Common Person on the street definitions of common Java/Programming terms

Contents

[WWH - The Why, What and How 2](#_Toc105584300)

[Abstract Class 5](#_Toc105584301)

[Access Modifiers 5](#_Toc105584302)

[API 5](#_Toc105584303)

[Array 5](#_Toc105584304)

[Block 5](#_Toc105584305)

[Brackets 5](#_Toc105584306)

[Class 5](#_Toc105584307)

[Compile-time / Checked Exception 5](#_Toc105584308)

[Runtime / Unchecked Exception 5](#_Toc105584309)

[Composition 5](#_Toc105584310)

[Abstraction 5](#_Toc105584311)

[Concrete Class 6](#_Toc105584312)

[Constant 6](#_Toc105584313)

[Constructor 6](#_Toc105584314)

[Declaration 6](#_Toc105584315)

[Encapsulation 6](#_Toc105584316)

[Global Variable 6](#_Toc105584317)

[Inheritance 6](#_Toc105584318)

[Instantiation 6](#_Toc105584319)

[Interface 6](#_Toc105584320)

[Local Variable 6](#_Toc105584321)

[Main method 6](#_Toc105584322)

[Member 6](#_Toc105584323)

[Method 7](#_Toc105584324)

[Object 7](#_Toc105584325)

[Overloading 7](#_Toc105584326)

[Overriding 7](#_Toc105584327)

[Parameters 7](#_Toc105584328)

[Polymorphism 7](#_Toc105584329)

[Return Values 7](#_Toc105584330)

[Runtime Exceptions 7](#_Toc105584331)

[Scope 7](#_Toc105584332)

[Static 7](#_Toc105584333)

Three Questions

## WWH - The Why, What and How

If ever learning a new concept, it is crucial to understand the

1. Why – Why do I need to use this concept? This is the most crucial aspect. There is no point using something unless you know WHY you are using it!
2. How – How will I use the components to complete the task, at a high level?
3. What – What do I need to use to operate the concept on a lower level? Are there other components needed? Do I need to set up an environment in order to see this concept working properly?

So, taking these heading, we will explain each concept’s functionality. This will form a better (if not comprehensive) understanding of the concept. We will also incorporate the use of real-world exercises that convey these concepts in a more accessible manner.

This can be applied to very high-level concepts (how do engines work) or to a sub-component concept (how does a spark plug work in a gasket or how do the gears affect the speed of the wheels?). Obviously, if there is too much complexity in one part, break up the explanation into subsequent levels.

How does this happen?

Small, controlled explosions move a crankshaft that moves the wheels

Why do we need an engine?

To provide power to move the car



What parts are used in this process?

Sparkplugs, petrol, gears and crankshaft are the main parts

|  |  |  |
| --- | --- | --- |
| Abstract Class | An abstract class is a class that the programmer wishes not to be instantiated directly.  It does not have to contain abstract methods but usually does.  In order to use the functionality of an abstract class the programmer would have to either:   1. Extend the abstract class into a new class and insert their own interpretation of any abstract methods. 2. Use polymorphism to use the abstract classes functionality. That would have to be a subclass of the abstract class existing to do this. | An abstract class would be used if a programmer wishes to transfer responsibility to any other programmer who wishes to use their code. |
| Abstract Method | This is a method with no actual code. This is shown by the absence of { } ‘s  It is made specifically so any programmer who wishes to use them, will have to implement their interpretation of the method’s functionality.  Abstract methods are in [abstract classes](#_Abstract_Class) and [interfaces](#_Interface)  It is the opposite to a [concrete method](#_Concrete_Method) |  |
| Access Modifiers | This is a setting that states who can access/use |  |
| API |  |  |
| Array | An array is a grouping of related data items, e.g., Marks in an exam, words in a sentence.  [ ] - Square brackets denote an array. |  |
| Block | A block is a section of code. Usually, curly brackets denote the start and end of a block of code { }  A block can be used to limit the effectiveness or time of execution of specific parts of code. |  |
| Brackets | Brackets are vital in Java programming. There are three primary brackets every programmer should know and their associated functionality   1. { } - Curly brackets denote block of code, either a class, method or construct (if statement, loop) 2. ( ) - Round brackets denote a method, either it's signature or an invocation point (calling the method) 3. [ ] - Square brackets denote an array. |  |
| Class | A class is the Java name for a program. |  |
| Compile-time / Checked Exception | An exception which will occur when the associated class is compiled. It is more severe than a run-time / unchecked exception.  All compile time exceptions are a subclass of the Exception class BUT not or RuntimeException class. |  |
| Runtime / Unchecked Exception |  |  |
| Composition | Composition is when an object class is composed of other object classes, typically which are written in relation to the overall | A CAR class maybe composed of Wheels, Doors and an Engine object.  Where there are several objects |
| Abstraction | This is where the details of a class are hidden, but the code can be used.  Similar to the notion of a “black Box£ |  |
| Concrete Class | A concrete class is a class that the user can instantiate/create objects from.  It is the opposite to an [abstract class](#_Abstract_Class). |  |
| Concrete Method | This is a method with actual code. This is shown by the presence of { } ‘s  It is the opposite to an [abstract method](#_Abstract_Method) |  |
| Constant | A constant is a variable, method or class that is fixed in value and cannot be changed.  The word final denotes a constant. |  |
| Constructor | A constructor is used to [instantiate](#Instantiation) objects. All classes will have at least one constructor to aid in creating objects of it’s type.  A constructor is recognizable in   1. It shares the same name as the class 2. Is similar in structure to a method 3. But will not have a return value |  |
| Declaration | This is when a variable is created and given a datatype (and maybe an initial value) |  |
| Encapsulation | This is where class/object variables are kept private (to prohibit disruption), and the object’s copy can then be accessed/modified via   1. Getters 2. Setters |  |
| Extends | This is the keyword to denote that this class [inherits](#_Inheritance) functionality from a super class |  |
| Exception Handling | This is where the programmer assumes responsibility for any Exceptions that may occur in their code, as opposed to the JVM (which would give a trace, but would force the program to crash).  Elements   1. Try … Catch 2. Finally |  |
| Final | Keyword to denote that this variable is a [constant](#_Constant) |  |
| Finally | A block of code used in Exception Handling.  It is typically used to clean up any administrative needs e.g. close any open IO resources |  |
|  |  |  |
| Global Variable | Opposite to a [Local Variable](#_Local_Variable), a Global Variable has far more scope and can be referred to/ used in a class.  Depending on the level it’s access modifier, it may also be used in other classes. |  |
| Inheritance | The ability of an object to access the already existing attributes and behaviour of any of it’s parent classes.  This is extremely beneficial as it allows for the sharing and future development of code.  It is made possible by the use of the keyword [extends](#_Extends) in the class heading |  |
| Instantiation | Instantiation means create an object of a class.  An object class may have several constructors to allow for different means of creating the object with different amounts of information. | String x = new String(); |
| Interface | An interface is a very bear definition of an object.  Typically, the only items that appear in an interface are   1. [constants](#Constant) 2. abstract object methods.   An interface should not contain constructors | an interface is typically used to define the very bare functionality of an object.  They are used, like abstract classes, to force the programmer to implement their interpretation of the object’s functionality. |
| Local Variable | A local variable is a variable that exists only in a method, Including the main method. the scope of this variable is restricted to the method and method alone.  The opposite to a local variable is a [global variable](#_Global_Variable). |  |
| Main method | The main method is the default start point for every Java application. It originates from at that time programmers used the command prompt to compile and execute their code. |  |
| Member | A member is an item of a class. This maybe   1. A variable 2. A method |  |
| Method | A method is the Java term for a function. This contains specific code to do a task.  A method may have input arguments / parameters and may also have a return value. |  |
| Object | An object is either an item or process represented by a Java class. An object is needed as the importance of the object itself in the overall application makes it necessary to have its own set of   1. Attributes (Variables) 2. Behaviour | A student may need to be represented in a library.  Thus a class with the following could be created   1. Variables    1. Course    2. Year 2. Methods    1. BorrowBook    2. PayFine |
| Overloading | Overloading is where a method shares the same name as another method BUT has different input parameters  These parameters can differ in terms of   1. Type – One method could have ints, the copy could have a String as an input 2. Number of inputs 3. Order of inputs   A method may overload another method in the same class, which is not possible in [overriding](#_Overriding) |  |
| Overriding | Overriding is where a method Overloading is where a method shares the same name as another method and has the same input parameters  A method cannot override another method in the same class, as opposed to [overloading](#_Overloading)  Overriding is a key concept as it allows for [polymorphism](#_Polymorphism) |  |
| Parameters | These are inputs into a method / function e.g. two numbers that are then added together |  |
| Polymorphism | Polymorphism is the notion of a hybrid object, which is essentially a simpler object but with the specific, defined behaviour of one of it’s subclasses.  It is used when the programmer wishes to make limited use of a subclass and retain most of the properties of the parent class/  It is made possible via [inheritance](#_Inheritance) and [overriding](#_Overriding) |  |
| Return Values |  |  |
| Runtime Exceptions |  |  |
| Scope | Scope refers to the boundaries in which a variable can be used. It may be very localized in terms of a local variable but could be more expensive in terms of global variable. |  |
| Static | Static refers to either a variable or a method that can be used without creating / [instantiating](#Instantiation) an associated object |  |