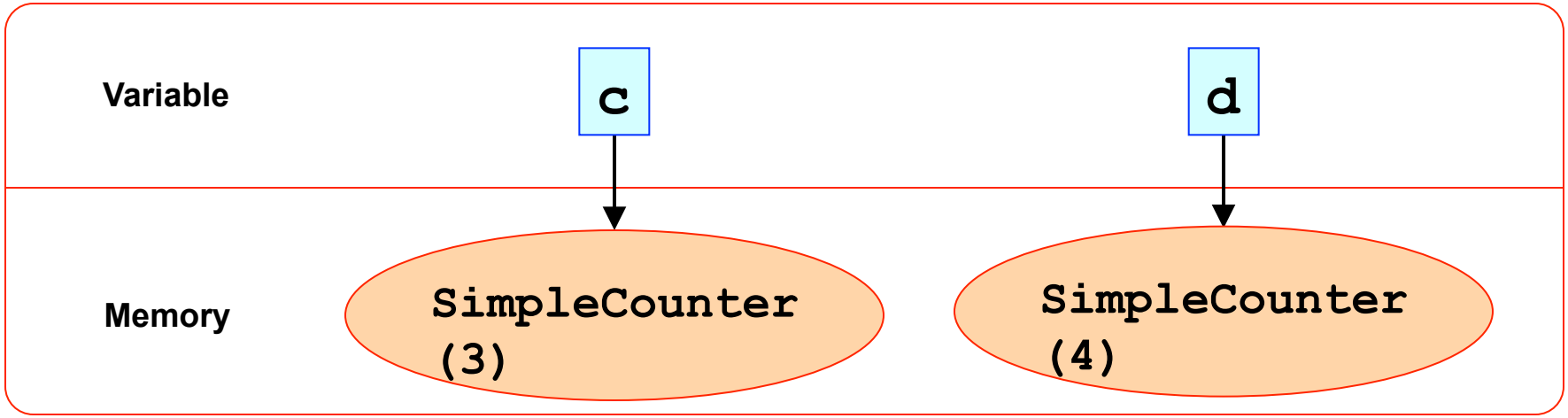
What's the end result of doing this ?



Garbage Collection

```
SimpleCounter c = new SimpleCounter(3);
SimpleCounter d = new SimpleCounter(4);
```

What's the end result of doing this ?

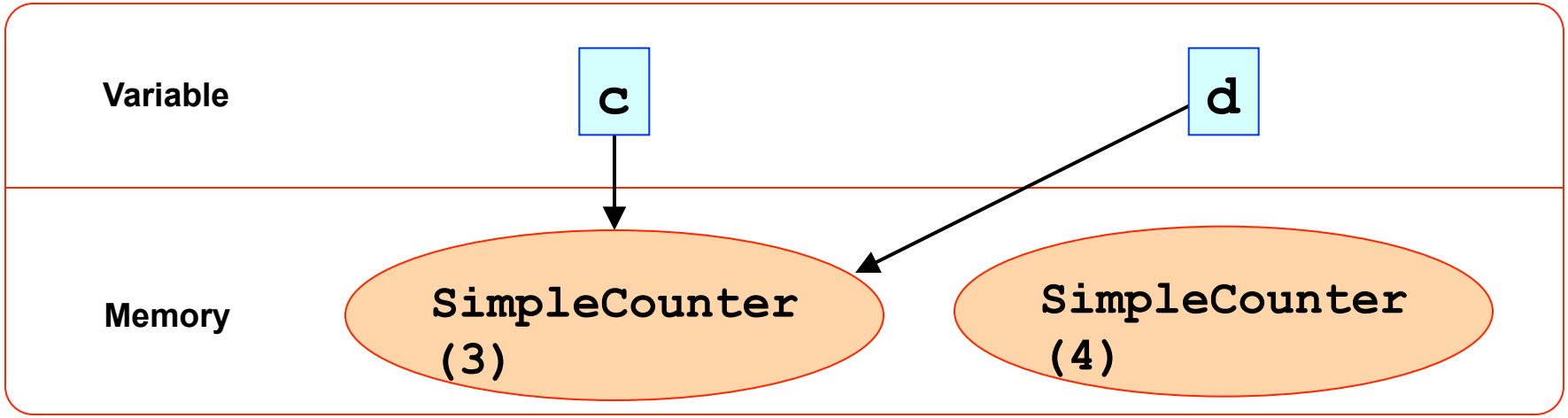


Garbage Collection

Add just one extra line:

```
SimpleCounter c = new SimpleCounter(3);
SimpleCounter d = new SimpleCounter(4);
d = c;
```

Then:



Garbage Collection

- The second `SimpleCounter` object has now been "forgotten" – there is no way to refer to it.
- Java will automatically detect this, and will eventually delete the "lost" object, freeing up the memory it occupied.
- Other OO languages, most notably C++ (see Module 10) do not do this. Special care must be taken to delete objects and failure to do so can cause a "memory-leak".

Passing Objects as Arguments

1. Primitive types : the function that is called receives a copy of the variable that is passed. The variable in the calling code is unchanged, whatever happens in the called function.

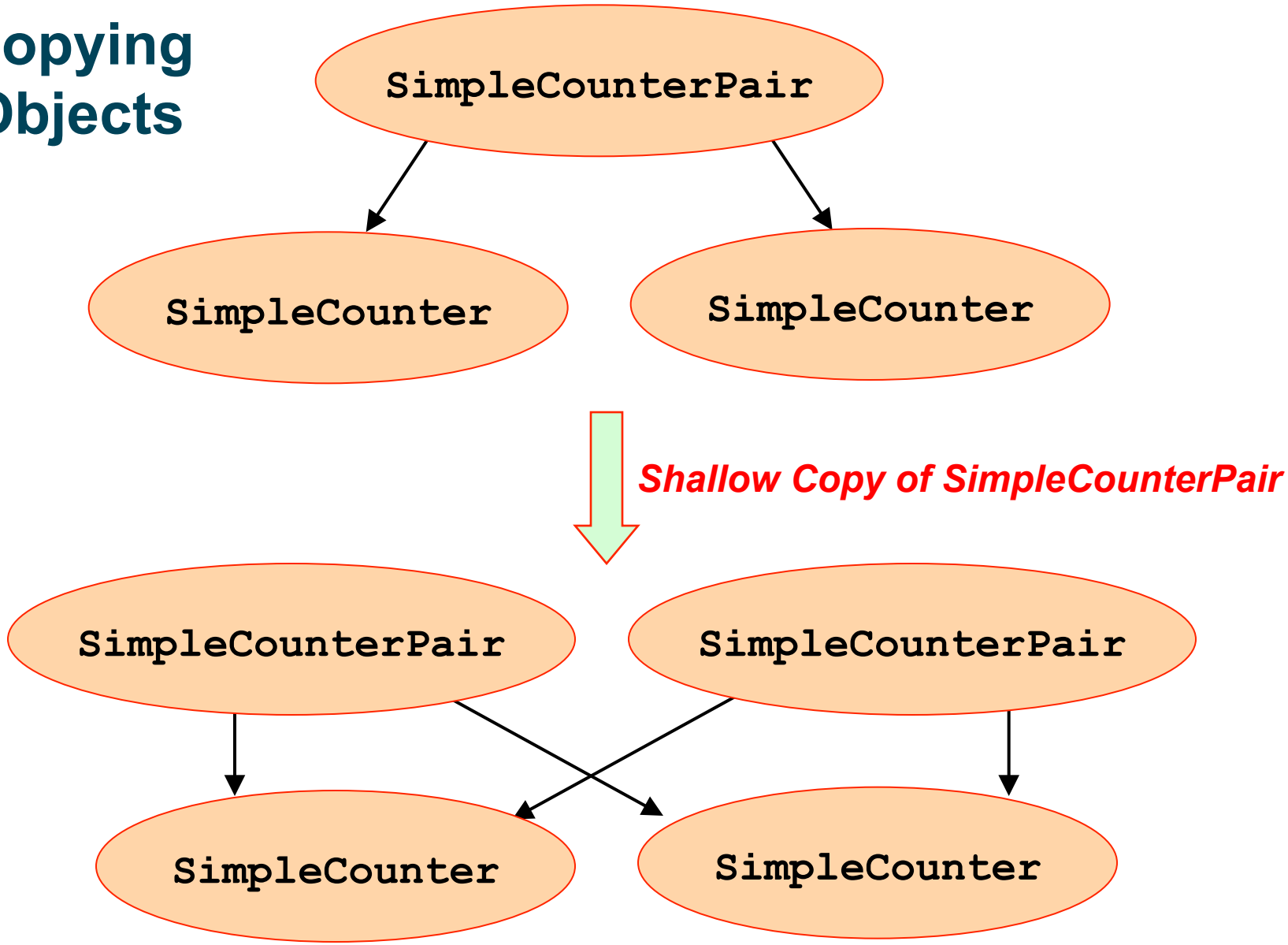
```
public static void main(String[] args) {

    int a = 1;
    changeIt(a);
    System.out.println("a is still "+a);

}

public static void changeIt(int i) {
    i = 2;
}
```

Passing Objects as Arguments

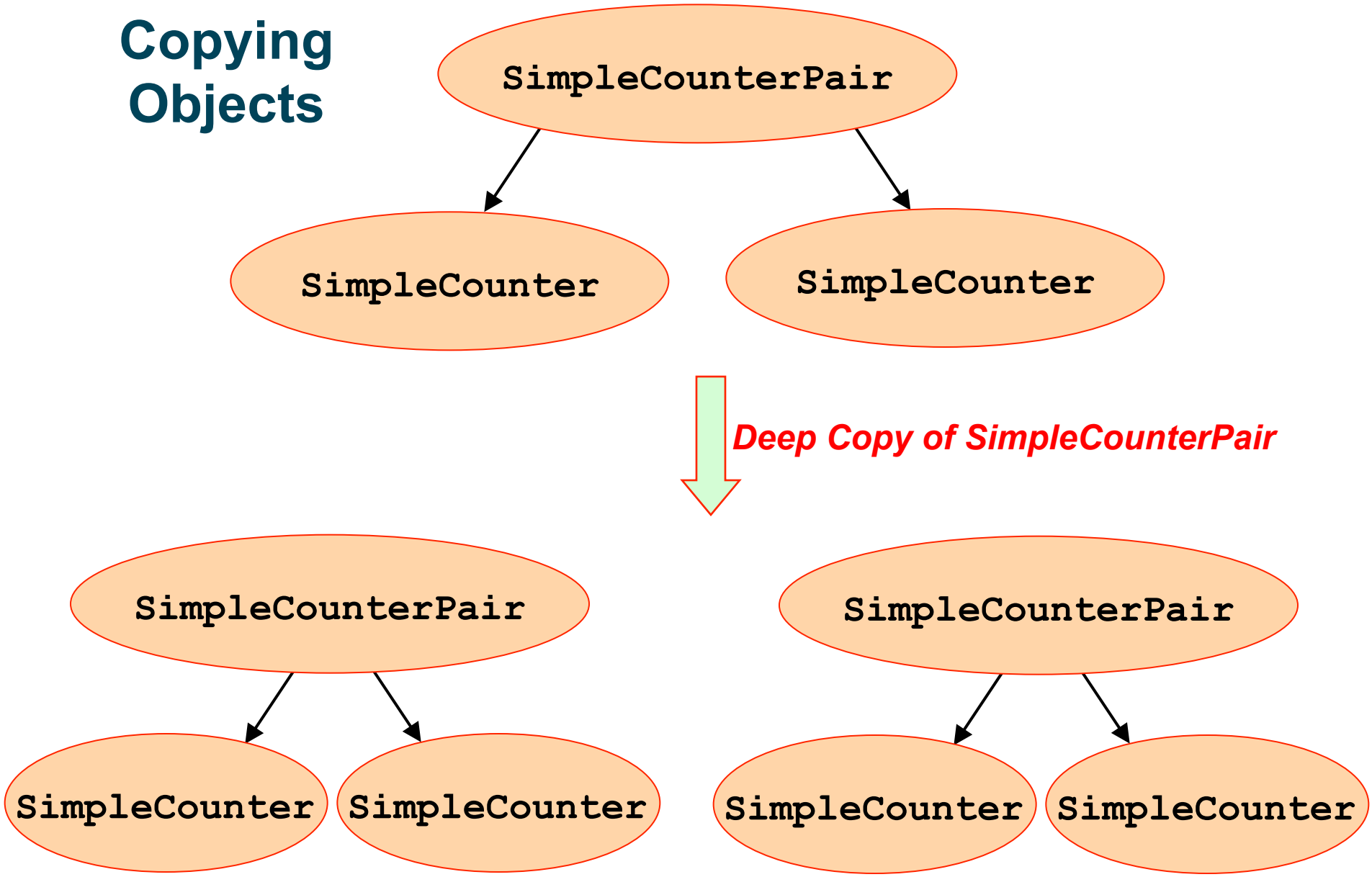
2. Object types: the same thing, but now the value that is copied is the reference to the original object. Watch out!
   - This means that the called function can modify the original object through its copied reference to that object.
   - Think about good OO design. Should the function be allowed to modify the object whose reference has been passed ?
   - Should the object be rendered *immutable* ? (see notes for further details).

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

    SimpleCounter a = new SimpleCounter(1);
    changeIt(a);
    System.out.println("a is now "+a.getCounter());

}

public static void changeIt(SimpleCounter c) {
    c.setCounter(2);
}
```

Copying Objects



Copying Objects



Copying Objects

- Copying objects is best achieved through the implementation of a proper `clone` method - see notes for details.
- The default `clone` method (provided by Java) does a shallow copy.
- A deep copy must be (carefully) implemented by the programmer. For example :

```
protected Object clone() throws CloneNotSupportedException {

    SimpleCounter copy1 = (SimpleCounter) this.first.clone();
    SimpleCounter copy2 = (SimpleCounter) this.second.clone();

    SimpleCounterPair copy = new SimpleCounterPair();

    copy.first = copy1;
    copy.second = copy2;

    return (Object) copy;

}
```

Testing Object Equality

1. Primitive types : the operator "==" compares the values of variables, as we would expect.
2. Object types : the operator "==" compares the values of the two references. They will only be identical if they refer to the *same object*.
   - This is rarely what you want. More naturally, you might want to say that two `SimpleCounter` objects are equal to one another when their internal data are equal.
   - Better to define and use an "`equals`" method. Eclipse can help you form such a method looking something like this (see notes for further details):

Testing Object Equality

```
public boolean equals(Object obj) {

   if (this == obj)
      return true;
   if (this == null)
      return false;
   if (getClass() != obj.getClass())
      return false;
   final SimpleCounter other = (SimpleCounter) obj;
   if (counter != other.counter)
      return false;
   return true;
}
```

- Note that it is up to the programmer to decide which of the Object's data should be compared in the definition of the "`equals`" method.

Summary

- It's important to be aware of what is really happening when Objects are assigned to variables, passed as arguments to functions and copied.
- These slides do not cover all the material in the first part of Module 5. Consult the notes for details of:
  - mutable and immutable Objects;
  - the `null` reference;
  - Using Eclipse to generate `equals` and `hashCode` methods.
- You can start working on the Module 5 exercises.
- You will hear more about collections, but all the information you need is in the notes.