First Year Software Workshop

Week 3

First Year Labs

- The First Year Labs on the timetable are optional drop-in sessions.
- You can get help from the demonstrators on any of your practical subjects.
- Ground floor lab:

  Monday 10 – 12

  Thursday 2 – 4

  Friday 11 – 1, 2 - 3

Lab files and solutions

- You can find all the work we do in Tuesday's labs on Canvas.
- You can also find my solutions to the exercises.
- Go to the Software Workshop page and click on the "modules" link for a week by week set of files.

Question from quiz

- What does the following code do?

```
for(int x = 0; x < 10; x++);
{
   System.out.println(x+1);
}
```

Answer – but why?

```
% javac MyTest.java
MyTest.java:7: error: cannot find symbol
        System.out.println(x+1);
                           ^
  symbol:   variable x
  location: class MyTest
1 error
```

Arrays

- We use arrays when we want to collect together a fixed number of items.
- We use the square bracket notation

```
int[] numberList = new int[10];

String[] text = new String[100];
```

Arrays are faster

- When you create an array, space for the whole fixed amount is set up.
- When you access an element of an array it is very quick for the computer to find it.
- How?

Address of array[k] =

address of array + k * size of element

ArrayList is sometimes slow

- When adding new elements to the ArrayList, it may be necessary to expand the underlying array.
- When removing elements from the ArrayList, it needs to move everything to close the gap.
- Once an element is in the ArrayList, it is quick to find.
- See question 4 of homework.

Accessing elements of arrays

- The square brackets notation enables us to access elements as if they were variables:

```
int[] numbers = new int[10];
numbers[5] = 6 * 9;
numbers[3] = numbers[5] + 2;
numbers[5]++;
```

Watch out – until you assign values to elements, they have default values.

Arrays and for-loops

- We often use for-loops to access and manipulate arrays:

```
for(int i = 0; i < array.length; i++)
{
   // do something with array[i]
}
```

Classes

- A typical java program comprises of lots of classes.
- Each class is chosen to represent some key type of data for the program
- Many classes are easy to write – they follow a formulaic pattern.

Exercise

- List the different things you should put in a class definition.

Defining classes

- Class name – begins with a capital
- Fields/attributes – the data involved. Should be **private**.
- Constructors – how to create objects of this class type (same name as class).
- Get methods – how to access the information
- Set methods – how to change the information
- Other helpful methods

Data fields (attributes)

- The fields (or attributes) specify all the data associated with an object belonging to your class.
- They should be **private**.
- This enables you to maintain ***consistency.***
- It also enables you to change your ***implementation***.

Example - consistency

```
public class PatientMonitor
{
   private double temperature;
   private boolean fever;

   // etc……
}
```

Example - consistency

```
public void setTemperature(double temp)
{
  this.temperature = temp;
  this.fever = (this.temperature > 38.0)
}
```

Example - implementation

- In the Student class, we used an array to list the modules that a student is taking.
- If the rules changed so that students could take a varying number of modules, we might change this to an ArrayList.
- As long as the header of the get and set methods don't change, programs that use our class will keep working the same.

Constructors

- Constructors have the same name as the class
- They initialise values, allocate space, create the objects required.
- They often initialise fields to *default* values.
- You can have more than one constructor.
- Typically you might have one that assigns default values and one that allows values to be specified.

Multiple constructors - example

```
public PatientMonitor()
{
   this.temperature = 37.0;
   this.fever = false;
}

public PatientMonitor(double temperature)
{
   this.temperature = temperature;
   this.fever = (temperature > 38.0);
}
```

Get methods

- Get methods are used to return values from the data fields, or to calculate useful information.
- The format is:

```
public <field type> getFieldName()
{
   return <the information>
}
```

Set methods

- Set methods change the data.
- Be careful they maintain consistency
- You might want to limit what can be changed.
- The format is

```
public void setFieldName( <new data> )
{
   //make the changes
   //nothing to return
}
```

Other helpful methods

- You should always write a **toString** method. This will help with testing your code.
- Another useful method is the **equals** method. For example, it is used by the ArrayList **contains** method.
- Depending on your program, there might be lots of other methods you want to write.

Equals versus ==

- In java, the == symbol tests for ***identity***.
- For class types this means “is it exactly the same object”
- For basic types it is the same as equality.

```
int x = 3;
int y = 3;
System.out.println(x == y); //prints true
```

Equals versus ==

- What happens?

```
ArrayList<String> x = new ArrayList<String>();
ArrayList<String> y = new ArrayList<String>();

System.out.println(x == y);
System.out.println(x.equals(y));
```

Equals versus ==

- What happens?

```
Integer x = new Integer(3);
Integer y = new Integer(3);
System.out.println(x == y);
System.out.println(x.equals(y));
```

Equals versus ==

- What happens?

```
int a = 2;
double b = 2.0;
System.out.println(a == b);
```

Equals versus ==

- What happens?

```
String s = "hello";
String t = "hello";

System.out.println(s == t);
System.out.println(s.equals(t));
```

Equals versus ==

- What happens?

```
String s = "hello";
String t = "hello world";
String v = t.substring(0, 5);

System.out.println(s == v);
System.out.println(s.equals(v));
```

null

- If you create an array of objects of some class type, but don't create the objects, they will take on the default value **null**.

```
String[] text = new String[10];

//prints null
System.out.println(text[1]);

//prints true
System.out.println(text[1] == null);
```

Null pointer exception

```
String[] text = new String[10];
System.out.println(text[1]); //null
System.out.println(text[1] == null); //true
System.out.println(text[1].equals(null));
```

The first line allocates space for 10 String objects, but does not initialise them.

```
Exception in thread "main"
java.lang.NullPointerException
   at NullTest.main(NullTest.java:6)
```

Using the keyword **null**

- This is fine:

```
String[] a = new String[10];
if(a[1] != null)
{
   String s = a[1].toUpperCase();
}
```

Undefined objects

- However, this gives a compiler error:

```
String b;
System.out.println(b);
```

```
% javac MyTest.java
MyTest.java:8: error: variable b might
not have been initialized
    System.out.println(b);
                       ^
1 error
```

Undefined objects

- This also gives a compiler error:

```
    int x;
    System.out.println(x);

% javac MyTest.java
MyTest.java:8: error: variable x might
not have been initialized
    System.out.println(x);
                       ^
1 error
```

Undefined objects

- But what does this do?

```
int[] numbers = new int[10];
for(int i = 0; i < numbers.length; i++)
{
   System.out.println(numbers[i]);
}
```

- Exercise: try this for boolean, char and double.

Don’t forget!

- Lab assignment by midnight Sunday
- On-line quiz by midnight Sunday
- Drop-in help labs Monday, Thursday, Friday