05 Using Objects

[Close all sections] Only visible sections will be included when printing this document.

References: L&L 1.6, 2.6, 3.1–3.5, 4.4, 7.3, Appendix M

▽ Creating and Interacting with Objects

Why objects?

Primitive types allow us to model simple things: numbers, characters and Boolean values. As soon as a problem becomes more complex they don't provide enough power to solve it.

Objects allow us to:
- model complex real-world (and imaginary) things that are described by many pieces of data; and
- organise the code that will manipulate and process that data

What is an object?

An object (in programming) has:

1. Identity: it's not *just* a value, like a primitive type.
2. State: It can contain many variables (both primitives and other objects). Their value at any given time represents the object's state. These are referred to as its *attributes* or *members*.
3. Behaviour: an object's *methods* provide ways to modify its state or have it perform some other action. Calling a method is sometimes referred to as 'sending a message' to the object.

A formal definition:
- An object is a collection of services that we can tell it to perform for us
- The services are defined by methods in a class that defines the object
- The services may change the state of the object
- The state of an object is recorded in instance variables

Classes

- A class is a 'blueprint' (pattern) for an object
- It is the model or pattern from which objects are created
  - `String` class is used to define String objects
  - `Turtle` class is used to define Turtle objects
  - `Scanner` class is used to define Scanner objects
- Can have many objects of the same class
  - Will each have a different identity
  - Will each have the same behaviours
  - Will probably each have different states

Creating objects

- A variable either holds a primitive type, or it holds a *reference* to, i.e. the memory address of, an object
- An object reference is defined with the same pattern as a primitive type variable:
  - **ClassName identifier** ;
  - e.g., `String title;`
- The actual object is created (*instantiated*) with the `new` keyword:
  - `title = new String(); //an empty string`

The `new` keyword is followed by a special method, called a *constructor*, that has the same name as the object's class. Because a constructor is a method, it may accept additional information (see Methods below).

Examples

```
String title;
Turtle fred;
Scanner sc;

title = new String(); //creates a new, empty string
fred = new Turtle();
//Scanner's constructor takes an argument
sc = new Scanner( System.in );

//Or

Turtle arthur = new Turtle();
//arthur is a different Turtle object to fred
```

Behind the Scenes: Heap versus Stack memory

The memory available to a program is divided (by the operating system) into two parts: stack and heap.

The stack stores data declared inside a method (including housekeeping data used by the computer to keep track of execution). Note that `main()` is just another method. The stack can store primitive types and object *references* (since these are actually memory addresses and so are equivalent to integers).

Each object (and its variables) is stored on the heap. An object reference points to its location in the heap.

This means that assigning an object reference to another object reference doesn't copy the object (just like copying someone's address doesn't clone their house). We'll return to this later in the unit.

Class libraries & the Java Standard Library

The many classes of objects are organised into *packages* (and *subpackages*). Package names are, by convention, all lowercase and a dot `.` is used to separate higher level packages from their subpackages, as in `java.awt.event`.

The Java standard library has many packages containing a very large number of classes for performing a variety of actions: building graphical user interfaces, network communication, database connectivity, data processing, etc.

One of the java packages is `java.lang`, which contains many classes that are so integral to Java that they can be considered part of the language (just like its reserved words, special symbols and syntax).

There is also a huge number of third-party Java libraries for a wide variety of tasks.

Importing a class to use it in your program

So the compiler knows which class, in which package, you wish to use you must either:

1. use its fully qualified name, including its package, as in `java.util.Scanner` (because the `Scanner` class is in the `java.util` package); or
2. Import it at the top of the .java source file where you want to use it:
   - Above `public class` add `import package.name.and.ClassName;` (such as `import java.util.Scanner;`)
   - Then that class's name can be used directly to declare variables of that type, as in `Scanner sc;`

`java.lang` is always available

There's no need to import any class in the `java.lang` package. These are so frequently used, and so essential to Java, that the compiler always knows about them.

Wildcards in import statements

All classes in a package can be imported by using a `*` wildcard at the end of the import statement. For instance,

```
import java.util.*;
```

imports everything from the `java.util` package.

This is not good practice though, because (1) it's a bit slower and (2) eventually there will be two classes in different packages that share the same name and the compiler won't know which one you mean.

Calling methods

- Syntax
  - **objectName** . **methodName** (**arguments**);
- Arguments
  - information passed to the method that it needs to complete its task
  - known as *parameters* in the method definition; *arguments* are the actual values passed when the method is called

Different methods accept zero or more arguments (it depends on what the method needs to do its work).

Some methods return a value (much like a mathematical function). If your program needs this value, it must be assigned to a variable (otherwise it will be forgotten), as in:

```
int origLength = someString.length();
```

Discovering what a method needs as input & what it returns

The method *header* is the first line of its declaration and indicates:
- the level of access (how visible the method is outside of the class where it is declared); the methods you will interact with are all `public` (can be seen and used from outside an object);
- the return type, if any
  - `void` indicates that the method does not return a value
  - the name of a primitive type or class name indicates that the method returns a value of that primitive type or an object of that class

Form of a method header:

access **return type** **methodName** (**parameter list**)

where a **parameter list** is an ordered list types and identifiers, as in:

```
int height, int depth, double density
```

When *calling* a method it is only essential to pass data of the correct type for each position. The identifiers are only used within the method.

Here is the method header for the `substring()` method of the String class, which extracts a sequence of characters from part of the String:

```
public String substring(int startAt, int endBefore)
```

which indicates that the method has *public* access, returns an object of type *String*, is named *substring*, and accepts two `int` parameters.

Constructors are methods too

Constructors are special methods that can declare parameters—the information they require to initialise an object. But they do not have a return type (and don't even declare `void` as a return type).

There's more on this in Part 11 Creating Your Own Data Types

▽ Some Useful Classes

String

- No import needed (in `java.lang`)
- Instantiation
  - like other objects: `String s = new String("not necessary");`
  - or with a String literal: `String s2 = "literal value";`
- Special features:
  - immutable (cannot be changed; any methods that look like they change a String actually return a new String object)
  - Strings can be joined using the concatenation operator `+`
  - Strings contain an array (like a list) of `char`s, which can be accessed by position, starting at 0. The position is the offset from the start of the string.

Some useful String methods

Method header (see Appendix M of L&L for explanation of function)	Example usage (assumes earlier declarations)
`char charAt(int idx)`	`char let = s.charAt(1); //second character in s`
`int indexOf(char c)`	`int pos = s.indexOf('p'); //first position of 'p'`
`int compareTo(String str)`	`int ord = s.compareTo(ns); //< 1 if s belongs before ns` //0 if equal, > 1 is s belongs after ns
`boolean equals(String str)`	`boolean b = s.equals("hello"); //true if s matches "hello", including its case`
`String substring(int startAt, int endBefore)`	`String ss = s.substring(0,8); //first 8 characters of s`

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

Or see L&L Appendix M

Scanner

- Can be used to read user input from the keyboard
- Import with `import java.util.Scanner;`
- Instantiation: `Scanner sc = new Scanner(System.in);` to be able to read data from standard input (typically the keyboard)
- Warnings:
  - Its methods of the form *nextType* read the next value (with type *Type*, such as `nextInt` to read an `int`) by reading characters up to the next whitespace character
  - Its `next()` method reads the next word (String), but...
  - `nextLine()` reads an entire line (up to the carriage return), including the whitespace
  - These differences may be important depending on what you want your program to do.

Some useful Scanner methods

Method header	Example usage (assumes earlier declarations)
`int nextInt()`	`int nInt = sc.nextInt();`
`double nextDouble()`	`double nDoub = sc.nextDouble();`
`String next()`	`String word = sc.next();`
`String nextLine()`	`String line = sc.nextLine();`

See also https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html

Activity: Simple *Star Wars* name generator

The *Star Wars* series of films (and many novels) are renowned for the characters odd sounding names (at least to speakers of most European languages). There are a variety of algorithms for generating similar sounding names from a person's own name, place of birth, parents' names, and so on. Make a *Star Wars* first name generator according to the following algorithm:

```
Algorithm: Star Wars First Name Generator

Variables:
Scanner sc
String givenName, user's given name
String familyName, user's family name
String swName, their generated Star Wars name

Steps:
create Scanner object sc
prompt user for their given name
givenName = next word from keyboard
prompt user for their family name
familyName = next word from keyboard
swName = first 3 letters of givenName + first 2 letters of familyName
display "Your Star Wars first name is " + swName
```

Tip: If you implement this algorithm as written, then a person with the given and family names of "John" and "Smith", would be told their *Star Wars* name is "JohSm". Ignoring difficulties of pronouncing this, it looks odd having the upper case 'S' in the middle of the name. There is an additional String method, not mentioned above, that would be useful for making the output look better, called `toLowerCase()`, which creates a copy of the original string, but all in lower case letters. Try adding that and then rerun your solution.

A complete implementation, which includes the use of `toLowerCase()`:
Show Solution

Turtle

- Can be used to draw simple line graphics in a window
- Obtain the code from the KIT101 MyLO site
- Import with `import kit101.turtle.Turtle;`
- Instantiate with `Turtle t = new Turtle();`
- Turtle's initial state is:
  - position in the centre of its world
  - direction is east (right on the screen)
  - its pen is down (`t.isPenDown()` returns `true`)
  - pen colour `java.awt.Color.BLACK`

Some useful Turtle methods

Method header	Example usage (assumes earlier declarations)
`void move(double dist)`	`t.move(50); //moves Turtle 50 units in current direction`
`void moveTo(int x, int y)`	`t.moveTo(10,50);`
`void turn(double deg)`	`t.turn(90); //Turtle has turned 90 degrees anticlockwise`
`void penUp()`	`t.penUp(); //changes pen to up position`
`void penDown()`	`t.penDown(); //changes pen to down position`
`boolean isPenDown()`	`boolean down = t.isPenDown(); //true if pen is down`
`public void setColor(java.awt.Color c)`	`t.setColor(java.awt.Color.RED); //pen colour is now red`

Documentation will be made available on the MyLO site.

Random

- Generates (pseudo)random numbers
- Instantiation options:
  - `Random rand = new Random();` (randomised using the current time)
  - `Randomd rand2 = new Random(123);` (randomised with the given *seed* value)
- Psuedorandom, not truly random
  - Computers do not generate truly random numbers, but sequences of numbers that are sufficiently close to random
  - Setting the random seed allows the same sequence to be produced

Some useful Random methods

Method header	Example usage (assumes earlier declarations)
`int nextInt(int limit)`	`int r = rnd.nextInt(5); //a random integer between 0 and 4`
`double nextDouble()`	`double p = rnd.nextDouble(); //a random real number between 0 and 1`
`void setSeed(long seed)`	`rnd.setSeed(2933); //a particular sequence of pseudorandom numbers now selected`

See also https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

Activity: Random-sized triangles

Download a copy of the Turtle code (it accompanies varies lectures on the topic of using objects and is also in the KIT101 Support Library, available in this section of MyLO) and place the kit101 folder (containing the turtle subfolder) in the same location as the program you are about to write. Implement the following algorithm to draw two triangles of different random sizes, both with their lower-left corner at the centre of the drawing area.

```
Algorithm: Random Triangles

Variables:
Random rand, random number generator
Turtle t, turtle graphics object
int length, triangle side length

Steps:
create Random object rand with random seed 23 (hint: this means `new Random(23)`{.java})
create Turtle object t

Draw Triangle 1
length = next int between 10 and 200 from rand
t move(length)
t turn(120)
t move(length)
t turn(120)
t move(length)
t turn(120)

Draw Triangle 2
length = next int between 10 and 200 from rand
t move(length)
t turn(120)
t move(length)
t turn(120)
t move(length)
```

Tip: To get a random integer between a lower bound *l* and upper bound *u*, when *l* is not zero use the following:

```
rand.nextInt(u - l + 1) + l
```

The `u - l + 1` passed to `nextInt()` requests a random number between zero and `u - l`, inclusive, which is the number of alternatives in the range we want, and then the `+ l` shifts the generated value to be in the correct *range*. Try out some examples on paper with $l = 5$ and $u = 10$.

Try out your program. Rerun it a couple of times. Are you always getting the same outcome? Delete the 23 from the call to Random's constructor (make it `new Random()`) and then try it again a few times in a row.

A complete implementation, which still uses a fixed random seed for repeatability:
Show Solution

▽ Class members

Objects of a class may share a single copy of some data and some methods. For instance, a hypothetical Person class may have a single method to take a census (it belongs to all instances, not just one) or a single, shared Planet variable (again, Planet is an imaginary class name).

These methods and data are declared with the `static` keyword (and are often called static methods or static data).

To access static data or a static method, use the class name instead of an object's name. Using the fictitious example from above, this could be:

```
Person.planet
Person.takeCensus()
```

The class `java.awt.Color` can be instantiated to represent different colour variants. It also holds a number of constant `Color` values (they are marked `static final`) representing different, commonly-used colours. For example, `Color.BLACK` and `Color.GREEN` are two different `Color` objects, one representing black and the other green.

Classes with no instances

Some classes exist *only* to provide useful methods. An example is the `Math` class, which cannot be instantiated (its constructor is not `public` but `private`). It holds constant values for π (in Math.PI) and e (in Math.E) plus a great many mathematics-related methods.

Useful methods of the Math class

Method header	Example usage (assumes earlier declarations)
`static int abs(int num)`	`int absDiff = Math.abs(7 - 10); //diff becomes 3, not -3`
`static double abs(double num)`	`double absDiff = Math.abs(7 - 10.5); //diff becomes 3.5, not -3.5`
`static int min(int num1, int num2)`	`int minVal = Math.min(10, 45); //minVal becomes 10`
`static double pow(double num1, double num2)`	`double sq = Math.pow(5, 2);` `double sqrt = Math.pow(5, 0.5);`
`static double sqrt(double num)`	`double rt = Math.sqrt(5);`

See also https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

A word on the System class

- No import is required because it's in `java.lang`
- Instantiation is not possible
- It contains a number of static utility methods and final objects that can be used to interact with the local system.
- In particular, it has input and output 'streams' for 'standard input', 'standard output' and 'standard error':
  - `System.in` (you will mostly use in combination with a `Scanner`)
  - `System.out` (you will use frequently)
  - `System.err` (you will use very rarely in this unit)
- `System.out` (and `System.err`) is a PrintStream object, which has useful methods for formatting and printing Strings and other data. Two main variants:
  - `println` outputs data and a carriage return, while `print` does not
  - there are versions of both for all the primitive types, plus String and any other object