University of
HUDDERSFIELD

CFS2160: Software Design and Development

Lecture 12: Objects All Around

Plus one last loop.

Tony Jenkins
A.Jenkins@hud.ac.uk

Objects

Java makes sense when you believe that *everything* is an object.

But, there are things that are not objects - `int`, `double`, `boolean` - and some of Java's heritage lurks behind the scenes.

Today we will fill in some of the details that will explain all of this.

Strings

`String` is, as we know, a class.

It is defined in the `java.lang` package.

All the details are in the docs.

https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

There are a few small details that can cause problems for the uninitiated.

Testing Equality

A common need is to test a variable's value to see if it equals some value of interest.

For a primitive type (`int`, `double`, `boolean`), this is easy as long as we remember the "double equals":

```
if (input == 1) { …
```

It is slightly more complicated with Strings (because they are objects).

Testing Equality

A common need is to test a variable's value to see if it equals some value of interest.

For a primitive type (`int`, `double`, `boolean`), this is easy as long as we remember the "double eq

```
if (input ==
```

It is slightly more complicated with S

objects).

This is as good a moment as any
to point of that this:
`if (input = 1) {`
is perfectly fine, and is sometimes
what is needed.
(It's `True` here.)

Comparing Objects

Suppose we have this very simple class.

A "Person" is represented by a String and an integer.

```
public class Person {

    private String name;
    private int age;

.
.
.

Person p1 = new Person ("Bob", 35);
Person p2 = new Person ("Jim", 46);
```

Comparing Objects

Suppose we have this very simple class.

A "Person" is represented by a String and an integer.

These two `Person` objects are clearly not in any sense equal.

```
public class Person {

  private String name;
  private int age;

.
.
.

Person p1 = new Person ("Bob", 35);
Person p2 = new Person ("Jim", 46);
```

Comparing Objects

Suppose we have this very simple class.

A "Person" is represented by a String and an integer.

These two `Person` objects are *equal* in the sense that they contain the same values, but they are not *identical* in that they are different objects.

```
public class Person {

  private String name;
  private int age;


.
.
.

Person p1 = new Person ("Alf", 56);
Person p2 = new Person ("Alf", 56);
```

Comparing Objects

Suppose we have this very simple class.

A "Person" is represented by a String and an integer.

These two `Person` objects are *equal* in the sense that they contain the same values, but they are not *identical* in that they are different objects.

```
public class Person {

  private String name;
  private int age;

.
.
.
```

Last week we saw how to define comparison for classes like this. Equality can be defined in much the same way.

Comparing Objects

Suppose we have this very simple class.

A "Person" is represented by a String and an integer.

These two Person objects are *identical*. The two identifiers point to the same object.

```
public class Person {

  private String name;
  private int age;

.
.
.

Person p1 = new Person ("Alf", 56);
Person p2 = p1;
```

Comparing Objects

Suppose we have this very simple class.

A "Person" is represented by a String and an integer.

These two Person objects are *identical*. The two identifiers point to the same object.

(So now changing the values in p1 will now also change the values in p2.)

```
public class Person {

  private String name;
  private int age;

.
.
.

Person p1 = new Person ("Alf", 55);
Person p2 = p1;
```

Comparing Objects

Suppose we have this very simple class.

A "Person" is represented by a String and an integer.

These two Person objects are *identical*. The two identifiers point to the same ~~object~~ area of memory.

(So now changing the values in p1 will now also change the values in p2.)

```
public class Person {

  private String name;
  private int age;

.
.
.

Person p1 = new Person ("Alf", 55);
Person p2 = p1;
```

Comparing Strings

So, there are two ways to compare objects:

- ➢ Test Identity.
- ➢ Test Equality.

Usually, with Strings, you probably want the second.

```
String foo, bar;

// Test Identity
if (foo == bar) {

// Test Equality
if (foo.equals (bar)) {

// Test Literal Value
if (foo.equals ("baz")) {
```

Comparing Strings

So, there are two ways to compare objects:

- Test Identity.
- Test Equality.

Closely related to what we did last week with a `compareTo` method is defining an `equals` method for any class.

```
String foo, bar;

// Test Identity
if (foo == bar) {

// Test Equality
if (foo.equals (bar)) {

// Test Literal Value
if (foo.equals ("baz")) {
```

Wrapper Classes

As we know, Java supports several *primitive* types:

`int`, `boolean`, `double`, `float` ...

Each of these also has a convenient "wrapper class" that does basically the same thing, but in an object-oriented way.

This is useful in cases where an object is required, but the value is in a primitive-type variable.

Wrapper Classes

As we know, Java supports several *primitive* types:

`int`, `boolean`, `double`, `float` ...

Each of these also has a convenient "wrapper class" that does basically the same thing, but in an object-oriented way.

Conversion between corresponding classes is usually automatic: called *autoboxing* and *unboxing*.

Wrapper Classes

The wrapper class for `int` is called `Integer`.

```
Integer i = new Integer (10);
```

Wrapper Classes

The wrapper class for `int` is called `Integer`.

If the value is used in a context of a primitive type, it is *unboxed*:

```
Integer i = new Integer (10);

i ++;
```

Wrapper Classes

The wrapper class for `int` is called `Integer`.

If the value is used in a context of a primitive type, it is *unboxed*.

If a wrapper class is assigned a primitive value, it is *autoboxed*:

```
Integer i = new Integer (10);

i ++;

Integer j;

j = 1;

if (j.equals (1)) {
  // Code executes.
}
```

Wrapper Classes

A common use case is to convert a `String` (entered from the keyboard, say) into a numerical value.

```
Scanner in = new Scanner (System.in);

System.out.print ("Enter a Number: ");
String entered = in.nextLine ();

Integer num = Integer.parseInt (entered);

num *= 2;
System.out.println (num);
```

Wrapper Classes

A common use case is to convert a `String` (entered from the keyboard, say) into a numerical value.

If the conversion fails, an Exception will be generated, so we can check if the user did enter a number.

But we haven't done Exceptions yet.

```
Scanner in = new Scanner (System.in);

System.out.print ("Enter a Number: ");
String entered = in.nextLine ();

Integer num = Integer.parseInt (entered);

num *= 2;
System.out.println (num);
```

Exception

```
Scanner in = new Scanner (System.in);

System.out.print ("Enter a Number: ");
String entered = in.nextLine ();

try {
  Integer num = Integer.parseInt (entered);
  num *= 2;
  System.out.println (num);
}
catch (NumberFormatException e) {
  System.out.println ("Enter a Number!");
}
```

Iteration

We are familiar with *iteration*:

- ➢ Scanning a collection to find a matching object.
- ➢ Listing all the entries in a collection.

We have met *determinate* and *indeterminate* iteration, along with the Java code to achieve them.

There is one further (object-oriented) alternative …

Iterator

Collections have an `iterator` method.

It returns an `Iterator` object.

The `Iterator` class has three methods:

- ➢ `boolean hasNext ()`
- ➢ `E next ()`
- ➢ `void remove ()`

Iterator

```
Iterator <ElementType> it = myCollection.iterator ();

while (it.hasNext ()) {

  // Call it.next () to get the next object.
  // Do something with that object.

}
```

Iterator

```
Iterator <ElementType> it = myCollection.iterator ();

while (it.hasNext ()) {

   // Call it.next () to get the ne
   // Do something with that objec

}
```

So an `iterator` is an object that can iterate through a collection. (This is basically the same as a for-each loop.)

Iterating

So, given a simple class representing "Jobs" in a "to-do" application.

```
public class Job {

  private String title;
  private int priority;
  private boolean finished;
```

Iterating

So, given a simple class representing "Jobs" in a "to-do" application.

And another that represents the list of jobs to do.

```
public class Job {

  private String title;
  private int priority;
  private boolean finished;

public class ToDoList {

  private ArrayList <Job> jobs;
```

Iterating

So, given a simple class representing "Jobs" in a "to-do" application.

And another that represents the list of jobs to do.

An iterator can be used to print out all the jobs.

```
public void listAllJobs () {

  Iterator <Job> it = jobs.iterator ();

  while (it.hasNext ()) {
    Job j = it.next ();
    System.out.println (j);
  }
}
```

Iterating

So, given a simple class representing "Jobs" in a "to-do" application.

And another that represents the list of jobs to do.

Or to purge (remove, delete) all the completed jobs.

```
public void listAllJobs () {

  Iterator <Job> it = jobs.iterator ();

  while (it.hasNext ()) {
    Job j = it.next ();
    if (j.isFinished ()) {
      it.remove ();
    }
  }
}
```

Collections: Arrays

Java provides a rich set of "Collections".

(So do many other modern languages.)

An array is an older collection type, still available in Java, and worth knowing about because it is ubiquitous.

Arrays exist in C (and C++) and Java follows much the same syntax.

Collections: Arrays

Java provides a rich set of "Collections".

(So do many other modern languages.)

An array is an older collection type, still available in Java, and worth knowing about because it is u

Arrays exist in C (and C++) and Java t syntax.

Python doesn't really have arrays (Lists are a bit like them), but there will always be something array-like in any language.

Collections: Arrays

Java provides a rich set of "Collections".

(So do many other modern languages.)

An array is an older collection type, still available in Java, and worth knowing about because it is u

Arrays exist in C (and C++) and Java t syntax.

Most of the time `ArrayList` (or `List`) are what you need in Java, but there is one case where an old-style array can be useful.

Fixed-size Collections

Sometimes the maximum collection size can be predetermined.

A special fixed-size collection type is available: an *array*.

Unlike the flexible *List* collections, arrays can store object references or primitive-type values.

Arrays use a special syntax.

Fixed-size Collections

Sometimes the maximum collection size can be predetermined.

A special fixed-size collection type is available: an *array*.

Unlike the flexible *List* collections, ar references or primitive-type values.

If you want an `ArrayList` of integers, you actually need an `ArrayList` of `Integers`.

Arrays use a special syntax.

Fixed-size Collections

Sometimes the maximum collection size can be predetermined.

A special fixed-size collection type is available: an *array*.

Unlike the flexible *List* collections, ar references or primitive-type values.

This is not valid:

```
ArrayList <int> marks;
```

This is valid:

```
ArrayList <Integer> marks;
```

Arrays use a special syntax.

Arrays

Suppose we wanted to store the number of hits to a website over 24 hours.

Say we have some logs and need to analyse them.

Arrays

Suppose we wanted to store the number of hits to a website over 24 hours.

An array can be used here because we want to store a fixed amount of a primitive type.

```
public class LogAnalyzer () {

  private int [] hourCounts;
```

Arrays

Suppose we wanted to store the number of hits to a website over 24 hours.

An array can be used here because we want to store a fixed amount of a primitive type.

It's created in the constructor.

```
public class LogAnalyzer () {

  private int [] hourCounts;

  public LogAnalyzer () {
    hourCounts = new int [24];
```

Arrays

Suppose we wanted to store the number of hits to a website over 24 hours.

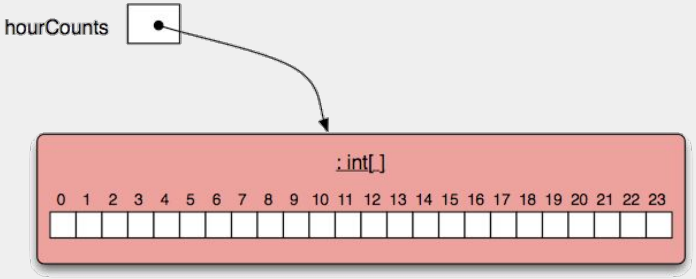
An array can be used here because we want to store a fixed number of a primitive type.

It can be visualised like this.

```
public class LogAnalyzer () {

  private int [] hourCounts;

  public LogAnalyzer () {
    hourCounts = new int [24];
```



Using an Array

Elements in the array are referenced using `[]`.

```
hourCounts [1]
```

Using an Array

Elements in the array are referenced using [].

Elements can be used just like any other variable.

```
hourCounts [1]

hourCounts [1] = 0;
hourCounts [3] ++;
adjusted = hourCounts [8] - 3;
```

Using an Array

Elements in the array are referenced using [].

Elements can be used just like any other variable.

The value in the brackets is the *index*. It can be a variable too.

```
hourCounts [1]

hourCounts [1] = 0;
hourCounts [3] ++;
adjusted = hourCounts [8] - 3;

int busy = 7;
System.out.println (hourCounts [busy]);
```

Using an Array

Elements in the array are referenced using [].

Elements can be used just like any other variable.

The value in the brackets is the *index*. It can be a variable too.

```
hourCounts [1]

hourCounts [1] = 0;
hourCounts [3] ++;
ad


                                       y]);
```

As we might expect, the lowest index is zero.
The highest index is therefore one less that the array's size.

Using an Array

It is often a good idea to store the size of the array in a constant.

Suppose we want to find the mean of six marks on a test.

This would work, but is a *bad thing* to do.

```
marks = new int [6];
```

Using an Array

It is often a good idea to store the size of the array in a constant.

Suppose we want to find the mean of six marks on a test.

This would work, but is a *bad thing* to do.

This is much better, because we can now easily change the number of marks.

```
final int NUMBER_OF_MARKS = 6;

marks = new int [NUMBER_OF_MARKS];
```

Using an Array

It is often a good idea to store the size of the array in a constant.

Suppose we want to find the mean of six marks on a test.

This would work, but is a *bad thing* to do.

This is much better, because we can now easily change the number of marks.

```
final int NUMBER_OF_MARKS = 6;

marks = new int [NUMBER_OF_MARKS];
```

To see why, we need to meet the final type of loop.

Looping through an Array

There are two variations of the for loop:

➢ for-each

➢ for

The "for loop" is often used to iterate a fixed number of times.

It is often used to iterate over every element of an array …

```
final int NUMBER_OF_MARKS = 6;

marks = new int [NUMBER_OF_MARKS];
```

Looping through an Array

There are two variations of the for loop:

➢ for-each

➢ for

```
final int NUMBER_OF_MARKS = 6;

marks = new int [NUMBER_OF_MARKS];
```

The "for loop" is often used to iterate a fixed number of times.

It is often used to iterate over every element of an array …

Here we will want to iterate ~~6~~ NUMBER_OF_MARKS times.

Looping through an Array

There are two variations of the for loop:

- for-each
- for

The "for loop" is often used to iterate a fixed number of times.

It is often used to iterate over every element of an array …

```
final int NUMBER_OF_MARKS = 6;

marks = new int [NUMBER_OF_MARKS];
```

A `for` loop can always be written as a `while` loop.
It's basically a shorthand for a very common case.

Looping through an Array

A for loop is defined like so:

```
for (initial; condition; post-action)
{
  // Statements
}
```

```
final int NUMBER_OF_MARKS = 6;

marks = new int [NUMBER_OF_MARKS];
```

Which is identical to:

```
initial;
while (condition) {
   // Statements
  post-action;
}
```

Looping through an Array

It's probably easier to see with an example.

```
final int NUMBER_OF_MARKS = 6;

marks = new int [NUMBER_OF_MARKS];

for (int i = 0; i < 6; i ++) {
  // Process the i'th element.
}
```

Average Mark

```
Scanner in = new Scanner (System.in);

int [] marks;
marks = new int [NUMBER_OF_MARKS];

for (int i = 0; i < NUMBER_OF_MARKS; i ++) {
    System.out.print ("Enter a Mark: ");
    marks [i] = in.nextInt ();
}

int totalMarks = 0;
for (int i = 0; i < NUMBER_OF_MARKS; i ++) {
    totalMarks += marks [i];
}

System.out.println ("Average Mark: " + totalMarks / (NUMBER_OF_MARKS * 1.0));
```

Assessment

There is no log book this term.

This means that I can show you *my* solutions to the practicals.

But remember that there are many ways to write a program, so just copying code from mine will probably not work …

IntelliJ Demo Time