Java Development Kit (**JDK**) is Kit which provides the environment to **develop and execute(run)** the Java program. JDK is a kit(or package) which includes two things

1. Development Tools(to provide an environment to develop your java programs)
2. JRE (to execute your java program).

Note : JDK is only used by Java Developers. The Java Software Development Kit (Java SDK) is the JRE plus the Java compiler, and a set of other tools. It converts source file (.java) to bytecode code (.class file).

Java Runtime Environment (JRE):

JRE is an installation package which provides environment to **only run (not develop / compile)** the java program(or application) onto your machine. JRE is only used by them who only wants to run the Java Programs i.e. end users of your system.

Java Virtual machine (JVM):

JVM is a very important part of both JDK and JRE because it is contained or inbuilt in both.

The Java Virtual Machine is like a computer. It can execute Java bytecode just like a PC can execute assembler instructions.

The Java Virtual Machine is implemented for several different operating systems, like Windows, Mac OS, Linux, IBM mainframes, Solaris etc. Thus, if your Java program can run on a Java Virtual Machine on Windows, it can normally also run on a Java Virtual Machine on Mac OS or Linux. Sometimes there are a OS specific issues that make your applications behave differently, but most of the time they behave very much alike. Sun referred to this as "Write once, run anywhere".

The Java Virtual Machine is a program itself. You start up the JVM and tell it what Java code to execute. This is typically done via a command line interface (CLI), like e.g. bash, or the command line interface in Windows. On the command line you tell the JVM what Java class (bytecode) to execute. Whatever Java program you run using JRE or JDK goes into JVM and

JVM is responsible for **executing the java program line by line** hence it is also known as interpreter.

Java Byte Code:

Java programs written in the Java language are compiled into Java bytecode which can be executed by the Java Virtual Machine.
The Java bytecode is stored in binary .class files.

Java APIs:

The Java language enables you to package components written in the Java language into APIs (Application Programming Interfaces) which can be used by others in their Java applications. Java comes bundled with a lot such components. These components are known as the standard Java APIs . These APIs enable your Java programs to access the local file system, the network and many other things.

The standard Java APIs provide a lot of basic functionality which you would otherwise have had to program yourself. Thus, the APIs help you develop your applications faster.

The standard Java APIs are available to all Java applications. The standard Java APIs come bundled with the Java Runtime Environment (JRE) or with the Java SDK which also includes a JRE.

Ref: http://tutorials.jenkov.com/java/what-is-java.html#language

---

Java syntax:

A Java file can contain the following elements:

- Package declaration
- Import statements

- Type declaration
    - Fields
    - Class initializers
    - Constructors
    - Methods

```
package javacore;
import java.util.HashMap;
public class MyClass {
   protected final String name = "John";
   {
      //class initializer
   }

   public MyClass() {
   }

   public String getName() {
      return this.name;
   }

   public static void main(String[] args) {
   }
}
```

Project ??

A Java project is a collection of Java files (and possibly other files) that belong together in a project. For instance, a larger Java program may consist of multiple (hundreds or thousands) of Java files.

Package ??

Java packages are a mechanism to group Java classes that are related to each other, into the same "module" (package). When a Java project grows bigger, for instance an app or API, it is useful to split the code into multiple Java classes, and the classes into multiple Java packages.

When you divide classes into multiple Java packages, it becomes easier to figure out where a certain class you are looking for is.

A Java package is like a directory in a file system. In fact, on the disk a package is a directory. All Java source and class files of classes belonging to the same package are located in the same directory.

Java packages can contain subpackages. Java packages can thus make up what is called a package structure. A Java package structure is like a directory structure. Its a tree of packages, subpackages and classes inside these classes. A Java package structure is indeed organized as directories on your hard drive, or as directories inside a zip file (JAR files).

Here is a screenshot of an example Java package structure:

At the top you see a directory called "src". This is the source root directory. It is not a Java package itself. Inside this directory, all subdirectories correspond to Java packages. Thus, "collections", "com", "concurrency" etc. are all Java packages (which are also directories on the disk). In the screenshot above the Java packages are illustrated with a folder icon.

I have expanded two of the sublevel Java packages, so you can see the classes inside. The classes are illustrated using a little blue circle with a C inside, in the screenshot above.

The full path of a subpackage is its name, with all ancestor package names in front of it, separated by dots. For instance, the full path of "navigation" subpackage is:

`Com.jenkov.navigation`

Similarly, the fully qualified name of a Java class includes its package name. For instance, the full qualified name of the "Page" class, is:

`com.jenkov.navigation.Page`

Creating a Java Package Structure

To create a Java package you must first create a source root directory on your hard disk. The root directory is not itself part of the package structure. The root directory contains all the Java sources that need to go into your package structure.

Once you have created a source root directory you can start adding subdirectories to it. Each subdirectory corresponds to a Java package. You can add subdirectories inside subdirectories to create a deeper package structure.

Adding Classes to Packages

In order to put add Java classes to packages, you must do two things:

1. Put the Java source file inside a directory matching the Java package you want to put the class in.
2. Declare that class as part of the package.

Putting the Java source file inside a directory structure that matches the package structure, is pretty straightforward. Just create a source root directory, and inside that, create directories for each package and subpackage recursively. Put the class files into the directory matching the package you want to add it to.

When you have put your Java source file into the correct directory (matching the package the class should belong to), you have to declare inside that class file, that it belongs to that Java package. Here is how you declare the package inside a Java source file:

```
package com.jenkov.navigation;
public class Page {
   ...
}
```

Java package naming convention:

Java packages are always written in lowercase letters. Not like Java classes, where the first letter is usually a capital letter.

Importing Classes From Other Java Packages

If a class A needs to use the class B, you must reference class B inside class A. If class A and B are located in the same Java package, the Java compiler will accept references between the two classes. Here is an example of that:

```
// B.java

public class B {

   public void doIt() {
      // do something...
   }
}

// A.java

public class A {

   public static void main(String[] args){
      B theBObj = new B();
```

```
        b.doIt();
    }
}
```

If the classes A and B are located in the same Java package, there is no problem with the code above. However, if class A and B are located in different Java packages, then class A must import class B in order to use it. Here is how that could look:

```
import anotherpackage.B;
public class A {
    public static void main(String[] args){
        B theBObj = new B();
        b.doIt();
    }
}
```

It is the first line in the example that imports class B. The example assumes that class B is located in a Java package named anotherpackage.

If class B had been located in a subpackage of anotherpackage you would have to list the full package and subpackage path to class B. For instance, if class B was located in package anotherpackage.utilthen the import statement would have looked like this:

```
import anotherpackage.util.B;
```

Importing All Classes From Another Package:

It is possible to import all classes of a package using the * character instead of a class name. Here is how such an import statement looks:

```
import anotherpackage.util.*;
```

Using Classes via Their Fully Qualified Class Name:

It is possible to use a class from another package without importing it with an import statement.

You could use this fully qualified class name to reference the TimeUtil class inside another class, like this:

```
public class A {

    public static void main(String[] args){

        anotherpackage.util.TimeUtil timeUtil =
            new anotherpackage.util.TimeUtil();

        timeUtil.startTimer();
    }
}
```

Built-in Java Packages:

The Java platform comes with a lot of built-in Java packages. These packages contain classes for all kinds of purposes that programmers often need, like reading and writing files from the local hard disk, sending and receiving data over networks and the internet, connecting to databases, and many, many other things.

A Simple Java Class Declaration :

Declaring a simple class without any variables, methods or any other instructions, looks like this in Java code:

```
public class MyClass {
}
```

This Java code needs to be located in a file with the same file name as the class and ending with the file suffix .java. More specifically, the file name has to be MyClass.java.

Once the file is located in a file matching its class name and ending with .java, you can compile it with the Java compiler from the Java SDK, or from inside your Java IDE (which is much easier).

If you locate a Java class inside a Java package, you have to specify the package name at the top of the Java file. Here is how the class from earlier looks with a package declaration added:

```
package myjavacode;

public class MyClass {

}
```

Java main method:

A Java program is a sequence of Java instructions that are executed in a certain order. Since the Java instructions are executed in a certain order, a Java program has a start and an end. To execute your Java program you need to signal to the Java Virtual Machine where to start executing the program. In Java, all instructions (code) have to be located inside a Java class.

A Java program starts by executing the main method of some class.

```
package myjavacode;
public class MyClass {
    public static void main(String[] args) {
    }
}
```

Java Variable Declaration

Exactly how a variable is declared depends on what type of variable it is (non-static, static, local, parameter).

Here are examples of how to declare variables of all the primitive data types in Java:

```
byte    myByte;
short   myShort;
char    myChar;
int     myInt;
long    myLong;
float   myFloat;
double  myDouble;
```

Here are examples of how to declare variables of the object types in Java:

```
Byte       myByte;
Short      myShort;
Character  myChar;
Integer    myInt;
Long       myLong;
Float      myFloat;
Double     myDouble;
String     myString;
```

Notice the uppercase first letter of the object types.
When a variable points to an object the variable is called a "reference" to an object.

Java Data types:

Data types into two groups:

- Primitive data types
- Object references

A variable takes up a certain amount of space in memory. How much memory a variable takes depends on its data type.

A variable of a primitive data type contains the value of the variable directly in the memory allocated to the variable. For instance, a number or a character.

A variable of an object reference type is different from a variable of a primitive type. A variable of an object type is also called a *reference*. The variable itself does not contain the object, but contains a *reference* to the object. The reference points to somewhere else in memory where the whole object is stored. Via the reference stored in the variable you can access fields and methods of the referenced object. It is possible to have many different variables reference the same object. This is not possible with primitive data types.

Primitive Data Types:

The Java language contains the following primitive data types:

Data type	**Description**

boolean	A binary value of either true or false
byte	8 bit signed value, values from -128 to 127
short	16 bit signed value, values from -32.768 to 32.767
char	16 bit Unicode character
int	32 bit signed value, values from -2.147.483.648 to 2.147.483.647
long	64 bit signed value, values from -9.223.372.036.854.775.808 to 9.223.372.036.854.775.808
float	32 bit floating point value
double	64 bit floating point value

Object Types:

Data type	**Description**
Boolean	A binary value of either true or false
Byte	8 bit signed value, values from -128 to 127
Short	16 bit signed value, values from -32.768 to 32.767
Character	16 bit Unicode character
Integer	32 bit signed value, values from -2.147.483.648 to 2.147.483.647
Long	64 bit signed value, values from -9.223.372.036.854.775.808 to 9.223.372.036.854.775.808
Float	32 bit floating point value
Double	64 bit floating point value

String	N byte Unicode string of textual data. Immutable

When you declare an object reference variable, the reference does not point to any object. You need to create (instantiate) an object first. Here is how that is done:

```
Integer myInteger;
myInteger = new Integer(45);
```

Ref : http://tutorials.jenkov.com/java/data-types.html

Classes & Objects:

Objects are instances of classes. When you create an object, that object is of a certain class. The class is like a template (or blueprint) telling how objects of that class should look. When you create an object, you say "give me an object of this class".

If you think of a factory producing lots and lots of the same items, then the class would be the blueprint / manual of how the finished product should look, and the objects would be each of the finished products.

Here is a simple diagram illustrating the principle of objects being of a certain class. The class determines what fields and methods the objects of that class have.

Here is an example of java class declaration,

```
public class Car {
}
```

Fields:

A Java field is a variable inside a class. For instance, in a class representing an employee, the Employeeclass might contain the following fields:

- name
- position
- salary
- hiredDate

The corresponding Java class could be defined like this:

```
public class Employee {
  String name;
  String position;
  int salary;
  Date hiredDate;
}
```

Constructors:

Constructors are a special kind of method that is executed when an object of that class is created. Constructors typically initialize the objects internal fields - if necessary.

```
public class Car {
   private String brand;
   public Car(String theBrand) {
      this.brand = theBrand;
   }
}
```

Constructor Overloading - Multiple Constructors for a Java Class

A class can have multiple constructors, as long as their signature (the parameters they take) are not the same. You can define as many constructors as you need. When a Java class contains multiple constructors, we say that the constructor is overloaded (comes in multiple versions). This is what*constructor overloading* means, that a Java class contains multiple constructors.

Here is a Java constructor overloading example:

```
public class MyClass {
  private int number = 0;
  public MyClass() {
  }
  public MyClass(int theNumber) {
    this.number = theNumber;
  }
}
```

The Java class above contains two constructors. The first constructor is a no-arg constructor, meaning it takes no parameters (no arguments). The second constructor takes an int parameter. Inside the constructor body the int parameter value is assigned to a field, meaning the value of the parameter is copied into the field. The field is thus initialized to the given parameter value.

The keyword this in front of the field name (this.number) is not necessary. It just signals to the compiler that it is the field named number that is being referred to.

Default, no-arg Constructor

You don't have to define a constructor for a class, but if you don't define any constructor, the Java compiler will insert a default, no-argument constructor for you. Thus, once the class is compiled it will always at least have a no-argument constructor.

If you do define a constructor for your class, then the Java compiler will not insert the default no-argument constructor into your class.

Calling a Constructor

You call a constructor when you create a new instance of the class containing the constructor. Here is a Java constructor call example:

```
MyClass myClassVar = new MyClass();
```

This example invokes (calls) the no-argument constructor for MyClass as defined earlier in this text.

In case you want to pass parameters to the constructor, you include the parameters between the parentheses after the class name, like this:

```
MyClass myClassVar = new MyClass(1975);
```

This example passes one parameter to the MyClass constructor that takes an int as parameter.

Calling Constructors in Superclasses

In Java a class can extend another class. When a class extends another class it is also said to "inherit" from the class it extends. The class that extends is called the subclass, and the class being extended is called the superclass.

A class that extends another class does not inherit its constructors. However, the subclass must call a constructor in the superclass inside one of the subclass constructors!

Look at the following two Java classes. The class Car extends (inherits from) the class Vehicle.

```
public class Vehicle {
  private String regNo = null;

  public Vehicle(String no) {
    this.regNo = no;
  }
}
```

```
public class Car extends Vehicle {
  private String brand = null;
  public Car(String br, String no) {
    super(no);
    this.brand = br;
  }
}
```

Using the keyword super refers to the superclass of the class using the super keyword. When superkeyword is followed by parentheses like it is here, it refers to a constructor in the superclass. In this case it refers to the constructor in the Vehicle class. Because Car extends Vehicle, the Car constructors must all call a constructor in the Vehicle.

Java Constructor Access Modifiers

The access modifier of a constructor determines what classes in your application that are allowed to call that constructor. The access modifiers are explained in more detail in the text on **Java access modifiers**.

For instance, if a constructor is declared protected then only classes in the same package, or subclasses of that class can call that constructor.

A class can have multiple constructors, and each constructor can have its own access modifier. Thus, some constructors may be available to all classes in your application, while other constructors are only available to classes in the same package, subclasses, or even only to the class itself (private constructors).

Ref : http://tutorials.jenkov.com/java/constructors.html

Methods:

Methods are groups of operations that carry out a certain function together. For instance, a method may add to numbers, and divide it by a third number. Or, a method could read and write data in a database etc.

Here is the Car class from before with a single, simple method named getBrand added:

```
public class Car {
  private String brand;

  public Car(String theBrand) {
    this.brand = theBrand;
  }

  public String getBrand() {
    return this.brand;
  }
}
```

Java Access modifier:

A *Java access modifier* specifies which classes can access a given class and its fields, constructors and methods. Access modifiers can be specified separately for a class, its constructors, fields and methods.

Classes, fields, constructors and methods can have one of four different Java access modifiers:

- private
- default (package)
- protected
- Public

Private Access modifier:

If a method or variable is marked as private (has the private access modifier assigned to it), then only code inside the same class can access the variable, or call the method. Code inside subclasses cannot access the variable or method, nor can code from any external class.

Classes cannot be marked with the private access modifier. Marking a class with the private access modifier would mean that no other class could access it, which means that you could not really use the class at all. Therefore the private access modifier is not allowed for classes.

Here is an example of assigning the private access modifier to a field:

```
public class Clock {
  private long time = 0;
}
```

The member variable time has been marked as private. That means, that the member variable timeinside the Clock class cannot be accessed from code outside the Clock class.

private Constructors

If a constructor in a class is assigned the private Java access modifier, that means that the constructor cannot be called from anywhere outside the class. A private constructor can still get called from other constructors, or from static methods in the same class. Here is a Java class example illustrating that:

```
public class Clock {
  private long time = 0;
  private Clock(long time) {
    this.time = time;
  }
  public Clock(long time, long timeOffset) {
    this(time);
    this.timeOffset = timeOffset;
  }
}
```

default (package) Access Modifier

The default Java access modifier is declared by not writing any access modifier at all. The default access modifier means that code inside the class itself as well as code inside classes in the same package as this class, can access the class, field, constructor or method which the default access modifier is assigned to. Therefore, the default access modifier is also sometimes referred to as the package access modifier.

Subclasses cannot access methods and member variables (fields) in the superclass, if they these methods and fields are marked with the default access modifier, unless the subclass is located in the same package as the superclass.

Here is an default / package access modifier example:

```
public class Clock {
  long time = 0;
}
public class ClockReader {
  Clock clock = new Clock();
  public long readClock{
    return clock.time;
  }
}
```

The time field in the Clock class has no access modifier, which means that it is implicitly assigned the default / package access modifier. Therefore, the ClockReader class can read the time member variable of the Clock object, provided that ClockReader and Clock are located in the same Java package.

Protected Access Modifier:

The protected access modifier provides the same access as the default access modifier, with the addition that subclasses can access protected methods and member variables (fields) of the

superclass. This is true even if the subclass is not located in the same package as the superclass.

```
public class Clock {
   protected long time = 0; // time in milliseconds
}

public class SmartClock() extends Clock{
   public long getTimeInSeconds() {
      return this.time / 1000;
   }
}
```

In the above example the subclass SmartClock has a method called getTimeInSeconds() which accesses the time variable of the superclass Clock. This is possible even if Clock and SmartClock are not located in the same package, because the time field is marked with the protected Java access modifier.

Public Access Modifier:

The Java access modifier public means that all code can access the class, field, constructor or method, regardless of where the accessing code is located. The accessing code can be in a different class and different package.

It is important to keep in mind that the Java access modifier assigned to a Java class takes precedence over any access modifiers assigned to fields, constructors and methods of that class. If the class is marked with the default access modifier, then no other class outside the same Java package can access that class, including its constructors, fields and methods. It doesn't help that you declare these fields public, or even public static.

The Java access modifiers private and protected cannot be assigned to a class. Only to constructors, methods and fields inside classes. Classes can only have the default (package) and public access modifier assigned to them.

Access Modifier and Inheritance:

When you create a subclass of some class, the methods in the subclass cannot have less accessible access modifiers assigned to them than they had in the superclass. For instance, if a method in the superclass is public then it must be public in the subclass too, in case the subclass overrides the method. If a method in the superclass is protected then it must be either protected or public in the subclass.

While it is not allowed to decrease accessibility of an overridden method, it is allowed to expand accessibility of an overridden method. For instance, if a method is assigned the default access modifier in the superclass, then it is allowed to assign the overridden method in the subclass the public access modifier.

Array:

An *array* is a collection of variables of the same type. For instance, an array of int is a collection of variables of the type int. The variables in the array are ordered and each have an index.

Declaring an Array Variable in Java :

```
int[] intArray;
int intArray[];
String[] stringArray;
String stringArray[];
MyClass[] myClassArray;
MyClass myClassArray[];
```

You actually have a choice about where to place the square brackets [] when you declare an array in Java. The first location you have already seen. That is behind the name of the data type (e.g. String[]). The second location is after the variable name. The above all Java array declarations are actually all valid.

Instantiating an Array in Java

When you declare a Java array variable you only declare the variable (reference) to the array itself. The declaration does not actually create an array. You create an array like this:

```
int[] intArray;
intArray = new int[10];
```

This example creates an array of type int with space for 10 int variables inside.

Once an array has been created its size cannot be changed. In some languages arrays can change their size after creation, but in Java an array cannot change its size once it is created. If you need an array-like data structure that can change its size, you should use a **List**.

The previous Java array example created an array of int which is a primitive data type. You can also create an array of object references. For instance:

```
String[] stringArray = new String[10];
```

Java Array Literals:

The Java programming language contains a shortcut for instantiating arrays of primitive types and strings. If you already know what values to insert into the array, you can use an array literal. Here is how how an array literal looks in Java code:

```
int[] ints2 = new int[]{ 1,2,3,4,5,6,7,8,9,10 };
```

Notice how the values to be inserted into the array are listed inside the { ... } block. The length of this list also determines the length of the created array.

Actually, you don't have to write the new int[] part in the latest versions of Java. You can just write.

```
int[] ints2 = { 1,2,3,4,5,6,7,8,9,10 };
```

It is the part inside the curly brackets that is called an *array literal*.

This style works for arrays of all primitive types, as well as arrays of strings. Here is a string array example:

```
String[] strings = {"one", "two", "three"};
```

Array Length

You can access the length of an array via its length field. Here is an example:

```
int[] intArray = new int[10];
int arrayLength = intArray.length;
```

Iterating Arrays:

```
String[] stringArray = new String[10];
for(int i=0; i < stringArray.length; i++) {
   stringArray[i] = "String no " + i;
}

for(int i=0; i < stringArray.length; i++) {
   System.out.println( stringArray[i] );
}
```

You can also iterate an array using the "for-each" loop in Java. Here is how that looks:

```
int[] intArray = new int[10];
for(int theInt : intArray) {
   System.out.println(theInt);
}
```

The for-each loop gives you access to each element in the array, one at a time, but gives you no information about the index of each element. Additionally, you only have access to the value. You cannot change the value of the element at that position. If you need that, use a normal for-loop as shown earlier.

Multidimensional Java Arrays

```
int[][] intArray = new int[10][20];
```

Iterating Multidimensional Arrays

```
int[][] intArray = new int[10][20];
for(int i=0; i < intArrays.length; i++){
   for(int j=0; j < intArrays[i].length; j++){
      System.out.println("i: " + i + ", j: " + j);
   }
}
```

Arrays Class:

Java contains a special utility class that makes it easier for you to perform many often used array operations like copying and sorting arrays, filling in data, searching in arrays etc. The utility class is calledArrays and is located in the standard Java package java.util.

Remember, in order to usejava.util.Arrays in your Java classes you must import it. Here is how importing java.util.Arrayscould look in a Java class of your own:

```
package myjavaapp;
import java.util.Arrays;
public class MyClass{
   public static void main(String[] args) {
   }
}
```

Copying Arrays

You can copy an array into another array in Java in several ways.

Copying an Array by Iterating the Array

The first way to copy an array in Java is to iterate through the array and copy each value of the source array into the destination array. Here is how copying an array looks using that method:

```
int[] source = new int[10];
int[] dest = new int[10];
for(int i=0; i < source.length; i++) {
  source[i] = i;
}
for(int i=0; i < source.length; i++) {
  dest[i] = source[i];
}
```

Copying an Array Using Arrays.copyOf()

The second method to copy a Java array is to use the Arrays.copyOf() method. Here is how copying an array using Arrays.copyOf() looks:

```
int[] source = new int[10];
for(int i=0; i < source.length; i++) {
  source[i] = i;
}
int[] dest = Arrays.copyOf(source, source.length);
```

The Arrays.copyOf() method takes 2 parameters. The first parameter is the array to copy. The second parameter is the length of the new array. This parameter can be used to specify how many elements from the source array to copy.

Copying an Array Using Arrays.copyOfRange()

The third method to copy a Java array is to use the Arrays.copyOfRange() method. TheArrays.copyOfRange() method copies a range of an array, not necessarily the full array. Here is how copying a full array using Arrays.copyOfRange() in Java looks:

```
int[] source = new int[10];
for(int i=0; i < source.length; i++) {
  source[i] = i;
```

```
}
int[] dest = Arrays.copyOfRange(source, 0, source.length);
```

The Arrays.copyOfRange() method takes 3 parameters. The first parameter is the array to copy. The second parameter is the first index in the source array to include in the copy. The third parameter is the last index in the source array to include in the copy (excluded - so passing 10 will copy until and including index 9).

Converting Arrays to Strings With Arrays.toString()

You can convert an Java array of primitive types to a String using the Arrays.toString() method. Here is an example of how to convert an array of int to a String using Arrays.toString():

```
int[] ints = new int[10];
for(int i=0; i < ints.length; i++){
   ints[i] = 10 - i;
}
System.out.println(java.util.Arrays.toString(ints));
```

The returned String(which is printed) looks like this:

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Sorting Arrays:

You can sort the elements of an array using the Arrays.sort() method. Sorting the elements of an array rearranges the order of the elements according to their sort order. Here is an Arrays.sort() example:

```
java.util.Arrays.sort(arr);
```

Ref : http://tutorials.jenkov.com/java/arrays.html

Java strings:

Strings in Java are objects. Therefore you need to use the new operator to create a new Java String object. Here is a Java String instantiation (creation) example:

```
String myString = new String("Hello World");
```

Java String Literals

Java has a shorter way of creating a new String:

```
String myString = "Hello World";
```

Concatenating Strings:

```
String one = "Hello";
String two = "World";
String three = one + " " + two;
```

String length:

```
String string = "Hello World";
int length = string.length();
```

Accessing sub-strings:

```
String string1 = "Hello World";
String substring = string1.substring(0,5);
```

After this code is executed the substring variable will contain the string Hello.

Searching in Strings With indexOf()

You can search for substrings in Strings using the indexOf() method. Here is an example:

```
String string1 = "Hello World";
int index = string1.indexOf("World");
```

The index variable will contain the value 6 after this code is executed. The indexOf() method returns the index of where the first character in the first matching substring is found.

If the substring is not found within the string, the indexOf() method returns -1;

There is a version of the indexOf() method that takes an index from which the search is to start. That way you can search through a string to find more than one occurrence of a substring. Here is an example:

```
String theString = "is this good or is this bad?";
String substring = "is";

int index = theString.indexOf(substring);
while(index != -1) {
   System.out.println(index);
   index = theString.indexOf(substring, index + 1);
}
```

The Java String class also has a lastIndexOf() method which finds the last occurrence of a substring.

```
String theString = "is this good or is this bad?";
String substring = "is";
int index = theString.lastIndexOf(substring);
System.out.println(index);
```

The output printed from this code would be 21 which is the index of the last occurrence of the substring "is".

Matching a String Against a Regular Expression With matches()

The Java String matches() method takes a regular expression as parameter, and returns true if the regular expression matches the string, and false if not.

Here is a matches() example:

```
String text = "one two three two one";
boolean matches = text.matches(".*two.*");
```

Comparing Strings

Java Strings also have a set of methods used to compare Strings. These methods are:

- equals()
- equalsIgnoreCase()
- startsWith()
- endsWith()
- compareTo()

Trimming Strings With trim()

```
String text = " And he ran across the field ";
String trimmed = text.trim();
```

After executing this code the trimmed variable will point to a String instance with the value "And he ran across the field".

Replacing Characters in Strings With replace()

The Java String class contains a method named replace() which can replace characters in a String. The replace() method does not actually replace characters in the existing String. Rather, it returns a new String instance which is equal to the String instance it was created from, but with the given characters replaced. Here is a Java String replace() example:

```
String source = "123abc";
String replaced = source.replace('a', '@');
```

After executing this code the replaced variable will point to a String with the text:

```
123@bc
```

replaceFirst()

The Java String replaceFirst() method returns a new String with the first match of the regular expression passed as first parameter with the string value of the second parameter.

Here is a replaceFirst() example:

```
String text = "one two three two one";
String s = text.replaceFirst("two", "five");
```

This example will return the string "one five three two one".

replaceAll()

The Java String replaceAll() method returns a new String with all matches of the regular expression passed as first parameter with the string value of the second parameter.

Here is a replaceAll() example:

```
String text = "one two three two one";
String t = text.replaceAll("two", "five");
```

This example will return the string "one five three five one".

Splitting Strings With split()

The Java String class contains a split() method which can be used to split a String into an **array** of String objects. Here is a Java String split() example:

```
String source = "A man drove with a car.";
String[] occurrences = source.split("a");
```

After executing this Java code the occurrences array would contain the String instances:

```
"A m"
"n drove with "
" c"
"r."
```

The source String has been split on the a characters. The Strings returned do not contain the acharacters. The a characters are considered delimiters to split the String by, and the delimiters are not returned in the resulting String array.

The String split() method exists in a version that takes a limit as a second parameter. Here is a Java String split() example using the limit parameter:

```
String source = "A man drove with a car.";
int limit = 2;
String[] occurrences = source.split("a", limit);
```

The limit parameter sets the maximum number of elements that can be in the returned array. If there are more matches of the regular expression in the String than the given limit, then the array will contain limit - 1 matches, and the last element will be the rest of the String from the last of the limit - 1matches. So, in the example above the returned array would contain these two Strings:

```
"A                                                                    m"
"n                drove                with                a                car."
```

The first String is a match of the a regular expression. The second String is the rest of the String after the first match.

Running the example with a limit of 3 instead of 2 would result in these Strings being returned in the resulting String array:

```
"A                                                                    m"
"n                drove                with                "
"                                                                     car."
```

Notice how the last String still contains the a character in the middle. That is because this String represents the rest of the String after the last match (the a after 'n drove with ').

Converting Numbers to Strings With valueOf()

The Java String class contains a set of overloaded static methods named valueOf() which can be used to convert a number to a String. Here are some simple Java String valueOf() examples:

```
String intStr = String.valueOf(10);
System.out.println("intStr = " + intStr);

String flStr = String.valueOf(9.99);
System.out.println("flStr = " + flStr);
```

The output printed from this code would be:

```
intStr = 10
flStr = 9.99
```

Converting Objects to Strings

The Object class contains a method named toString(). Since all Java classes extends (inherits from) the Object class, all objects have a toString() method. This method can be used to create a String representation of the given object. Here is a Java toString() example:

```
Integer integer = new Integer(123);
String intStr = integer.toString();
```

Getting Characters:

It is possible to get a character at a certain index in a String using the charAt() method. Here is an example:

```
String theString = "This is a good day to code";
System.out.println( theString.charAt(0) );
System.out.println( theString.charAt(3) );
```

This code will print out:

```
T
s
```

since these are the characters located at index 0 and 3 in the String.

Ref : http://tutorials.jenkov.com/java/strings.html

---

for loop:

```
for(int i=0; i < 10; i++) {
  System.out.println("i is: " + i);
}
```

The Java for each Loop

In Java 5 a variation of the for loop was added. This variation is called the for each loop. Here is a Java for each loop example:

```
String strings = {"one", "two", "three" };
for(String aString : strings) {
   System.out.println(aString);
}
```

Switch statement:

```
char penaltyKick = 'L';

switch (penaltyKick) {
        case 'L': System.out.println("Messi shoots to the left and scores!");
                  break;
        case 'R': System.out.println("Messi shoots to the right and misses the goal!");
                  break;
        case 'C': System.out.println("Messi shoots down the center, but the keeper blocks it!");
                  break;
         default:
                  System.out.println("Messi is in position...");
}
```

Inheritance:

Inheritance can be an effective method to share code between classes that have some traits in common, yet allowing the classes to have some parts that are different.

Here is diagram illustrating a class called Vehicle, which has two subclasses called Car and Truck.

The Vehicle class is the superclass of Car and Truck. Car and Truck are subclasses of Vehicle. The Vehicle class can contain those fields and methods that all Vehicles need (e.g. a license plate, owner etc.), whereas Car and Truck can contain the fields and methods that are specific to Cars and Trucks.

Class Hierarchies

Superclasses and subclasses form an inheritance structure which is also called a *class hierarchy*. At the top of the class hierarchy you have the superclasses. At the bottom of the class hierarchy you have the subclasses.
A class hierarchy may have multiple levels, meaning multiple levels of superclasses and subclasses. A subclass may itself be a superclass of other subclasses etc.

What is Inherited?

When a subclass extends a superclass in Java, all protected and public fields and methods of the superclass are inherited by the subclass. By *inherited* is meant that these fields and methods are part of of the subclass, as if the subclass had declared them itself. protected and public fields can be called and referenced just like the methods declared directly in the subclass. Fields and methods with default (package) access modifiers can be accessed by subclasses only if the subclass is located in the same package as the superclass. Private fields and methods of the superclass can never be referenced directly by subclasses.

Java Only Supports Singular Inheritance

The Java inheritance mechanism only allows a Java class to inherit from a single superclass (singular inheritance). In some programming languages, like C++, it is possible for a subclass to inherit from multiple superclasses (multiple inheritance). Since multiple inheritance can create

some weird problems, if e.g. the superclasses contain methods with the same names and parameters, multiple inheritance was left out in Java.

Declaring Inheritance in Java

In Java inheritance is declared using the extends keyword. You declare that one class *extends* another class by using the extends keyword in the class definition. Here is Java inheritance example using the extends keyword:

```
public class Vehicle {
   protected String licensePlate = null;

   public void setLicensePlate(String license) {
      this.licensePlate = license;
   }
}

public class Car extends Vehicle {
   int numberOfSeats = 0;

   public String getNumberOfSeats() {
      return this.numberOfSeats;
   }
}
```

Inheritance and Type Casting

It is possible to reference a subclass as an instance of one of its superclasses. For instance, using the class definitions from the example in the previous section it is possible to reference an instance of the Carclass as an instance of the Vehicle class. Because the Car class extends (inherits from) the Vehicleclass, it is also said to **be** a Vehicle.

Here is a Java code example:

```
Car     car     = new Car();
Vehicle vehicle = car;
```

First a Car instance is created. Second, the Car instance is assigned to a variable of type Vehicle. Now the Vehicle variable (reference) points to the Car instance. This is possible because the Car class inherits from the Vehicle class.
As you can see, it is possible to use an instance of some subclass as if it were an instance of its superclass.

The process of referencing an object of class as a different type than the class itself is called type casting.

Upcasting and Downcasting

You can always cast an object of a subclass to one of its superclasses. This is referred to as *upcasting*(from a subclass type to a superclass type).
It may also be possible to cast an object from a superclass type to a subclass type, but only if the object really is an instance of that subclass (or an instance of a subclass of that subclass). This is referred to as*downcasting* (from a superclass type to a subclass type). Thus, this example of downcasting is valid:

```
Car     car     = new Car();

// upcast to Vehicle
Vehicle vehicle = car;

// downcast to car again
Car     car2    =  (Car) vehicle;
```

Overriding Methods:

In a subclass you can override (redefine) methods defined in the superclass. Here is a Java method override example:

```
public                  class                   Vehicle                 {
   String               licensePlate            =                       null;
```

```
  public void setLicensePlate(String licensePlate) {
    this.licensePlate = licensePlate;
  }
}

public class Car extends Vehicle {
  public void setLicensePlate(String license) {
    this.licensePlate = license.toLowerCase();
  }
}
```

Notice how both the Vehicle class and the Car class defines a method called setLicensePlate(). Now, whenever the setLicensePlate() method is called on a Car object, it is the method defined in the Car class that is called. The method in the superclass is ignored.

To override a method the method signature in the subclass must be the same as in the superclass. That means that the method definition in the subclass must have exactly the same name and the same number and type of parameters, and the parameters must be listed in the exact same sequence as in the superclass. Otherwise the method in the subclass will be considered a separate method.

In Java you cannot override private methods from a superclass. If the superclass calls a private method internally from some other method, it will continue to call that method from the superclass, even if you create a private method in the subclass with the same signature.

The @override Annotation:

If you override a method in a subclass, and the method is all of a sudden removed or renamed or have its signature changed in the superclass, the method in the subclass no longer overrides

the method in the superclass. But how do you know? It would be nice if the compiler could tell you that the method being overridden no longer overrides a method in the superclass, right?

This is what the Java @override annotation is for. You place the Java @override annotation above the method that overrides a method in a superclass. Here is Java @override example:

```
public class Car extends Vehicle {
  @Override
  public void setLicensePlate(String license) {
    this.licensePlate = license.toLowerCase();
  }
}
```

Calling Superclass Methods

If you override a method in a subclass, but still need to call the method defined in the superclass, you can do so using the super reference, like this:

```
public class Car extends Vehicle {
  public void setLicensePlate(String license) {
    super.setLicensePlate(license);
  }

}
```

In the above code example the method setLicensePlate() in the Car class, calls the setLicensePlate() method in the Vehicle class.

The instanceof Instruction

Java contains an instruction named instanceof. The instanceof instruction can determine whether a given object is an instance of some class. Here is a Java instanceof example:

```
Car car = new Car();
boolean isCar = car instanceof Car;
```

After this code has been executed the isCar variable will contain the value true.

As you can see, the instanceof instruction can be used to explore the inheritance hierarchy. The variable type used with the instanceof instruction does not affect its outcome. Look at thisinstanceof example:

```
Car car = new Car();
Vehicle vehicle = car;
boolean isCar = vehicle instanceof Car;
```

Even though the vehicle variable is of type Vehicle, the object it ends up pointing to in this example is a Car object. Therefore the vehicle instanceof Car instruction will evaluate to true.

Fields and Inheritance

As mentioned earlier, in Java fields cannot be overridden in a subclass. If you define a field in a subclass with the same name as a field in the superclass, the field in the subclass will hide (shadow) the field in the superclass. If the subclass tries to access the field, it will access the field in the subclass.

If, however, the subclass calls up into a method in the superclass, and that method accesses the field with the same name as in the subclass, it is the field in the superclass that is accessed.

Here is Java inheritance example that illustrates how fields in subclasses shadow (hides) fields in superclasses:

```
public class Vehicle {
  String licensePlate = null;
  public void setLicensePlate(String licensePlate) {
    this.licensePlate = licensePlate;
  }
```

```
    public String getLicensePlate() {
        return licensePlate;
    }
}


public class Car extends Vehicle {
    protected String licensePlate = null;
    @Override
    public void setLicensePlate(String license) {
        super.setLicensePlate(license);
    }
    @Override
    public String getLicensePlate() {
        return super.getLicensePlate();
    }

    public void updateLicensePlate(String license){
        this.licensePlate = license;
    }
}
```

Notice how both classes have a licensePlate field defined.

Both the Vehicle class and Car class has the methods setLicensePlate() and getLicensePlate(). The methods in the Car class calls the corresponding methods in the Vehicle class. The result is, that eventually both set of methods access the licensePlate field in the Vehicle class.

The updateLicensePlate() method in the Car class however, accesses the licensePlate field directly. Thus, it accesses the licensePlate field of the Car class. Therefore, you will not get the same result if you call setLicensePlate() as when you call the updateLicense() method.

Look at the following lines of Java code:

```
Car car = new Car();
car.setLicensePlate("123");
car.updateLicensePlate("abc");
System.out.println("license plate: "
    + car.getLicensePlate());
```

This Java code will print out the text 123.

Constructors and Inheritance

The Java inheritance mechanism does not include constructors. In other words, constructors of a superclass are not inherited by subclasses. Subclasses can still call the constructors in the superclass using the super() contruct. In fact, a subclass constructor is required to call one of the constructors in the superclass as the very first action inside the constructor body. Here is how that looks:

```
public class Vehicle {
  public Vehicle() {
  }
}

public class Car extends Vehicle{
  public Car() {
    super();
    //perform other initialization here
  }
}
```

Notice the call to super() inside the Car constructor. This super() call executes the constructor in the Vehicle class.

You may have seen Java classes where the subclass constructors did not seem to call the constructors in the superclass. Maybe the superclass did not even have a constructor. However, the subclass constructors have still called superclass constructors in those case. You just could not see it. Let me explain why:

If a class does not have any explicit constructor defined, the Java compiler inserts an implicit no-arg constructor. Thus, a class always has a constructor. Therefore the following version of Vehicle is equivalent to the version shown just above:

```
public class Vehicle {
}
```

Second, if a constructor does not explicitly call a constructor in the superclass, the Java compiler inserts an implicit call to the no-arg constructor in the superclass. That means that the following version of the Carclass is actually equivalent to the version shown earlier:

```
public class Car extends Vehicle{
  public Car() {
  }
}
```

In fact, since the constructor is now empty, we could leave it out and the Java compiler would insert it, and insert an implicit call to the no-arg constructor in the superclass. This is how the two classes would look then:

```
public class Vehicle {
}
public class Car extends Vehicle{
}
```

Even though no constructors are declared in these two classes, they both get a no-arg constructor, and the no-arg constructor in the Car class will call the no-arg constructor in the Vehicle class.

If the Vehicle class did not have a no-arg constructor, but had another constructor which takes parameters, the Java compiler would complain. The Car class would then be required to declare a constructor, and inside that constructor call the constructor in the Vehicle class.

Final Classes and Inheritance

A class can be declared final. Here is now that looks:

```
public final class MyClass {
}
```

A final class cannot be extended. In other words, you cannot inherit from a final class in Java.

Ref : http://tutorials.jenkov.com/java/inheritance.html

Exception:

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

Checked exceptions :

A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

```
import java.io.File;
import java.io.FileReader;

public class FilenotFound_Demo {

   public static void main(String args[]) {
      File file = new File("E://file.txt");
      FileReader fr = new FileReader(file);
   }
}
```

If you try to compile the above program, you will get the following exceptions.

```
C:\>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
      FileReader fr = new FileReader(file);
```

Unchecked exceptions:

An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

```
public class Unchecked_Demo {

  public static void main(String args[]) {
    int num[] = {1, 2, 3, 4};
    System.out.println(num[5]);
  }
}
```

If you compile and execute the above program, you will get the following exception.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

Catching Exceptions

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following −

```
try {
  // Protected code
} catch (ExceptionName e1) {
  // Catch block
}
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

```
import java.io.*;

public class ExcepTest {

  public static void main(String args[]) {
    try {
      int a[] = new int[2];
      System.out.println("Access element three :" + a[3]);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception thrown :" + e);
    }
    System.out.println("Out of the block");
  }
}
```

This will produce the following result −

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

Multiple Catch Blocks:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following −

```
try {
  // Protected code
} catch (ExceptionType1 e1) {
  // Catch block
} catch (ExceptionType2 e2) {
  // Catch block
} catch (ExceptionType3 e3) {
  // Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Catching Multiple Type of Exceptions:

Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code. Here is how you would do it −

```
catch (IOException|FileNotFoundException ex) {
  logger.log(ex);
  throw ex;
}
```

The Throws/Throw Keywords :

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

Try to understand the difference between throws and throw keywords, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a RemoteException

```
import java.io.*;
public class className {

  public void deposit(double amount) throws RemoteException {
    // Method implementation
    throw new RemoteException();
  }
  // Remainder of class definition
}
```

The Finally Block:

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax.

```
public class ExcepTest {

  public static void main(String args[]) {
    int a[] = new int[2];
    try {
      System.out.println("Access element three :" + a[3]);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception thrown :" + e);
    } finally {
      a[0] = 6;
      System.out.println("First element value: " + a[0]);
      System.out.println("The finally statement is executed");
    }
  }
}
```

This will produce the following result −

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

Note the following

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

User-defined Exceptions

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes −

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below

```
class MyException extends Exception {
}
```

You just need to extend the predefined **Exception** class to create your own Exception. These are considered to be checked exceptions. The following **InsufficientFundsException** class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

InsufficientFundsException.java

```
import java.io.*;

public class InsufficientFundsException extends Exception {
  private double amount;

  public InsufficientFundsException(double amount) {
    this.amount = amount;
  }

  public double getAmount() {
    return amount;
  }
}
```

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

CheckingAccount.java

```
import java.io.*;
```

```
public class CheckingAccount {
  private double balance;
  private int number;

  public CheckingAccount(int number) {
    this.number = number;
  }

  public void deposit(double amount) {
    balance += amount;
  }

  public void withdraw(double amount) throws InsufficientFundsException {
    if(amount <= balance) {
      balance -= amount;
    }else {
      double needs = amount - balance;
      throw new InsufficientFundsException(needs);
    }
  }

  public double getBalance() {
    return balance;
  }

  public int getNumber() {
    return number;
  }
}
```

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

BankDemo.java

```
public class BankDemo {

  public static void main(String [] args) {
    CheckingAccount c = new CheckingAccount(101);
    System.out.println("Depositing $500...");
    c.deposit(500.00)d;

    try {
      System.out.println("\nWithdrawing $100...");
      c.withdraw(100.00);
```

```
        System.out.println("\nWithdrawing $600...");
        c.withdraw(600.00);
    } catch (InsufficientFundsException e) {
        System.out.println("Sorry, but you are short $" + e.getAmount());
        e.printStackTrace();
    }
  }
}
```

Compile all the above three files and run BankDemo. This will produce the following result −

```
Depositing $500...

Withdrawing $100...

Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
        at CheckingAccount.withdraw(CheckingAccount.java:25)
        at BankDemo.main(BankDemo.java:13)
```

Ref : https://www.tutorialspoint.com/java/java_exceptions.htm

---

Servlet

A **servlet** is a Java programming language class that is used to extend the capabilities of servers that host applications accessed by means of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.

The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servletinterface, which defines life-cycle methods. When implementing a generic service, you can use or extend the GenericServlet class provided with the Java Servlet API. The HttpServlet class provides methods, such as doGet and doPost, for handling HTTP-specific services.

Consider a scenario where requesting a login page for gmail.com. Generally Http request key elements are Http method, Http Url & form parameter. Whereas Http response has status code, Content-type & content.

Where does servlet comes into picture ??

When login page of gmail (www.gmail.com) is requested, it is a static page, request from client goes into server. Server process the request and sends back html page.

Once user name & password given, resultant page (home page) is not static (different for different users). Here web server cannot able to handle the request.

To handle dynamic request from client, Servlet comes into play.

Here, helper application is nothing but a servlet. Servl.et is a special kind of java program which resides on the server and does not contain main method.

What is a web container ?

How does a web server communicate to Servlet ? Web Container (also called “Servlet engine”)

- Servlet engine is used to communicate the web server & Servlet.
- Server lives and dies within a web container.
- Servlet has call-back function which the container knows and that is responsible for calling methods in the Servlet.

How does Container handles the request:

- From web container, it creates request object & response object. It send those objects to Servlet.
- Web container will create thread for each request and calls the Servlet.

- Web container calls the call back function (for eg., doGet() if GET request method). So all functionalities are to be handled inside doGet() / doPost().
- Once Servlet process the request, it creates response object and sends back to web container.
- From web container, it converts to Http Response and sends to web server.

What is the role of web Container?

- Communication support
- Life cycle management - Web container knows the call back methods on Servlet. It knows when to invoke such method.
- Multi- threading support - For every request that comes to particular servlet, web container creates thread.
- Securtiy - Since client cannot talk to Servlet directly, we can keep all kind of security in web container.
- JSP support.

How does a container know which Servlet the client has requested for?

Suppose we have 5 servlets which can do different kind of jobs. There is a file called **web.xml** in web container. It contains the information about the Servlet. It maps url and corresponding Servlet.

Apache Tomcat:

- Apache Tomcat is a web container which allows to run Servlet and Java Server Pages(JSP) based web applications.
- Apache Tomcat also provides by default HttpConnector on port 8080. i.e, Tomcat can also used as HTTP Server.
- But its performance is not as good as the performance of designated web server like Apache HTTP Server.

Google App Engine:

- GAE is a cloud computing platform for developing and hosting web applications in Google managed data centers.
- GAE offers automatic scaling for web-applications (as number of request increases for an application).

Static keyword:

The static keyword in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

1) Java static variable

If you declare any variable as static, it is known static variable.

- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable

- It makes your program **memory efficient** (i.e it saves memory).

Example:

Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created.All student have its unique rollno and name so instance data

member is good.Here, college refers to the common property of all objects.If we make it static,this field will get memory only once.

```
class Student8{
  int rollno;
  String name;
  static String college ="ITS";

  Student8(int r,String n){
  rollno = r;
  name = n;
  }
}
```

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

Example1 :

```
class Student9{
    int rollno;
    String name;
    static String college = "ITS";

    static void change(){
    college = "BBDIT";
    }

    Student9(int r, String n){
    rollno = r;
    name = n;
    }
}
```

Example 2:

```
class Calculate{
  static int cube(int x){
  return x*x*x;
  }

  public static void main(String args[]){
  int result=Calculate.cube(5);
  System.out.println(result);
  }
}
```

Restrictions for static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

Example:

```
class A{
 int a=40;//non static

 public static void main(String args[]){
  System.out.println(a);
 }
}
```

Output: Compiler Time Error

3) Java static block

- It is used to initialize the static data member.
- It is executed before main method at the time of classloading.

Example:

```
class A2{
  static{System.out.println("Static block is invoked");}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}
```

Output:
Static block is invoked
Hello main

---

final keyword in Java:

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

- variable
- method
- class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable:

```
class Bike9{
 final int speedlimit=90;
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
```

```
 obj.run();
 }
}
```

Output: Compile Time Error

2) Java final method

If you make any method as final, you cannot override it.

Example

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

Output: Compile Time Error

3) Java final class

If you make any class as final, you cannot extend it.

```
final class Bike{}
class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}
  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Output: Compile Time Error

Q) Is final method inherited?
Ans) Yes, final method is inherited but you cannot override it.

Que) Can we initialize blank final variable?
Yes, but only in constructor. For example:

Example:

```
class Bike10{
  final int speedlimit;//blank final variable

  Bike10(){
  speedlimit=70;
  System.out.println(speedlimit);
  }

  public static void main(String args[]){
    new Bike10();
 }
}
```

Q) What is final parameter?
If you declare any parameter as final, you cannot change the value of it.

```
class Bike11{
  int cube(final int n){
   n=n+2; // can't be changed as n is final
   n*n*n;
  }
  public static void main(String args[]){
    Bike11 b=new Bike11();
    b.cube(5);
 }
}
```

Output: Compile Time Error.

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

```
class A{
```

```
  static final int data;//static blank final variable
  static{ data=50;}
  public static void main(String args[]){
    System.out.println(A.data);
  }
}
```

Singleton:

The purpose of the Singleton class is to control object creation, limiting the number of objects to only one. The singleton allows only one entry point to create the new instance of the class.

Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static fields. Singletons are often useful where you have to control the resources, such as database connections or sockets.

Create Singleton class :

To implement the Singleton class, the simplest way is to make the constructor of the class as private. There are two approaches for the initialization.

1. Eager initialization:

In eager initialization, the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a Singleton class.
By making the constructor as private you are not allowing other class to create a new instance of the class you want to create the Singleton. Instead, you are creating one public static method (commonly name as for *getInstance()*) to provide the single entry point to create the new instance of the class.

```
public class SingletonClass {
  private static volatile SingletonClass sSoleInstance = new SingletonClass();
  private SingletonClass(){}
```

```
  public static SingletonClass getInstance() {
     return sSoleInstance;
  }
}
```

This approach has one drawback. Here instance is created even though client application might not be using it. This might be a considerable issue if your

Singleton class in creating a database connection or creating a socket. This may cause the memory leak problem. The solution is to create the new instance of the class, when needed. This can be achieved by Lazy Initialization method.

2. Lazy initialization:

Opposite to Eager initialization, here you are going to initialize new instance of the class in *getInstance()* method it self. This method will check if there is any instance of that class is already created? If yes, then our method (*getInstance()*) will return that old instance and if not then it creates a new instance of the singleton class in JVM and returns that instance. This approach is called as Lazy initialization.

```
public class SingletonClass {
  private static SingletonClass sSoleInstance;
  private SingletonClass(){}  //private constructor.
  public static SingletonClass getInstance(){
     if (sSoleInstance == null){ //if there is no instance available... create new one
        sSoleInstance = new SingletonClass();
     }
```

Ref : https://android.jlelse.eu/how-to-make-the-perfect-singleton-de6b951dfdb0

ArrayList :

Arraylist class implements List interface. It is widely used because of the functionality and flexibility it offers. Most of the developers **choose Arraylist over Array** as it's a very good alternative of traditional java arrays.

ArrayList is a resizable-array implementation of the List interface. It implements all optional list operations, and permits all elements, including null.

The issue with arrays is that they are of fixed length so if it is full you cannot add any more elements to it, likewise if there are number of elements gets removed from it the memory consumption would be the same as it doesn't shrink. On the other ArrayList can dynamically grow and shrink after addition and removal of elements.

Apart from these benefits ArrayList class enables us to use predefined methods of it which makes our task easy. Let's see the ArrayList example first then we will discuss it's methods and their usage.

Code:

```
ArrayList<String> obj = new ArrayList<String>();

/*This is how elements should be added to the array list*/
obj.add("Ajeet");
obj.add("Harry");
obj.add("Chaitanya");
obj.add("Steve");
obj.add("Anuj");

/* Displaying array list elements */
System.out.println("Currently the array list has following elements:"+obj);

/*Add element at the given index*/
obj.add(0, "Rahul");
obj.add(1, "Justin");

/*Remove elements from array list like this*/
obj.remove("Chaitanya");
obj.remove("Harry");

System.out.println("Current array list is:"+obj);

/*Remove element from the given index*/
obj.remove(1);
```

```
            System.out.println("Current array list is:"+obj);
```

Methods of ArrayList class

In the above example we have used methods such as add and remove. However there are number of methods available which can be used directly using object of ArrayList class. Let's discuss few of the important methods.

1) **add(Object o)**: This method adds an object o to the arraylist.

```
obj.add("hello");
```
This statement would add a string hello in the arraylist at last position.

2) **add(int index, Object o)**: It adds the object o to the array list at the given index.

```
obj.add(2, "bye");
```
It will add the string bye to the 2nd index (3rd position as the array list starts with index 0) of array list.

3) **remove(Object o)**: Removes the object o from the ArrayList.

```
obj.remove("Chaitanya");
```
This statement will remove the string "Chaitanya" from the ArrayList.

4) **remove(int index)**: Removes element from a given index.

```
obj.remove(3);
```
It would remove the element of index 3 (4th element of the list – List starts with o).

5) **set(int index, Object o)**: Used for updating an element. It replaces the element present at the specified index with the object o.

```
obj.set(2, "Tom");
```
It would replace the 3rd element (index =2 is 3rd element) with the value Tom.

6) **int indexOf(Object o)**: Gives the index of the object o. If the element is not found in the list then this method returns the value -1.

```
int pos = obj.indexOf("Tom");
```

This would give the index (position) of the string Tom in the list.

7) **Object get(int index)**: It returns the object of list which is present at the specified index.

```
String str= obj.get(2);
```
Function get would return the string stored at 3rd position (index 2) and would be assigned to the string “str”. We have stored the returned value in string variable because in our example we have defined the ArrayList is of String type. If you are having integer array list then the returned value should be stored in an integer variable.

8) **int size()**: It gives the size of the ArrayList – Number of elements of the list.

```
int numberofitems = obj.size();
```

9) **boolean contains(Object o)**: It checks whether the given object o is present in the array list if its there then it returns true else it returns false.

```
obj.contains("Steve");
```
It would return true if the string “Steve” is present in the list else we would get false.

10) **clear():** It is used for removing all the elements of the array list in one go. The below code will remove all the elements of ArrayList whose object is obj.

```
obj.clear();
```

Ref : https://beginnersbook.com/2013/12/java-arraylist/

Initialize array List:

Here is the normal way to initialize

```
ArrayList<T> obj = new ArrayList<T>();
        obj.add("Object o1");
        obj.add("Object o2");
        obj.add("Object o3");
```

For alternative way of initializing , refer https://beginnersbook.com/2013/12/how-to-initialize-an-arraylist/

Iterating arrayList

```
ArrayList<Integer> arrlist = new ArrayList<Integer>();
        arrlist.add(14);
        arrlist.add(7);
        arrlist.add(39);
        arrlist.add(40);

        /* For Loop for iterating ArrayList */
        System.out.println("For Loop");
        for (int counter = 0; counter < arrlist.size(); counter++) {
            System.out.println(arrlist.get(counter));
        }

        /* Advanced For Loop*/
        System.out.println("Advanced For Loop");
        for (Integer num : arrlist) {
             System.out.println(num);
        }

        /*Looping Array List using Iterator*/
        System.out.println("Iterator");
        Iterator iter = arrlist.iterator();
        while (iter.hasNext()) {
           System.out.println(iter.next());
        }
```

ArrayList sorting:

```
        ArrayList<String> listofcountries = new ArrayList<String>();
          listofcountries.add("India");
          listofcountries.add("US");
          listofcountries.add("China");
          listofcountries.add("Denmark");


        /* Sort statement*/
          Collections.sort(listofcountries);

          /* Sorted List*/
          System.out.println("After Sorting:");
          for(String counter: listofcountries){
                        System.out.println(counter);
              }
```

-

Linked list:

LinkedList is a doubly-linked list implementation of the List and Deque interfaces. LinkedList allows for constant-time insertions or removals using iterators, but only sequential access of elements. In other words, LinkedList can be searched forward and backward but the time it takes to traverse the list is directly proportional to the size of the list.

```
LinkedList<String> linkedlist = new LinkedList<String>();

    linkedlist.add("Item1");
    linkedlist.add("Item5");
    linkedlist.add("Item3");
    linkedlist.add("Item6");
    linkedlist.add("Item2");

    /*Display Linked List Content*/
    System.out.println("Linked List Content: " +linkedlist);

    /*Add First and Last Element*/
    linkedlist.addFirst("First Item");
    linkedlist.addLast("Last Item");
    System.out.println("LinkedList Content after addition: " +linkedlist);

    /*This is how to get and set Values*/
    Object firstvar = linkedlist.get(0);
    System.out.println("First element: " +firstvar);
    linkedlist.set(0, "Changed first item");
    Object firstvar2 = linkedlist.get(0);
    System.out.println("First element after update by set method: " +firstvar2);

    /*Remove first and last element*/
    linkedlist.removeFirst();
    linkedlist.removeLast();
    System.out.println("LinkedList after deletion of first and last element: " +linkedlist);

    /* Add to a Position and remove from a position*/
    linkedlist.add(0, "Newly added item");
    linkedlist.remove(2);
    System.out.println("Final Content: " +linkedlist);
```

Methods of linked list :

```
LinkedList<String> llistobj  = new LinkedList<String>();
```

1) **boolean add(Object item)**: It adds the item at the end of the list.

llistobj.add("Hello");
It would add the string “Hello” at the end of the linked list.

2) **void add(int index, Object item)**: It adds an item at the given index of the the list.

llistobj.add(2, "bye");
This will add the string “bye” at the 3rd position(2 index is 3rd position as index starts with 0).

3) **boolean addAll(Collection c)**: It adds all the elements of the specified collection c to the list. It throws NullPointerException if the specified collection is null. Consider the below example –

```
LinkedList<String> llistobj = new LinkedList<String>();
ArrayList<String> arraylist= new ArrayList<String>();
arraylist.add("String1");
arraylist.add("String2");
llistobj.addAll(arraylist);
```

This piece of code would add all the elements of ArrayList to the LinkedList.

4) **boolean addAll(int index, Collection c)**: It adds all the elements of collection c to the list starting from a give index in the list. It throws NullPointerException if the collection c is null and IndexOutOfBoundsException when the specified index is out of the range.

llistobj.add(5, arraylist);
It would add all the elements of the ArrayList to the LinkedList starting from position 6 (index 5).

5) **void addFirst(Object item)**: It adds the item (or element) at the first position in the list.

llistobj.addFirst("text");
It would add the string “text” at the beginning of the list.

6) **void addLast(Object item)**: It inserts the specified item at the end of the list.

llistobj.addLast("Chaitanya");
This statement will add a string “Chaitanya” at the end position of the linked list.

7) **void clear()**: It removes all the elements of a list.

llistobj.clear();
8) **Object clone()**: It returns the copy of the list.

For e.g. My linkedList has four items: text1, text2, text3 and text4.

```
Object str= llistobj.clone();
 System.out.println(str);
```
Output: The output of above code would be:

[text1, text2, text3, text4]

9) **boolean contains(Object item)**: It checks whether the given item is present in the list or not. If the item is present then it returns true else false.

```
boolean var = llistobj.contains("TestString");
```
It will check whether the string “TestString” exist in the list or not.

10) **Object get(int index)**: It returns the item of the specified index from the list.

```
Object var = llistobj.get(2);
```
It will fetch the 3rd item from the list.

11) **Object getFirst()**: It fetches the first item from the list.

```
Object var = llistobj.getFirst();
```
12) **Object getLast()**: It fetches the last item from the list.

```
Object var= llistobj.getLast();
```
13) **int indexOf(Object item)**: It returns the index of the specified item.

```
llistobj.indexOf("bye");
```
14) **int lastIndexOf(Object item)**: It returns the index of last occurrence of the specified element.

```
int pos = llistobj.lastIndexOf("hello);
```
integer variable pos will be having the index of last occurrence of string “hello”.

15) **Object poll()**: It returns and removes the first item of the list.

```
Object o = llistobj.poll();
```
16) **Object pollFirst()**: same as poll() method. Removes the first item of the list.

Object o = llistobj.pollFirst();
17) **Object pollLast()**: It returns and removes the last element of the list.

Object o = llistobj.pollLast();
18) **Object remove()**: It removes the first element of the list.

llistobj.remove();
19) **Object remove(int index)**: It removes the item from the list which is present at the specified index.

llistobj.remove(4);
It will remove the 5th element from the list.

20) **Object remove(Object obj)**: It removes the specified object from the list.

llistobj.remove("Test Item");
21) **Object removeFirst()**: It removes the first item from the list.

llistobj.removeFirst();
22) **Object removeLast()**: It removes the last item of the list.

llistobj.removeLast();
23) **Object removeFirstOccurrence(Object item)**: It removes the first occurrence of the specified item.

llistobj.removeFirstOccurrence("text");
It will remove the first occurrence of the string “text” from the list.

24) **Object removeLastOccurrence(Object item)**: It removes the last occurrence of the given element.

llistobj.removeLastOccurrence("String1);
It will remove the last occurrence of string “String1”.

25) **Object set(int index, Object item)**: It updates the item of specified index with the give value.

llistobj.set(2, "Test");
It will update the 3rd element with the string “Test”.

26) **int size()**: It returns the number of elements of the list.

llistobj.size();

Iterating linked list:

```
    /*for loop*/
    System.out.println("**For loop**");
    for(int num=0; num<linkedlist.size(); num++)
    {
            System.out.println(linkedlist.get(num));
    }

    /*Advanced for loop*/
    System.out.println("**Advanced For loop**");
    for(String str: linkedlist)
    {
            System.out.println(str);
    }

    /*Using Iterator*/
    System.out.println("**Iterator**");
    Iterator i = linkedlist.iterator();
    while (i.hasNext()) {
            System.out.println(i.next());
    }
```

/* Using List Iterator we can iterate the list in both the directions(forward and backward). Along with traversing, we can also modify the list during iteration, and obtain the iterator's current position in the list.*/

```
ListIterator listIt = linkedlist.listIterator();

   // Iterating the list in forward direction
   System.out.println("Forward iteration:");
   while(listIt.hasNext()){
      System.out.println(listIt.next());
   }

   // Iterating the list in backward direction
   System.out.println("\n Backward iteration:");
   while(listIt.hasPrevious()){
      System.out.println(listIt.previous());
   }
```

-

Hash Map:

HashMap is a Map based collection class that is used for storing Key & value pairs, it is denoted as HashMap<Key, Value> or HashMap<K, V>. This class makes no guarantees as to the order of the map. It is similar to the Hashtable class except that it is unsynchronized and permits nulls(null values and null key).

It is not an ordered collection which means it does not return the keys and values in the same order in which they have been inserted into the HashMap. It does not sort the stored keys and Values. You must need to import java.util.HashMap or its super class in order to use the HashMap class and methods.

Code:

```
HashMap<Integer, String> hmap = new HashMap<Integer, String>();

    /*Adding elements to HashMap*/
    hmap.put(12, "Chaitanya");
    hmap.put(2, "Rahul");
    hmap.put(7, "Singh");
    hmap.put(49, "Ajeet");
    hmap.put(3, "Anuj");

    /* Display content using Iterator*/
    Set set = hmap.entrySet();
    Iterator iterator = set.iterator();
    while(iterator.hasNext()) {
       Map.Entry mentry = (Map.Entry)iterator.next();
       System.out.print("key is: "+ mentry.getKey() + " & Value is: ");
       System.out.println(mentry.getValue());
    }

    /* Get values based on key*/
    String var= hmap.get(2);
    System.out.println("Value at index 2 is: "+var);

    /* Remove values based on key*/
    hmap.remove(3);
    System.out.println("Map key and values after removal:");
    Set set2 = hmap.entrySet();
```

```
    Iterator iterator2 = set2.iterator();
    while(iterator2.hasNext()) {
       Map.Entry mentry2 = (Map.Entry)iterator2.next();
       System.out.print("Key is: "+mentry2.getKey() + " & Value is: ");
       System.out.println(mentry2.getValue());
     }
```

HashMap Class Methods

Here is the list of methods available in HashMap class. I have also covered examples using these methods at the end of this post.

1. **void clear()**: It removes all the key and value pairs from the specified Map.
2. **Object clone()**: It returns a copy of all the mappings of a map and used for cloning them into another map.
3. **boolean containsKey(Object key)**: It is a boolean function which returns true or false based on whether the specified key is found in the map.
4. **boolean containsValue(Object Value)**: Similar to containsKey() method, however it looks for the specified value instead of key.
5. **Value get(Object key)**: It returns the value for the specified key.
6. **boolean isEmpty()**: It checks whether the map is empty. If there are no key-value mapping present in the map then this function returns true else false.
7. **Set keySet()**: It returns the Set of the keys fetched from the map.
8. **value put(Key k, Value v)**: Inserts key value mapping into the map. Used in the above example.
9. **int size()**: Returns the size of the map – Number of key-value mappings.
10. **Collection values()**: It returns a collection of values of map.
11. **Value remove(Object key)**: It removes the key-value pair for the specified key. Used in the above example.
12. **void putAll(Map m)**: Copies all the elements of a map to the another specified map.

Iterating Hash map:

```
HashMap<Integer, String> hmap = new HashMap<Integer, String>();
      //Adding elements to HashMap
      hmap.put(11, "AB");
      hmap.put(2, "CD");
      hmap.put(33, "EF");
      hmap.put(9, "GH");
      hmap.put(3, "IJ");

      //FOR LOOP
```

```
        System.out.println("For Loop:");
        for (Map.Entry me : hmap.entrySet()) {
          System.out.println("Key: "+me.getKey() + " & Value: " + me.getValue());
        }

        //WHILE LOOP & ITERATOR
        System.out.println("While Loop:");
        Iterator iterator = hmap.entrySet().iterator();
        while (iterator.hasNext()) {
            Map.Entry me2 = (Map.Entry) iterator.next();
          System.out.println("Key: "+me2.getKey() + " & Value: " + me2.getValue());
        }
```

ArrayList vs HashMap in Java

1) **Implementation**: **ArrayList** implements List Interface while **HashMap** is an implementation of Map interface. List and Map are two entirely different collection interfaces.

2) **Memory consumption**: ArrayList stores the element's value alone and internally maintains the indexes for each element.

HashMap stores key & value pair. For each value there must be a key associated in HashMap. That clearly shows that memory consumption is high in HashMap compared to the ArrayList.

3) **Order**: ArrayList maintains the insertion order while HashMap doesn't. Which means ArrayList returns the list items in the same order in which they got inserted into the list. On the other side HashMap doesn't maintain any order, the returned key-values pairs are not sorted in any kind of order.

4) **Duplicates**: ArrayList allows duplicate elements but HashMap doesn't allow duplicate keys (It does allow duplicate values).

5) **Nulls**: ArrayList can have any number of null elements. HashMap allows one null key and any number of null values.

6) **get method**: In ArrayList we can **get** the element by specifying the index of it. In HashMap the elements is being fetched by specifying the corresponding key.

Iterator vs List Iterator:

1) Iterator is used for traversing List and Set both.
We can use ListIterator to traverse List only, we cannot traverse Set using ListIterator.

2) We can traverse in only forward direction using Iterator.

Using ListIterator, we can traverse a List in both the directions (forward and Backward).

3) We cannot obtain indexes while using Iterator

We can obtain indexes at any point of time while traversing a list using ListIterator. The methods nextIndex() and previousIndex() are used for this purpose.

4) We cannot add element to collection while traversing it using Iterator, it throws ConcurrentModificationException when you try to do it.

We can add element at any point of time while traversing a list using ListIterator.

5) We cannot replace the existing element value when using Iterator.

By using set(E e) method of ListIterator we can replace the last element returned by next() or previous() methods.

6) Methods of Iterator:

- hasNext()
- next()
- remove()

Methods of ListIterator:

- add(E e)
- hasNext()

- hasPrevious()
- next()
- nextIndex()
- previous()
- previousIndex()
- remove()
- set(E e)

HashMap:

Great! Another useful built-in data structure in Java is the *HashMap*.

```
HashMap<String, Integer> restaurantMenu = new HashMap<String, Integer>();
```

In the example above, we create a `HashMap` object called `restaurantMenu`. The `restaurant` HashMap will store `keys` of `String` data types and `values` of type `Integer`.

HashMap : Insertion & Access

```
restaurantMenu.put("Turkey Burger",13);
restaurantMenu.put("Naan Pizza",11);
restaurantMenu.put("Cranberry Kale Salad",10);

System.out.println(restaurantMenu.get("Naan Pizza"));

System.out.println(restaurantMenu.size());
for (String item : restaurantMenu.keySet()) {
        System.out.println("A " + item + " costs " + restaurantMenu.get(item) + " dollars.");
}
```

Logging :

Here is an overview diagram of how the Java Logging API works:

All logging is done via a Logger instance. Loggers gather the data to be logged into a LogRecord. The LogRecord is then forwarded to a Handler. The Handler determines what to do with the LogRecord. For instance, the LogRecord can be written to disk, or sent over the network to a surveillance system.

Both Logger's and Handler's can pass the LogRecord through a Filter which determines whether the LogRecord should be forwarded or not.

A Handler can also use a Formatter to format the LogRecord as a string before it is sent to the external disk or system.

Log Level

Whenever a message is logged, this message is logged with a certain log level. The level is an integer which determines how important the message is. The higher the number (level) is, the more important the message is.

A Logger can have a minimum log level set on it, which determines if the message is forwarded to a Handler or not. This is not a Filter, even though it has the same effect. For instance, all messages below a certain level can be suppressed.

Creating Logger :

The java.util.Logger class is the main access point to the Java logging API.

The most common way of using the Java Logging API is to create a Logger in each class that needs to log. This instance is typically made static and final, meaning all instances of that class use the same Logger instance. Here is an example:

```
public class LoggingExamples {
    private static final Logger logger =
```

```
        Logger.getLogger(LoggingExamples.class.getName());
}
```

As you can see from this example, it is common practice to use the class name including package name as name for the Logger. The name of the Logger to create is passed as string parameter to the Logger.getLogger() method.

Logger Hierarchy:

The string passed as parameter to the getLogger() factory method is the name of the Logger to create. You can choose the name freely, but the name implies where the Logger is located in the Logger hierarchy. Every . (dot) in the name is interpreted as a branch in the hierarchy. Look at these names:

```
Logger logger  = Logger.getLogger("myApp");
Logger logger1 = Logger.getLogger("myApp.user");
Logger logger2 = Logger.getLogger("myApp.admin");
Logger logger3 = Logger.getLogger("myApp.admin.import.user");
```

These names are all valid. They also imply a hierarchy. The name "myApp" is at the top of the hierarchy. The two names "myApp.user" and "myApp.admin" are children of the "myApp" name. The name "myApp.admin.import.user" is a branch of the name "myApp.admin.import", which is again a branch of the "myApp.admin" name.

Now, if you call getParent() on logger3, you will get the Logger with the name myApp.admin. The parent of that Logger is named myApp.user etc.

You can obtain the name of a Logger using the getName() method, in case you need it. Here is an example:

```
String name = logger.getName();
```

The log() methods:

The log() group of methods will log a message at a certain log level. The log level is passed as parameter. Use one of the Level constants as parameter. Log level is covered in more detail in its own text.

Some of the log() methods can take object parameters. These object parameters are inserted into the log message, before it is being logged. The merging of object parameters into the message is only performed, if the message is not filtered out, either by a Filter, or because of too low log level. This improves performance in the cases where the message is filtered out.

- Here is a log() example:

```
Logger logger = Logger.getLogger("myLogger");

logger.log(Level.SEVERE, "Hello logging");
```

And here is what is logged to the console (default log destination) :

```
08-01-2012 14:10:43 logging.LoggingExamples main
SEVERE: Hello logging
```

- Here is an example that inserts a parameter into the message:

```
logger.log(Level.SEVERE, "Hello logging: {0} ", "P1");
```

And here is what is being logged:

```
08-01-2012 14:45:12 logging.LoggingExamples main
SEVERE: Hello logging: P1
```

Notice how the object parameter value P1 is inserted at the place in the log message where the {0} is located. The 0 is the index of the object parameter to insert.

- Here is an example that logs a message with multiple object parameters to be inserted into the log message:

```
logger.log(Level.SEVERE, "Hello logging: {0}, {1}", new Object[] {"P1", "P2"});
```

Here is what is being logged:

```
08-01-2012 14:45:12 logging.LoggingExamples main
SEVERE: Hello logging: P1, P2
```

Notice again how the object parameters are inserted into the log message instead of the {0} and {1}tokens. As mentioned earlier, the number inside the token refers to the index of the object parameter to insert, in the object parameter array passed to the log() message.

- Here is an example that logs a Throwable:

```
logger.log(Level.SEVERE, "Hello logging",
    new RuntimeException("Error"));
```

The logp() Methods

The logp() methods work like the log() methods, except each method take an extra two parameters: The sourceClass and sourceMethod parameter.
These two parameters are intended to tell from what class and method the log message originated. In other words, which class and method was the "source" of the log message.

Refer : http://tutorials.jenkov.com/java-logging/logger.html#the-log-methods

The Last Log Methods

The Logger also has the following methods for logging:
entering(String sourceClass, String sourceMethod);
entering(String sourceClass, String sourceMethod, Object param1);
entering(String sourceClass, String sourceMethod, Object[] params);

exiting (String sourceClass, String sourceMethod);
exiting (String sourceClass, String sourceMethod, Object result);

fine (String message);
finer (String message);
finest (String message);

config (String message);
info (String message);
warning (String message);
severe (String message);

Each of these methods corresponds to a log level. For instance, finest(), finer(), fine(), info(), warning() and severe() each corresponds to one of the log levels. Logging message using one of these methods corresponds to calling the log() method

---
-

An overview of App Engine:

- At the highest level, an App engine application is made up of one or more services, which can be configured to use different run times and to operate with different performance settings .
- Services let developers factor large applications into logical components that can share App Engine features such as Memcache and communicate in a secure fashion.

Versions and instances :

- Each service consists of source code and config file. The files used by the service represent a version of the service.
- Having versions of a service allows to roll back to particular service (or) to use traffic splitting to gradually increase traffic to newly deployed versions of the service.

- Each service and each version must have a name. Choose a unique name for each service and each version.
- While running, a particular version will have one or more instances. App engine by default., scales the number of instances running up and down to match the load, thus providing consistent performance for the application all time and minimizes the idle instances.
- While uploading a version of a service, instance class & config files specifies the scaling type. It controls how instances are created.

Datastore :

Google Datastore is a No SQL document database built for automatic scaling, high performance and ease of application development.

Features :

1. **Atomic transaction :**
   Cloud datastore can execute a set of operations where either all succeed or none occur.

2. **High availability of reads and writes**
   Cloud datastore runs in Google data centers, which use redundancy to minimize impacts from points of failure.

3. **Massive scaling with high performance**
   Datastore uses a distributed architecture to automatically manage scaling.

4. **Encryption**
   Datastore automatically encrypts all data before it is written to disk and automatically decrypts the data when read by authorized user.

Concept	**Datastore**	**Relational database**
Category of object	Kind	Table
One object	Entity	Row
Individual data for an object	Property	Field
Unique id for an Object	Key	Primary key

1. Unlike rows in relational database table, entities of same kind can have different properties.
2. Different entities can have properties with same name but different value types.
3. Does not support ‘join’ operation.
4. Does not support ‘sub query’ .

Creating and Using Entity Keys

Each entity in Cloud Datastore has a *key* that uniquely identifies it. The key consists of the following components:

- The *namespace* of the entity, which allows for multitenancy
- The *kind* of the entity, which categorizes it for the purpose of Cloud Datastore queries
- An optional *ancestor path* locating the entity within the Cloud Datastore hierarchy.
- An *identifier* for the individual entity, which can be either
    - a *key name* string
    - an integer *numeric ID*

Because the identifier is part of the entity's key, the identifier is associated permanently with the entity and cannot be changed. You assign the identifier in either of two ways:

- Specify your own *key name* string for the entity.
- Let Cloud Datastore automatically assign the entity an integer *numeric ID*.

Specifying a key name for an entity

To assign a key name to an entity, provide the name as the second argument to the constructor when you create the entity:

```
Entity employee = new Entity("Employee", "asalieri");
```

To let Cloud Datastore assign a numeric ID automatically, omit this argument:

```
Entity employee = new Entity("Employee")
```

Using ancestor paths

Entities in Cloud Datastore form a hierarchically structured space similar to the directory structure of a file system. When you create an entity, you can optionally designate another entity as its *parent;* the new entity is a *child* of the parent entity (note that unlike in a file system, the parent entity need not actually exist). An entity without a parent is a *root entity.* The association between an entity and its parent is permanent, and cannot be changed once the entity is created. Cloud Datastore will never assign the same numeric ID to two entities with the same parent, or to two root entities (those without a parent).

An entity's parent, parent's parent, and so on recursively, are its *ancestors;* its children, children's children, and so on, are its *descendants.* A root entity and all of its descendants belong to the same *entity group.*The sequence of entities beginning with a root entity and

proceeding from parent to child, leading to a given entity, constitute that entity's *ancestor path.* The complete key identifying the entity consists of a sequence of kind-identifier pairs specifying its ancestor path and terminating with those of the entity itself:

[Person:GreatGrandpa, Person:Grandpa, Person:Dad, Person:Me]

For a root entity, the ancestor path is empty and the key consists solely of the entity's own kind and identifier:

[Person:GreatGrandpa]

To designate an entity's parent, provide the parent entity's key as an argument to the Entity()constructor when creating the child entity. You can get the key by calling the parent entity's getKey()method:

```
Entity employee = new Entity("Employee");
datastore.put(employee);
Entity address = new Entity("Address", employee.getKey());
datastore.put(address);
```

If the new entity also has a key name, provide the key name as the second argument to the Entity()constructor and the key of the parent entity as the third argument:

```
Entity address = new Entity("Address", "addr1", employee.getKey());
```

Generating keys

Applications can use the class KeyFactory to create a Key object for an entity from known components, such as the entity's kind and identifier. For an entity with no parent, pass the kind and identifier (either a key name string or a numeric ID) to the static method KeyFactory.createKey() to create the key. The following examples create a key for an entity of kind Person with key name "GreatGrandpa" or numeric ID 74219:

```
Key k1 = KeyFactory.createKey("Person", "GreatGrandpa");
Key k2 = KeyFactory.createKey("Person", 74219);
```

If the key includes a path component, you can use the helper class KeyFactory.Builder to build the path. This class's addChild method adds a single entity to the path and returns the builder itself, so you can chain together a series of calls, beginning with the root entity, to build up the

path one entity at a time. After building the complete path, call getKey to retrieve the resulting key:

```
Key k =
    new KeyFactory.Builder("Person", "GreatGrandpa")
        .addChild("Person", "Grandpa")
        .addChild("Person", "Dad")
        .addChild("Person", "Me")
        .getKey();
```

Class KeyFactory also includes the static methods keyToString and stringToKey for converting between keys and their string representations:

```
String personKeyStr = KeyFactory.keyToString(k);
Key personKey = KeyFactory.stringToKey(personKeyStr);
Entity person = datastore.get(personKey);
```

Creating Entities

Data objects in Cloud Datastore are known as *entities*, each of which is categorized under a particular *kind* for the purpose of queries. For instance, if you are writing a human resources application you might represent each employee with an entity of kind Employee.

The following example creates an entity of kind Employee, populates its property values, and saves it to data store:

```
Entity employee = new Entity("Employee", "asalieri");
employee.setProperty("firstName", "Antonio");
employee.setProperty("lastName", "Salieri");
employee.setProperty("hireDate", new Date());
employee.setProperty("attendedHrTraining", true);

DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
datastore.put(employee);
```

Note : If you don't provide a key name, Cloud Datastore automatically generates a numeric ID for the entity's key:

```
Entity employee = new Entity("Employee");
```

Embedded entities

You may sometimes find it convenient to embed one entity as a property of another entity. This can be useful, for instance, for creating a hierarchical structure of property values within an entity. The Java class EmbeddedEntity allows you to do this:

```
EmbeddedEntity embeddedContactInfo = new EmbeddedEntity();

embeddedContactInfo.setProperty("homeAddress", "123 Fake St, Made, UP 45678");

embeddedContactInfo.setProperty("phoneNumber", "555-555-5555");

embeddedContactInfo.setProperty("emailAddress", "test@example.com");


employee.setProperty("contactInfo", embeddedContactInfo);
```

Retrieving entities

```
Entity emp = datastore.get(employee.getKey());
(or)
Key k = KeyFactory.createKey("Employee", “asalieri”);
Entity emp = datastore.get(k);
```

Updating entities

To update an existing entity, modify the attributes of the Entity object, then pass it to the DatastoreService.put() method. The object data overwrites the existing entity. The entire object is sent to Cloud Datastore with every call to put().

Note : The Cloud Datastore API does not distinguish between creating a new entity and updating an existing one. If the object's key represents an entity that already exists, the **put()** method overwrites the existing entity. You can use a transaction to test whether an entity with a given key exists before creating one.

Deleting entities

Given an entity's key, you can delete the entity with the DatastoreService.delete() method:

```
Key employeeKey = KeyFactory.createKey("Employee", “asalieri”);
datastore.delete(employeeKey);
```

Deleting entities in bulk

You can use the GCP Console to delete all entities of a given kind, or all entities of all kinds, in the default namespace:

1. Go to the Cloud Datastore Admin page:
2. Go to the Cloud Datastore Admin page.
3. If you have not already enabled Cloud Datastore Admin functionality, click **Enable Datastore Admin**.
4. Select the entity kind(s) you want to delete.
5. Click **Delete Entities**. Note that bulk deletion takes place within your application, and thus counts against your quota.

Using batch operations

You can use the batch operations if you want to operate on multiple entities in a single Cloud Datastore call.

Here is an example of a batch call:

```
Entity employee1 = new Entity("Employee");
Entity employee2 = new Entity("Employee");
Entity employee3 = new Entity("Employee");
List<Entity> employees = Arrays.asList(employee1, employee2, employee3);
datastore.put(employees);
```

These batch operations group all the entities or keys by entity group and then perform the requested operation on each entity group in parallel, which is faster than making separate calls for each individual entity because they incur the overhead for only one service call. If the batch uses multiple entity groups, the work for all groups is done in parallel on the server side.

A batch put() or delete() call may succeed for some entities but not others. If it is important that the call succeed completely or fail completely, use a transaction with all affected entities in the same entity group.

Retrieving query results

After constructing a query, you can specify a number of retrieval options to further control the results it returns.

Retrieving a single entity

To retrieve just a single entity matching your query, use the method PreparedQuery.asSingleEntity():

```
Query q = new Query("Person") .setFilter(new FilterPredicate("lastName",
FilterOperator.EQUAL, targetLastName));
PreparedQuery pq = datastore.prepare(q);
Entity result = pq.asSingleEntity();
```

This returns the first result found in the index that matches the query. (If there is more than one matching result, it throws a TooManyResultsException.)

Iterating through query results :

When iterating through the results of a query using the PreparedQuery.asIterable() and PreparedQuery.asIterator() methods, Cloud Datastore retrieves the results in batches. By default each batch contains 20 results, but you can change this value using FetchOptions.chunkSize(). You can continue iterating through query results until all are returned or the request times out.

Retrieving selected properties from an entity

To retrieve only selected properties of an entity rather than the entire entity, use a *projection query*. This type of query runs faster and costs less than one that returns complete entities.

Similarly, a *keys-only query* saves time and resources by returning just the keys to the entities it matches, rather than the full entities themselves. To create this type of query, use the Query.setKeysOnly() method:

```
Query q = new Query("Person").setKeysOnly();
```

Setting a limit for your query

You can specify a *limit* for your query to control the maximum number of results returned in one batch. The following example retrieves the five tallest people from Cloud Datastore:

```
private List<Entity> getTallestPeople() {
  DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
  Query q = new Query("Person").addSort("height", SortDirection.DESCENDING);
  PreparedQuery pq = datastore.prepare(q);
  return pq.asList(FetchOptions.Builder.withLimit(5));
}
```

Datastore Queries

A Datastore *query* retrieves *entities* from Cloud Datastore that meet a specified set of conditions.

A typical query includes the following:

- An *entity kind* to which the query applies
- Optional *filters* based on the entities' property values, keys, and ancestors
- Optional *sort orders* to sequence the results

When executed, a query retrieves all entities of the given kind that satisfy all of the given filters, sorted in the specified order. Queries execute as read-only.

Filters

A query's *filters* set constraints on the properties, keys, and ancestors of the entities to be retrieved.

Property filters

A *property filter* specifies

- A property name
- A comparison operator

- A property value

For example:

```
Filter propertyFilter =
    new FilterPredicate("height", FilterOperator.GREATER_THAN_OR_EQUAL, minHeight);
Query q = new Query("Person").setFilter(propertyFilter);
```

Key filters

To filter on the value of an entity's key, use the special property Entity.KEY_RESERVED_PROPERTY:

```
Filter keyFilter =
    new FilterPredicate(Entity.KEY_RESERVED_PROPERTY, FilterOperator.GREATER_THAN, lastSeenKey);
Query q = new Query("Person").setFilter(keyFilter);
```

Special query types

Some specific types of query deserve special mention:

Kindless queries

A query with no kind and no ancestor filter retrieves all of the entities of an application from Datastore. This includes entities created and managed by other App Engine features, such as statistics entities and Blobstore metadata entities (if any). Such *kindless queries* cannot include filters or sort orders on property values. They can, however, filter on entity keys by specifying Entity.KEY_RESERVED_PROPERTYas the property name:

```
Filter keyFilter =
    new FilterPredicate(Entity.KEY_RESERVED_PROPERTY, FilterOperator.GREATER_THAN, lastSeenKey);
Query q = new Query().setFilter(keyFilter);
```

Ancestor filters

You can filter your Datastore queries to a specified *ancestor*, so that the results returned will include only entities descended from that ancestor:

```
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
Entity tom = new Entity("Person", "Tom");
Key tomKey = tom.getKey();
datastore.put(tom);

Entity weddingPhoto = new Entity("Photo", tomKey);
weddingPhoto.setProperty("imageURL", "http://domain.com/some/path/to/wedding_photo.jpg");

Entity babyPhoto = new Entity("Photo", tomKey);
babyPhoto.setProperty("imageURL", "http://domain.com/some/path/to/baby_photo.jpg");

Entity dancePhoto = new Entity("Photo", tomKey);
dancePhoto.setProperty("imageURL", "http://domain.com/some/path/to/dance_photo.jpg");

Entity campingPhoto = new Entity("Photo");
campingPhoto.setProperty("imageURL", "http://domain.com/some/path/to/camping_photo.jpg");

List<Entity> photoList = Arrays.asList(weddingPhoto, babyPhoto, dancePhoto, campingPhoto);
datastore.put(photoList);

Query photoQuery = new Query("Photo").setAncestor(tomKey);

// This returns weddingPhoto, babyPhoto, and dancePhoto,
// but not campingPhoto, because tom is not an ancestor
List<Entity> results =
    datastore.prepare(photoQuery).asList(FetchOptions.Builder.withDefaults());
```

Kindless ancestor queries

A kindless query that includes an ancestor filter will retrieve the specified ancestor and all of its descendants, regardless of kind. This type of query does not require custom indexes. Like all kindless queries, it cannot include filters or sort orders on property values, but can filter on the entity's key:

```
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();

Entity tom = new Entity("Person", "Tom");
Key tomKey = tom.getKey();
datastore.put(tom);

Entity weddingPhoto = new Entity("Photo", tomKey);
```

```
weddingPhoto.setProperty("imageURL", "http://domain.com/some/path/to/wedding_photo.jpg");

Entity weddingVideo = new Entity("Video", tomKey);
weddingVideo.setProperty("videoURL", "http://domain.com/some/path/to/wedding_video.avi");

List<Entity> mediaList = Arrays.asList(weddingPhoto, weddingVideo);
datastore.put(mediaList);

// By default, ancestor queries include the specified ancestor itself.
// The following filter excludes the ancestor from the query results.
Filter keyFilter =
    new FilterPredicate(Entity.KEY_RESERVED_PROPERTY, FilterOperator.GREATER_THAN,
tomKey);
Query mediaQuery = new Query().setAncestor(tomKey).setFilter(keyFilter);
// Returns both weddingPhoto and weddingVideo,
// even though they are of different entity kinds
List<Entity> results =
    datastore.prepare(mediaQuery).asList(FetchOptions.Builder.withDefaults());
```

Projection queries

Sometimes all you really need from the results of a query are the values of a few specific properties. In such cases, you can use a *projection query* to retrieve just the properties you're actually interested in, at lower latency and cost than retrieving the entire entity; see the Projection Queries page for details.

Sort orders

A query *sort order* specifies

- A property name
- A sort direction (ascending or descending)

```
Query q1 = new Query("Person").addSort("lastName", SortDirection.ASCENDING);

// Order by height, tallest to shortest:
```

```
Query q2 = new Query("Person").addSort("height", SortDirection.DESCENDING);
```

If a query includes multiple sort orders, they are applied in the sequence specified. The following example sorts first by ascending last name and then by descending height:

```
Query q =
    new Query("Person")
        .addSort("lastName", SortDirection.ASCENDING)
        .addSort("height", SortDirection.DESCENDING);
```

If no sort orders are specified, the results are returned in the order they are retrieved from Cloud Datastore.

Query interface example

The low-level Java Datastore API provides class Query for constructing queries and the PreparedQuery interface for retrieving entities from Datastore:

```
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();

Filter heightMinFilter =
    new FilterPredicate("height", FilterOperator.GREATER_THAN_OR_EQUAL, minHeight);

Filter heightMaxFilter =
    new FilterPredicate("height", FilterOperator.LESS_THAN_OR_EQUAL, maxHeight);

// Use CompositeFilter to combine multiple filters
CompositeFilter heightRangeFilter =
    CompositeFilterOperator.and(heightMinFilter, heightMaxFilter);

// Use class Query to assemble a query
Query q = new Query("Person").setFilter(heightRangeFilter);

// Use PreparedQuery interface to retrieve results
PreparedQuery pq = datastore.prepare(q);

for (Entity result : pq.asIterable()) {
  String firstName = (String) result.getProperty("firstName");
```

```
  String lastName = (String) result.getProperty("lastName");
  Long height = (Long) result.getProperty("height");

  out.println(firstName + " " + lastName + ", " + height + " inches tall");
}
```

Notice the use of FilterPredicate and CompositeFilter to construct filters. If you're setting only one filter on a query, you can just use FilterPredicate by itself:

```
Filter heightMinFilter =
    new FilterPredicate("height", FilterOperator.GREATER_THAN_OR_EQUAL, minHeight);
Query q = new Query("Person").setFilter(heightMinFilter);
```

However, if you want to set more than one filter on a query, you must use CompositeFilter, which requires at least two filters.

Projection Queries

Most Cloud Datastore queries return whole entities as their results, but often an application is actually interested in only a few of the entity's properties. *Projection queries* allow you to query Cloud Datastore for just those specific properties of an entity that you actually need, at lower latency and cost than retrieving the entire entity.

Projection queries are similar to SQL queries of the form:

```
SELECT name, email, phone FROM CUSTOMER
```

You can use all of the filtering and sorting features available for standard entity queries, subject to the limitations described below. The query returns abridged results with only the specified properties (`name`, `email`, and `phone` in the example) populated with values; all other properties have no data.

```
Query projection_query = new Query("Employee");
projection_query.addProjection(new PropertyProjection("firstName",String.class));
PreparedQuery projection = datastore.prepare(projection_query);
```

Limitations on projections

Projection queries are subject to the following limitations:

- **Only indexed properties can be projected.**
- Projection is not supported for properties that are not indexed, whether explicitly or implicitly. Long text strings (Text) and long byte strings (Blob) are not indexed.
- **The same property cannot be projected more than once.**

- **Properties referenced in an equality (`EQUAL`) or membership (`IN`) filter cannot be projected.**
  For example,

  `SELECT A FROM kind WHERE B = 1`

  is valid (projected property not used in the equality filter), as is

  `SELECT A FROM kind WHERE A > 1`

  (not an equality filter), but

  `SELECT A FROM kind WHERE A = 1`

  (projected property used in equality filter) is not.

- **Results returned by a projection query should not be saved back to Cloud Datastore.**
  Because the query returns results that are only partially populated, you should not write them back to Cloud Datastore.

Properties and value types

The data values associated with an entity consist of one or more *properties.* Each property has a name and one or more values. A property can have values of more than one type, and two entities can have values of different types for the same property. Properties can be indexed or unindexed (queries that order or filter on a property *P* will ignore entities where *P* is unindexed). An entity can have at most 20,000 indexed properties.

Cloud Datastore supports a variety of data types for property values. These include, among others:

- Integers
- Floating-point numbers
- Strings
- Dates
- Binary data

For a full list of types, see Properties and value types.

Projections and multiple-valued properties

Projecting a property with multiple values will not populate all values for that property. Instead, a separate entity will be returned for each unique combination of projected values matching the query. For example, suppose you have an entity of kind Foo with two multiple-valued properties, A and B:

```
entity = Foo(A=[1, 1, 2, 3], B=['x', 'y', 'x'])
```

Then the projection query

```
SELECT A, B FROM Foo WHERE A < 3
```

will return four entities with the following combinations of values:

```
A = 1, B = 'x'
A = 1, B = 'y'
A = 2, B = 'x'
A = 2, B = 'y'
```

Task Queue:

The Task Queue API lets applications perform work, called *tasks*, asynchronously outside of a user request. If an app needs to execute work in the background, it adds tasks to *task queues*. The tasks are executed later, by worker services.

Push queues and pull queues

Task queues come in two flavors, *push* and *pull*. The manner in which the Task Queue service dispatches task requests to worker services is different for the different queues.

Push queues dispatch requests at a reliable, steady rate. They guarantee reliable task execution. Because you can control the rate at which tasks are sent from the queue, you can control the workers' scaling behavior and hence your costs.

Because tasks are executed as App Engine requests targeted at services, they are subject to stringent deadlines. Tasks handled by automatic scaling services must finish in ten minutes. Tasks handled by basic and manual scaling services can run for up to 24 hours.

Pull queues do not dispatch tasks at all. They depend on other worker services to "lease" tasks from the queue on their own initiative. Pull queues give you more power and flexibility over

when and where tasks are processed, but they also require you to to do more process management. When a task is leased the leasing worker declares a deadline. By the time the deadline arrives the worker must either complete the task and delete it or the Task Queue service will allow another worker to lease it.

All task queue tasks are performed asynchronously. The application that creates the task is not notified whether or not the task completed, or if it was successful. The task queue service provides a retry mechanism, so if a task fails it can be retried a finite number of times.

Push Queues in Java

Push queues run tasks by delivering HTTP requests to App Engine worker services. The requests are delivered at a constant rate. If a task fails, the service will retry the task, sending another request. You must write a handler for every kind of task you use. A single service can have multiple handlers for different kinds of tasks, or you can use different services for different task types.

The task deadline

When a worker service receives a push task request, it must handle the request and send an HTTP response before a deadline that depends on the scaling type of the worker service.

Automatic scaling services must finish before 10 minutes have elapsed. Manual and basic scaling services can run up to 24 hours.

An HTTP response code between 200–299 indicates success, all other values indicate the task failed. If the task fails to respond within the deadline, or returns an invalid response value, the task will be retried.

Retrying a failed task

If a push task request handler returns an HTTP status code outside the range 200–299, or fails to return any response before the task deadline occurs, App Engine retries the task until it succeeds. The system backs off gradually to avoid flooding your application with too many requests, but schedules retry attempts for failed tasks to recur at a maximum of once per hour.

Creating Push Queues:

Using queue.xml to create queues

To process a task, you must add it to a push queue. App Engine provides a default push queue, named default, which is configured and ready to use with default settings. If you want, you can just add all your tasks to the default queue, without having to create and configure other queues.

If you need additional queues, or want to change the default queue configuration, you can do so in the queue.xml file for your application, which you upload to App Engine.

Note that you cannot create queues dynamically in your program. You can create up to **10 queues for free** applications, and up to **100 queues for billing-enabled** applications.

```
<?xml version="1.0" encoding="UTF-8"?>
 <queue-entries>
   <queue>
     <name>queue-blue</name>
     <target>v2.task-module</target>
   </queue>
   <queue>
     <name>queue-red</name>
     <rate>1/s</rate>
   </queue>
 </queue-entries>
```

All tasks added to queue-blue are sent to the target module v2.task-module. The refresh rate of queue-red is changed from 5/s to 1/s. Tasks will be dequeued and sent to their targets at the rate of 1 task per second.

Note: Although App Engine might appear to process tasks in the order in which they are enqueued, it is normal for tasks to be executed in arbitrary order, so your implementation should not assume that tasks are executed serially or in any other order**.**

Note: If you delete a queue, you must wait approximately 7 days before creating a new queue with the same name.

Defining the push queue processing rate

You can control the rate at which tasks are processed in each of your queues by defining other directives, such as <rate>, <bucket-size>, and <max-concurrent-requests>.

The task queue uses token buckets to control the rate of task execution. Each named queue has a token bucket that holds tokens, up to the maximum specified by the bucket_size, **or a maximum of 5 tokens** if you don't specify the bucket size.

Each time your application executes a task, a token is removed from the bucket. Your app continues processing tasks in the queue until the queue's bucket runs out of tokens. App Engine

refills the bucket with new tokens continuously based on the **<rate>** that you specified for the queue.

If your queue contains tasks to process, and the queue's bucket contains tokens, App Engine simultaneously processes as many tasks as there are tokens. This can lead to bursts of processing, consuming system resources and competing with user-serving requests.

If you want to prevent too many tasks from running at once or to prevent datastore contention, you use <max-concurrent-requests>.

The following samples shows how to set <max-concurrent-requests> to limit tasks and also shows how to adjust the bucket size and rate based on your application's needs and available resources:

```
<?xml version="1.0" encoding="UTF-8"?>
<queue-entries>
  <queue>
    <name>optimize-queue</name>
    <rate>20/s</rate>
    <bucket-size>40</bucket-size>
    <max-concurrent-requests>10</max-concurrent-requests>
  </queue>
</queue-entries>
```

Setting storage limits for all queues

You can use queue.xml to define the total amount of storage that task data can consume over all queues. To define the total storage limit, include an element named <total-storage-limit> at the top level:

```
<?xml version="1.0" encoding="UTF-8"?>
<queue-entries>
  <total-storage-limit>120M</total-storage-limit>
  <queue>
    <name>foo</name>
    <rate>35/s</rate>
  </queue>
</queue-entries>
```

The value is a number followed by a unit: B for bytes, K for kilobytes, M for megabytes, G for gigabytes, T for terabytes. For example, 100K specifies a limit of 100 kilobytes. If adding a task would cause the queue to exceed its storage limit, the call to add the task will fail.

The default limit is **500M(500 megabytes)** for free apps. **For billed apps there is no limit** until you explicitly set one.

Configuring the maximum number of concurrent requests

You can further control the processing rate by setting <max-concurrent-requests>, which limits the number of tasks that can execute simultaneously.

If your application queue has a rate of 20/s and a bucket size of 40, tasks in that queue execute at a rate of 20/s and can burst up to 40/s briefly. These settings work fine if task latency is relatively low; however, if latency increases significantly, you'll end up processing significantly more concurrent tasks. This extra processing load can consume extra instances and slow down your application.

For example, let's assume that your normal task latency is 0.3 seconds. At this latency, you'll process at most around 40 tasks simultaneously. But if your task latency increases to 5 seconds, you could easily have over 100 tasks processing at once. This increase forces your application to consume more instances to process the extra tasks, potentially slowing down the entire application and interfering with user requests.

You can avoid this possibility by setting <max-concurrent-requests> to a lower value. For example, if you set <max-concurrent-requests> to 10, our example queue maintains about 20 tasks/second when latency is 0.3 seconds. However, when the latency increases over 0.5 seconds, this setting throttles the processing rate to ensure that no more than 10 tasks run simultaneously.

```
<?xml version="1.0" encoding="utf-8"?>
<queue-entries>
  <queue>
    <name>optimize-queue</name>
    <rate>20/s</rate>
    <bucket-size>40</bucket-size>
    <max-concurrent-requests>10</max-concurrent-requests>
  </queue>
</queue-entries>
```

Creating Push Tasks:

Creating a new task:

To create and enqueue a task, get a Queue using the QueueFactory, and call its add() method. You can get a named queue specified in the queue.xml file using the getQueue() method of the factory, or you can get the default queue using getDefaultQueue(). You can call the Queue's

add() method with a TaskOptions instance (produced by TaskOptions.Builder, or you can call it with no arguments to create a task with the default options for the queue.

```
Queue queue = QueueFactory.getDefaultQueue();
queue.add(TaskOptions.Builder.withUrl("/worker").param("key", key));
```

Specifying the worker service

When a task is popped off its queue, the Task Queue service sends it on to a worker service. Every task has a *target* and a *url*, which determine what service and handler will ultimately perform the task.

target

The target specifies the service that will receive the HTTP request to perform the task. It is a string that specifies a service/version/instance in any one of the canonical forms. The most often-used ones are:

```
service
version.service
Instance.version.service
```

The target string is prepended to the domain name of your app. There are three ways to set the target for a task:

- Explicitly declare the target when you construct the task.
- Include a target directive when you define a queue in the queue.xml, as in the definition of queue-blue. All tasks added to a queue with a target will use that target, even if a different target was assigned to the task at construction time.
- If no target is specified according to either of the previous two methods, then the task's target is the version of the service that enqueues it. Note that if you enqueue a task from the default service and version in this manner, and the default version changes before the task executes, it will run in the new default version.

url

The url selects one of the handlers in the target service, which will perform the task.

The url should match one of the handler URL patterns in the target service. The url can include query parameters if the method specified in the task is GET or PULL. If no url is specified the default URL /_ah/queue/[QUEUE_NAME] is used, where [QUEUE_NAME] is the name of the task's queue.

Passing data to the handler

You can pass data to the handler as query parameters in the task's URL; but only if the method specified in the task is GET or PULL.

TaskOptions.Builder constructor has methods to add data as the payload of the HTTP request, and as parameters, which are added to the URL as query parameters.

params

Do not specify params if you are using the POST method along with a payload, or if you are using the GET method and you've included a url with query parameters.

Naming a task

When you create a new task, App Engine assigns the task a unique name by default. However, you can assign your own name to a task by using the name parameter. An advantage of assigning your own task names is that named tasks are de-duplicated, which means you can use task names to guarantee that a task is only added once. De-duplication continues for 9 days after the task is completed or deleted.

Note that de-duplication logic introduces significant performance overhead, resulting in increased latencies and potentially increased error rates associated with named tasks. These costs can be magnified significantly if task names are sequential, such as with timestamps. So,

if you assign your own names, we recommend using a well-distributed prefix for task names, such as a hash of the contents.

If you assign your own names to tasks, note that the maximum name length is 500 characters, and the name can contain uppercase and lowercase letters, numbers underscores, and hyphens.

Adding tasks asynchronously

By default, the Task Queue API calls are synchronous. For most scenarios, synchronous calls work fine. For instance, adding a task is usually a fast operation: the median time to add a task is 5 ms and 1 out of every 1000 tasks can take up to 300 ms. Periodic incidents, such as back-end upgrades, can cause spikes to 1 out of every 1000 tasks taking up to 1 second.

If you are building an application that needs low latency, the Task Queue API provides asynchronous calls that minimize latency.

Consider the case where you need to add 10 tasks to 10 different queues (thus you cannot batch them). In the worst case, calling queue.add() 10 times in a loop could block up to 10 seconds, although it's very rare. Using the asynchronous interface to add tasks to their respective queues in parallel, you can reduce the worst-case latency to 1 second.

If you want to make asynchronous calls to a task queue, use the asynchronous methods provided by the Queue class. Call get on the returned Future to force the request to complete. When asynchronously adding tasks in a transaction, you should call get() on the Future before committing the transaction to ensure that the request has finished.

Creating Task Handlers

App Engine executes tasks by sending HTTP requests to your application. You must provide a request handler to execute your task code. The mapping from the request URL to the code is declared in your service's web.xml, just like any other request handler. Because you control how to map task requests to a handler, you're free to organize your task handlers. If your application

processes many different kinds of tasks, you can add all the handlers to a single service, or you can distribute them among multiple services.

Writing a push task request handler

The Task Queue service creates an HTTP header and sends it to an instance of the worker service specified by the task's target. App Engine sends Task Queue requests from the IP address `0.1.0.2`.

Your handler does not need to be written in the same language that created and enqueued the task if you write it in a separate service.

When you write you handler, follow these guidelines:

- The code should return an HTTP status code within the range 200–299 to indicate success. Any other code indicates that the task failed.
- Push tasks have a fixed completion deadline that depends on the scaling type of the service that's running them. Automatic scaling services must finish before 10 minutes have elapsed. Manual and basic scaling services can run up to 24 hours. If your handler misses the deadline, the Task Queue service assumes the task failed and will retry it.
- When a task's execution time nears the deadline, App Engine raises a `DeadlineExceededException` before the deadline is reached, so you can save your work or log whatever progress was made.
- It is important to consider whether the handler is idempotent. App Engine's Task Queue API is designed to provide "at least once" delivery; that is, if a task is successfully added, App Engine will deliver it at least once. Note that in some rare circumstances, multiple task execution is possible, so your code must ensure that there are no harmful side-effects of repeated execution.

Task Queue uses the HTTP code in the handler's response to determine if the task succeeded. Task Queue ignores all other fields in the response. No data is sent back to your application. If a task fails, the Task Queue service will retry the task by sending another request.

The response is only seen by the Task Queue service to determine if the task succeeded. The service discards the response, so your app will never see any of the data. You should not include any application data in the response. If a task fails, the Task Queue service will retry the task by sending another request.

Securing task handler URLs

If a task performs sensitive operations (such as modifying data), you might want to secure its worker URL to prevent a malicious external user from calling it directly. You can prevent users from accessing task URLs by restricting access to App Engine administrators. Task requests themselves are issued by App Engine and can always target restricted URL.

You can read about restricting URLs at Security and Authentication.

Deleting Tasks & Queues:

Deleting task from a queue:

```
Queue q = QueueFactory.getQueue("queue1");
q.deleteTask("foo")
```

To delete all task from a queue:

```
Queue queue = QueueFactory.getQueue("foo");
queue.purge();
```

Warning: Do not create new tasks immediately after purging a queue. Wait at least a second. Tasks created in close temporal proximity to a purge call will also be purged.

Disabling queues

You can disable, or pause, a queue by removing its definition from your `queue.yaml` file and then upload the `queue.yaml` file.

If you disable a queue by omitting it from the `queue.yaml` and uploading it, the queue is set to "disabled" and its rate is set to 0. Any tasks that remain on the queue or new tasks that are added to the queue will not be processed. You can re-enable the disabled queue by uploading a new `queue.yaml` file with the queue defined.

To delete a queue:

1. Remove the queue definition from your `queue.yaml` file.
2. Upload the change to your `queue.yaml` file.
3. Delete the queue in the GCP Console, select the queue and click **Delete queue**:

If you delete a queue from the GCP Console, you must wait 7 days before recreating with the same name.

Retrying Failed Push tasks

Push tasks executing in the task queue can fail for many reasons. If a task fails to execute (by returning any HTTP status code outside of the **range 200–299**), App Engine retries the task until it succeeds. By default, the system gradually reduces the retry rate to avoid flooding your application with too many requests, but schedules retry attempts to recur at a maximum of once per hour until the task succeeds.

Push queues and pull queues differ in how they retry tasks, as described in the following sections.

Retrying tasks

In push queues, you can specify your own scheme for task retries by adding the retry parameters element in queue.xml. This addition allows you to specify the maximum number of times to retry failed tasks in a specific queue. You can also set a time limit for retry attempts and control the interval between attempts.

The following example demonstrates various retry scenarios:

- In fooqueue, tasks are retried up to seven times and for up to two days from the first execution attempt. After both limits are passed, it fails permanently.
- In barqueue, App Engine attempts to retry tasks, increasing the interval linearly between each retry until reaching the maximum backoff and retrying indefinitely at the maximum interval (so the intervals between requests are 10s, 20s, 30s, ..., 190s, 200s, 200s, ...).
- In bazqueue, the retry interval starts at 10s, then doubles three times, then increases linearly, and finally retries indefinitely at the maximum interval (so the intervals between requests are 10s, 20s, 40s, 80s, 160s, 240s, 300s, 300s, ...).

```
<?xml version="1.0" encoding="utf-8"?>
<queue-entries>
  <queue>
    <name>fooqueue</name>
    <rate>1/s</rate>
    <retry-parameters>
      <task-retry-limit>7</task-retry-limit>
      <task-age-limit>2d</task-age-limit>
    </retry-parameters>
  </queue>
  <queue>
    <name>barqueue</name>
    <rate>1/s</rate>
    <retry-parameters>
      <min-backoff-seconds>10</min-backoff-seconds>
      <max-backoff-seconds>200</max-backoff-seconds>
      <max-doublings>0</max-doublings>
```

```
    </retry-parameters>
  </queue>
  <queue>
    <name>bazqueue</name>
    <rate>1/s</rate>
    <retry-parameters>
      <min-backoff-seconds>10</min-backoff-seconds>
      <max-backoff-seconds>200</max-backoff-seconds>
      <max-doublings>3</max-doublings>
    </retry-parameters>
  </queue>
</queue-entries>
```

Pull Queue:

In push queues tasks are delivered to a worker service based on the queue's configuration. In pull queues the worker service must *ask* the queue for tasks. The queue responds by allowing that worker unique access to process the task for a specified period of time, which is called a *lease*.

Using pull queues, you can also group related tasks using tags and then configure your worker to pull multiple tasks with a certain tag all at once. This process is called *batching*.

If a worker cannot process a task before its lease expires, it can either renew the lease or let it expire, at which point another worker can acquire it. Once the work associated with a task is complete, the worker must delete it.

Using pull queues requires your code to handle some functions that are automated in push queues:

Scaling your workers

Your code needs to scale the number of workers based on processing volume. If your code does not handle scaling, you risk wasting computing resources if there are no tasks to process; you also risk latency if you have too many tasks to process.

Deleting the tasks

Your code also needs to explicitly delete tasks after processing. In push queues, App Engine deletes the tasks for you. If your worker does not delete pull queue tasks after processing, another worker will re-process the task. This wastes computing resources and risks errors if tasks are not idempotent.

Pull queues in the App Engine standard environment are created by setting a property in a configuration file called queue.xml. Pull queues can also be set up outside of the App Engine standard environment using Cloud Tasks, now in alpha.

Creating Pull Queues:

You create pull queues using the queue.xml file for your application. The process is the same as creating named push queues, with a specialized directive, <mode>pull</mode>, added to the file.

```
<?xml version="1.0" encoding="UTF-8"?>
<queue-entries>
  <queue>
    <name>my-queue-name</name>
    <mode>pull</mode>
  </queue>
</queue-entries>
```

Disabling queues

You can disable, or pause, a queue by removing its definition from your queue.xml file and then uploading the updated file.

If you disable a queue by omitting it from the queue.xml and uploading it, the queue is set to "disabled" and its rate is set to 0. Any tasks that are in the queue or new tasks that are added to the queue will not be processed. You can re-enable the disabled queue by uploading a new queue.xml file with the queue defined.

You can also pause a queue from within the Task Queues page in the GCP Console

To delete a queue:

4. Remove the queue definition from your queue.yaml file.
5. Upload the change to your queue.yaml file.
6. Delete the queue in the GCP Console, select the queue and click **Delete queue**

Creating Pull tasks:

First you need the name of the queue, which is defined in queue.xml. Then you use the builder and TaskOptions.Method.PULL to add the task .

First, get the queue using the queue name defined in the queue.xml:

```
Queue q = QueueFactory.getQueue("pull-queue");
```

Then use the queue's add() method with TaskOptions.Method.PULL to place tasks in a pull queue named pull-queue:

```
q.add(TaskOptions.Builder.withMethod(TaskOptions.Method.PULL)
    .payload(content.toString()));
```

Leasing Pull Tasks:

Once tasks are in a pull queue, a worker can lease them. After the tasks are processed the worker must delete them.

After the tasks are in the queue, a worker can lease one or more of them using the leaseTasks()method. There may be a short delay before tasks recently added using add() become available via leaseTasks().

When you request a lease, you specify the number of tasks to lease (up to a maximum of 1,000 tasks) and the duration of the lease in seconds (up to a maximum of one week). The lease duration needs to be long enough to ensure that the slowest task will have time to finish before the lease period expires. You can modify a task lease using modifyTaskLease().

Leasing a task makes it unavailable for processing by another worker, and it remains unavailable until the lease expires.

The following code sample leases 100 tasks from the queue pull-queue for one hour:

```
List<TaskHandle> tasks = q.leaseTasks(3600, TimeUnit.SECONDS, numberOfTasksToLease);
```

Batching with tags:

Not all tasks are alike; your code can "tag" tasks and then choose tasks to lease by tag. The tag acts as a filter.

```
q.add(TaskOptions.Builder.withMethod(TaskOptions.Method.PULL)
    .payload(content.toString())
    .tag("process".getBytes()));
```

Then lease the filtered tasks:

```
List<TaskHandle> tasks = q
    .leaseTasksByTag(3600, TimeUnit.SECONDS, numberOfTasksToLease, "process");
```
