## Language Summary and Terms

Please see the power points and worksheets for detailed descriptions and examples of the following terms. Also be prepared to support your learning with scholarly reading.

### 1. Primitive Types

The simplest programming entity is the primitive variable. Java provides C-like primitive variable types notably

**boolean** can be true or false

**byte** 0 to 255 Unsigned 8-bit integer

**char** U+0000  (or 0) to U+ffff (or 65,535) Unicode 16-bit character

**long**  –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 Signed 64-bit integer

**int** -2,147,483,648 to 2,147,483,647 Signed 32-bit integer

**short** -32,768 to 32,767 Signed 16-bit integer

**float** ±1.5 × 10−45 to ±3.4 × 1038 7 digits precision

**double** ±5.0 × 10−324 to ±1.7 × 10308 15-16 digit precision

You **declare** variables by stating variable type and some name. You initialise variables with the **assignment operator** ( = ), though ensure floats use an ‘f’ suffix when assigning values with decimal points.

For example…

#### int x=10;

#### float PI=3.1414f;

**double e= 2.7182818284;**

Declaring a string variable in Java

#### String mystring;

Assigning a string value…

#### myString="hello world";

### 2. Arrays

Arrays are lists of variables. Declaring arrays of variables in Java is especially tidy…

#### float[] prime\_numbers = new float[10];

#### double[][] matrix = new double[3][3];

And you assign or reference values with indexes

#### prime\_numbers[0]=1.0f;

#### prime\_numbers[1]=2.0f;

#### prime\_numbers[2]=3.0f;

#### matrix[2][0]=0;

#### float val2= prime\_numbers[1];

### 3. Arithmetic operators

You can add, subtract, multiply and divide variables with arithmetic operators

#### ( + - \* / )

For example…

#### z=x+y;

#### average=(a+b+c)/3;

And **recast** if needed. For example

#### myInt=(int)myFloat;

### 4. Equality, Relational and Conditional operators

To control the flow of your program you test variable values and perform appropriate tasks using ‘if’ statements and operators such as…

== equals

>, < greater than, less than

>=,<= greater than or equal, less than or equal

For example…

#### float x=0, y=0;

#### if(x==10)

#### y=20;

#### else

#### y=3;

You can use conditional operators && for And, || for OR.

#### if(myString.equals("hello") && happy == true)

#### {

#### reply = "hello to you too";

#### }

#### else

#### {

#### reply = "be that way";

#### }

### 5. Loops

Loops repeat blocks of code. You can loop until a condition is met…

#### while(i < 10)

#### {

#### sum = sum + prime\_numbers[i];

#### i = i + 1;

#### }

You can also have ‘for loops’ of the form…

#### for(initial value, loop while true, increment/decrement)

#### {

#### …statements to run every iteration…

#### }

For example…

#### int i;

#### for(i = 0; i<9; i++)

#### {

#### val = prime\_numbers[i];

#### sum = sum + val;

#### }

You can use the **break** statement to break out of a loop when a condition is met

#### for(i = 0; i<9; i=i+1)

#### {

#### val = prime\_numbers[i];

#### sum = sum + val;

#### if(sum>100)

#### break;

#### }

And use c**ontinue** to skip code within a loop to jump to the next **iteration**.

#### for(i = 0; i<9; i+=1)

#### {

#### if(i=5)

#### continue;

#### val = prime\_numbers[i];

#### sum = sum + val;

#### }

Note the three different possible ways of **incrementing** i (i++, i=i+1, i+=1) after each iteration**.** All add one though you can use the last two ways to add larger step values , e.g. i+=3.

### 6. Arrays and Loops

As well as arrays of primitives, you can also manage collections of objects with ‘for’ loops. For example the following outputs the successive string values within the ‘names’ array.

**String [] names = new String[3];**

**names[0]="Fred";**

**names[1]="Mary";**

**names[2]="Bob";**

**for(String tmp:names)**

**{**

**System.*out*.println("name is "+tmp);**

**}**

### 7. Switch statements

Switch statements are also useful for selecting code based on integer options. For example…

#### switch (myoption) {

#### case 1:

#### mystring="Kellys Eye";

#### break;

#### case 2:

#### mystring="One Little Duck";

#### break;

#### case 13:

#### mystring="Unlucky for some";

#### break;

#### case 66:

#### mystring="Clickety Click";

#### break;

#### default:

#### mystring="I don’t know this bingo call";

#### }

Note the use of ‘break’ to break from the ‘switch’ statement if the correct ‘case’ is matched. Any statements below ‘default’ are executed if no cases are matched or ‘break’ has not been used earlier.

### 8. Methods

Methods are sections of code that can be called repeatedly. You pass **arguments** to methods which may return a resulting value or void. All Java methods are defined within classes.

Here’s a simple method definition. The first line of the method is called its **signature** or prototype, and the variables that define its inputs and output respectively are called **parameters** and **return type.**

#### float myAdd(float a, float b)

#### {

#### float c;

#### c=a+b;

#### return c;

#### }

This somewhat useless function adds two integers and returns an answer. The call to this method could be …

#### C = myAdd(1, 2);

Note: when passing a primitive variable to a method as an argument, a **copy of the variable** is passed. Assigning this value in the method won’t change the value of the primitive variable in the calling/parent method.

### 9. Passing object references

When passing an object name to a method as an argument, a **reference to the object** is passed. Assigning values to the object’s components in the method will change values of the object’s components in the calling/parent method.

So for example the following class defines a simple 2D vector

#### class Vector2D {

#### public double x,y;

#### }

(Note x and y should really be private and accessed by **accessor** and **mutator** methods such as getX(), setX(), etc.)

If we write a method to double the size of a vector

#### public void doubleSize(Vector2D vector)

#### {

#### vector.x \*= 2;

#### vector.y \*= 2;

#### 

#### }

Call it with the following will double its values.

**Vector2D v = new Vector2D();**

**v.x=1;**

**v.y=2;**

**doubleSize(v);**

**System.*out*.println("v.x="+v.x + "v.y="+v.y);**

### 10. Classes and Encapsulation

A Class **encapsulates** data and functionality in fields and methods. **Fields** are member variables of the class that represent the current **state** of the object. **Methods** define the **behaviour** of the object.

**class Student {**

**String name;**

**int age;**

**public Student(String name, int age) {**

**this.name=name;**

**this.age=age;**

**}**

**public void someMethod() {**

**// do something**

**}**

**}**

### 11. Object Instantiation

We can **instantiate** objects from classes. Objects are **instances** of classes, i.e. they occupy memory and have state and behaviour. The method called to set the object’s state at instantiation is called a **constructor** method. For example…

**Student student = new Student("Bob",18);**

### 12. Aggregations of Objects

We create aggregations of these objects with lists (e.g. ArrayLists, Sets, Stacks, Maps, etc) and manage them with ‘**for each loops**’.

**ArrayList <Student> students=new ArrayList<Student>();**

**students.add(new Student("Bob",18));**

**students.add(new Student("Mary",21));**

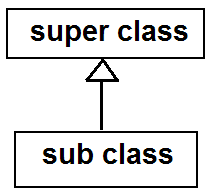
**for(Student s:students)**

**s.someMethod();**

Note the <> symbols around Student in the declaration and instantiation of the ArrayList. These are **generics** thatspecify at compile time that the ArrayList’s methods such as add() will take arguments of type Student.

### 13. Class Inheritance

Class definitions can extend prewritten classes using the ‘**extends’** keyword.



In doing so the **sub class** inherits fields and methods already declared and defined in the **super class**. We can add new fields and methods to the sub class. We can also hide methods declared static (called **class methods**) or **override** none static methods (called **instance methods**) with new method definitions.

**class CompStudent extends Student {**

**float height;**

**public void someMethod() {**

**// override what came before**

**}**

**}**

The super class will itself be a sub class of another, with an inheritance tree leading back to an Object class.

**14. Access Modifiers**

Access modifiers define the access to an object’s member methods and fields. Access modifiers include **public**, **private**, **protected** (access granted to subclasses) and the default ‘**package private**’ (access granted to members of the same package).

**15. Abstract methods**

Abstract methods are methods that are declared but not defined. For example

**public void someMethod();**

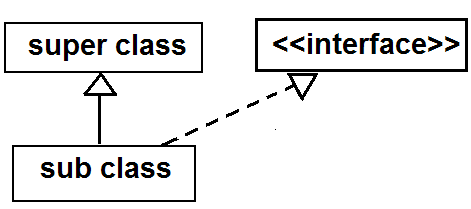
**16. Abstract classes**

An abstract class can include abstract methods that the sub class must provide **concrete implementations** for. A concrete implementation is a method that is both declared and defined.

Abstract classes can also have fields and defined methods just like any other class.

**17. Realising Interfaces**

As well as classes Java has interfaces.



An interface can declare unimplemented *abstract* methods that the sub class must provide concrete implementations for. When you use an interface in a class definition you **realise** or **implement** the interface. You define the interface with the **‘interface’** keyword. You implement the interface with the **‘implements’** keyword.

**interface Singing {**

**public void sing();**

**}**

**class CompStudent implements Singing {**

**public void sing() {**

**// do something concrete**

**}**

**}**

A subclass can only directly **extend** one class, but can **implement** many interfaces. Interfaces can’t have none constant fields or methods already defined.

**18. Substitutability**

If B is a subclass of A and/or a realisation of interface I, you can point B with a reference of type A or I respectively.

**class B extends A implements I {**

**}**

**B b = new B();**

**A a = new B();**

**I i = new B();**

**19. Polymorphism**

Different objects with a common super class or interface can be managed with a reference of the super class or interface type, yet exhibit different overridden behaviours.

**class Animal { void speak( // makes a sound ); }**

**class Cat extends Animal { void speak( …meow… ); }**

**class Dog extends Animal { void speak( …woof… ); }**

**Animal a = new Cat();**

**Animal b = new Dog();**

**a.speak(); // meows**

**b.speak(); // goes woof**

**20. Anonymous Inner Classes**

Anonymous Inner Class definition is a shorthand way define a new class that implements an interface or extends a super class. You can define an Anonymous Inner Classes, using the syntax

**SomeInterface() { … some code defining the class’s members …}**

**SomeSuperClass() { … some code defining the class’s members …}**

The new class has no name (i.e. it’s anonymous) and so instances are pointed to with a reference of the super class type. For example

**EventHandler bHandler = new EventHandler() {**

**public void handle(Event event) {**

**// some code**

**}};**

**21. API**

An Application Programming Interface is a set of tools (e.g. a library of classes) for building software applications.

**22. UML**

The Unified Modelling Language (UML) is a standard way to visualise the design of software applications. This includes specifications for class diagrams, use case diagrams, etc.