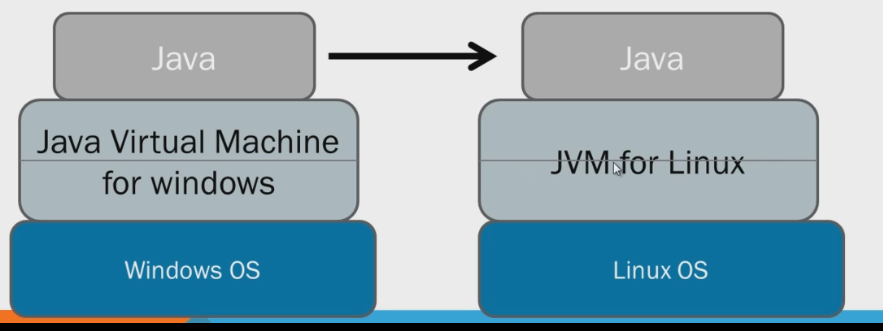
**Java is platform independent:**

* Java is independent of the platform on which it is running.
* After compiling java code gets converted to byte code (1s and 0s).This byte code can be run on any platform irrespective of the platform where it was compiled.
* JVM is platform specific but not the java code. JVM takes care of running java byte code.



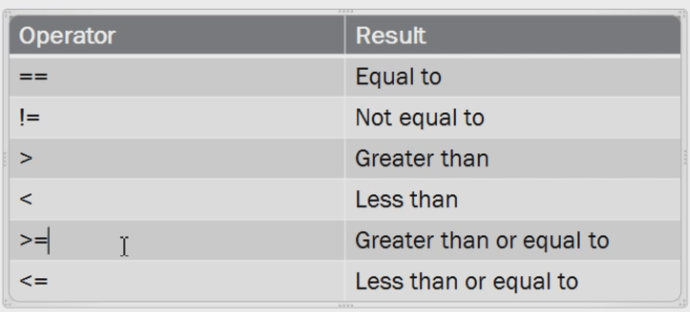
**JDK: Java development kit:**

* Includes JVM, Compiler and related files.

**Displaying Environment Variables in Windows**

* In Windows Command-Prompt the syntax is **echo %PATH%**
* To get a list of all environment variables enter the command **set**
* To send those variables to a text file enter the command **set > filename.txt**

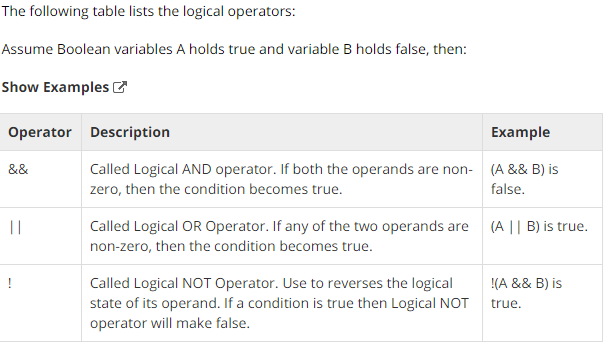
**Relational operators:**



**Arithmetic Operators:**



**Logical operators:**



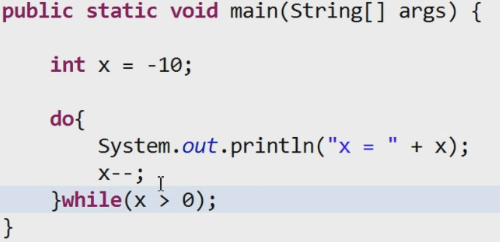
**Decision making:**

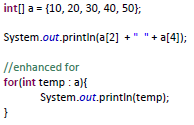
* If
* If else
* If else if
* Conditional Operator ( ? : ):
  + *b = (a > 10) ? 20: 30;*
* Switch case
  + In switch, when there is no case found then the default case will get executed.
  + The variable used in a switch statement can only be a byte, short, int, char, String



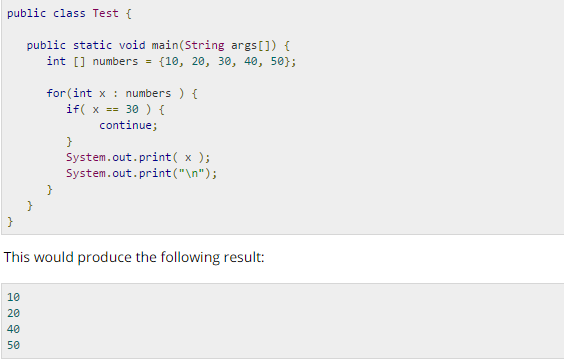
**Loops**

* For loop
* Enhanced for loop(for each loop)
* While loop
* Do while loop
* Difference between do and do-while statement
  + The do-while statement runs at least once. In the following case -10 will be printed once even though while condition is not matched.



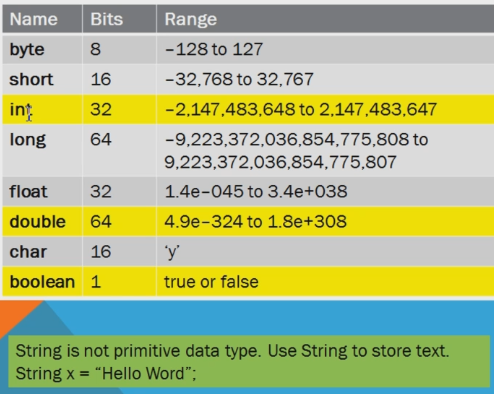


* **Continue:**
  + The continue keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

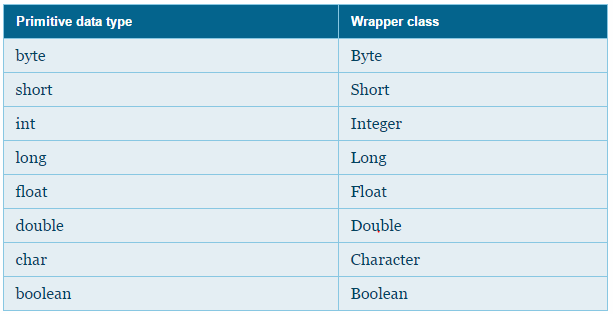


**Variable Data types:**

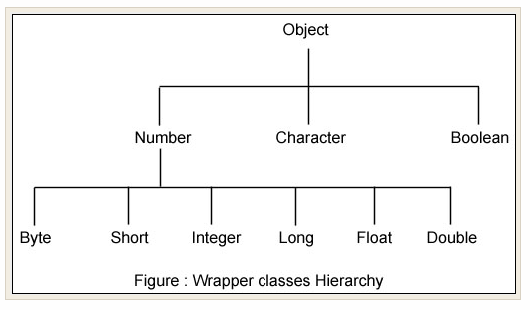
* There are two data types available in Java:
  + Primitive Data Types
  + Reference/Object Data Types
* **Primitive Data Types:**
  + Primitive data types are predefined by the language and named by a keyword.
  + Java has 8 primitive data types.



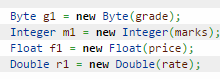
* **Wrapper classes:**
  + As the name says, a wrapper class wraps (encloses) around a data type and gives it an object appearance.
  + Every primitive data type has corresponding wrapper class.



* + Following is the hierarchy of the above **classes**.

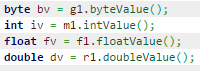


* + Creating objects of the Wrapper classes (Wrapping)

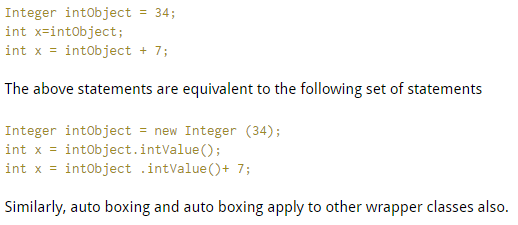




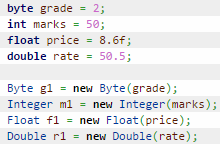
* + Retrieving the value wrapped by a wrapper class object(Unwrapping)



* + Auto wrapping and unwrapping.
    - Creating a wrapper class object using the constructors and retrieving the values wrapped by those objects using the methods as shown above can become quite cumbersome. As an alternative, there exists auto wrapping and unwrapping.
    - New wrapper object can be created by specifying the value to be wrapped just as we would do for a primitive data type variable. Also, the value can be retrieved and used in a simple way by specifying the object name. Look at the following code:



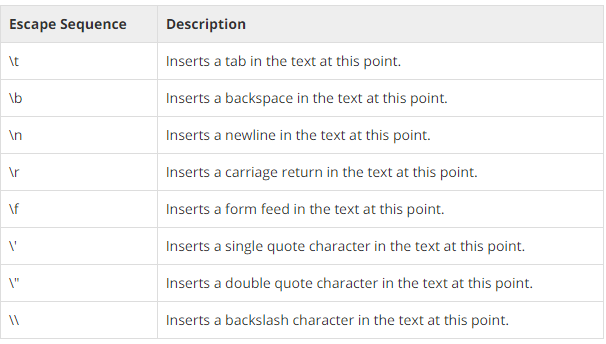
* + There are mainly two uses with wrapper classes.
    - To convert simple data types into **objects**, that is, to give object form to a data type; here constructors are used.



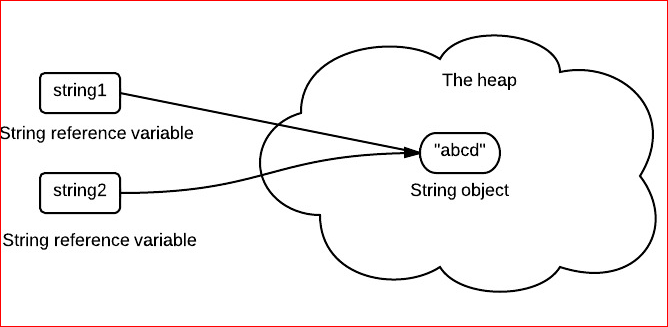
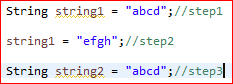
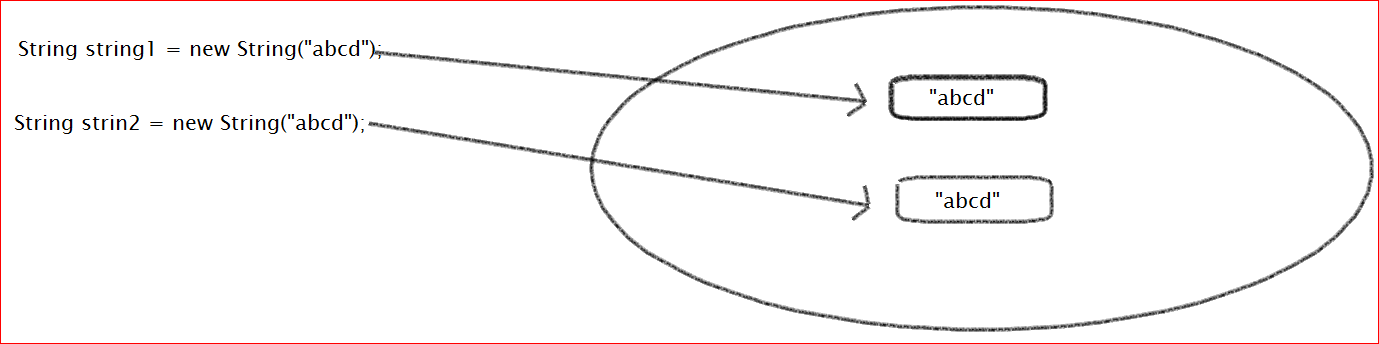
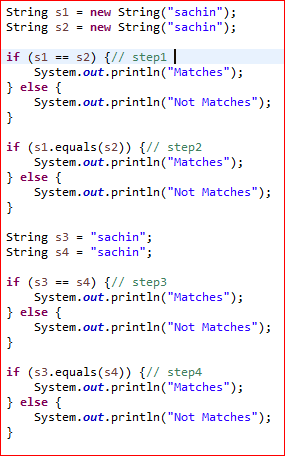
* + - To provide several utility methods which are required for primitive data types.
    - To convert strings into data types (known as parsing operations), here methods of type parseXXX() are used.



* + Utility methods in Wrapper classes
    - valueOf()
    - xxxValue()
      * To get primitive value of wrapper object.
    - parseXxx()
      * To convert string to primitive.
    - toString()
      * To convert primitive to string.
* **char data type.**
  + **char** is a primitive data type.
    - * 
  + It’s mandatory to use single quotes for characters. Double quotes are for strings
  + Array of Characters:
    - 
  + Its corresponding wrapper class is **Character**.
    - * 
* **Escape characters:**
  + A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler.



**Strings:**

* String holds a string of characters.
* String is an array of characters.
* Creating a string:
  + Using string literals:
    - *String name = "James Dean";*
  + Using “new” keyword
    - *String name = new String("James Dean");*
  + Both the methods create new String object. Here the string ‘name’ is object of String class.
* String is not a wrapper class
* Strings are Immutable. What does it mean?
  + String immutability comes into picture when we create a string using string literals and not when we create using a new keyword.
  + String pool (String intern pool) is a special storage area in Java heap. When a string is created and if the string already exists in the pool, the reference of the existing string will be returned, instead of creating a new object and returning its reference.
  + The following code will create only one string object in the heap.
  + 
  + 
  + Here “abcd” is a string object and string1 and string2 are references to this object.
  + Here object “abcd” is immutable and the references string1 and string2 are immutable.
  + In the following example:
    - 
    - In step1 ,a new object “abcd” gets created and string1 is reference to it
    - In step2, a new object “efgh” gets created and string1 dereferences “abcd” and references ”efgh”. In this step we are not modifying the “abcd” object that got created in step1. After step2 there are two objects in the string pool.
    - In step 3, we are not creating “abcd” object again.string2 references to already existing “abcd” object which is already present in string pool
    - This example explains that the object “abcd” once created couldn’t be changed. Only thing we can do is create a new object and reference it.
  + What is the Use of string immutability?
    - For proper memory management.
* When string objects are created using a new keyword, this is how memory is allocated to them. In the following example though both the objects have the same string value they are two different objects unlike above scenario(where strings are created using string literals)
  + 
* The following example:
  + Strings s1 and s2 are created using new keyword. So s1 and s2 refer to two different objects.
  + Strings s3 and s4 are created using string literals. So objects s3 and s4 reference the same object.
  + Step1 prints “Not Matches” as “==” compares the objects and the values.
  + Step2 prints “Matches” as “equals()” method compares values of the objects and not the objects.
  + Step 3 and Step 4 print ‘Matches’ as both s3 and s4 reference the same object and not two different objects.
  + 
  + **StringBuilder and StringBuffer:**
    - Since String is immutable in Java, whenever we do String manipulation like concatenation, substring etc, it generates a new String and discards the older String for garbage collection.
    - These are heavy operations and generate a lot of garbage in the heap. So Java has provided StringBuffer and StringBuilder class that should be used for String manipulation.
    - StringBuffer and StringBuilder are mutable objects in java and provide append(), insert(), delete() and substring() methods for String manipulation.
  + **When to use which one :**
    - If a string is going to remain constant throughout the program, then use a String class object because a String object is immutable.
    - If a string can change (example: lots of logic and operations in the construction of the string) and will only be accessed from a single thread, using a StringBuilder is good enough.
    - If a string can change, and will be accessed from multiple threads, use a StringBuffer because StringBuffer is synchronous so you have thread-safety.
  + To concatenate :
    - String :
      * str = str + “anotherstring”
      * str = str.concat(“anotherstring”)
    - StringBuilder
      * sb = sb.append(“anotherstring”)
    - StringBuffer
      * sb = sb.append(“anotherstring”)

**Variable Types:**

* There are three kinds of variables in Java:
  + Local variables
  + Instance variables
  + Class/static variables
* **Local variables:**
  + Variables defined inside methods, constructors or blocks are called local variables.
  + The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
  + Access modifiers cannot be used for local variables.
  + There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.
  + Local variables are implemented at stack level internally.
* **Instance variables:**
  + Instance variables are variables within a class but outside any method.
  + Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
  + Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
  + Access modifiers can be given for instance variables.
  + Normally, it is recommended to make these variables private (access level).
  + Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null.
  + Values can be assigned during the declaration or within the constructor.
  + Instance variables can be accessed directly by calling the variable name inside the class in which they are declared.
* **Class variables/Static Variables:**
  + Class variables are variables declared within a class, outside any method, with the static keyword.
  + There would only be one copy of each class/static variable per class, regardless of how many objects are created from it.
  + Static variables are stored in static memory.
  + Static variables are created when the program starts and destroyed when the program stops.
  + Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
  + Static variables can be accessed by calling with the class name .ClassName.VariableName.
  + When declaring class variables as public static final, then variable names (constants) are all in uppercase. If the static variables are not public and final the naming syntax is the same as instance and local variables.
  + A local variable cannot be declared static.

**Static:**

* Whatever **not static** data variables and methods are one per object not one per class.
* Static is one per class and not one per object.
* Static methods can only access other static methods and static variables.
* Even before an object of a class gets created, even before a constructor gets fired, a static block of code gets fired.
* If we create multiple objects of a class, constructors get called once per each object but the static block of code is only fired once before any object gets created.
* Static methods can be directly called using class names.
  + **<class\_name>.<static\_method>()**





* In the above example you can see that a fresh copy of a non-static variable is created per each object but only one copy of the static variable is created per class which is used by all objects. It is evident that when object2 is called ,static variable its value is 3 but not zero.

**Why is the main method is Static?**

* *public static void main(String args[])*
* Java program processing starts from the main() method which is a mandatory part of every Java program.
* **Main** method is the entry point for execution.
* **Public**: because it should be accessible from everywhere
* **Void**: Because it doesn’t return anything.
* **Static**:
  + There should be only one entry point of execution. Since static is one per class even though multiple objects are created for the class containing the main method ,only one main method exists among all objects.
  + If the main method is not declared static, when multiple objects are created for the class containing the main method, multiple main methods can be called one per object which means multiple entry points of execution or multiple main methods running at time which is not acceptable.
* **(String args[]):**
  + Sometimes we might want to pass information into a program when you run it. This is accomplished by passing command-line arguments to main( ).
  + A command-line argument is the information that directly follows the program's name on the command line when it is executed. They are stored as strings in the String array passed to main( )

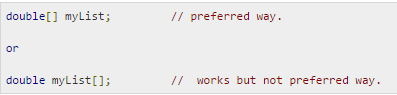


**Final:**

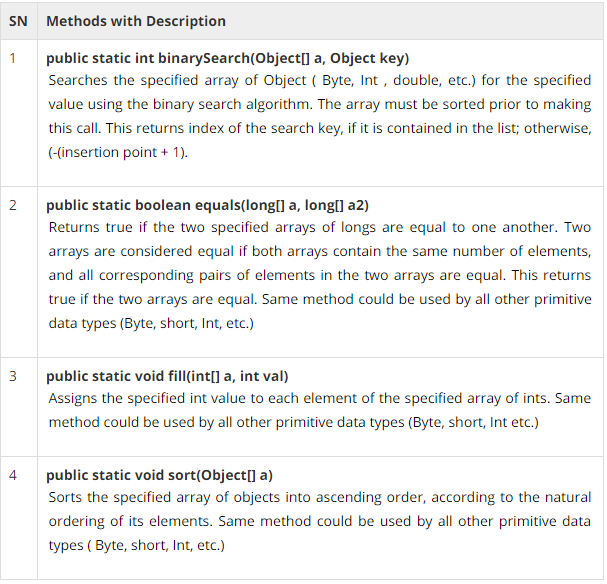
* **final** keyword can be applied to data, methods & classes.
* **final** data is a constant. Cannot be changed.
* **final** methods cannot be overridden.
* **final** classes cannot be subclassed.

**Arrays:**

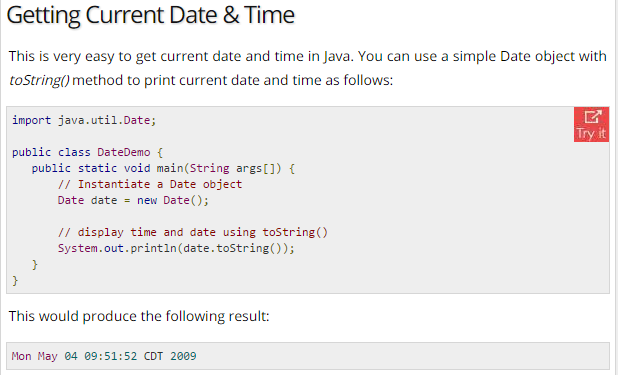
* Consists of fixed-size homogeneous data.
* For int array default value is zero
* For String array default value is null
* Declaring an array :
  + *Int[] a = new int[5]*
  + *String[] s= new String[4]*
  + *int[][] TwoDim = new int[4][3];*



* The Arrays Class:
  + The **java.util.Arrays** class contains various **static** methods for sorting and searching arrays, comparing arrays, and filling array elements. These methods are overloaded for all primitive types.
  + These are readymade methods.



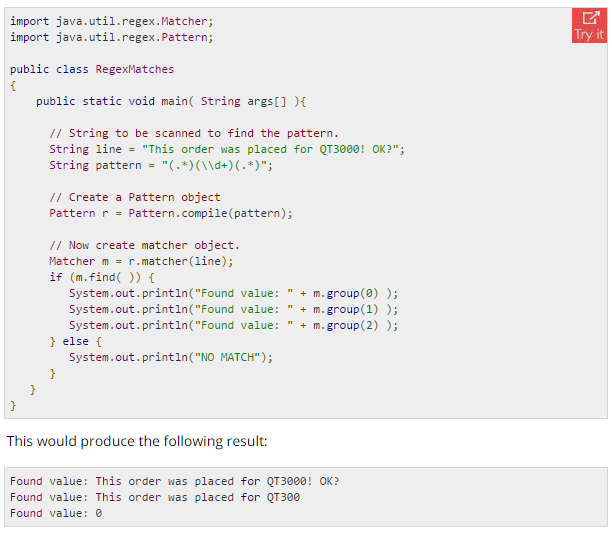
**Date Class:**

* The java.util.Date class represents a specific instant in time, with millisecond precision
  + - 
* **Sleeping for a While:**
  + You can sleep for any period of time from one millisecond up to the lifetime of your computer. For example, following program would sleep for 10 seconds:
    - 
* **Measuring Elapsed Time:**
  + Sometimes, we may need to measure point in time in milliseconds. So let's re-write above example once again:

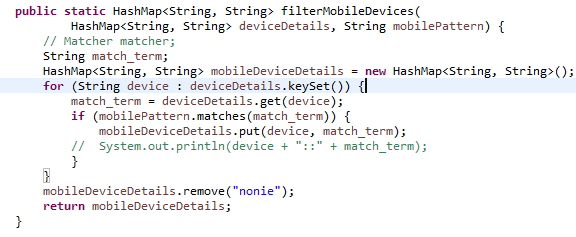


**Regular Expressions:**

* Java provides the java.util.regex package for pattern matching with regular expressions.
* The java.util.regex package primarily consists of the following three classes:
  + **Pattern** Class:
    - A Pattern object is a compiled representation of a regular expression
  + **Matcher** Class:
    - A Matcher object is the engine that interprets the pattern and performs match operations against an input string
  + **PatternSyntaxException**:
    - A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.



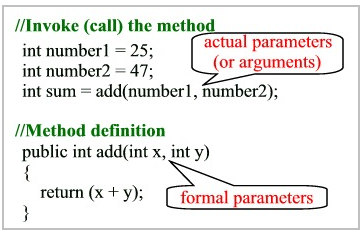
* Pattern matching can also be done using **String.matches**



* [What's the difference between String.matches and Matcher.matches?](http://stackoverflow.com/questions/2469244/whats-the-difference-between-string-matches-and-matcher-matches)
  + A Matcher is created on on a precompiled regexp, while String.matches must recompile the regexp every time it executes, so it becomes more wasteful the more often you run that line of code.

**Java Methods:**

* A Java method is a collection of statements that are grouped together to perform an operation.
* Formal and Actual parameter:
  + Parameters passed during method declaration are formal parameters
  + Parameters passed during calling method are actual parameters.



* Java supports only pass by value .It doesn’t support pass by reference.

**Class:**

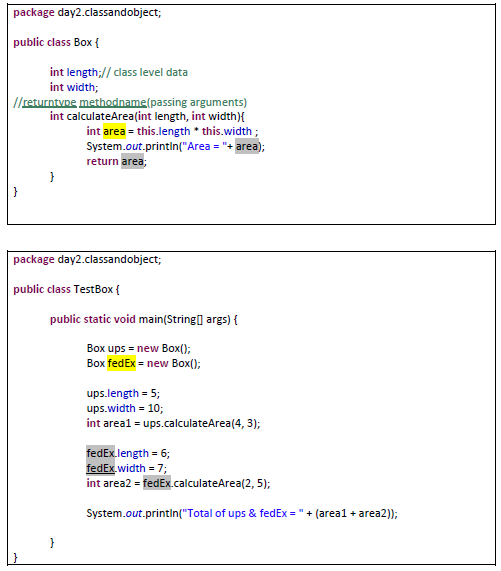
* + Is a template for an object.
  + Class contains data(variables) and methods which act on data.
  + Every class extends **Object** class by default. This is by default in Java.

**Object:**

* + Object is a copy or instance of a class.

**This:**

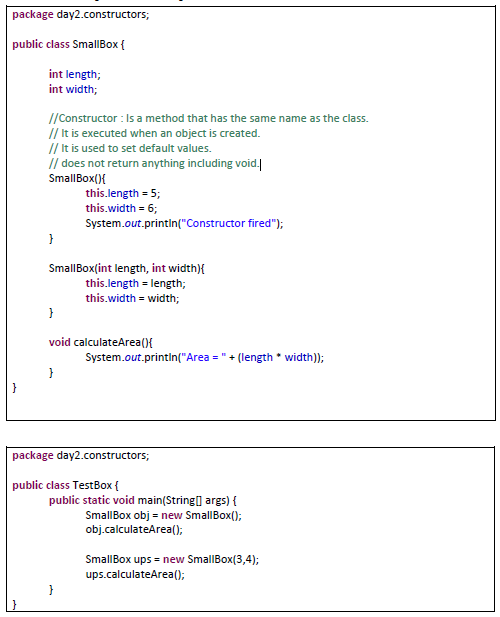
* To access class level data



* In the above example though parameters are passed in calculateArea(4,3) method, class level data(5,10) will be considered for calculation.

**Constructor:**

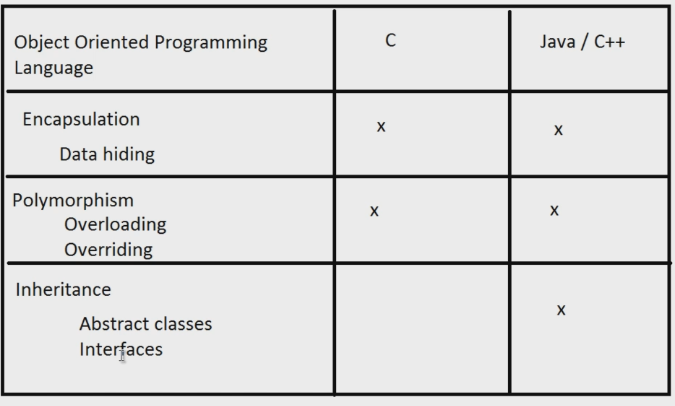
* Constructor is a method that has the same name as the class.
* It is executed when an object is created.
* It is used **to set default values.**
* It does not return anything including void.
* Every class has a constructor. If we do not explicitly write a constructor for a class the Java compiler builds a default constructor for that class.



* The above class has two constructors one with arguments and another without arguments. So when creating an object constructor is called based on number of arguments.

**OOPS**

* A programming language is called Object oriented if it satisfies the following features
  + Object
  + Class
  + Inheritance
  + Encapsulation(Data hiding)
    - Achieved using Access Modifiers
  + Polymorphism
    - Overloading
    - Overriding
  + Abstraction
    - Abstract classes
    - Interfaces



**Encapsulation:**

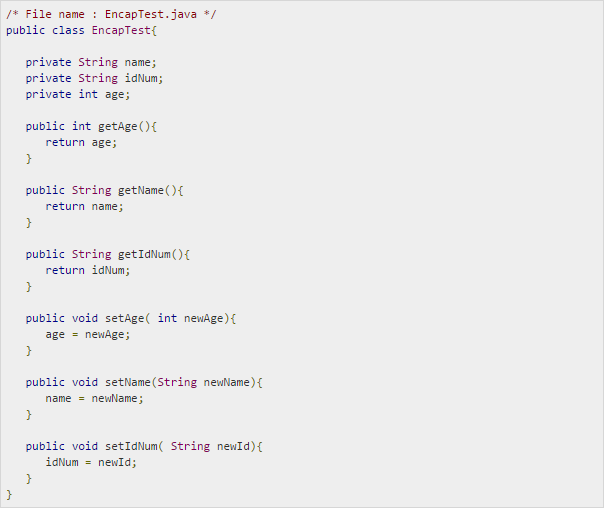
* Encapsulation is capsuling data and methods which act on the data.
* Encapsulation is also referred to as data hiding.
* Data hiding:

Data hiding is basically not hiding data .It about protecting data from accidental manipulation.

* Data hiding is achieved using access modifiers.



* As a general rule, always declare data as private and use public getters and setters.
* As a general rule, always declare data as private and methods as public.



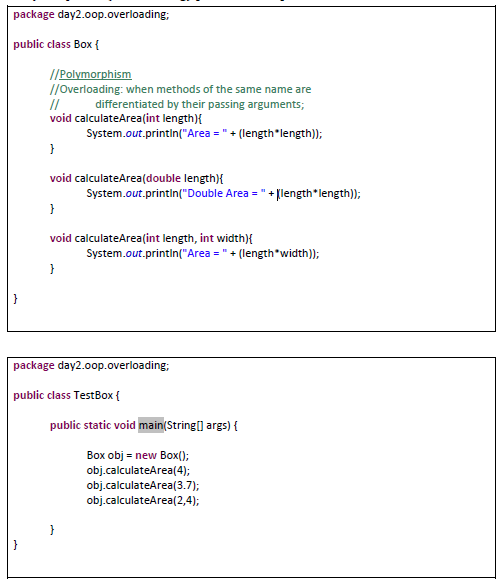
* The public methods are the access points to this class' fields from the outside java world. Normally, these methods are referred to as getters and setters. Therefore any class that wants to access the variables should access them through these getters and setters.



* Benefits of Encapsulation:
  + The fields of a class can be made read-only or write-only.
  + A class can have total control over what is stored in its fields.
  + The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

**Polymorphism: Overloading**

* When methods of the same name are differentiated by their passing arguments.

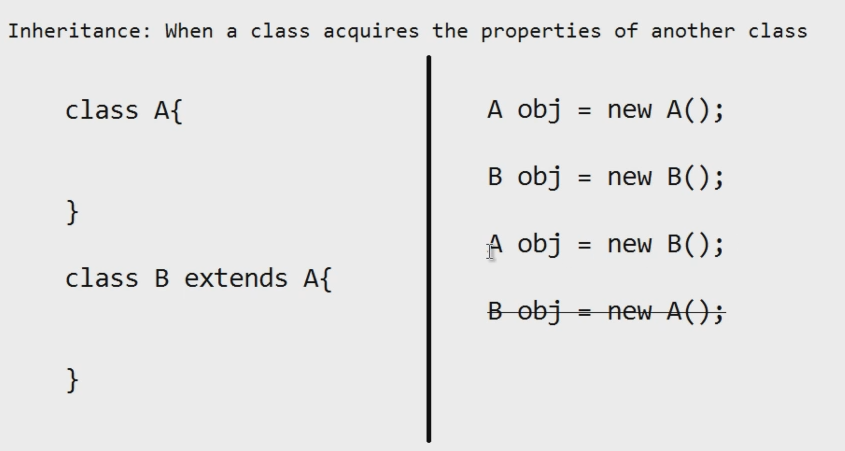


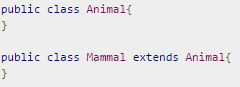
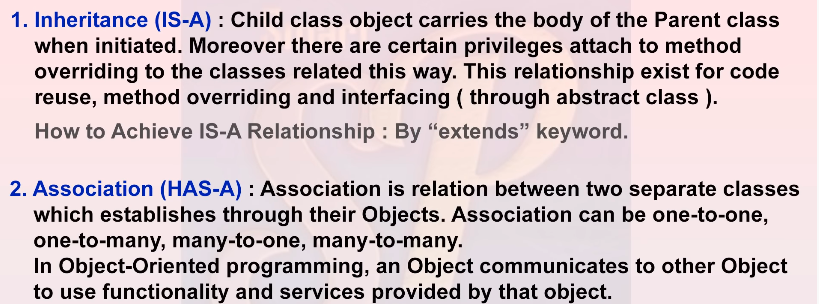
**Polymorphism: Overriding**

* When a method in subclass has same signature (name and parameters) then the method in subclass takes precedence over the method present in parent class

**Inheritance:**

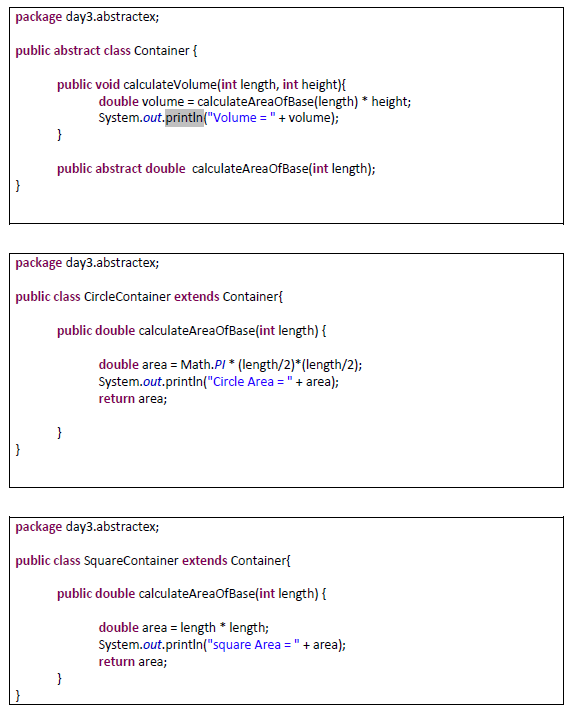
* A class acquires methods and data (variables) of another class.
* When to inherit a class:
  + When I want to reuse methods of some other class
  + I don’t have source code for parent class
  + Parent class belongs to some other project

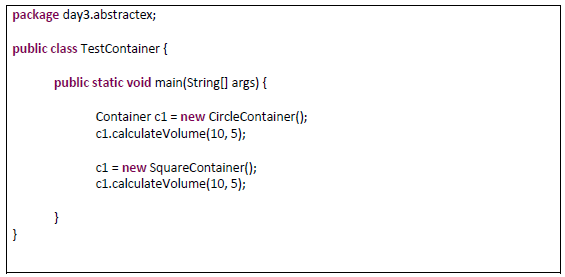


* Java only supports single inheritance. This means that a class cannot extend more than one class. Therefore following is illegal:
  + - 
    - Child can not have more than one parent
* However, a class can implement one or more interfaces. This has made Java get rid of the impossibility of multiple inheritances.
* IS-A Relationship:
  + IS-A is a way of saying: This object is a type of that object.
  + 
  + We can say that Mammal IS-A Animal
  + 
  + HAS -A relationship is achieved by creating an object of a class in another class using ‘new’ keyword

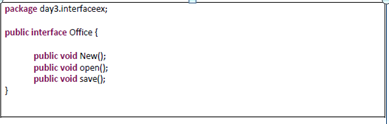
**Abstract class:**

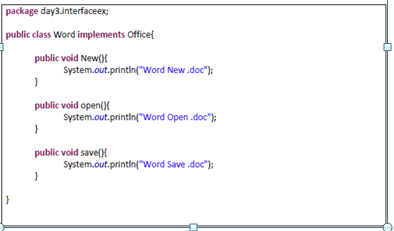
* A class which has empty methods and fully defined methods.
* Abstract class cannot be instantiated.
* A Method must be declared abstract if there is no implementation for it.
* A class having abstract methods should be declared abstract.
* A child class that inherits an abstract method must override (implement) it. If they do not, they must be abstract and any of their children must override it.
* Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.
* **Concrete Class:** A derived class that implements all the missing functionality of abstract class is called a *concrete class**.*

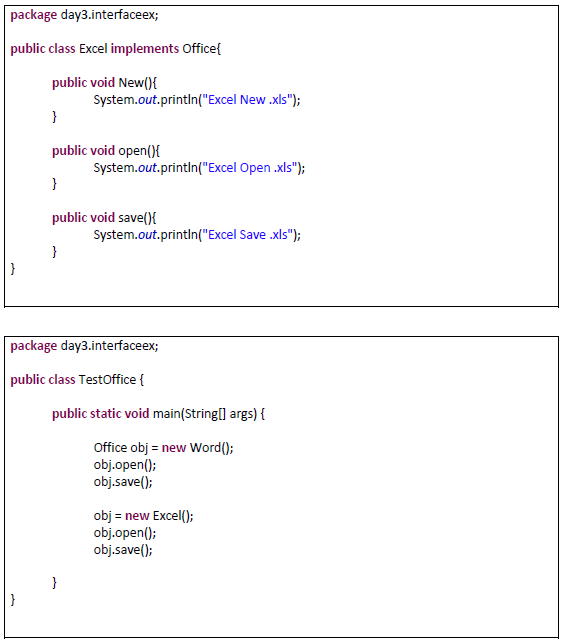




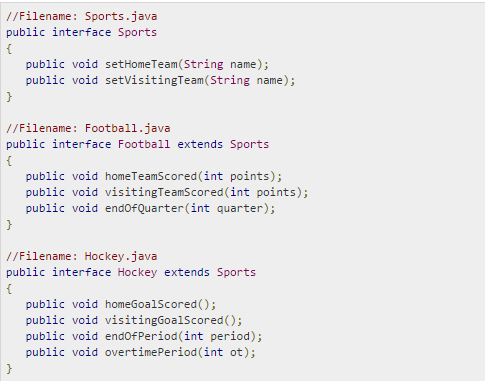
**Interface:**

* A class with only empty (or abstract) methods.(*From Java 8, an Interface can contain implemented methods which should be declared using default/static keywords*)
* Used to bring consistency/uniformity across various classes.
* An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts.
* When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviours of the interface.
* An interface is similar to a class in the following ways:
  + An interface can contain any number of methods.
  + An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
  + The bytecode of an interface appears in a .class file.
  + Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.
* However, the interface is different from a class in several ways, including:
  + You cannot instantiate an interface.
  + An interface does not contain any constructors.
  + All of the methods in an interface are abstract.
  + An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
  + An interface is not extended by a class; it is implemented by a class.
  + An interface can extend multiple interfaces.
* Interfaces have the following properties:
  + An interface is implicitly abstract. You do not need to use the abstract keyword when declaring an interface.
  + Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
  + Methods in an interface are implicitly public.
* Implementing Interfaces:
  + A class extends another class
  + A class extends an abstract class
  + A class **implements** an interface.
  + If a class does not implement all the methods of the interface, the class must declare itself as abstract. A child class that inherits an abstract method must override (implement) it. If they do not, they must be abstract and any of their children must override it.
    - 

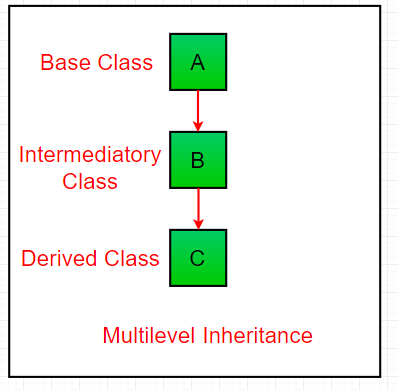
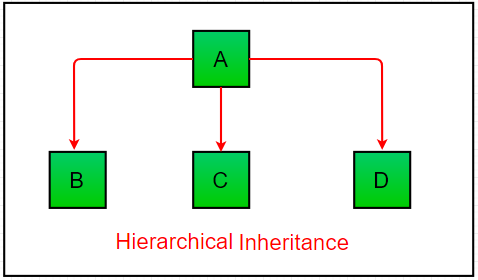
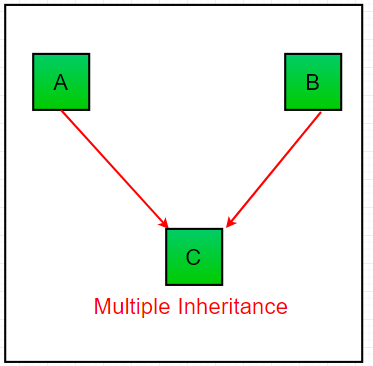
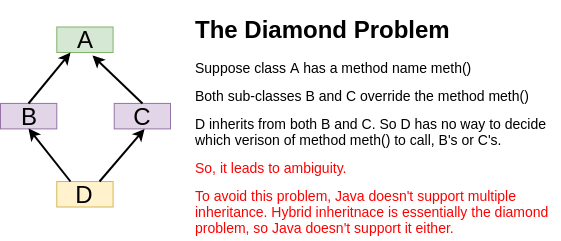


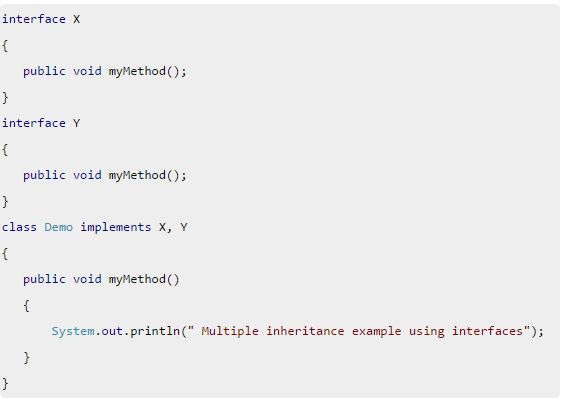


* Extending Interfaces:
  + An interface can extend another interface.



**Multiple Inheritance in Java:**

* This is Allowed:
  + In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B
  + 
* This is Allowed:
  + In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class.
  + In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C
  + 
* This is allowed:
  + Multiple kids to a parent is allowed
  + one class serves as a superclass (base class) for more than one sub class.In below image, the class A serves as a base class for the derived class B,C and D.
  + 
* This is Not allowed:
  + One kid with multiple parents is not allowed
  + In Multiple inheritance ,one class can have more than one superclass
  + Java does not support [multiple inheritance](https://www.geeksforgeeks.org/java-and-multiple-inheritance/) with **classes**.
  + In Java, we **can** achieve multiple inheritance only through [**Interfaces**](http://quiz.geeksforgeeks.org/interfaces-in-java/).
  + In the image below, Class C is derived from interface A and B.
  + 
  + Why Java doesn’t support multiple inheritance?
  + Multiple inheritance can cause ambiguity in few scenarios. One of the most common scenarios is Diamond problem.
  + What is the diamond problem?
    - Consider the below diagram which shows multiple inheritance as Class D extends both Class B & C.
    - 
* **How is Multiple Inheritance achieved?**
  + By Implementing Multiple Interfaces:
  + By Extending Multiple Interfaces:
* Implementing Multiple Interfaces:
  + A class can implement multiple interfaces but a class can extend only one class.
  + This is how multiple inheritance is achieved in Java.



* Extending Multiple Interfaces:
  + A Java class can only extend one parent class. Multiple inheritance is not allowed.
  + An interface can extend more than one parent interface.
  + For example, if the Hockey interface extended both Sports and Event, it would be declared as:
    - 

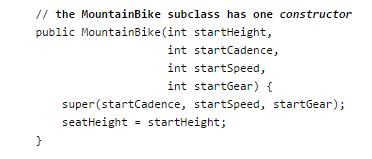
**Super:**

* Super keyword has three uses:
  + super is used to refer to the immediate parent class instance variable.
  + super is used to invoke the immediate parent class method.
  + super() is used to invoke the immediate parent class constructor.
* To access super class methods and data.
  + If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword super.
  + This is useful only in case of method overriding.



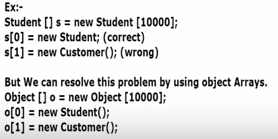
* To invoke immediate parent class constructor

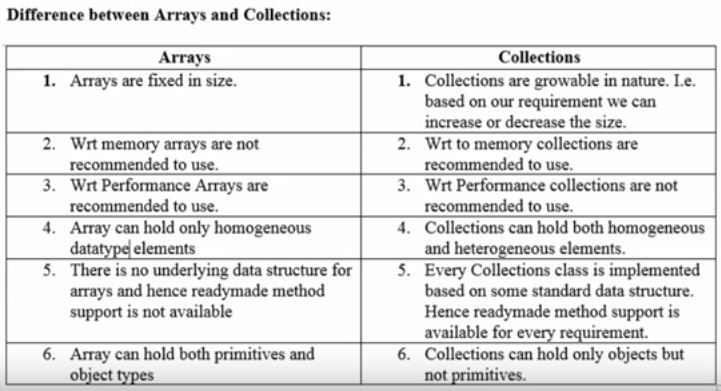




* Invocation of a superclass constructor must be the first line in the subclass constructor.
* With super(), the superclass no-argument constructor is called. With super(parameter list), the superclass constructor with a matching parameter list is called.

**Collections:**

* Limitations of Arrays:
  + Arrays are of fixed length. We cannot add additional entries in it.
  + Arrays can hold only homogeneous data only.
    - 
  + Readymade methods are not available. We need to write code explicitly. For example, sorting or searching for an element in an array doesn’t have ready-made methods. We need to write code for that.
* Collections provide solutions to overcome limitations of Arrays.
  + Collections are Growable in nature.
  + Collections can hold both homogeneous and heterogeneous data.
  + Every collection is implemented based on some standard data structure. Hence readymade support is available.

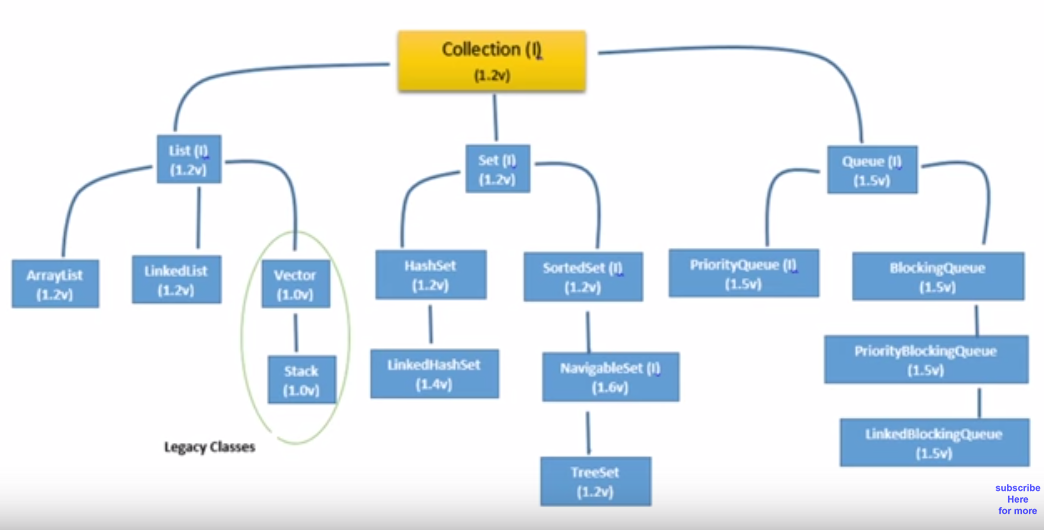


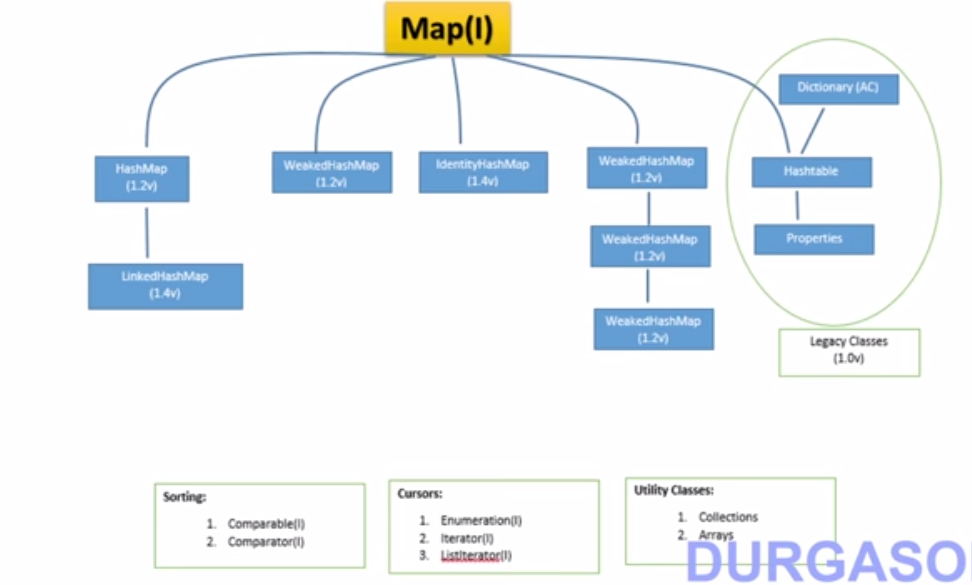
* Collection Framework:
  + Framework which defines several interfaces and classes required to implement collections.
  + Cursors
  + Utility classes.
  + Sorting

Collection Interfaces:

Collections framework has 9 interfaces. These 9 interfaces can be grouped into 2 groups. To one group of interfaces ‘Collection’ interface is the root interface and to another group of interfaces ‘Map’ interface is the root interface.

* There are different types of collections.
  + ArrayList
  + HashMap
  + LinkList
  + Set
  + Map
* Map Interface is not a child interface of Collection interface even h.
* Collection is not the root interface of collections framework since Map interface is not the child interface of Collection interface.





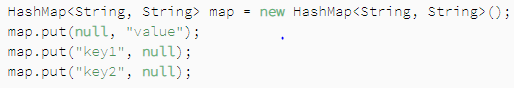
**Non generic list:**

**Can add anything irrespective of data type to list.**

**Generic list**

**Can add only specific data type to list.**

****

* **ArrayList:**
  + ArrayList implements List Interface.
  + ArrayList<Integer> arrayListName = new ArrayList(Integer);
    - OR
  + List<Integer> arrayListName = new ArrayList(Integer);
  + Note: Primitive data types(int,str) should not be passed as arguments but classes should be passed (Integer,String)
* **HashSet:**
  + Implements Set interface
  + Very similar to ArrayList
* **Difference between List and Set:**
  + List is an ordered collection it maintains the insertion order, Where as Set doesn't
  + List can have duplicates but Set can not
  + List can have any number of null values but Set can have only one null value
* **HashMap**
  + Implements Map Interface
* **Hashtable** 
  + Implements Map Interface
* **Difference between HashMap and Hashtable** 
  + HashMap is non synchronized and not thread safe whereas hashtable is synchronized
  + HashMap allows adding one Entry with null as key as well as many entries with null as value
    - 
  + Hashtable doesn't allow null at all
    - Neither null key nor null values are allowed in Hashtable
    - If key is null or value is null, Null pointer exception is thrown
* **When to Choose HashMap Over Hashtable?**
  + We should use HashMap for an unsynchronized or single threaded application.
  + It is worth mentioning that since JDK 1.8, Hashtable has been deprecated. However, [ConcurrentHashMap](https://www.baeldung.com/java-concurrent-map) is a great Hashtable replacement. We should consider ConcurrentHashMap to use in applications with multiple threads.

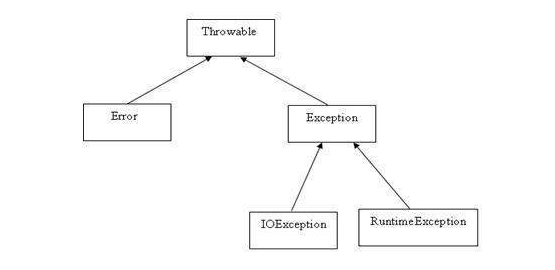
**Exception Handling:**

**Types of Errors:**

* + **Compile time errors:** 
    - These are syntactical errors. They are not called exceptions
  + **Runtime errors:**
    - These are called exceptions.
    - Exceptions are always runtime. They are never at compile time

**Exception Hierarchy:**

* **Exception** class and **Error** classes are subclasses of Throwable class.
* The Exception class has two main subclasses: IOException class and RuntimeException Class.



* **Exceptions:**
  + An unexpected event which disturbs the normal flow of execution.
  + Exceptions are recoverable
  + Example:
    - File not found exception. If a file is not found we can make code to go with the default file.
* **Errors**:
  + These are not exceptions at all, but the problems that arise beyond the control of the programmer.
  + Errors are not recoverable.
  + Errors are typically ignored in your code because you can rarely do anything about an error.
  + Example,
    - Out of memory error
    - stack overflow

**Try-Catch**

* If ever we feel a block of code may throw some error then surround it by try-catch block.
* Exception can be of two types:
  + Checked exception(Compile time exceptions)
  + Runtime exception
* We can have multiple catch one for each type of exception.
  + *(Exception e)* => For all types of exceptions.
  + *(ArrayIndexOutOfBoundsException e)*
  + *(ArithmeticException e)*

**Try-Catch-Finally:**

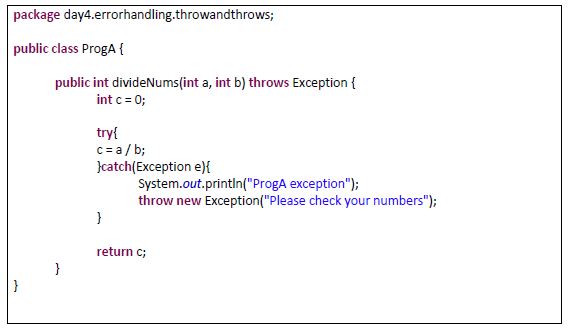
* Whether there is an exception or not ,”Finally” block of code gets executed.
* “Finally” is like a door to get out of a try catch block.
* “Finally” block mostly used to close connections

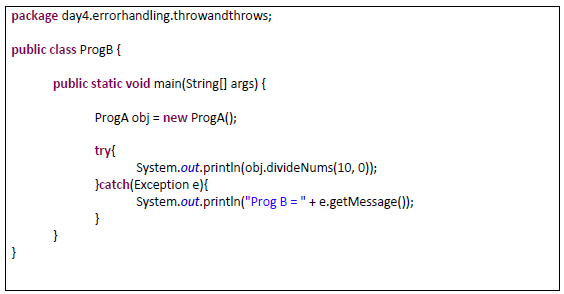


**Exception handling: Throw and Throws.**

**Throws:**

* + An exception occurred in one program is thrown to another program using ‘throws’.
  + Suppose program B is calling a method in program A. If there is a chance of exception in the method in program A, then program A has to throw that exception back to calling program (program B), Program B has to catch that exception.
  + Program A should mention “throws Exception” and should “throw” back that exception.
  + Program B should catch that exception.

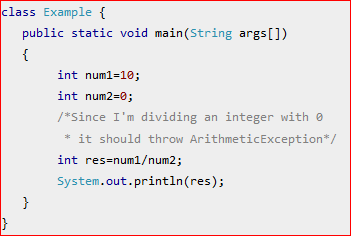




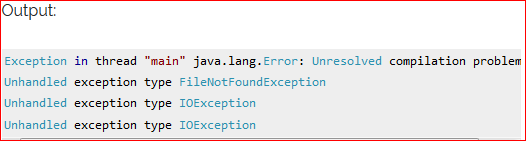
* + A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas.



**Throw:**

* + **throw** keyword is used to throw an exception explicitly.
  + Only objects of the Throwable class or its subclasses can be thrown.
  + Program execution stops on encountering throw statement, and the closest catch statement is checked for matching type of exception.
* **Checked Exceptions and Unchecked Exceptions**:
  + **Important thing here is, exception occurs only at run time** and the errors which occur during compile time are not called exceptions. Unchecked and Checked exceptions are categorized based on whether they are checked during compile time or not. Just that they are checked during compile time and not that they occur during compile time.
  + **Unchecked exceptions**:
    - These are not checked at compile time. It means if your program is throwing an unchecked exception and even if you didn’t handle/declare that exception, the program won’t give a compilation error.
    - Most of the times these exception occurs due to the bad data provided by user during the user-program interaction. It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately.
    - All Unchecked exceptions are direct subclasses of RuntimeException class.
      * 
    - If you compile this code, it would compile successfully however when you run it, it would throw ArithmeticException. That clearly shows that unchecked exceptions are not checked at compile-time, they are being checked at runtime.
    - Here are the few most frequently seen unchecked exceptions –
      * NullPointerException
      * ArrayIndexOutOfBoundsException
      * ArithmeticException
      * IllegalArgumentException
  + **Checked exceptions**:
    - Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using [try-catch block](http://beginnersbook.com/2013/04/try-catch-in-java/) or it should declare the exception using [throws keyword](http://beginnersbook.com/2013/04/difference-between-throw-and-throws-in-java/), otherwise the program will give a compilation error.
    - It is named as **checked exception** because these exceptions are **checked** at Compile time.
    - Let’s understand this with this **example**: In this example we are reading the file **myfile.txt** and displaying its content on the screen. In this program there are three places where a checked exception is thrown as mentioned in the comments below. FileInputStream which is used for specifying the file path and name, throws FileNotFoundException. The read() method which reads the file content throws IOException and the close() method which closes the file input stream also throws IOException.

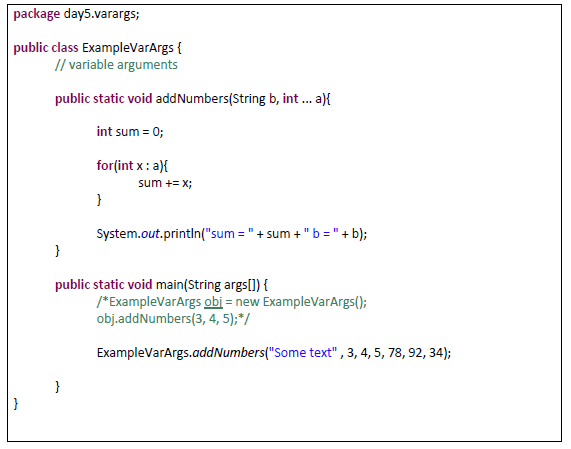




* + - **Why this compilation error?** As mentioned in the beginning that checked exceptions gets checked during compile time. Since we not handled/declared the exceptions, our program gave the compilation error.
    - **How to handle this exception?**
      * Using try –catch
      * Using throws
    - Here are the few other Checked Exceptions –
      * SQLException
      * IOException
      * DataAccessException
      * ClassNotFoundException
      * InvocationTargetException

**Variable arguments:**

* If we want to pass variable number of parameters, then there is a way to achieve this using “…a”.



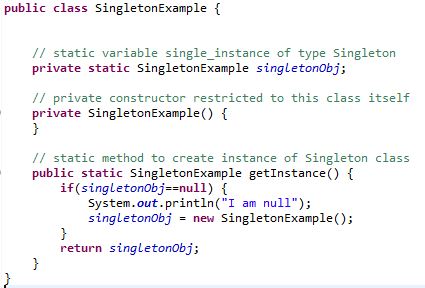
**Java Package:**

* In simple, it is a way of categorizing the classes and interfaces.
* When developing applications in Java, hundreds of classes and interfaces will be written, therefore categorizing these classes is a must as well as makes life much easier.

**Import statements:**

* In Java if a fully qualified name, which includes the package and the class name, is given then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class.
* For example, the following line would ask compiler to load all the classes available in directory java\_installation/java/io :
  + **import java.io.\*;**

**Singleton Class**

* Singleton is a design pattern. There is no keyword called ‘Singleton’
* In object-oriented programming, a singleton class is a class that can have only one object (an instance of the class) at a time.
* After the first time, if we try to instantiate the Singleton class, the new variable also points to the first instance created. So whatever modifications we do to any variable inside the class through any instance, it affects the variable of the single instance created and is visible if we access that variable through any variable of that class type defined.
* To design a singleton class:
  + Make constructor as private.
  + Write a static method that has the return type object of this singleton class.
* Normal class vs Singleton class:
  + Difference in normal and singleton class in terms of instantiation is that, For normal class we use constructor, whereas for singleton class we use getInstance() method
* 
* **Note** : Don't get confused singleton object with static variable
  + static variable is one per class irrespective of the number of objects
  + But Singleton class can have only one object. You can not create multiple objects for singleton class

**Pass By Value/ Pass By reference**

* <https://blog.penjee.com/passing-by-value-vs-by-reference-java-graphical/>