JAVA SYNTAX SUMMARY

General Definitions

Abstraction: the ability to ignore details and focus attention on higher levels of a problem.

Modularisation: the process of dividing something into parts, each of which can be built separately.

Overloading: when multiple constructors and/or methods in the same class have the same name. They are then defined by their parameter(s).

Access Modifiers: Used for fields, constructors and methods. 'public' is used for methods which are accessible by other classes; 'private' methods are only accessible within that class.

Information Hiding: the principle stating that internal details of a class's implementation should be hidden from other classes.

Coupling: the level to which changes in one class mean changes are required in another class. Loose coupling is desired. Coupling is highly affected by cohesion, responsibility-driven design and encapsulation.

Implicit Coupling: when one class depends on internal information from another, but the dependence is not immediately obvious (class will often still compile when changes are made, just return the wrong data).

Cohesion: how well a unit of code maps to a logical task. Good cohesion means that **one** unit (method or class) is responsible for **one** well-defined task. Cohesion can be achieved with Responsibility-Driven Design.
Cohesion helps readability, and the potential for reuse of code.

Responsibility-Driven Design: the process of designing classes by assigning each specific responsibilities. Each class should be responsible for handling its own data, and methods manipulating that data should be stored in that class.

Code Duplication: when the same segment of code is present more than once in an application; an indication of bad design (bad cohesion).

Encapsulation: a guideline suggesting that only information about **what** a class can do should be visible to the outside, not **how** is does that. If implemented, the implementation of a class, and how it stores its data, can be changed without affecting any other classes (known as *Localising Change*).

Refactoring: restructuring of existing classes and methods to adapt them to changed functionality and requirements.

Inheritence: the extension of one class into another.

Polymorphism: 'many types'; where one object can be used in situations where a different type to its

declared type is called for (e.g. polymorphic variables - when subclass can be used when superclass is called for).

Autoboxing: performed automatically when a primitive type is used in a context requiring a wrapper type.

Type Checking: uses an objects static type; when the compiler checks code to see if it is feasible (e.g. if the object has the method called).

Java Programs

Java programs are created through the primary use of classes, objects and methods.
Classes are used to define the type, or general cases for objects, and the actions that can be performed with those objects. From a class, one can instantiate an instance of that class, with specific attributes (such as dimensions and colour) that the programmer has decided upon; this is known as an 'object'. The object can then be interacted with using the methods defined in its class, changing its *state* (the set of all attributes defining that object – Java refers to attributes as *fields*).
Methods are are series of instructions, separated into blocks of code with curly brackets, each individual statement terminated with the use of a semi-colon.
The line of code
public static void main(String[] args)
denotes the start of the 'actual' program, that is, the steps that will be executed during runtime.

Classes

A class definition follows the general form
<access modifier(s)> class <identifier>
Each class can have fields, which define the state of each object created from it. Fields are set in the class' *constructor*, which is a 'special' method defined as such:
public <class name>(<parameter(s)>)
Within the constructor the default values of each of the fields should be set, using parameters if the user needs to define any of these values. Methods can then be written to use the fields in the class.

Class Variable

Indicated by the keyword static, it is a field stored in the class itself, therefore there is only ever one copy of the variable at any time, regardless of the number of instances of the class that exist.

Class Method

Can be invoked without an instance of the class. Denoted by the keyword *static*. The method can then be called with <class name>.<method>. Most commonly used as the main method. A class method cannot access any instance fields (because instance fields are associated with individual instances of the class); can only access class variables. A class method may also not call an instance method from the class (because it has no object to call it from), it can only call other class methods.

Constants

Indicated by the keyword final. They are like variables, but cannot change their value during the execution of the application. Must be declared with a value. Often class variables are constant. By

convention, constants are usually all caps.

Generic Classes
A class which requires a second parameter to define types of its fields and/or other variables. e.g. an ArrayList requires another parameter to define what type of objects the list will store.

Class Interface
Name of class, general purpose, constructors, methods, parameters and return types. Everything to use the class, but not know how it specifically works.

Inner Class
A class that is declared inside another class.

Class Implementation
Source code of a class.

Immutable Object
An object where the contents or state cannot be changed once it has been created (e.g. String).

Mutable Object
An object where the contents or state can be changed after its creation.
Note: multiple constructors can exist (known as 'overloading'). The constructor will then be chosen based on the parameter(s) input.
Note2: classes can be input from packages with the syntax import <class name>;
Note3: classes can be imported using import <qualifiedclassname>.

Abstract Class:
a class that is not intended for creating objects. Its purpose is to serve as a superclass for other classes. Can have a mixture of abstract and concrete methods. Defined with the keyword 'abstract'. A subclass of an abstract class must implement all of the abstract class' methods, or it too is an abstract class.
e.g. public abstract class MyClass

Concrete Class
A class which does have an implementation for every method.

Access Modifiers
Used to define what objects (if any) can access an element in a class. Access modifiers can be applied to an objects methods and fields.

Public
Anything can access it.

Private
Only the encapsulating class/object can access it, for fields use getter/setter methods.

Protected

Somewhere between public and private; allows access from within the encapsulating class, and any subclasses, but no other classes. Usually reserved for methods and constructors (applying protected access to fields rather than private ruins encapsulation).

Object Equality

Reference equality

When two variables reference the exact same object (i.e. the same memory location); does not take into account the content, but only the memory location. Test for reference equality with double equals, '=='.
The .equals() method inherited from the object class uses reference equality, unless overridden to do differently.

Content Equality

Checks whether two objects are the same internally; checks whether the two states of the objects match (i.e. all the fields are the same); does not take into account the objects memory location.
Note: if writing your own content equality .equals() method, first include a check to make sure the object passed as a parameter is of the right type (i.e. are they both students/vehicles?) before checking the fields – otherwise you'll likely be calling fields that don't even exist.
Note: if the .equals() method is overridden, you should also override the hashCode() method.

hashCode()

Needs to be overridden if equals() is, because two objects which are equal according to equals() must have the same return value from hashCode(). hashCode() returns an integer value that represents an object.

Object Types

A way of differentiating between the type of the variable, or the type of the object stored within; caters for issues raised by inheritance.

Static Type

The type of an object, as declared in the source code; will never change.

Dynamic Type

The type of object currently stored (e.g. takes into account changes to the object, and casting etc.; changes depending on dynamic behaviour of the program throughout runtime.
e.g. Vehicle c1 = new Car();
Static type is Vehicle, dynamic type at the time of creation is Car, but may change.
e.g. for(Vehicle v: garage)
Static type is vehicle, dynamic type could be any subtype of vehicle.
Note: static type is used for type checking.

Methods

A method signature is the combination of the methods name and its parameters. Access modifiers and return types are not officially recognised as part of the method signature, though some references do include them as being so. For example, the line of code
public void addNumbers(int X, int Y)

would have the method signature of addNumbers(int, int) under the official definition, with some references also including 'public void'.

Accessor Method
This is a method which returns a piece of information about the state of an object.

Mutator Method
A method which changes the state of its object.

Internal Method Call
A method call made by another method in the same class. <method name> (<parameter(s)>)

External Method Call
A method call made to a method in a different class. <object name>.<method name>(<parameter(s)>) - known as *dot notation*.

Method Polymorphism
The same method call may at different times execute different methods, depending on the dynamic type of the variable used to make that call. e.g. how a different method is called when inheritance is used, depending on the subtype called.

Abstract Method
A method consisting of only the method signature, with no implementation. Defined with the keyword abstract, and a semi-colon at the end of the definition.
e.g. abstract public void myMethod();

Concrete Method
A method which does have an implementation.

Overriding
If subtypes and supertypes have the same method, with the exact same signature, the subtype is described as having overridden the method in the supertype. The overriding method takes precedence when that method is called on the subtype object – this is because at runtime, methods from the dynamic type of an object are executed, not the static type.
You can, however, call an overridden method with the use of the keyword 'super'.
e.g. super.print()
Will call the print method from a supertype, even if that method was overridden in the calling subtype.
Note: if a method is called that doesn't exist in the subtype, the compiler searches the supertype, and continues up the class hierarchy until the method is found.

Parameters
Parameters are a way of 'sending' data to a method. Parameters are declared in the parentheses of the method, the same way regular variables are, with an identifier and data type. When the method is called, the parameter must be given a value, by including a form of data within the parentheses of the method call. Throughout the method, the data 'sent' to the method through the parameter(s) can be accessed through the chosen identifier.
e.g.
public void multiply(int X, int Y)
has two parameters, 'X' and 'Y', of type *integer*. When calling this method, X and Y can be given values in the following manner:

multiply(2, 3)
Note: Methods can have any number of parameters, each declaration separated by a comma.
Note2: Names of parameters can be referred to as *formal parameters* (which are limited in scope to their defining constructor or method), whereas the actual value referred to as *actual parameters*.

Functions

As well as taking input, methods can also return a value. This functionality must be declared in the method header, with the data type of the return value. For example,
string getName()
denotes a method which returns a value of the type 'string'. If a method does not return any value, the word 'void' must be included in its header to indicate so. For example, to return the value stored in a variable called 'name', the line
return name;
would be used.

Conditional Statements/Loops

If Statement

General form:
if(<condition>) {
<statement(s)>
}
else {
<statement(s)>
}

For Loop

Executes statements a set number of times.
for(<counter>; <condition>; <counter changes>){<instructions>}
The first part (counter) is often a variable declaration. (e.g. “int I = 0”).

For Each Loop

for(<element type> <element identifier>: <collection>{instructions}
For each element in collection, execute the commands in the curly brackets.

While Loop

while(<condition>){instructions}
Continue to execute the instructions, so long as the condition remains true (pre-condition loop).

Do-While Loop

Similar to the While loop, but the condition is not tested until the statements have run at least once.

```
Do {
        <statement(s)>;
} while (<condition>);
```

Switch (like a case select)

switch (<variable identifier>){
case <value>: <statement>; break;
case <value>: <statement>; break;
}
Note: without 'break' at the end of each line, the compiler 'falls over' into the next line and continues executing statements until it reaches a break, regardless of whether the variable meets the case requirement.
e.g.
switch (number){
case 1: System.out.println("The number was 1.");
case 2: System.out.println("The number was 2."); break;
}
Would output "The number was 1.The number was 2.", if the number was 1.

InstanceOf

obj instanceof Class
This checks if the 'obj' is an instance of the 'Class'.
Returns true if the dynamic type obj is Class, or any of Class' subtypes.

Data Types

Note: Headings written with the format <full name> (<name used when declaring>)
Data types define the way in which data can be manipulated.

Integer (int)

Whole numbers.

String (string)

Denotes a section of text. When assigning text to a string variable, the text must be enclosed in double quotes.
Note: when comparing two strings, always use:
<first string>.equals("second string");

Boolean (boolean)

A variable that can hold only two values – 'true' or 'false'.

Collections:

A group of objects. When defining, we must declare the type of collection and what type of elements it contains. The data type contained within a collection is defined between angular (<>) brackets.

Array

A fixed size collection. Can 'directly' store primitive types *and* objects.
Declared with:

<data type>[] <identifier>;
and then created with
<identifier> = new <data type>[<number of items>];
Values in an array can be accessed by:
<identifier>[<index>]
where indexes go from 0 – the number of items – 1.
Note: all arrays contain a field *length*.

ArrayList

A flexible sized collection. Can store only 'directly' store objects (auto-boxing allows it to store primitive types).
Must import java.util.ArrayList to get access to the class.
ArrayList<<data type>>();
A list of items of the defined data type, of any length. Use add() to add items, use size() to return the number of items in the list. Each item is referred to by its index (starts at 0; removing items is messy, as all the items after it move down an index), and can be returned by its index with <identifier>.get(index).
Note: known as a 'collection' object (objects which can store an arbitrary number of objects).
Note2: each ArrayList has an iterator (of type Iterator) which can be accessed with <identifier>.iterator();

Iterator

Must import java.util.Iterator to get access to the class.
Has two super-useful methods: hasNext() and next().
Iterator<element type> <identifier> = <arraylist identifier>.iterator();
hasNext() returns true/false, depending on whether there is another element for the iterator to move to, and next() moves the iterator to the next index.
iterator.remove() should be used to remove elements from a collection, as you are cycling through (but normal ArrayList remove method should be used for everything else). This avoids exceptions caused by removing an index and confusing the iterator, because the iterator performing the removal means that it knows what's going on.

HashMap

A collection which stores pairs of objects, being keys and values. Values can be looked up by using the key. Ideal for one way lookup (where they key is always known), but terrible for reverse lookup (using the value to find the key). Defined with:
HashMap<<key data type>, <value data type>>
'put' and 'get' are the most useful methods.
<identifier>.put(<key>, <value>);
<identifier>.get(<key>);

Set

A set is much like a list, except that it only allows unique elements (i.e. the same element cannot be stored twice).
Set<<data type>> <identifier>;

Enumerator

A set of items, which must be specified, and then cannot be changed. An enumerator is essentially a name for a group of classes (the enumerator values). Defined as follows:
public enum <enumerator identifier>
{
<value1>, <value2>, <value3>;
}
This code is treated as if it were an entire class. By convention, the values are capitalised.
Enumerators are not objects, but its type values are objects (so could be compared with value1 == value2).
Enumerators can be defined with a parameter value (e.g. GO("go")).
Enumerators can have a constructor (this is not defined as public or private, because the enumerator itself is not a class), which is used for the initialisation of the enumerated values (which are objects).
The enumerator can also have methods, which can be used on its objects.
Values in an enumerator can be iterated over like so:
for (<enum identifier> <identifier> : <enum identifier>.values()){ do stuff...}

Declaring Variables

The declaration of variables follows the following general form
<public or private> <data type> <identifier>
where the data type is one of the recognised and support Java data types, and the identifier a name of the programmers choosing, which can be used thereon in to refer to the data stored in that variable.

e.g.
Private String Variable With Identifier 'firstname'
private string firstname;

Public Integer Variable With Identifier 'age'
public int age;

Note: variables are only available to the method or constructor that declared them. Fields are available throughout the class; *local variables* are those declared within a method (commonly given a value in the same line as their declaration).
Local variable Declaration:
<data type> <identifier> = <value (optional)>;
Note: local variables do not have an access modifier.

Class and Object Diagrams

Class Diagram

Shows the relationship between classes; a *static* view.

Object Diagram

Shows objects (and their field types and field values) at one particular moment during runtime; a *dynamic* view.

Fields

Fields are the 'attributes' of an object, which make up an objects state. There are two types of fields:

Primitive Field:

A field of a data type predefined by the java language (e.g. int, string); stored as a literal value.

Object Field:

A field whose type is a class defined by the programmer; stored as a reference to an object.
Note: if a field is an object field, it is not enough to simply define it as being of that class' type. You must initialise it to be a *new* object (e.g. new <class name>)
Syntax:
private <field type> <identifier>;

Inheritance

public class Class1 extends Class2
The first class has all the functionality of the second; referred to as Class1 inheriting from Class2. This is helpful for avoiding code duplication and maintenance issues. If one wants to use methods from the first class, refer to it using super.methodname.
In a class diagram, inheritance is shown with an arrow with a hollow head extending from the subclass to the superclass.
Inheritance is know as an 'is-a' relationship; Class1 'is-a' Class2, and can be used as such.
Class1 is known as a 'subtype' of Class2.
Variables may hold objects of their declared type, or any subtypes of that declared type.
An *inheritance hierarchy* is a group of linked classes formed through inheritance relationships.
A subclass cannot access private methods of its superclass, but other classes can access public methods of both sub and superclasses. A superclass' public methods can be called by the subclass as if they were directly part of the subclass.

Substitution:

When a subtype is used where an instance of a supertype is expected; does not work both ways (cannot used or declare supertype when subtype is expected).
Note: Execution of the superclass' constructor must be the first line of the subclass' constructor, with the line super(<superclass constructor parameters>).
Note: all classes with no explicit superclass inherit from **Object**; thus, all classes on some level extend from object.
Note: Java allows a class to inherit through 'extends' from only one class; it does, however, allow the implementation of as many interfaces as the programmer wants.

Interfaces

A specification of a type, including the type name and a set of methods, but does not include any implementation for any of those methods. Using an interface ensures that the class implementing that interface includes a certain set of methods. Indicated with the keyword 'interface'.
e.g. public interface myInterface{}

Any class can 'implement' this interface, but it must override all the methods defined in the interface, with its own implementation.
e.g. public class myClass implements myInterface{}
An interface defines a type, thus polymorphism applies the same it would as if the interface were a superclass.
Note: interfaces are usually preferable to an abstract class if given a choice.

Casting

When the type of a variable is explicitly stated with that variable.
(<cast type>) <variable name>
Explicitly states that variable to be of the defined type. Can be used to get around the restriction of using a supertype where subtype is called for, but only if the programmer knows that supertype will *always* be an instance of the subtype whenever the code is called (otherwise errors).
The compiler can not do this for you, because it suffers from 'type loss', where it only knows the declared type of a variable, as it interprets code line-by-line.

Autoboxing

Performed automatically when a primitive type value is used in a context requiring a wrapper type. Compiler automatically wraps the primitive type in its appropriate wrapper object so that a primitive type can be directly added to a collection without extra effort by the programmer.
e.g. int, boolean and char are not object types, but primitive types; therefore, because they do not extend from object, it is impossible to add them to a collection.

Wrapper Classes

Every primitive type in Java has a corresponding wrapper class. A wrapper class holds a primitive type, but represents it as an object.
Note: the reverse to autoboxing, **unboxing**, is also performed automatically.

Graphical User Interfaces (GUI)

Components

Individual parts the GUI is built from (e.g. button, menu, slider); placed on a frame by either adding to the *menu bar* of the *content pane*.

Layout

How to arrange components on the screen; achieved by using a layout manager.

Event Handling

The task of redirecting events (e.g. mouse click), so that something happens when they trigger it. If a user activates a component, the system generates an event; this sends a notification to the appropriate part of the program (the event listener), which then triggers the appropriate action.

GUI Libraries

AWT (Abstract Window Toolkit; import java.awt.*) and Swing (import javax.swing.*). Swing builds on the work in AWT, but still makes use of some AWT classes. For overridden classes, we use the Swing

variation, which is denoted by a 'J' at the beginning of the class name (e.g. Frame is AWT; JFrame is Swing).
Note: custom Swing components can be made by creating a class which extends **JComponent**.
JFrame: a top-level window which contains all other components; can be resized and moved.
frame.getContentPane() returns the content pane, to which things can be added
e.g. Container contentPane = frame.getContentPane();
frame.pack(); // makes the window fit the preferred size and layout; always call after adding/resizing components
frame.setVisible(true); // make frame visible on screen
Note: it is popular to extend a class from JFrame, so that the JFrame does not have to be instantiated, but the methods can be called as if they were internal.
JLabel: a component that can display text and/or an image.
e.g. Container contentPane = frame.getContentPane();
JLabel label = new Jlabel("hello world.");
contentPane.add(label);
JMenuBar: only ever one per window; holds JMenu and JMenuItem components.
e.g. JMenuBar menubar = new JMenuBar();
frame.setJMenuBar(menubar);
JMenu: represent a single menu (e.g. 'File'); often held in a menu bar
e.g. JMenu filemenu = new JMenu("File");
menubar.add(filemenu);
JMenuItem: and single menu item (e.g. 'Open'); held inside a menu.
e.g. JMenuItem openitem = new JmenuItem("Open");
filemenu.add(openitem);

Event Listeners

An object can listen to component events by implementing the appropriate *event listener* interface (must implement the right interface given the component it wants to listen to – e.g. *JMenuItem* components raise *ActionEvents*, so a listener must implement the *ActionListener* interface.). Any object can become an event listener for any event. If it does, it will receive a notification about any event it listens to.
When a button is clicked/menu item selected, the component raises an **ActionEvent**. When a mouse is clicked or moved, a **MouseEvent** is raised. When a frame is closed, a **WindowEvent** is generated. There are a many types of event.
Can have a single object listen for multiple events, or have a separate event listener for each object.

One Listener for them All:

If implementing one event listener for all objects, the frame is often used. The class header needs to implement the ActionListener interface, and implement the method public void actionPerformed(ActionEvent e). This class can then be registered as an action listener to an object by defining it as such in that object (e.g. openItem.addActionListener(this);). If the class is the central ActionListener, it must first figure out what event triggered the method call before it can respond.

Inner Classes:

This is a class that is declared inside another class. Instances of an inner class are attached to instances of the enclosing class. The inner class can see and access private fields/methods from its enclosing class, as it is considered part of the enclosing class just like any other method.
Therefore, the programmer could define separate inner classes for each event possible, and define that inner class to be the action listener for the event. This avoids the need to search for which event happened, but still places all the action listeners in the same central location.
e.g. class MyClass
{

```
class OpenActionListener implements ActionListener
{
public void actionPerformed(ActionEvent e){...}
}
class SaveActionListener implements ActionListener
{
public void actionPerformed(ActionEvent e){...}
}
}
JMenuItem open = new JMenuItem("Open");
open.addActionListener(new OpenActionListener());
JMenuItem save = new JMenuItem("Save");
save.addActionListener(new SaveActionListener());
```

Anonymous Inner Classes:

A class without a name, declared as a parameter within another class. Very useful for implementing event listeners, as they are used where only a single instance of the implementation is required (like with menu items). They essentially provide the same functionality as inner classes, but with a syntactical shortcut because the class is not created as specifically, as it does not have a name, and is defined closer to the registration of its 'listener' status.

Anonymous inner classes are created without naming the class, and with the immediate creation of a single instance of the class.

```
e.g. JMenuItem openItem = new JMenuItem("Open");
openItem.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
openFile();
}
})
```

Instead of a name, the anonymous class names only its supertype (often an abstract class or interface), instantiates that supertype, then defines a block of implementation of its supertypes abstract methods, as is done with any other class.

Anonymous inner classes are able to access fields/methods of their enclosing class, and local variables/parameters of the method in which they are defined (however, any local variables accessed must be declared as **final** variables).

Layout

Swing uses layout managers to arrange components in a frame. Each separate container holding components has an associated layout manager. Layouts can be nested for more sophisticated interfaces. Layout managers are objects in their own rights, and need to be instantiated before being set to a component.

Layout is defined on a contentPane with:

```
conentPane.setLayout(new <layouttype>());
```

FlowLayout: Arranges all components sequentially from left to right; leaves components at their preferred size, and centers them horizontally – if horizontal space is not enough, the components wrap around to a second line. Can be set to align components left or right.

BorderLayout: places up to five components in an arranged pattern (center, top, bottom, right, left) – more components can be added by adding them to a single JPanel, then adding that panel to one of the positions. Position is defined as being CENTER, NORTH, SOUTH, EAST or WEST. When resized, middle component is stretch both directions; east and west change in height, and north and south change in only width.

```
e.g. label = new JLabel();
contentPane.add(label, BorderLayout.NORTH);
```

BoxLayout: lays out components either horizontally or vertically; does not wrap when resized.
GridLayout: lays components in an evenly spaced grid. Number of rows and columns can be specified, and the layout manager will keep all components the same size.

Containers

Containers appear to other components to be a single component, but they can contain multiple components within them. Each container has its own layout manager attached. The most useful/used container is the JPanel.

JPanel:

Can be inserted as a component into a frames content pane, and then have its own components assigned to it. Various JPanel's, each with their own layout, can be combined to make sophisticated layouts.

Borders

Can be used to group components or add space between them. Every Swing component can have a border.

- BevelBorder
- CompoundBorder
- EmptyBorder
- EtchedBorder
- TitledBorder

e.g. JPanel contentPane = (JPanel) frame.getContentPane();
contentPane.setBorder(new EmptyBorder(6, 6, 6, 6));
Note: you must cast the contentPane as being a JPanel, as Container does not have the setBorder method, but JPanel does.
Note2: can only implement borders after importing the border package (javax.swing.border).

Dialog

Modal Dialog: blocks all interaction with other parts of an application until the dialog is closed; forces user to deal with the dialog.
Non-Modal Dialog: allows user to ignore the dialog and continue interaction with other aspects of the application.
Dialogs are often implemented using the **JDialog** class. Modal dialogs with a standard structure can be implmented using convenience methods in JOptionPane.
JOptionPane: includes three standard dialogs; these are in static methods, so do not need to be instantiated.
Message Dialog: displayed a message with an OK button
Confirm Dialog: displays some text and selection buttons (e.g. yes, no, cancel)
Input Dialog: displays some text and a text field for the user to respond.
e.g. JoptionPane.showMessageDialog(frame, “Some Text”, JoptionPane.<message type (e.g. INFORMATION_MESSAGE) – changes icon in message window>);

General Syntax

Assignment Statement
<variable to assign to> = <expression>;

Print To Console
System.out.println("<text>" + <variable>);
Note: if the parameter to this method (or System.out.print()) is not a String object, the method automatically calls the objects toString() method.

Operators
+: addition (and string concatenation – if one argument in an addition is a string, all with be string)
-: subraction
*: multiplication
/: division
%: modulus (remainder after division)

Logic Operators
&&: and
| |: or
!: not

Substring
Returns a string containing the characters from the lowest index, to the last index (does not return the last index) in the nominated string, the first character in the string being index 0..
<string identifier>.substring(<lowest index>, <last index>);
e.g. Calling name.substring(0, 4) on "Benjamin", would return "Benj".

String Length
length(<string>) returns an integer representing the length of the nominated string.

This
this.<some identifier>
This is used to specifically call an object, field, method or other item from within the same class.

Javadoc
The accepted method of commenting java programs. A javadoc comment is shown by an extra asterisk.
e.g.
/**
*Javadoc comment.
*/
Key Symbols:
@version

@author
@param
@return
- information should go after the symbol for proper formatting.