Java - Notes - Chapter 1 - The Java Programming Environment

+ Java language is both a technology and a platform.

+ Language:

- Programming language is a notation to write program.
  e.g. C, C++, C# etc.
- Every programming language has following elements
  * Syntax -
  * Semantics -
  * Type System [Data types] -
  * Operator Set -
  * Built in features -
  * Standard library and runtime system. -
- Java has all above elements hence Java is considered as programming language.

+ Technology :

- Using language and technology we can develop the application.
- Let's consider example of asp.net. Asp.net helps us to create Web application and web services so it is classified as technology but it is not a language. To implement it we should use C# as a programming language.
- Using java, it is possible to create different types of application hence it is considered as technology.

+ Platform :

- A platform is the hardware or software environment in which a program runs.
- Platforms can be described as a combination of the operating system and underlying hardware(e.g MS Windows, Linux, Solaris OS, Mac OS etc) or software only platform(e.g MS.NET, Java).
- The Java platform has two components:
  * Java virtual machine[JVM]
  * Java application programming interface[Java API].

+ History of Java :

- Java goes back to 1991, when a group of Sun engineers, led by Patrick Naughton and James Gosling, wanted to design a small computer language that could be used for consumer devices like cable TV switchboxes. Since these devices do not have a lot of power or memory, the language had to be small and generate very tight code. Also, as different manufacturers may choose different CPU's, it was important that the language not be tied to any single architecture. The project was code-named "Green"

- The requirements for small, tight, and platform-neutral code led the team to design a portable language that generated intermediate code for a virtual machine. The Sun people came from a UNIX background, so they based their language on C++.

- Gosling decided to call his language The people at Sun later realized that Oak was the name of an existing computer language, so they changed the name to Java.

Java - Notes - Chapter 1 - The Java Programming Environment

- In 1992, the Green project delivered its first product, called "*7."It was an extremely intelligent remote control. Unfortunately, no one was interested in producing this at Sun, and the Green people had to find other ways to market their technology. However, none of the standard consumer electronics companies were interested either. The group then bid on a project to design a cable TV box that could deal with emerging cable services such as video-on-demand. They did not get the contract.

- The Green project spent all of 1993 and half of 1994 looking for people to buy its technology. No one was found.

- First Person was dissolved in 1994. While all of this was going on at Sun, the World Wide Web part of the Internet was growing bigger and bigger. The key to the World Wide Web was the browser translating hypertext pages to the screen. In 1994, most people were using Mosaic, a noncommercial web browser that came out of the supercomputing center at the University of Illinois in 1993.

- In the SunWorld interview, Gosling says that in mid-1994, the language developers realized that we could build a real cool browser. It was one of the few things in the client/server mainstream that needed some of the weird things we'd done: architecture-neutral, real-time, reliable, secure-issues that weren't terribly important in the workstation world. So we built a browser

- The actual browser was built by Patrick Naughton & Jonathan Payne and evolved into the HotJava browser, which was designed to show off the power of Java. The builders made the browser capable of executing Java code inside web pages. This "proof of technology was shown at SunWorld on May 23, 1995, and inspired the Java craze that continues today.

+ Java Platforms:

- Java is not specific to any processor or operating system as Java platforms have been implemented for a wide variety of hardware and operating systems with a view to enable Java programs to run identically on all of them. Different platforms target different classes of device and application domains:

- **Java Card:** A technology that allows small Java-based applications to be run securely on smart cards and similar small-memory devices.

- **Java ME (Micro Edition) :** Specifies several different sets of libraries (known as profiles) for devices with limited storage, display, and power capacities. It is often used to develop applications for mobile devices, PDAs, TV set-top boxes, and printers.

- Java SE (Standard Edition) : Java Platform, Standard Edition or Java SE is a widely used platform for development and deployment of portable code for desktop environments

- Java EE (Enterprise Edition) : Java Platform, Enterprise Edition or Java EE is a widely used enterprise computing platform developed under the Java Community Process. The platform provides an API and runtime environment for developing and running enterprise software, including network and web services, and other large-scale, multi-tiered, scalable, reliable, and secure network applications.

JDK and JRE :

- SDK is software development kit which is required to develop application.
- SDK = software development tools + documentation + [libraries + runtime environment].
- JDK is java platforms SDK. It is a software development environment used for developing Java applications and applets.
- JDK - java tools + java docs + JRE.,
- JRE - Java API (java class libraries) + Java virtual machine(jvm).
- All core java fundamental classes are part of rt.jar file. JVM and "rt.jar" is integrated part of JRE.
- To develop application it is necessary to have both i.e. JDK and JRE on developer's machine.
- To deploy java application on client's machine, it is necessary to install only JRE on client's machine.

JDK + JRE ⟶ To develop.
JRE ⟶ To deploy & run on client machine

+ Java SE Naming and Versions:

- The Java Platform name has changed a few times over the years.
- Java was first released in January 1996 and was named Java Development Kit, abbreviated JDK.
- Version 1.2 was a large change in the platform and was therefore rebranded as Java 2. Full name: Java 2 Software Development Kit, abbreviated to Java 2 SDK or J2SDK.
- Version 1.5 was released in 2004 as J2SDK 5.0 â€“dropping the "1."from the official name and was further renamed in 2006. Sun simplified the platform name to better reflect the level of maturity, stability, scalability, and security built into the Java platform. Sun dropped the "2" from the name. The development kit reverted back to the name "JDK" from "Java 2 SDK". The runtime environment has reverted back to "JRE" from "J2RE."

- JDK 6 and above no longer use the "dot number"at the end of the platform version.

+ Directory structure of JDK :

- "/usr/lib/jvm/java-8-openjdk" is a installation directory of JDK in Ubuntu. Following is the directory structure of JDK.

Directory structure	Description
openjdk	[The name may be different].
|	
|- bin	The compiler and other java tools.
|- docs	Library documenation in HTML format.
|- include	Files for compiling native methods.
|- jre	Java runtime environment files.
|- lib	Library files.
|- src	The library source code[Extract src.zip].
|- man	

+ Simple Hello World Application:

- Let us write simple "Hello World" application using editor[e.g vim/gedit/Leafpad].

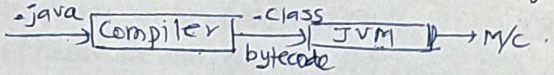
```
-       class MyProgram
        {
                public static void main( String[] args )
                {
                        System.out.println("Hello World.");
                }
        }
```

- Steps to Compile and execute java application.

```
export PATH=/usr/bin/                   //To locate java tools

javac -d . MyProgram.java               //output : MyProgram.class

export CLASSPATH=.;                     //To locate .class file

java MyProgram                          //To execute java app
```

- In the Java programming language, all source code is first written in plain text files ending with the .java extension. Those source files are then compiled into .class files by the javac compiler. A .class file does not contain code that is native to your processor; it instead contains byte codes - the machine language of the Java Virtual Machine (Java VM). The java launcher tool then runs your application with an instance of the Java Virtual Machine.

+ Bytecode :



- java compiler is responsible for generating bytecode.
- It is architecure neutral code. Bytecode is also called as a virtual or managed code. -Using "javap -c"command we can view bytecode.
- Bytecode is an object oriented assembly language code which can be understand by the java virtual machine [JVM].

+ Path and Classpath :

- PATH is an operating systems environment variable which is used to locate "javac" file.

- CLASSPATH is a java platforms environment varibale which is used to locate ".class" file.

+ Let us seperate .java file from .class file. Consider the following directory structure.

```
- Complex[ dir ]*
|
|- src[ dir ]
|       |
```

```
|       |- MyProgram.java
|- bin[ dir ]
|
```

- Now, let us compile and execute MyProgram.java file from Complex directory.

```
export PATH=/usr/bin/;

javac -d ./bin/ ./src/MyProgram.java

export CLASSPATH=./bin/;

java MyProgram
```

- After compilation MyProgram.class file will be stored in bin folder. See the following directory structure.

```
Complex[ dir ]*
|
|- src[ dir ]
|       |
|       |- MyProgram.java
|- bin[ dir ]
|       |
|       |- MyProgram.class
|
```

+ Java 8 New Features:

- The newest version of the Java platform, Java 8, was released more than a year ago.There are very few good reasons to do this, because Java 8 has brought some important improvements to the language.

- There are many new features in Java 8.

1. Lambda expressions.
2. Stream API for working with Collections.
3. Asynchronous task chaining with CompletableFuture.
4. Java Date and Time API.

Java Notes - Chapter 2 - The Java Buzzwords

Authors of java have written an influential "White Paper" that explains their design goals and accomplishments.

+ Simple :

- Since syntax of java is simpler than syntax of C/C++, Java programming language is considered as simple.
- Consider the following points:
  * In Java, there is no need for header files.
  * It does not support pointer and pointer arithmetic.
  * It does not structure and union.
  * It does not support friend concept, operator overloading and virtual base class.
  * It does not support multiple [implementation] inheritance. no diamond problem
  * It supports new operator but not delete operator.
  * There is no concept of declaration and definition separately.

- Another aspect of being simple is being small. One of the goals of java is to enable the construction of software that can run stand-alone in small machines. The size of the basic interpreter and class support is about 40K; the basic standard libraries and thread support add another 175K.

+ Object Oriented:

- Since java supports all major and minor pillars of oops, it is considered as object oriented programming language.
- Java programming language was influenced from its previous successors programming language like C++. Java developers did not just took everything and implemented in Java but they analyzed the current challenges in the existing language and then included what is necessary.
- The object oriented model in Java is simple and easy to extend and also the primitive types such as integers, are retained for high-performance.

there are wrapper classes for every primitive type.

+ Robust :

- Following features of Java make it Robust.

(1) * Platform Independent: Java program are written once and executed on any platform this makes the job of developer easier to develop programs and not code machine dependent Coding.

(2) * Object Oriented Programming Language: helps to break the complex code into easy to understand objects and manage high complexity programs in distributed team environment.

Automatic.

(3) * Memory Management: In traditional programming language like C, C++ user has to manage memory by allocating and deallocating memory which leads to memory leaks in the program. In Java, memory management is taken care by the Java Virtual Machine and safe from memory crashes. All the allocation and clean of the memory is done automatically.

(4) * Exception Handling: In Java, developers are forced to handle the exception during the development time. All the possible exception are errored out during the compilation of the program. This way when the exception happens during runtime there is proper exception handling mechanism already coded in the program.

+ Architecture-Neutral : (only depends on the JVM.)

- The major challenge when Java was developing is to have programming language with which a program can be developed and executed anytime in future. With changing environments of hardware, processor, Operating system there was need to have program still adopt to this architecture changes
- Java code does not depend on the underlying architecture and only depends on it JVM thus accomplish the architecture neutral programming language.
- MS.NET is single platform for multiple technology and Java is single technology for multiple platforms.

+ Portable : (Runs on any platform that has JVM in it.)

- Java programs are not dependant on underlying hardware or operating system. The ability of java program to run on any platform makes java portable.
- Number of datatypes and their size is constant / same on all the platforms which makes java portable.
- Since java is portable java do not support sizeof operator.

Bytecode → JVM → Machine code

+ Secure :

- When Java programs are executed they donâ€™t instruct commands to the machine directly. Instead Java Virtual machine reads the program (ByteCode) and convert it into the machine instructions. This way any program tries to get illegal access to the system will not be allowed by the JVM. Allowing Java programs to be executed by the JVM makes Java program fully secured under the control of the JVM.

+ High Performance:

- Performance of java is slower than performance of C and C++.

- When java programs are executed, JVM does not interpret entire code into machine instructions. If JVM attempts to do this then there will huge performance impact for the high complexity programs. JVM was intelligently developed to interpret only the piece of the code that is required to execute and untouched the rest of the code. (Hotspot profiler.) In JIT compiler.

- Just In Time compiler (JIT) has taken place of interpreter which helps to increase performance of the application.

+ Multithreaded :

- If we want to utilize hardware resources(e.g CPU)efficiently then we should use threrad.
- Since java supports multithreading, it is easy to write program that responds to the user actions and helps developers to just implement the logic based on the user action instead to manage the complete multi-tasking solution.

+ Dynamic :

- Methods of the class are by default considered as a virtual.
- Java is designed to adapt to an evolving environment hence it is truly dynamic programming language.
- Java programs access various runtime libraries and information inside the compiled code (Bytecode). This dynamic feature allows to update the pieces of libraries without affecting the code using it.

+ Distributed :

- Due tot RMI(Remote Method Invocation), java is considered as a distributed.
- using RMI, program can invoke method of another program across a network and get the output.

[

+ Coding Conventions:

- Camel case naming convention: In this case of naming convention, first character of each word except first word should be capital.

e.g parseInt.

- Use "camel case" naming convention for following elements

* fields
* method
* method parameter
* local variable

- Pascal case naming convention : In this case of naming convention, first character of each word including first word should be capital.

e.g StringBuilder.

- Use "pascal case" naming convention for following elements.

* Type name[Interface/class/enum].
* File name.

- package name must be in lower case.
- Constant must be in upper case only.

+ Simple Java Program :

- Java is pure object oriented programming language so you can not write anything global. In C/C++ we write main function globly but in Java it must be member of class. For method names java follows camel case naming convention so name of entry point method is in lowercase.

- Consider the following code:

```
import java.lang.*;     //Optional to import
class Program <- class
{
        public static void main( String[] args ) <- Main Method.
        {
                System.out.println("Hello SunBeam");
        }
}
```

+ Characteristics of entry point method :

- main is considered as entry poit method in java.
- syntax : â€œpublic static void main(String[] args)â€• .
- JVM invokes main method.

- we can overload main method in java.
- per class we can write entry point method.

+ java.lang.System class :

```
package java.lang;
import java.io*;
public final class System extends Object
{
        //Fields
        public static final InputStream in;
        public static final PrintStream out;
        public static final PrintStream err;
        //Methods
        public static Console console();
        public static void exit(int status);
        public static void gc();
        public static void loadLibrary(String libname);
        ....
}//end of System class.
```

+ System.out.println :

- System : It is a final class declared in java.lang package and java.lang package is declared in rt.jar file.
- Out : It is an object of PrintStream class and. Out is declared as public static final field in System class.
- Println : It is non static overloaded method of PrintStream class.

+Java.io.PrintStream class :

```
package system.io;
public class PrintStream extends ....
{
        //Constructors
        public void flush();
        public void print(String s);
        //Other overloaded print methods
        public void println(String x);
```

*System - Final class (java.lang).
*Out - Object of PrintStream class.
*Println - Non static overloaded metho of PrintStream

```
Final class System{
  Public static final PrintStream Out;
}
```

```
    //Other overloaded println methods
    public PrintStream format( String format, Object... args );
    public void close();
}//end of class
```

+ Comments :

- If we want to maintain documentation of source code then we should use comments.
- Java supports three types comments
  - * //Single line comment.
  - * /* Multi line comments */
  - * /** Documentation comments */
- If we want to generate java code documentation using java source code then we should use
- javadoc tool. It comes with jdk. using javadoc we can generate documentation in HTML format.

+ Data Types :

- Data type describes three things:
  1. How much memory is required to store the data
  2. Which kind of data memory can hold
  3. which operation we can perform on the data.
- Java is strictly as well as strongly type checked language. In java, data types are classified as:
  1. primitive / value types — Also Statically typed Language.
  2. non primitive / reference types ← Wrapper Classes.

- Value Types :

Type	Size	Default value
boolean	undefined	false
byte	1 byte	0
char	2 bytes	'\u0000'
short	2 bytes	0
int	4 bytes	0
long	8 bytes	0L
float	4 bytes	0.0f
double	8 bytes	0.0d

- Wrapper Class: (java.lang.) → To make it reference type.

- Java has designed classes corresponding to every primiitive type. It is called as wrapper class.
- All wrapper classes are final classes i.e we can extend it.
- All wrapper classes are declared in java.lang package.
- Consider the hierarchy:

```
* java.lang.Object
        |- Boolean      < final >
        |- Character    < final >
        |- Number       < abstract >
                |- Byte   < final >
                |- Short      < final >
                |- Integer    < final >
                |- Long       < final >
                |- Float  < final >
                |- Double     < final >
```

- Widening: We can convert state of object of narrower type into wider type. it is called as widening

```
int num1 = 10;
double num2 = num1;        //widening (int to double)
```

- Narrowing : We can convert state of object of wider type into narrower type. It is called narrowing.

```
double num1 = 10.5;
int num2 = ( int ) num1;   //narrowing (double to int.)
```

- Reference Type:

- Only 4 types are reference types in java
- following are the reference types in java
  1. interface
  2. class
  3. enum
  4. array

+ Command Line Arguments:

```
public class Program
{
   public static void main(String[] args)
   {
        int num1 = Integer.parseInt( args[ 0 ] );
        float num2 = Float.parseFloat( args[ 1 ] );
        double num3 = Double.parseDouble( args[ 2 ] );
        String operator = args[ 3 ];
        switch( operator )
        {
        case "+":
          double result = num1 + num2 + num3;
          System.out.println("Result :          "+result);
        }
   }
}
//Input from termial : java Program 10 20.1f 30.2d +
```

+ Console Input Output :

- In java there is no standard way to take input from console.
- we can use any one of the following way to take input :

1. Console is a class declared in java.io package

```
Console console = System.console();
String text = console.readLine();
```

2. JOptionPane is a class declared in javax.swing package

```
String text = JOptionPane.showInputDialog("Enter text");
JOptionPane.showMessageDialog(null, text);
```

3. BufferedReader is class declared in java.io package

```
BufferedReader reader = null;
reader = new BufferedReader(new InputStreamReader(System.in));
String text = reader.readLine();
```

4. Scanner is a class declared in java.util package.

```
Scanner sc = new Scanner(System.in);
String text = sc.nextLine();
```

+ Boxing : int to Int, etc. (Primitive to Reference)

- It is the process of converting state of object of value type into reference type.
- Consider the example:

```
int number = 10;  ← value
String strNumber = String.valueOf( number );  ← Reference
```

- If boxing is done implictly then it is called autoboxing.
- Consider the example:

```
Object obj = 10;        //AutoBoxing
```

it is internally works like:

```
Object obj = new Integer( 10 );
```

+ UnBoxing : (Reference to premitive.)

- It is the process of converting state of object of reference type into value type.
- Considert the example: Int to int.

```
String strNumber = "125";  ← Reference
int number = Integer.parseInt( strNumber );  ← value
```

- If unboxing is done impliitly then it is called as autounboxing
- Considert the example:

```
Integer obj = new Integer(10);
int number = obj;       //AutoUnboxing
```

Java Notes - Chapter 4 - Package

+ Path :

- It is operating system platform's environment variable which is used to locate java tools.
- export PATH=/usr/bin/

+ Classpath :

- It is Java platforms environment variable which is used to locate .class file.
- export CLASSPATH=./bin

+ Package:

- If we want to group functionally equivalent or functionally related classes together then we should use package.
- Package can contain following elements:
  - * Sub package
  - * Interface
  - * Class
  - * Enum
  - * Error
  - * Exception
  - * Annotation
- **Advantages of package**
  - * To avoid name clashing/ collision/ ambiguity
  - * To group functionally related classes together.
- If we want add class in package then we should use package keyword.

e.g

```
package p1;
class Complex
{       }
```

- package statement must be the first statement in .java file.

```
import java.util.Scanner;
package p1;     //Error
class Complex
{       }
```

- we can not declare multiple packages in single .java file.

1 file → 1 package.

```
package p1;
package p2;     //Error
class Complex
{       }
```

- package name is physically mapped with folder.
- Default access modifier of the class is package(default).

```
package p1;
??? class Complex       // ??? --> package level private
{       }
```

- If we want to access any Type outside the package then we should declare that type public.
- class can have either package or public access modifier.

```
package p1;
public class Complex
{       }
```

- Name of public class and .java file must be same hence in single .java file we cannot write multiple public classes.
- .java file can contain multiple non-public classes but It can contain at the most only one public class.

1 public class.
Many private class.

- If we want to use packaged class outside the package then we should use either fully qualified class name or import statement.
- Consider the example:

```
class Program
{
	public static void main( String[] args )
	{
		p1.Complex c1 = new p1.Complex(); //Ok
	}
}
or

import p1.Complex;   //Look Here
class Program
{
	public static void main( String[] args )
	{
		Complex c1 = new Complex();		//Ok
	}
}
```

- We can access packaged class in unpackaged class.
- If we declare class without package then it is considered as a member of default package. We cannot import default package hence it is not possible to access unpackaged class in packaged class.

"Complex.java"

```
public class Complex
{
	//TODO
}
```

"Program.java"

```
package p2;
public class Program
{
	public static void main( String[] args )
	{
		Complex c1 = new Complex();		//Error
	}
}
```

- We can add multiple classes in single package. In this case we can access class directly within same package.

"Complex.java"

```
package p1;
public class Complex
{
	//TODO
```

```
}

"Program.java"
package p1;
public class Program
{
        public static void main( String[] args )
        {
                Complex c1 = new Complex();         //Ok
        }
}
```

- we can declare package inside package. It is called sub package.

```
package p1.p2;          //Here p2 is sub package
public class Complex
{
        //TODO
}
```

- If we want to access types outside package then we should use import statement and if we want to access static members of packaged class outside package directly then we should use static import.

```
import static java.lang.Math.PI;
import static java.lang.Math.pow;
import static java.lang.System.out;

public class Program
{
   public static void main(String[] args)
   {
        double radius = 10;
        double area = PI * pow( radius, 2);
        out.println("Area        :        "+area);
   }
}
```

- Most commonly used packages in java:
  * java.lang
  * java.util
  * java.io
  * java.net
  * java.rmi
  * java.lang.reflect
  * java.sql
  * java.rmi

- Since java.lang package is by default imported in every .java file, it is optional to import java.lang package.

+ Jar: -jar file. (group packages) ⟶ Reuse component.

- using jar tool we can group packages together. Jar tool creates jar file.
- If we want to create reusable component in java then we should take help of jar file.
- Steps to create jar file from eclipse:
  - * create java project and add classes in it.
  - * Right click on project--> Export-->Java-->Jar File--Next-->Brozse... -->.jar --> Finish.
- **Steps to add .jar file in runtime classpath or buildpath:**
  - * create java project.
  - * Right click on project-->Build Path-->Configure Build Path-->Java Build Path--> Libraries--> Add External Jars--> Select jar file and then click on ok.
  - * import package and use types.

Java Notes - Chapter 5 - OOPS - PART I

+ Class: (Template/Blue print for object.)

- It is a collection of fields and methods
  - * Field - data member is called field in java
  - * Method - member function is called method in java
- Class is reference type i.e instance of class get space on heap section. (object)
- We can achieve encapsulation using class.
- In java every class is by default extended from java.lang.Object class.

+Java.lang.Object:

- Object is non final concrete class declared in java.lang package.
- In java every class extends Object class directly or indirectly.
- Methods declared in Object class:

```
* protected Object clone() throws CloneNotSupportedException
* public boolean equals(Object obj)
* public int hashCode()
* public String toString()
* protected void finalize() throws Throwable
* public final Class<?> getClass()
* public final void wait()throws InterruptedException
* public final void wait(long timeout)throws InterruptedException
* public final void wait(long timeout,int nanos) throws InterruptedException
* public final void notify()
* public final void notifyAll()
```

+ Instance: of class

- Object is called instance in java.
- To create instance it is necessary to use new operator.
- Consider the following code snippet:

```
* Complex c1;       //reference
* new Complex();    //instance
* Complex c1 = new Complex();
  Complex c2 = c1;  //Shallow Copy of references.
```

+ Constructor: Special member function of a class.

- It is method of class which is used to initialize the object or instance.
- Java do not support default argument. Hence it is possible to call one constructor from another constructor. It is called constructor chaining.
- If we want to reuse implementation of existing constructor then we should use constructor chaining.
- using this statement we can achieve constructor chaining.
- this statement must be the first statement in constructor.

```
class Complex
{
   private int real;
   private int imag;
   public Complex()
   {
```

```
        this(0,0);      //ctor chaining
    }
    public Complex(int real, int imag)
    {
        this.real = real;
        this.imag = imag;
    }
}
```

+ Final variable:

- After storing value inside varible, if we dont want to modify value of that variable then we should declare such variable as final.
- final is keyword in java.
- we can declare reference as a final but we can not declare object as a final.

+ Static in java:

- In java we cannot declare local variable as a static.
- If we want to share value of field in all the instances of same class then we should declare field as static.
- Static field do not get space inside object hence we should access it using class name.
- To initialize the static field we should use static block. we can write multiple static blocks inside a class.
- JVM executes static block at the time of class loading.
- To access state of non-static member we should define non static method inside class and to access state of static member we should define static method inside a class.
- In non-static method we can access static as well as non-static members.
- Static method do not get this reference hence in static method we cannot access non-static member
- Using object/instance we can access non static members inside static method.

+ Singleton Class:

- a class from which we can create only one instance is called singleton class.
- Consider the following code.

```
class Singlton
{
  private Singleton()
  {   }
  private static Singleton singleton;
  public static Singleton getInstance()
  {
      if( singleton == null )
         singleton = new Singleton();
      return singleton;
  }
}
```

Java Notes - Chapter 6 - Array

+ Array:

- It is collection of same type of elements where each element get space continuously.
- It is collection of fixed elements i.e it cannot grow/shrink at runtime.
- Arrays are reference types in java i.e to create array instance we should use new operator.
- There are three types of array in java:
  - * Single dimensional
  - * Multi-dimensional
  - * Ragged Array

+ Single Dimensional Array:

```
- int arr[];                          //Array reference : Ok
- int[] arr;                          //Array reference : Recommended
- int[] arr = new int[ 3 ];           //Array instance
- int[] arr = new int[ ]{ 10, 20, 30 }; //Initialization
- int[] arr = { 10, 20, 30 };         //Initialization : valid
```

- Accessing elements using for loop

```
for( int index = 0; index < arr.length; ++ index )
    System.out.println( arr[ index ] );
```

- Accessing elements using for each loop

```
for( int element : arr )
    System.out.println( element );
```

+ Multi-Dimensional Array:

- Array of array which contains same number of elements is called Multi-dimensional Array.

```
- int arr[][];                        //Array reference : Ok
- int[] arr[];                        //Array reference
- int[][] arr;                        //Array reference : Recommended
- int[][] arr = new int[2][3];        //Array instance
- int[][] arr = new int[2][3]{ {10, 20, 30}, {40, 50, 60}};  //Initialization
- int[][] arr = { {10, 20, 30}, {40, 50, 60} };              //Initialization
```

- Accessing elements using for loop

```
for( int row = 0; row < arr.length; ++ row )
{
    for( int col = 0; col < arr[ row ].length; ++ col )
    {
        System.out.println(arr[row][col]);
    }
}
```

- Accessing elements using for each loop

```
for( int[] arrRef : arr )
{
    for( int element : arrRef )
    {
        System.out.println(element);
```

```
        }   }
```

+ Ragged Array:

- Array of array which contains diff. number of elements is called ragged Array.
- int arr[][]; //Array reference : Ok
- int[][] arr; //Array reference : Recommended
- int[][] arr = new int[3][]; //Array of references

```
arr[ 0 ] = new int[ 2 ];
arr[ 1 ] = new int[ 3 ];
arr[ 2 ] = new int[ 4 ];
```

- Accessing elements using for loop

```
for( int row = 0; row < arr.length; ++ row )
{
    for( int col = 0; col < arr[ row ].length; ++ col )
    {
        System.out.println(arr[row][col]);
    }
}
```

- Accessing elements using for each loop

```
for( int[] arrRef : arr )
{
    for( int element : arrRef )
    {
        System.out.println(element);
    }
}
```

+ Array of Value Types:

- If we create array of value type then it contains value. Default value is depends on default value of datatype.

e.g boolean[] arr = new boolean[3];

+ Array of Rerefences:

- If we create array of reference type then it contains reference. Default value is null.

e.g Complex[] arr = new Complex[3];

+ Array of Objects:

```
Complex[] arr = new Complex[ 3 ];  //Array of references
for( int index = 0; index < arr.length; ++ index )
```

```
arr[ index ] = new Complex();
```

+ In java, every element is pass to the function by value. If we want to pass element to the function by reference then we should use reference.

```
Public static void swap(int[] arr)
{
    int temp = arr[ 0 ];
    arr[ 0 ] = arr[ 1 ];
    arr[ 1 ] = temp;
}
public static void main(String[] args)
{
    int num1 = 10, num2 = 20;
    int[] arr = new int[ ]{ num1, num2 };
    Program.swap(arr);
    num1 = arr[ 0 ]; num2 = arr[ 1 ];
    System.out.println(num1+"    "+num2);
}
```

Java Notes - Chapter 7 - OOPS - Part II

+ Object Oriented Programming:

- It's a method of programming in which real world problems are solved using classes and objects.
- Java is object oriented programming language.
- A language which supports all major and minor pillars of OOP is called pure object orie programming language. 4 3

+ Major Pillars / Parts / Elements of OOPS:

1. - Abstraction
   * It is the process of getting essential things from object.
   * Abstraction may changes from object to object.
   * Using abstraction we can achieve simplicity.
2. **- Encapsulation**
   * Binding of data and code together is called encapsulation / Process of implementi abstraction is called encapsulation.
   * Using class we can achieve encapsulation.
3. **- Modularity-**
   * Process of dividing complex system into small modules is called modularity.
   * using package and jar file we can achieve modularity.
   * Using modularity we can minimize module dependancy.
4. **- Hierarchy**
   * Level / Order / ranking of abstraction is called hierarchy.
   * Main job of hierarchy is to achieve modularity.
   * 4 Types of heirarchy:
     # has-a / part-of --> Composition Employee has an address
     # is-a / kind-of --> Inheritance Employee is person.
     # use-a --> Dependancy
     # creates-a --> Instantiation

+ Minor Pillars / Parts / Elements of OOPS:

1. - Typing / Polymorphism
   * Ability of an object to take multiple forms is called polymorphism.
   * Using polymorphism we can reduce maintenance of the system.
   * We can achieve it using
     # Method Overloading ← Same name same class
     # Method overriding ← Same name diff. class
2. **- Concurrency**
   * Process of executing multiple task simultaneously is called concurrency.
   * We can achieve it using threading. ← In Java
3. **- Persistence**
   * Process of maintaing state of the object in file or database is called persistence.

+ Composition:

- Composition is also called as containment.
- If has-a relationship exist between two types then we shoud use composition.
  e.g Car has-a engine, person has address ← object declared outside.
- If object/component is part of another object then it is called as composition.
- In java composition do not implies physical containment. Composition is achieved using reference variable only.
- In other words object outside the object is called composition.
- Consider the example:

```
class Person
{
        private String name = new String();
        private Date birthDate = new Date();
        private Address currAddress = new Address();

        //TODO
}
```

+ Inheritance:

Parent/Base/Super
↓
Child/derived/sub.

- Inheritance is also called as generalization.
- If is-a relationship exist between two types then we should use inheritance.
- Without modifing existing class if we want to extend meaning of the class then we should use inheritance.
- In java, parent class is called super class and child class is called sub class.
- to create sub class we should use "extends" keyword.
- consider following example:

```
class Shape
{       }
class Circle extends Shape
{       }
```

- Sub class can extend at the most only one super class. i.e. multiple implementation inheritance is not allowed: ← In java, C++ supports it.

```
class A{        }
class B{        }
class C extends A, B   //Not Allowed in java
{       }
```

(Java supports only public mode of inheritance.)

	In same package			In diff package	
	Same class	Sub class	Non sub class	Sub class	Non Sub class
private	A	NA	NA	NA	NA
package	A	A	A	NA	NA
protected	A	A	A	A	NA
public	A	A	A	A	A

- If we create instance of sub class then first super class constructor gets called and then sub class constructor gets called.
- In Java class do not contain destructor. ← Managed by Garbage Collecter.
- using super statement, we can call any super class constructor from sub class constructor.
- If we want to access members of super class in method of sub class then we should use super keyword.

super();

+ Upcasting and Downcasting:

- we can convert reference of sub class into reference of super class. It is called upcasting.

```
e.g.  Employee emp = new Emloyee("Sandeep",33,45000);
      Person p = emp;        //Upcsting.
```

- We can convert reference of super class into reference of sub class. It is called downcasting.

```
e.g   Person p =  new Emloyee("Sandeep",33,45000);
      Employee emp = ( Employee ) p;      //Downcasting
```

- If downcasting fails then JVM throws ClassCastException.

- In java all the methods are by default virtual hence using super class reference variable we can call method of sub class. It is called dynamic method dispatch.

+ Rules of method overriding:

1. Signature of super class and sub class method should be same.
2. Access modifier of super class and sub class method should be same or it should be wider than super class method.

3. Return type of super class and sub class method should be same or it should be sub type of return type specified in super class method.
4. Method name, number of parameter and type of parameter pass to the method must be same.
5. Checked exception list specified in sub class method should be same ot it should be subset of exception list specified in super class method.

+ **Difference between == and equals method:**

- if we want to compare state of object of value type then we should use == operator.

```
int num1 = 10;
int num2 = 10;
if( num1 == num2 )
   System.out.println("Equal");
else
   System.out.println("Not Equal");
//Output : Equal
```

- if we want to compare state of variable of reference type then also we should use == operator.

```
Complex c1 = new Complex(10,20);
Complex c2 = new Complex(10,20);
if( c1 == c2 )
   System.out.println("Equal");
else
   System.out.println("Not Equal");
//Output : Not Equal
```

- if we want to compare state of object of reference type then we should use equals method.

```
Complex c1 = new Complex(10,20);
Complex c2 = new Complex(10,20);
if( c1.equals( c2 ) )
   System.out.println("Equal");
else
   System.out.println("Not Equal");
//Output : Equal
```

* if class do not contain equals method then super class equals method gets called. And if any super class do not contain equals method then Object class equals method gets called. Object.equals always compares object reference.

* To compare state of the object we should override equals method in class.
* For example:

```
class Complex
{
   private int real;
   private int imag;
   @Override
   public boolean equals(Object obj)
   {
        if( obj != null )
        {
           Complex other = (Complex)obj;
           if( this.real == other.real && this.imag == other.imag )
                return true;
        }
        return false;
   }
}
```

+ Final method:

- If implementation of super class method is logically complete then we should declare super class method as final.
- We can not override final method in sub class. But final method inherit into sub class.
- Some of the final methods are:
  * public final Class<?> getClass()
  * public final void wait() throws InterruptedException
  * public final void notify()
  * public final void notifyAll()

+ Final class:

- If implementation of all the methods is logically complete then instead of declaring method as final we should declare class as final.
- we can not extend final class i.e we can create sub class of final class.
- Some of the final classes are:
  * All Wrapper classes.
  * java.lang.System
  * java.lang.String
  * java.lang.StringBuffer
  * java.lang.StringBuilder
  * java.lang.Math

+ Abstract method: Even if 1 method is Ab. → declare class Abstract

- If implementation of super class method is logically 100% incomplete then we should declare super class method as abstract. declare abstract
- Abstract method do not contain body.
- If method is abstract then it is compulsory to declare class as abstract.
- if super class contains abstract method then sub class should override that method in sub class or sub class should be declared as abstract.
- we cannot override private, final and static method in sub class hence such keywords cannot be used with abstract method. not used.
- Some of the abstract methods declared in Number class are:

```
* public abstract int intValue()
* public abstract float floatValue()
* public abstract double doubleValue()
* public abstract long longValue()
```

+ Abstract class:

- If implementation of any class is logically incomplete then we should declare class as abstract. - If class contains abstract method then we should declare class as abstract.
- Without declaring method as abstract we can declare class as abstract.
- We cannot create instance of abstract class but we can create reference of abstract class.
- Abstract class can contain constructor.
- Some of the abstract classes are:
  * java.lang.Enum
  * java.lang.Number

+ Fragile base class problem:

- If we make changes in super class method then it is necessary to recompile all the sub classes is called fragile base class problem.
- To avoid fragile base class problem we should declare super type as a interface.

+ Interface: (Specification for the subclasses.)

- Set of rules is called standard and standard is also called as specification.
- If we want to define specification for the sub classes then we should use interface.
- There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, interfaces are such contracts.
- Interface is keyword in java.

Java Notes - Chapter 7 - OOPS - Part II

- Interface is reference type.
- Interface an contain only constants, method signatures, default methods, static methods, and nested types.
- Interfaces cannot be instantiated , they can only be implemented by classes or extended by other interfaces.
- Observe the following statements:

```
Interface : I1, I2, I3
Class     : C1, C2, C3

* I2 implements I1              //Incorrect
* I2 extends I1                 //correct : Interface inheritance
* I3 extends I1, I2             //correct : Multiple interface inheritance
* C2 implements C1              //Incorrect
* C2 extends C1                 //correct : Implementation Inheritance
* C3 extends C1,C2              //Incorrect : Multiple Implementation Inheritance
* I1 extends C1                 //Incorrect
* I1 implements C1              //Incorrect
* c1 implements I1              //correct : Interface implementation inheritance
* c1 implements I1,I2 //correct : Multiple Interface implementation inheritance
* c2 implements I1,I2 extends C1     //Incorrect
* c2 extends C1 implements I1,I2     //correct
```

- If we want to implement interface then we should use implements keyword.

```
interface A
{
   void f1();
}
class B implements A
{
   @Override
   public void f1()
   {    }
}
```

- If class implement multiple interfaces which contain methods with same signature then sub class can override it only once.

```
interface A
{
   void f1();
}
interface B
```

```
{
   void f1();
}
class C implements A, B
{
   @Override
   public void f1()
   {    }
}
```

- If class implement multiple interfaces which contain methods with same name and different return type then sub class can not override it.

```
interface A
{
   int f1();
}
interface B
{
   double f1();
}
class C implements A, B
{
   //Error: Can not override method in sub class
}
```

+ Default Methods:

- Designing interfaces have always been a tough job because if we want to add additional methods in the interfaces, it will require change in all the implementing classes.

As interface grows old, the number of classes implementing it might grow to an extent that it is not possible to extend interfaces.

That is why when designing an application, most of the frameworks provide a base implementation class and then we extend it and override methods that are applicable for our application.

- To overcome such problem, in java 8 , we can add default methods in interface.
- For creating a default method in java interface, we need to use " default " keyword with the method signature.

```
interface A
{
   void f1(String str);
   default void log(String str)
   {
```

```
        System.out.println("A log :"+str);
    }
}
```

- Notice that log(String str) is the default method in the Interface A. Now when a class will implem interface A, it is not mandatory to provide implementation for default methods of interface. This feature w help us in extending interfaces with additional methods, all we need is to provide a default implementatio

- We know that Java doesn't allow us to extend multiple classes because it will result in the "Diamond Problem" where compiler cannot decide which superclass method to use. With the default methods, the diamond problem would arise for interfaces too.

- Some key points about default method:

* Java interface default methods will help us in extending interfaces without having the fear of breaking implementation classes.

* Java interface default methods has bridge down the differences between interfaces and abstract classes.

* Java 8 interface default methods will help us in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.

* Java interface default methods will help us in removing base implementation classes, we can provide default implementation and the implementation classes can chose which one to override.

* One of the major reason for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions.

* A default method cannot override a method from java.lang.Object.

* Java interface default methods are also referred to as Defender Methods or Virtual extension methods.

+ Interface Static Method:

- Java interface static method is similar to default method except that we canâ€™t override them in the implementation classes.

- Important points about java interface static method:

* Java interface static method is part of interface, we can't use it for implementation class objects.

* Java interface static methods are good for providing utility methods, for example null check, collection sorting etc.

* Java interface static method helps us in providing security by not allowing implementation classes to override them.

* We can't define interface static method for Object class methods.

* We can use java interface static methods to remove utility classes such as Collections and move all of it's static methods to the corresponding interface, that would be easy to find and use.

+ Functional Interface:

- An interface with exactly one abstract method is known as Functional Interface.
- A new annotation @FunctionalInterface has been introduced to mark an interface as Functional Interface.
- @FunctionalInterface annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces. It's optional but good practice to use it.
- Functional interfaces enables us to use lambda expressions to instantiate them.
- java.util.function package contains functional interface that are targeted to lambda expression.

Java Notes - Chapter 8 - Exception Handling

+ Exception Handling:

- An exception is an event, which occurs during the execution of a program that disrupts the norm flow of the program's instructions.
- To manage OS resources carefully, we should handle exception.
- To handle exception we should use following keywords:
  * try - to inspect the exception for risky code.
  * throw - to generate the exception
  * catch - to handle the exception
  * throws - to redirect the exception
  * finally - to release the local resources.

Superclass

- java.lang.Throwable is super class of Error and exception in java.
- Exceptional conditions that are internal to the application are exceptions.
- Exceptional conditions that are external to the application, and that the application usually can recover from are errors.

+ Throwable class:

- Constructors:

* public Throwable()
* public Throwable(String message)
* public Throwable(Throwable cause)
* public Throwable(String message,Throwable cause)

- Methods:

* public final void addSuppressed(Throwable exception)
* public StackTraceElement[] getStackTrace()
* public Throwable getCause()
* public String getMessage()
* public void printStackTrace()

+ Types of Exception:

- Checked and UnChecked are the types of exception. These are the types of exception designed for compiler.

- Checked Exception

* java.lang.Exception and all its sub classes except RuntimeException class are classified as checked exception classes.
* It is compulsory to handle checked exception otherwise compiler generates error. ↳using throws.
* e.g InterruptedException, ClassNotFoundException etc.

- UnChecked Exception

* java.lang.RuntimeException and all its sub classes are classified as ~~checked~~ unchecked exception classes.
* It is optional to handle unchecked exception.

↳using try-catch.

* e.g NullPointerException, ArithmeticException, ClassCastException etc.

+ StackTrace : History of pgm execution until the exception.

- A stack trace provides information on the execution history of the current thread and lists the names of the classes and methods that were called at the point when the exception occurred.
- A stack trace is a useful debugging tool that you'll normally take advantage of when an exception has been thrown.

+ Syntax:

```
try
{
   BufferedReader reader = null;
   reader = new BufferedReader(new InputStreamReader(System.in));
   System.out.print("Enter num1      :       ");
   int num1 = Integer.parseInt(reader.readLine());
   System.out.print("Enter num2      :       ");
   int num2 = Integer.parseInt(reader.readLine());
   int result = num1 / num2;
   System.out.println("Result :        "+result);
   reader.close();
}
catch (NumberFormatException e)
{
   e.printStackTrace();
}
catch (ArithmeticException e)
{
   e.printStackTrace();
}
catch (IOException e)
{
   e.printStackTrace();
}
```

- It is possible to handle all the exceptions in single catch block.

```
try
{
   BufferedReader reader = null;
   reader = new BufferedReader(new InputStreamReader(System.in));
```

```
        System.out.print("Enter num1      :       ");
        int num1 = Integer.parseInt(reader.readLine());
        System.out.print("Enter num2      :       ");
        int num2 = Integer.parseInt(reader.readLine());
        int result = num1 / num2;
        System.out.println("Result :        "+result);
        reader.close();
    }
    catch (NumberFormatException | ArithmeticException | IOException e)
    {
        e.printStackTrace();
    }
```

- If you want to release local resources then you should define finally block.

```
Scanner sc = null;
try
{
    sc = new Scanner(System.in);
    System.out.print("Enter num1      :       ");
    int num1 = sc.nextInt();
    System.out.print("Enter num2      :       ");
    int num2 = sc.nextInt();
    int result = num1 / num2;
    System.out.println("Result :        "+result);
}
catch ( ArithmeticException | InputMismatchException e)
{
    e.printStackTrace();
}
finally
{
    sc.close(); ← Releases the sc object of Scanner.
}
```

- **try-with-resources**

  * A resource is an object that must be closed after the program is finished with it.
  * Instance of class which implements either AutoCloseable or Closeable interface is called resource.

* If we use resource with try then there is no need to close the resource in finally. When block of code gets executed normally or by throwing exception then close method of resource gets called automatically.

* consider the following code:

```
try( Scanner sc = new Scanner(System.in); )
{
   System.out.print("Enter num1      :        ");
   int num1 = sc.nextInt();
   System.out.print("Enter num2      :        ");
   int num2 = sc.nextInt();
   int result = num1 / num2;
   System.out.println("Result :        "+result);
}
catch ( ArithmeticException | InputMismatchException e)
{
   e.printStackTrace();
}
```

- A try-with-resources statement can have catch and finally blocks just like an ordinary try statement.
- Some exceptions are designed to handle. If we do not handle it then compiler generates error.

Consider the following code:

```
public static void print()
{
      for( int i = 1; i < 10; ++ i )
      {
         System.out.println(i);
         Thread.sleep(300);  //Compiler error
      }
}
```

- sleep method throws InterruptedException, which is checked exception. We must handle it.

```
public static void print()
{
      try
      {
         for( int i = 1; i < 10; ++ i )
         {
            System.out.println(i);
            Thread.sleep(300);        //Now Ok
         }
      }
```

```
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
```

- If we want to redirect exception to caller of the method then we can use throws clause.

```
public static void print() throws InterruptedException
{
    for( int i = 1; i < 10; ++ i )
    {
        System.out.println(i);
        Thread.sleep(300);
    }
}
```

- throws clause can be used to redirect any checked as well as unchecked exception.

+ Generic Catch:

- Exception class can keep reference of any checked as well as unchecked exception hence it is used to write generic catch.
- Remember, compiler do not allows us to handle super type exception first. So generic catch must appear at last. Use (Exception e) in last catch.

+ Summary:

- using try, catch, throw, throws and finally we can handle exception in java.
- try block may have multiple catch block but single try block must have at lest one catch/ finally or resource.
- we can handle multiple exceptions in single catch block also.
- generic catch must appear at last.
- Handling checked exceptions is compulsory. If we do not handle it then compler gives err.
- To handle checked exception we should use either try-catch block or we should redirect exception to caller method using throws clause.
- To release local resources we should use finally block.
- JVM always execute finally block. If we write System.exit(0) in try and catch block then JVM do execute finally block.

+ Chained Exceptions:

- Generally exceptions are handled by throwing new exception. This process of handling exception is called exception chaining.
- Following Methods and constructors in Throwable that support chained exceptions:

* Throwable(Throwable)
* Throwable(String, Throwable)
* Throwable initCause(Throwable)
* Throwable getCause()

- consider the following example:

```
try
{
        //TODO
}
catch (SQLException e)
{
    throw new SerevletException("Other SQLException", e); // Exception Chaining
}
```

Exception as a response.

+ Consider the following code snippets:

1.

```
int[] arr = new int[ ]{ 10, 20, 30 };
int element = arr[ 3 ]; //ArrayIndexOutOfBoundsException
```

2.

```
String str = "Sandeep";
char ch = str.charAt(7); //StringIndexOutOfBoundsException
```

3.

```
ArrayList<Integer> list = new ArrayList<>();
list.add(10);
list.add(20);
list.add(30);
int element = list.get(3); //IndexOutOfBoundsException
```

4.

```
Object obj = new Integer(10);
//downcasting
String str = ( String )obj; // ClassCastException
```

5.

```
String str = "A123";
int number = Integer.parseInt(str); // NumberFormatException
```

① Type Checking ② Bounded-Type Para. ③ Wild card.

Java Notes - Chapter 9 - Generics

+ Generics: (Template in C++)

- Like object, we can pass data type as a argument to the function. Hence parameterized type is generics. In C++, it is called as template.

- First let us discuss generic class using java.lang.Object class. Consider the following code.

```
class Box
{
	private Object object;
	public Object getObject()
	{
		return object;
	}
	public void setObject(Object object)
	{
		this.object = object;
	}
}
```

1.

```
Box box = new Box();
int num1 = 10;
box.setObject(num1);//box.setObject(new Integer(num1));
int num2 = (int)box.getObject();	//Ok
```

2.

```
Box box = new Box();
Date dt1  = new Date();
box.setObject(dt1);				//UpCasting
Date dt2 = (Date)box.getObject();	//DownCasting
```

3.

```
Box box = new Box();
Date dt1  = new Date();
box.setObject(dt1);				//UpCasting
String str = (String)box.getObject();	//DownCasting : ClassCastException
```

+ Why Generics?

1. Generics gives us stronger type checking at compile time.
2. It completely eliminates explicit type casting.
3. Using generics we can write generic algorithm.

Let's us consider following code.

```
class Box< T >          //T is Type parameter
{
        private T object;
        public T getObject()
        {
                return object;
        }
        public void setObject(T object)
        {
                this.object = object;
        }
}
```

1. * Type-Check at runtime.

```
Box<int> box = new Box<int>();      //Error
```

In above code, int is type argument. Type argument used in generics must be reference type.

2.

```
Box<Integer> box = new Box<Integer>();    //OK
```

If we want to store primitive values in generic collection then we should use wrapper class.

3.

```
Box<Date> box = new Box<Date>();
box.setObject(new Date());
String str = (String)box.getObject();  //Compiler error : src and dest must be Date  type.
```

4.

```
Box<Date> box = new Box<Date>();
box.setObject(new Date());
Date date = box.getObject();  //OK
```

+ Most Commonly used type parameters in java:

T - Type
N - Number
E - Element
K - Key

V - Value
A, U - Second Type Parameters.

We can specify multiple type parameters too.

```
interface Map<K,V>
{
	void put( K key, V value );
	K getKey();
	V getValue();
}
class HashTable<K,V> implements Map<K,V>
{
	private K key;
	private V value;
	public void put( K key, V value )
	{
		//TODO : Assignment for programmer
	}
	@Override
	public K getKey()
	{
		return this.key;
	}
	@Override
	public V getValue()
	{
		return this.value;
	}
}
public class Program
{
	public static void main(String[] args)
	{
		Map<Integer,String> map = new HashTable<>();
		map.put(1, "Sandeep");
		map.put(2, "Prathamesh");
		map.put(3, "Soham");
	}
}
```

+ Bounded Type Parameter :

- Sometimes we need to put restrictions on a type that can we used as type argument. Using bounded type parameter we can put restriction on type argument.

```
class Box<N extends Number>      //N extends Number => is bounded type parameter.
{
        private N object;
        public N getObject()
        {
                return object;
        }
        public void setObject(N object)
        {
                this.object = object;
        }
}
```

- as shown in above code, box object can store only numbers.

1.
```
Box<Integer> box = new Box<>();          //Allowed
```
2.
```
Box<String> box = new Box<>();           //Not allowed
```

+ Wild Card:

- In generics ? is called wild card which indicates unknown type.
- There are 3 types of wild card in generics.
  1. UnBounded Wild Card.
  2. Upper Bounded Wild Card.
  3. Lower Bounded Card.

+ UnBounded Wild Card :

```
public static void printList( List<?> list )
{
        for (Object object : list)
        {
                System.out.println(object);
        }
```

```
}

List<Integer> list1 = new ArrayList<>();
//TODO: Add element in list1
Program.printList(list1);

List<String> list2 = new Vector<>();
//TODO: Add element in list2
Program.printList(list2);

List<Date> list3 = new LinkedList<>();
//TODO: Add element in list3
Program.printList(list3);
```

- As shown in above code, list can store reference of any List<> collection which contains any type of element.

+ Upper Bounded Wild Card:

```
public static void printList( List<? extends Number > list )
{
        for (Number number : list)
        {
                System.out.println(number);
        }
}
List<Integer> list1 = new ArrayList<>();
//TODO: Add element in list1
Program.printList(list1);

List<Double> list2 = new Vector<>();
//TODO: Add element in list2
Program.printList(list2);

List<String> list3 = new ArrayList<>();
//TODO: Add element in list3
Program.printList(list3);               //Not Allowed
```

- As shown in above code, list can store reference of any List<> collection which contains only number.

+ Lower Bounded Wild Card:

```
public static void printList( List<? super Integer > list )
{
	for (Object object : list)
	{
		System.out.println(object);
	}
}
```

```
List<Integer> list1 = new ArrayList<>();
//TODO: Add element in list1
Program.printList(list1);

List<Double> list2 = new Vector<>();
//TODO: Add element in list2
Program.printList(list2);		//Not Allowed

List<String> list3 = new ArrayList<>();
//TODO: Add element in list3
Program.printList(list3);		//Not Allowed
```

- As shown in above code, list can store reference of any List<> collection which contains elements of integer or its super type.

```
public static void printList( List<Number > list )
{
	//TODO
}
public static void main(String[] args)
{
	List<Integer> list1 = new ArrayList<Integer>();
	Program.printList(list1);	//Error
}
```

- In case of type argument, inheritance is not allowed. As shown in above code, type argument used in List is Number and type argument used in ArrayList is integer. It must be same.

+ Marker Interface:

- An interface which do not contain any member is called marker interface.
- following are the marker interface:
  * java.lang.Cloneable
  * java.util.EventListener
  * java.util.RandomAccess
  * java.io.Serializable
  * java.rmi.Remote

+ Commonly used interfaces in java:

- java.lang.Cloneable
- java.lang.AutoCloseable
- java.lang.Comparable<T>
- java.util.Comparator<T>
- java.lang.Iterable<T>
- java.util.Iterator<E>
- java.lang.Runnable

+ Cloneable:

- It is interface declared in java.lang package.
- It is marker interface.
- consider the following code:

```
Complex c1 = new Complex();
Complex c2 = c1;       //Shallow Copy of references
```

- If we want to create new object from existing object then we should override clone() method.

```
@Override
public Complex clone() throws CloneNotSupportedException
{
    Complex other = ( Complex )super.clone();
    return other;
}
```

shallow copy.

- If we want to create shallow copy of current object then we should use "super.clone()".
- Without implementing Cloneable interface, if we try to use clone method then it throws CloneNotSupportedException.

+ Comparable:

- It is interface declared in java.lang package.
- "int compareTo(T o)" is method declared in Comparable interface.

- If we want to sort array of object of same type then we should use Comparable interface.
- Consider the following example:

```
class Employee implements Comparable<Employee>
{
   @Override
   public int compareTo(Employee other)
   {
        if( this.empid < other.empid )
           return -1;
        else if( this.empid > other.empid )
           return 1;
        else
           return 0;
   }
}
public class Program
{
   public static Employee[] getEmployees()
   {    }
   public static void main(String[] args)
   {
        Employee[] arr = Program.getEmployees();
        Arrays.sort(arr);
   }
}
```

+ Comparator:

- It is interface declared in java.util package.
- It is functional interface.
- "int compare(T o1,T o2)" is method of Comparator interface.
- If we want to sort array of objects of different types then we should use Comparator interface.
- Consider following example:

```
abstract class Person{ }
class Student extends Person{        }
class Employee extends Person{       }
class SortById implements Comparator<Person>
{
```

```
    @Override
    public int compare(Person p1, Person p2)
    {
        //TODO
        return 0;
    }
}
public class Program
{
    public static Person[] getPersons()
    {   }
    public static void main(String[] args)
    {
        Person[] persons = Program.getPersons();
        Comparator<Person> comparator = new SortById();
        Arrays.sort(persons,comparator);
    }
}
```

+ Iterator: For each ( )

- It is smart pointer which is used to traverse the collection.
- In java for each loop is called iterator.
- We can traverse elements of array or object which implements java.lang.Iterable interface.

Consider the folloiwing code

```
class Node
{       }
class LinkedList implements Iterable<Integer>
{
    private Node head = null;
    //TODO
    @Override
    public Iterator<Integer> iterator()
    {
        Iterator<Integer> itr = new LinkedListIterator( this.head );
        return itr;
    }
}
```

```
class LinkedListIterator implements Iterator<Integer>
{
   Node trav;
   LinkedListIterator( Node trav )
   {
        this.trav = trav;
   }
   @Override
   public boolean hasNext()
   {
     return this.trav != null;
   }
   @Override
   public Integer next()
   {
     int data = trav.data;
     trav = trav.next;
     return data;
   }
}
```

Java Notes - Chapter 11 - Nested Class

+ Nested Class:

- In Java we can write class inside scope of another class, it is called as nested class.
- Outer class can be declared as package level private or public only but nested class can be dec as private, package level private, protected or public.
- If we want to design class for implementation of another class then it should be nested.
- Why Use Nested Classes?

* It is a way of logically grouping classes that are only used in one place:

If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together.

* It increases encapsulation:

Consider two top-level classes, A and B, where B needs access to members of A that wou otherwise be declared private. By hiding class B within class A, A's members can be decl private and B can access them. In addition, B itself can be hidden from the outside world

* It can lead to more readable and maintainable code:

Nesting small classes within top-level classes places the code closer to where it is used.

- Types of Nested Classes:

* Inner class:

If we declare nested class as a non-static then it is called as inner class.

```
#   class A                          //A.class
    {
        class B //Inner class       //A$B.class
        {
            //TODO
        }
    }
# A.B b = new A().new B();   //Instantiation
```

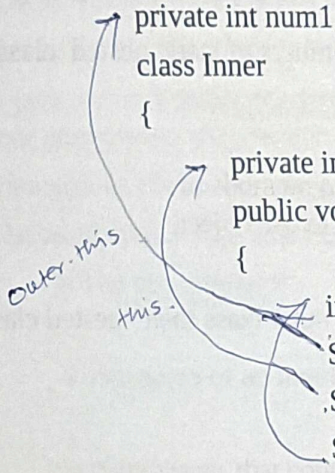
As with instance methods and variables, an inner class is associated with an instance of its encl class and has direct access to that object's methods and fields. Also, because an inner class is ass with an instance, it cannot define any static members itself.

For the sake of simplicity consider non static nested class as non-static method of the class.

using object (of inner class) we can access members of inner class inside method of outer class.

Members of Outer class are directly accessible in methods of inner class.

```
# Shadowing:
class Outer
{
    private int num1 = 10;
    class Inner
    {
        private int num1 = 20;
        public void print( )
        {
            int num1 = 30;
            System.out.println(Outer.this.num1); //10
            System.out.println(this.num1);       //20
            System.out.println(num1);            //30
        }
    }
}
```



If implementation of nested class is dependent on outer class then nested class should be declared as non static(Inner class).

There are two special kinds of inner classes: local classes and anonymous classes.

*** Static Nested class:**

```
# if we declare nested class as static then it is called as static nested  class.
#       class A                                          //A.class
        {
                static class B   //static nested class     //A$B.class
                {
                        //TODO
                }
        }
#       A.B b = new A.B();    //Instantiation
```

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested

class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.

for the sake of simplicity, consider static nested class as static method of the class.

Using object (of static nested class) we can access members of static nested class inside method outer class.

Static members of Outer class are directly accessible in methods of static nested class. To access non static members it is necessary to use object.

If implementation of nested class do not dependent on outer class then nested class should be declared as static.

- Example of nested class.

```
class LinkedList
{
	private static class Node
	{
		//TODO
	}
	private Node head;
	public class LinkedListIterator
	{
		Node node = head;
		//TODO
	}
}
```

+ Local Class:

- In java, we can write class inside block/method. It is called as local class.
- Local class can be categorized as:
  * Method local inner class
  * Method local anonymous inner class

- Method local inner class:

* Local class cannot be declared as static hence it is also called as method local inner class.

* Instance of local class are accessible within method only.

- Method Local Anonymous Inner Class:

* We can define class without name, it is called anonymous class.

* In java, we can create anonymous class within block/method only so it cannot be static. Hence anonymous class is also called as method local inner class.

* Anonymous classes enable you to make your code more concise.

* The anonymous class expression consists of the following:

The new operator

The name of an interface to implement or a class to extend.

Parentheses that contain the arguments to a constructor, just like a normal class instance creation expression.

Note: When you implement an interface, there is no constructor, so you use an empty pair of parentheses.

A body, which is a class declaration body.

* Example 1:

```
Runnable r = new Runnable()
{
	@Override
	public void run()
	{
		System.out.println("Inside run");
	}
};
r.run();
```

* Example 2:

```
abstract class Shape
{
	public abstract void calculateArea();
}
public class Program
```

```
{
        public static void main(String[] args)
        {
                Shape sh = new Shape()
                {
                        @Override
                        public void calculateArea()
                        {
                                System.out.println("calculateArea()");
                        }
                };
                sh.calculateArea();
        }
}
```

* Example 3:

```
Object obj = new Object()
{
        @Override
        public String toString()
        {
                return "string";
        }
};
System.out.println(obj.toString());
```

Java Notes - Chapter 12 - Collection Framework

An object which contains more than one element is called collection. In java data structure classes are called collection classes. Collection classes are also called as container classes. Library of reusable classes which are used to develop the application is called framework. Library of reusable collection classes which are used to develop java application is called java's collection framework.

Java is abstraction technology so, developer should not worry about implementation of collection rather he/she should worry about its use.

To use java collection framework, we should import java.util package. We are going to use following interfaces in collection framework.

java.lang.Iterable
java.util.Iterator
java.lang.Comparable
java.util.Comparator
java.util.Enumeration
java.util.Collection
java.util.List
java.util.ListIterator
java.util.Set
java.util.SortedSet
java.util.NavigableSet
java.util.Queue
java.util.Deque
java.util.Map
java.util.Map.Entry
java.util.SortedMap
java.util.NavigableMap

+ **Collection:**

- It is root interface in java collection framework.
- The JDK does not provide any direct implementations of this interface.
- This interface is typically used to pass collections around and manipulate them where maximum generality is desired.
- **Methods Of Collection Interface**
  * boolean add(E e)
  * boolean addAll(Collection<? extends E> c)
  * boolean contains(Object o)
  * boolean containsAll(Collection<?> c)

* boolean remove(Object o)
* boolean removeAll(Collection<?> c)
* boolean retainAll(Collection<?> c)
* void clear()
* boolean isEmpty()
* int size()
* Object[] toArray()
* <T> T[] toArray(T[] a)

* default boolean removeIf(Predicate<? super E> filter)
* default Stream<E> stream()
* default Stream<E> parallelStream()
* default Spliterator<E> spliterator()

+ List:

- It is sub interface of Collection interface.
- ArrayList, Vector, Stack, LinkedList etc implements List interface. These are reffered as List collection.
- List collections store data in ordered/sequential fashion [unsorted].
- From List collection, user can access element using integer index.
- In list collection we can insert duplicate elements.
- List collections allows us to insert multiple null elements.
- We can travserse elements of list collection using Iterator and ListIterator.
- If we want to store elements of reference type in List collection then reference type shoul contain equals method.
- Due to inheritance all above methods are implicitly methods of List. Following are List specific methods.

* void add(int index,E element)
* boolean addAll(int index, Collection<? extends E> c)
* E get(int index)
* E set(int index, E element)
* int indexOf(Object o)
* int lastIndexOf(Object o)
* ListIterator<E> listIterator()
* ListIterator<E> listIterator(int index)
* boolean remove(Object o)
* List<E> subList(int fromIndex, int toIndex)

- * default void sort(Comparator<? super E> c)
  * default void replaceAll(UnaryOperator<E> operator)

Following are the synchronized collection classes in java:

1. Vector -
2. Stack -
3. HashTable -

+ ArrayList: unsynchronized.

- It is dynamically growable/shrinkable array.
- It is unsynchronized collection.
- using Collections.synchronizedList() method, we can make it synchronized.
- It stores data in ordered/sequential fashion [unsorted].
- From ArrayList, user can access element using integer index.
- In ArrayList we can insert duplicate elements.
- It allows us to insert multiple null elements.
- We can travserse elements of ArrayList using Iterator and ListIterator.
- If we want to store elements of reference type in ArrayList then reference type should contain equals method.
- Initial capacity of ArrayList is 10 elements.
- If ArrayList is full then its capacity gets increased by half of its exsiting capacity. 10 → 15
- Ctor declared in ArrayList:
  * public ArrayList() //Initial capacity - 10
  * public ArrayList(int initialCapacity) ← declare capacity)
  * public ArrayList(Collection<? extends E> c)
- Methods declared in ArrayList:
  * public void ensureCapacity(int minCapacity)
  * public void trimToSize()
  * public void forEach(Consumer<? super E> action)

+ Vector: Synchronized.

- It is dynamically growable/shrinkable array.
- It is in java since JDK 1.0. It is legacy class.
- It is synchronized collection.
- It stores data in ordered/sequential fashion [unsorted].

- From Vector, user can access element using integer index.
- In Vector we can insert duplicate elements.
- It allows us to insert multiple null elements.
- We can traverse elements of Vector using Iterator , ListIterator and Enumeration.
- If we want to store elements of reference type in Vector then reference type should contain equals method.
- Initial capacity of Vector is 10 elements.
- If Vector is full then its capacity gets increased its exsiting capacity. (2x) 10 → 20.
- Ctor declared in Vector:
  * public Vector()
  * Vector(int initialCapacity)
  * public Vector(Collection<? extends E> c)
- Methods declared in Vector:
  * public void ensureCapacity(int minCapacity)
  * public void addElement(E obj)
  * public int capacity() //Not exist in ArrayList
  * public void copyInto(Object[] anArray)
  * public E elementAt(int index)
  * public Enumeration<E> elements()
  * public E firstElement()
  * public void forEach(Consumer<? super E> action)
  * public void insertElementAt(E obj,int index)
  * public E lastElement()
  * public boolean removeElement(Object obj)
  * public void removeElementAt(int index)
  * public void removeAllElements()
  * public void setElementAt(E obj,int index)
  * public void setSize(int newSize)
  * public void trimToSize()

+ Stack: LIFO.

- It is sub class of Vector
- It is synchronized collection
- The Stack class represents a last-in-first-out (LIFO) stack of objects.
- A more complete and consistent set of LIFO stack operations is provided by the Deque interface:

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

- Methods declared in Stack:
  - * public boolean empty()
  - * public E peek()
  - * public E pop()
  - * public E push(E item)
  - * public int search(Object o)

+ LinkedList:

- It implements List and Deque interface
- Its implementation is based on Doubly Linked List.
- Implementation of LinkedList is not synchronized.
- We can make it synchronized using Collections.synchronizedList() method.
- It stores data in ordered/sequential fashion [unsorted].
- From LinkedList, user can access element using integer index.
- In LinkedList we can insert duplicate elements.
- It allows us to insert multiple null elements.
- If we want to store elements of reference type in LinkedList then reference type should contain equals method.
- Ctor declared in LinkedList:
  - * public LinkedList()
  - * public LinkedList(Collection<? extends E> c) ← Type check / generics.
- Methods declared in LinkedList:
  - * public void addFirst(E e)
  - * public void addLast(E e)
  - * public Iterator<E> descendingIterator()
  - * public E getFirst()
  - * public E getLast()
  - * public E element()
  - * public E removeFirst()
  - * public E removeLast()

+ Enumeration:

- It is interface declared in java.util package
- Using Enumeration we can traverse collection only in forward direction.
  - The functionality of this interface is duplicated by the Iterator interface. In addition, Iterator adds an optional remove operation, and has shorter method names.
- Methods of Enumeration:

+ boolean hasMoreElements()
+ E nextElement()

- e.g

```
Vector<Integer> vector = new Vector<>();
vector.add(10);
vector.add(20);
vector.add(30);
Enumeration<Integer> e = vector.elements();
while( e.hasMoreElements())
{
        int element = e.nextElement();
        System.out.println(element);
}
```

+ **Iterator:** forward only.

- It is a interface declared in java.util package
- Iterator takes the place of Enumeration in the Java Collections Framework.
- Iterators differ from enumerations in two ways:
    * Iterators allow the caller to remove elements from the underlying collection during the iteration.
    * Method names have been improved. (short)
- Methods of Iterator:
    * boolean hasNext()
    * E next()
    * default void remove()
    * default void forEachRemaining(Consumer<? super E> action)
- e.g

```
Vector<Integer> vector = new Vector<>();
vector.add(10);
vector.add(20);
vector.add(30);
Iterator<Integer> itr = vector.iterator();
while( itr.hasNext())
{
        int element = itr.next();
        System.out.println(element);
```

}

+ **ListIterator:** Forward / Backward (only available for List)

- It is interface declared in java.util package
- It is sub interface of Iterator
- It is designed to use with List Collection only.
- It allows the programmer to traverse the list in either direction
- During travsersing we can add and remove elements
- Methods declared in ListIterator
  - * boolean hasNext()
  - * E next()
  - * boolean hasPrevious()
  - * E previous()
  - * void add(E e)
  - * void set(E e)
  - * void remove()
  - * int nextIndex()
  - * int previousIndex()
- e.g

```
Vector<Integer> vector = new Vector<>();
vector.add(10);
vector.add(20);
vector.add(30);

ListIterator<Integer> itr = vector.listIterator();
while( itr.hasNext())
        System.out.print(itr.next()+"  ");                    //10 20 30
System.out.println();
while(itr.hasPrevious())
        System.out.print(itr.previous()+"      ");            //30 20 10
```

+ **ConcurrentModificationException:**

- This exception may be thrown by methods that have detected concurrent modification of an object when such modification is not permissible.
- For example, it is not generally permissible for one thread to modify a Collection while another thread is iterating over it. In general, the results of the iteration are undefined under these

circumstances. Some Iterator implementations may choose to throw this exception if this behavio detected.

- Iterators that do this are known as fail-fast iterators.

- e.g

```
List<String> list = new ArrayList<String>();
list.add("1");
list.add("2");
list.add("3");
list.add("4");
list.add("5");
Iterator<String> itr = list.iterator();
while( itr.hasNext())
{
        String value = itr.next();
        System.out.println("Value     :        "+value);
        if( value.equals("3"))
                list.remove("3");       //ConcurrentModificationException
}
```

+ **Set:** (List with restrictions)

- It is sub interface of Collection.
- HashSet, LinkedHashSet, TreeSet etc. implements Set interface. These are Set Collections.
- Set models the mathematical set abstraction.
- The Set interface contains only methods inherited from Collection and adds the restricti duplicate elements are prohibited.
- Methods of Set interface:
    * boolean add(E e)
    * boolean addAll(Collection<? extends E> c)
    * boolean contains(Object o)
    * boolean containsAll(Collection<?> c)
    * boolean remove(Object o)
    * boolean removeAll(Collection<?> c)
    * boolean retainAll(Collection<?> c)
    * void clear()
    * boolean isEmpty()
    * int size()
    * Object[] toArray()

* <T> T[] toArray(T[] a)

* default boolean removeIf(Predicate<? super E> filter)
* default Stream<E> stream()
* default Stream<E> parallelStream()
* default Spliterator<E> spliterator()

+ SortedSet:

- It is sub interface of Set.
- It maintains its elements in ascending order, sorted according to the elements' natural ordering (Comparable) or according to a Comparator provided at SortedSet creation time.
- Methods declared in SortedSet:
  * E first()
  * E last()
  * SortedSet<E> headSet(E toElement)
  * SortedSet<E> tailSet(E fromElement)
  * SortedSet<E> subSet(E fromElement,E toElement)

+ NavigableSet:

- It is a sub interface of SortedSet.
- It is sortedSet having navigation methods.
- TreeSet class Implements NavigableSet interface.
- Methods declared in NavigableSet:
  * E ceiling(E e)
  * E floor(E e)
  * E higher(E e)
  * E lower(E e)
  * E pollFirst()
  * E pollLast()
  * Iterator<E> descendingIterator()
  * NavigableSet<E> descendingSet()

+ TreeSet:

- It is a class which implements NavigableSet interface.
- It is based on TreeMap collection.
- It do not contain duplicate elements
- We can not store null elements in TreeSet.

- It is unsynchronized collection.
- Using Collections.synchronizedSortedSet() method we can make it synchronized.
- It stores data in sorted format on the basis of either Comparable or Comparator.
- If we want to store elements of reference type in TreeSet then Reference type must implement Comparable or Comparator interface.

+ Searching Algorithm:

1. Linear search / Sequential Search
   - We can use it search element in any sorted as well as unsorted collection.
   - If collection contains large number of elements then it is time consuming

2. **Binary Search**
   - We can use it only on sorted array *
   - It is faster than Linear search

   break-list to half and compare.

3. **Hashing**
   - If we want to search element in constant time then we should use hashing
   - It is based on hashcode. An integer number that can be generated by processing state of object is called hashcode. Hash function is responsible for generating hashcode.
   - If objects are hashed to the same slot then it is called as collision.
   - To avoid collision we can use collision removal techniques(Open addressing, separate chaining etc.)
   - In seperate chaining, array of linked list is maintained. Each linked list is called as bucke
   - Load Factor = (no of buckets) / bo of elements.

+ HashSet:

- It is a class wich implements Set interface.
- It is based on HashTable.
- It do not contain duplicate elements
- It can contain null element
- It is unsynchronized collection.
- using Collections.synchronizedSet we can make it sychronized.
- HashSet is much faster than TreeSet but do not give guarantee of ordering.
- An instance of Hashtable has two parameters that affect its performance: initial capacity and loa factor.
- The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created.
- One thing worth keeping in mind about HashSet is that iteration is linear in the sum of the num of entries and the number of buckets (the capacity). Thus, choosing an initial capacity that's too h can waste both space and time. On the other hand, choosing an initial capacity that's too low was time by copying the data structure each time it's forced to increase its capacity. If you don't specif

initial capacity, the default is 16. In the past, there was some advantage to choosing a prime number as the initial capacity. This is no longer true. Internally, the capacity is always rounded up to a power of two.

- If we want to add object of reference type in HashSet then refrence type should override equals and hashCode method.
- Ctor declared in HashSet:
  * public HashSet()
  * public HashSet(Collection<? extends E> c)
  * public HashSet(int initialCapacity, float loadFactor)
  * public HashSet(int initialCapacity)

- Methods declared in HashSet:
  * public boolean add(E e)
  * public void clear()
  * public Object clone()
  * public boolean contains(Object o)
  * public boolean isEmpty()
  * public Iterator<E> iterator()
  * public boolean remove(Object o)
  * public int size()
  * public Spliterator<E> spliterator()

+ LinkedHashSet:

- It is a subclass of HashSet
- Its implimentation is based on HashTable and LinkedList.
- It gives gurantee of order of elements.
- It runs nearly as fast as HashSet.
- It do not contain duplicate elements
- It can contain null element
- It is unsynchronized collection.
- using Collections.synchronizedSet we can make it sychronized.

+ Queue: FIFO .

- It is sub interface of Collection.
- PriorityQueue implements Queue interface
- Queue Collection maintains elements in FIFO.
- Queue implementations generally do not allow insertion of null elements

- Methods declared in Queue:
  * boolean add(E e)
  * E element()
  * boolean offer(E e)
  * E peek()
  * E poll()
  * E remove()

-

	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

+ Deque:

- The name deque is short for "double ended queue" and is usually pronounced "deck".
- It is sub interface of Queue.
- This interface defines methods to access the elements at both ends of the deque.
-

	First Element (Head)		Last Element (Tail)	
	Throws exception	Special value	Throws exception	Special value
Insert	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Examine	getFirst()	peekFirst()	getLast()	peekLast()

- LinkedList class implements Deque interface.

+ Map: (key-value) dup ✗ keys, dup ✓ values

- It is interface declared in java.util package.
- It is a part of collection framework but it do not extend Collection interface
- This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.

- Properties, HashTable, HashMap, LinkedHashMap, TreeMap etc implements Map interface. It is also called as Map collection.
- Map collection stores elements in key/Value pair format.
- A map cannot contain duplicate keys but it can contain duplicate values.
- Each key can map to at most one value.
- Abstract methods declared in Map:
    * void clear()
    * boolean containsKey(Object key)
    * boolean containsValue(Object value)
    * Set<Map.Entry<K,V>> entrySet()
    * V get(Object key)
    * boolean isEmpty()
    * Set<K> keySet()
    * V put(K key,V value)
    * void putAll(Map<? extends K,? extends V> m)
    * V remove(Object key)
    * int size()
    * Collection<V> values()

+ Map.Entry:

- It is nested interface declared in Map.
- Methods of Entry interface:
    * K getKey()
    * V getValue()
    * V setValue(V value)

+ HashTable: Synchronized.

- It is class which implements Map interface.
- Since it has implemented Map interface, it stores elements in key/value pair format.
- In HashTable Key and value can not be null.
- Key must be unique, value can be duplicate
- It is synchronized collection.
- To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.
- Ctor declared in HashTable:
    * public Hashtable()
    * public Hashtable(int initialCapacity)

* public Hashtable(int initialCapacity, float loadFactor)
* public Hashtable(Map<? extends K,? extends V> t)

+ HashMap: Unsynchronized

- It is Hash table based implementation of the Map interface.
- The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.
- In HashMap key and value can be null
- Key must be unique, value can be duplicate
- Implementation of HashMap is unsynchronized.
- Using Collections.synchronizedMap method we can make it synchronized.
- To successfully store and retrieve objects from a HashMap, the objects used as keys must implement the hashCode method and the equals method.
  (store key. Retrieve)
- Ctor declared in HashMap:
  * public HashMap()
  * public HashMap(int initialCapacity)
  * public HashMap(int initialCapacity, float loadFactor)
  * public HashMap(Map<? extends K,? extends V> t)

+ LinkedHashMap:

- Hash table and linked list implementation of the Map interface, with predictable iteration order
- This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries.
- In LinkedHashMap key and value can be null
- Key must be unique, value can be duplicate
- Implementation of LinkedHashMap is unsynchronized.
- Using Collections.synchronizedMap method we can make it synchronized.

+ SortedMap:

- It is sub interface of Map.
- A SortedMap is a Map that maintains its entries in ascending order, sorted according to the keys' natural ordering, or according to a Comparator provided at the time of the SortedMap creation.
- Methods declared in Sorted map.
  * Comparator<? super K> comparator()
  * K firstKey()
  * SortedMap<K,V> headMap(K toKey)

* K lastKey()
* SortedMap<K,V> subMap(K fromKey, K toKey)
* SortedMap<K,V> tailMap(K fromKey)

+ **NavigableMap:**

- It is sub interface of SortedMap.
- It is sortedMap having navigation methods. A NavigableMap may be accessed and traversed in either ascending or descending key order.
- Methodes of NavigableMap:
  * Map.Entry<K,V> ceilingEntry(K key)
  * K ceilingKey(K key)
  * NavigableSet<K> descendingKeySet()
  * NavigableMap<K,V> descendingMap()
  * Map.Entry<K,V> firstEntry()
  * Map.Entry<K,V> floorEntry(K key)
  * K floorKey(K key)
  * Map.Entry<K,V> higherEntry(K key)
  * Map.Entry<K,V> lastEntry()
  * Map.Entry<K,V> lowerEntry(K key)
  * K lowerKey(K key)
  * NavigableSet<K> navigableKeySet()
  * Map.Entry<K,V> pollFirstEntry()
  * Map.Entry<K,V> pollLastEntry()

+ **TreeMap:**

- A Red-Black tree based NavigableMap implementation.
- The map is sorted according to the natural ordering of its keys(Comparable), or by a Comparator provided at map creation time.
- In TreeMap, Key cannot be null but value can be null.
- It is unsynchronized collection.
- To make it synchronized we should use Collections.synchronizedSortedMap() method.
- If we want to use object of reference type as a key then reference type should implement either Comparable or Comparator interface.

+ Persistence:

- It is minor pillar of oops.
- It is the process of maintain state of the object either file or databases.

+ File:

- A container which holds collection of records on HDD is called file.
- General classification of files:
  - * Binary files (.dat)
    - can open with specific application.
    - requires less processing hence it is faster.
    - e.g. jpg, mp3, .class etc
  - * Text files(.txt)
    - can open with any text editor.
    - requires more processing hence it is slower.

+ Stream:

- It is an object which is used to perform operations on file(read/write/append)

+ Standard Streams in java:

- System.in --> associated with keyboard
- System.out --> associated with monitor
- System.err --> associated with monitor

+ File related terminology:

- Path : It consist of root directory, sub directories, path seperator and file names.
  e.g c:\Sandeep\Java\SimpleHello\src\Program.java
- Absolute path : It is a path of file from root directory.
  e.g c:\Sandeep\Java\SimpleHello\src\Program.java
- Relative path : It is path of file from curret directory.
  e.g .\src\Program.java
- Path Seperators:

Windows	-	\
Linux	-	/
Mac OS	-	. (dot)

+ File IO:

- To manipulate files, we should use interfaces and classes declared in java.io package.
- Types declared in io package are device independent,

- Interfaces declared in java.io package:
    * FilenameFilter
    * Flushable
    * Closeable
    * DataInput
    * DataOutput
    * ObjectInput
    * ObjectOutput
    * Serializable

- Classes declared in java.io package:
    * Console
    * File

    * InputStream
    * OutputStream
    * FileInputStream
    * FileOutputStream
    * BufferedInputStream
    * BufferedOutputStream
    * DataInputStream
    * DataOutputStream
    * ObjectInputStream
    * ObjectOutputStream
    * PrintStream

    Above classes are required to manipulate binary files.

    * Reader
    * Writer
    * FileReader
    * FileWriter
    * BufferedReader
    * BufferedWriter
    * InputStreamReader

* PrintWriter

above classes are required to manipulate text files

+ java.io.File:

- It represents files and directories of of OS.
- Instances of the File class are immutable; that is, once created, the pathname represented by a File object will never change.
- File ctor:
  * File(String pathname)
- Some of the important methods of File class.
  * public boolean createNewFile() throws IOException
  * public boolean delete()
  * public boolean exists()
  * public String getName()
  * public long getFreeSpace()
  * public long getUsableSpace()
  * public long getTotalSpace()
  * public boolean isDirectory()
  * public boolean isFile()
  * public long lastModified()
  * public long length()
  * public File[] listFiles()
  * public File[] listFiles(FileFilter filter)
  * public boolean mkdir()

-

```
File file = new File("/media/sandeep/DOCUMENTS/E-Book/C++");
if( file.exists())
{
        File[] files = file.listFiles(new FilenameFilter()
        {
                @Override
                public boolean accept(File dir, String name)
                {
                        if(name.endsWith(".pdf"))
                                return true;
```

```
                    return false;
                }
            });
            for (File f : files)
                System.out.println(f.getName());
        }
```

- The java.nio.file package defines interfaces and classes for the Java virtual machine to access files, file attributes, and file systems. This API may be used to overcome many of the limitations of the ava.io.File class.

+ I/O Streams:

- An I/O Stream represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.
- Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.
- No matter how they work internally, all streams present the same simple model to programs that use them: A stream is a sequence of data. A program uses an input stream to read data from a source.

- java.lang.Object

|- java.io.InputStream
|- java.io.OutputStream
|- java.io.Reader
|- java.io.Writer

- Byte Streams :

* Programs use byte streams to perform input and output of 8-bit bytes.
* All byte stream classes are descended from InputStream and OutputStream.

//Program to write single character at a time in file

```
public static void writeRecord( String pathname ) throws Exception
{
    try( FileOutputStream outputStream = new FileOutputStream(pathname))
    {
        char ch = 'A';
```

```
            while( ch <= 'Z' )
            {
                outputStream.write(ch);
                ++ ch;
            }
        }
    }
    //Program to read single character at a time in file
    public static void readRecord( String pathname ) throws Exception
    {
        try( FileInputStream inputStream = new  FileInputStream(pathname))
        {
            int data;
            while( ( data = inputStream.read( ) ) != -1 )
            {
                char ch = (char)data;
                System.out.print(ch+" ");
            }
        }
    }
```

* Examples we've seen is unbuffered I/O. This means each read or write request is handled dire by the underlying OS. This can make a program much less efficient, since each such request ofte triggers disk access, network activity, or some other operation that is relatively expensive.

* To reduce this kind of overhead, the Java platform implements buffered I/O streams. Buffered streams read data from a memory area known as a buffer; the native input API is called only whe buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output A called only when the buffer is full.

* There are four buffered stream classes used to wrap unbuffered streams:
BufferedInputStream and BufferedOutputStream create buffered byte streams, while
BufferedReader and BufferedWriter create buffered character streams.

* A program can convert an unbuffered stream into a buffered stream :

```
inputStream = new BufferedInputStream(new FileInputStream(pathname));
outputStream = new BufferedOutputStream(new FileOutputStream(pathname));
```

- Data Streams:

* Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values.

* All data streams implement either the DataInput interface or the DataOutput interface.

* outputStream = new DataOutputStream(new BufferedOutputStream(...));

* A data output stream lets an application write primitive Java data types to an output stream .

* inputStream = new DataInputStream(new BufferedInputStream(...));

* A data input stream lets an application read primitive Java data types from an underlying input stream.

* Notice that DataStreams detects an end-of-file condition by catching EOFException,

* Also notice that each specialized write in DataStreams is exactly matched by the corresponding specialized read.

- Object Streams:

* Object streams support I/O of objects.

* The object stream classes are ObjectInputStream and ObjectOutputStream.

* These classes implement ObjectInput and ObjectOutput, which are subinterfaces of DataInput and DataOutput.

* readObject() is method of ObjectInputStream and writeObject is method of ObjectOutputStream.

* Process of converting state of the object into bytes is called serialization and converting bytes into Object is called deserialization.

* To serialize state, class must implement Serializable marker interface.

* Without implementing Serializable interface, if we try to serialize state of the object then JVM throws NotSerializableException.

* If we do not serialize state of any specific field then we should declare such field as transient.

* State of static and transient field can not be serialized.

* serialVersionUID is used to ensure that same class (that was used during serialization) is loaded during deserialization. serialVersionUID is used for version control of object.

* e.g. private static final long serialVersionUID = -5410036443708494562L;

* //Serialization

```
stream = new ObjectOutputStream( new BufferedOutputStream(...));
Employee emp1 = new Employee("Sandeep",33,45000.50f);
stream.writeObject( emp1 );
```

*
```
//Deserialization
stream = new ObjectInputStream(new BufferedInputStream(...));
Employee emp2 = ( Employee )stream.readObject();
//TODO
```

- Character Streams:

* The Java platform stores character values using Unicode conventions.

* Character stream I/O automatically translates this internal format to and from the local character set

* I/O with character streams is no more complicated than I/O with byte streams.

* All character stream classes are descended from Reader and Writer.

* Character streams are often "wrappers" for byte streams. The character stream uses the byte stream to perform the physical I/O, while the character stream handles tr between characters and bytes. FileReader, for example, uses FileInputStream, while FileWriter us FileOutputStream.

* There are two general-purpose byte-to-character "bridge" streams: InputStreamReader a OutputStreamWriter. Use them to create character streams when there are no prepackaged charac stream classes that meet your needs.

Java Notes - Chapter 14 - String Handling

+ Using Following classes, we can manipulate strings in java.

- java.lang.Character
- java.lang.String
- java.lang.StringBuffer
- java.lang.StringBuilder
- java.util.StringTokeniizer

+ ASCII versus UNICODE:

- ASCII and Unicode are two character encodings. Basically, they are standards on how to represent different characters in binary so that they can be written, stored, transmitted, and read in digital media.

- The main difference between the two is in the way they encode the character and the number of bits that they use for each.

- ASCII originally used seven bits to encode each character. This was later increased to eight with Extended ASCII to address the apparent inadequacy of the original.

- In contrast, Unicode uses a variable bit encoding program where you can choose between 32, 16, and 8-bit encodings.

- Major advantage of Unicode is that at its maximum it can accommodate a huge number of characters. Because of this, Unicode currently contains most written languages and still has room for even more. This includes typical left-to-right scripts like English and even right-to-left scripts like Arabic. Chinese, Japanese, and the many other variants are also represented within Unicode

- In order to maintain compatibility with the older ASCII, which was already in widespread use at the time, Unicode was designed in such a way that the first eight bits matched that of the most popular ASCII page. So if you open an ASCII encoded file with Unicode, you still get the correct characters encoded in the file.

+ Character:

- It is a wrapper class of char data type.

- In addition, this class provides several methods for determining a character's category (lowercase letter, digit, etc.) and for converting characters from uppercase to lowercase and vice versa.

- Character information is based on the Unicode Standard.

- The char data type are based on the original Unicode specification, which defined characters as fixed-width 16-bit entities.

- The range of legal code points is now U+0000 to U+10FFFF, known as Unicode scalar value.

- The set of characters from U+0000 to U+FFFF is sometimes referred to as the Basic Multilingual Plane (BMP). Characters whose code points are greater than U+FFFF are called supplementary characters.

- The Java platform uses the UTF-16 representation in char arrays and in the String and StringBuffer classes.

- More about utf:

  * Unicode Transformation Format(UTF) is an algorithmic mapping from every unicode code point to unique byte sequence.

  * UTF-8 and UTF-16 are variable length encodings.

  * UTF-8 is most common on web. UTF-16 is used by Java and Windows.UTF-8 and UTF-32 are used by linux and various Unix systems.

  * UTF-16 encodes characters into specific binary sequences using either one or two 16-bit sequences.

+ String :

- It is a final class declared in java.lang package

- It implements Serializable, Comparable<String>, CharSequence interfaces.

- In java String is not built in type. It is a reference type.

- A String represents a string in the UTF-16 format.

- we can create object of String with or without new operator.

  * String str1 = new String("Sandeep"); //OK

  * String str2 = "SunBeam"; //OK

- String str = "Soham" is equivalent to:

```
char data[] = {'S', 'o', 'h','a', 'm' };
String str = new String(data);
```

char array to string.

- If we try to modify string then new string object gets created i.e. String objects are constant/ immutable. Because String objects are immutable they can be shared.

- The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings.

- **Some of the ctor's of String:**

  * public String();

  * public String(char[] value)

  * public String(byte[] bytes)

  * public String(String original)

  * public String(StringBuffer buffer)

  * public String(StringBuilder builder)

- Methods declared in String:

* public char charAt(int index)
* public int codePointAt(int index)
* public int compareTo(String anotherString)
* public int compareToIgnoreCase(String str)
* public String concat(String str)
* public boolean endsWith(String suffix)
* public boolean equalsIgnoreCase(String anotherString)
* public static String format(String format,Object... args)
* public byte[] getBytes()
* public int indexOf(int ch)
* public int indexOf(String str)
* public String intern()
* public boolean isEmpty()
* public int lastIndexOf(int ch)
* public int lastIndexOf(String str)
* public int length()
* public boolean matches(String regex)
* public String[] split(String regex)
* public boolean startsWith(String prefix)
* public String substring(int beginIndex)
* public String substring(int beginIndex, int endIndex)
* public char[] toCharArray()
* public String toLowerCase()
* public String toUpperCase()
* public String trim()
* public static String valueOf(int i)
* overloaded valueOf methods.

- Some useful twisters:

1.
```
String str1 = new String("SunBeam");
String str2 = new String("SunBeam");
if( str1 == str2 )
        System.out.println("Equal");
else
        System.out.println("Not Equal");
//Output : Not Equal
```

2.
```
String str1 = new String("SunBeam");
String str2 = new String("SunBeam");
if( str1.equals(str2) )
        System.out.println("Equal");
else
        System.out.println("Not Equal");
//Output : Equal
```

3.
```
String str1 = "SunBeam";
String str2 = "SunBeam";
if( str1 == str2 )
        System.out.println("Equal");
else
        System.out.println("Not Equal");
//Output : Equal
```

4.
```
String str1 = "SunBeam";
String str2 = "SunBeam";
if( str1.equals( str2 ) )
        System.out.println("Equal");
```

```
else
        System.out.println("Not Equal");
//Output : Equal
```

5.
```
String str1 = "SunBeam";
String str2 = new String("SunBeam");
if( str1 == str2 )
        System.out.println("Equal");
else
        System.out.println("Not Equal");
//Output : Not Equal
```

6.
```
String str1 = "SunBeam";
String str2 = new String("SunBeam");
if( str1.equals( str2 ) )
        System.out.println("Equal");
else
        System.out.println("Not Equal");
//Output : Equal
```

7.
```
String str1 = "Sun"+"Beam";
String str2 = "SunBeam";
if( str1 == str2  )
        System.out.println("Equal");
else
        System.out.println("Not Equal");
//Output : Equal
```

8.
```
String str = "Sun";
```

```
String str1 = str + "Beam";
String str2 = "SunBeam";
if( str1 == str2  )
        System.out.println("Equal");
else
        System.out.println("Not Equal");
//Output : Not Equal
```

9.
```
String str = "Sun";
String str1 = (str + "Beam").intern();
String str2 = "SunBeam";
if( str1 == str2  )
        System.out.println("Equal");
else
        System.out.println("Not Equal");
//Output : Equal
```

* String created without new() adds value onto the string literal pool (constant pool) → search on google.

- Literal strings within the same class in the same package represent references to the same String object. same
- Literal strings within different classes in the same package represent references to the same String object. same
- Literal strings within different classes in different packages likewise represent references to the same String object. same
- Strings computed by constant expressions are computed at compile time and then treated as if they were literals.
- Strings computed by concatenation at run time are newly created and therefore distinct. diff.
- The result of explicitly interning a computed string is the same string as any pre-existing literal string with the same contents.

- String Pooling:

* When compiling source code, compiler must process each literal string and emit the string into metadata. If the same literal string appears several times in source code, emitting all of these strings into metadata will bloat the size of the resulting file.

* To remove this bloat, many compilers write the literal string into metadata only once.

* An ability of a compiler to merge multiple occurrences of single string into single instance can reduce the size. This process is called String Pooling.

+ Regular expression: //Assignment for reader

+ StringBuffer: (Synchronized) → slower.

- It is a final class declared in java.lang package.
- It implements Serializable and CharSequence interface.
- A string buffer is like a String, but can be modified i.e it is mutable string object.
- It is synchronized.
- equals and hashCode methods are not overiden in StringBuffer.
- It is slower than StringBuilder class.
- Ctor's declared in StringBuffer:
  * public StringBuffer() //Initial capacity - 16 char
  * public StringBuffer(int capacity)
  * public StringBuffer(String str) //Initial capacity - 16 + string length
- Some of the methods declared in StringBuffer:
  * public StringBuffer append(String str)
  * //Other overloaded append methods.
  * public int capacity()
  * public int length()
  * public char charAt(int index)
  * public void setCharAt(int index,char ch);
  * public StringBuffer reverse() //Not available in string
  * public String substring(int start)
  * public void trimToSize()

- Some useful twisters:

1.
```
StringBuffer sb1 = new StringBuffer("Sandeep");
StringBuffer sb2 = new StringBuffer("Sandeep");
if( sb1 == sb2)
        System.out.println("Equal");
else
        System.out.println("Not Equal");
//Output : Not Equal
```

2.
```
StringBuffer sb1 = new StringBuffer("Sandeep");
StringBuffer sb2 = new StringBuffer("Sandeep");
if( sb1.equals(sb2))
        System.out.println("Equal");
else
        System.out.println("Not Equal");
//Output : Not Equal
```

3.
```
String str1 = new String("Sandeep");
StringBuffer str2 = new StringBuffer("Sandeep");
if( str1  == str2 )        //Compiler Error
        System.out.println("Equal");
else
        System.out.println("Not Equal");
```

4.
```
String str1 = new String("Sandeep");
StringBuffer str2 = new StringBuffer("Sandeep");
if( str1.equals(str2) )
```

```
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
    //Output : Not Equal
```

+ StringBuilder: (Unsynchronized) faster

- It is a final class declared in java.lang package.
- It implements Serializable and CharSequence interface.
- A string buffer is like a String, but can be modified i.e it is mutable string object.
- It is unsynchronized.
- equals and hashCode methods are not overiden in StringBuilder
- It is faster than StringBuffer class.
- Ctor's declared in StringBuilder:
  - * public StringBuilder() //Initial capacity - 16 char
  - * public StringBuilder(int capacity)
  - * public StringBuilder(String str) //Initial capacity - 16 + string length
- This class provides an API compatible with StringBuffer, but with no guarantee of synchronization.
- **Some useful Twisters:**

1.
```
StringBuilder sb1 = new StringBuilder("SunBeam");
StringBuilder sb2 = new StringBuilder("SunBeam");
if( sb1 == sb2 )
    System.out.println("Equal");
else
    System.out.println("Not Equal");
//Output : Not Equal
```

2.
```
StringBuilder sb1 = new StringBuilder("SunBeam");
StringBuilder sb2 = new StringBuilder("SunBeam");
if( sb1.equals(sb2) )
```

```
            System.out.println("Equal");
        else
            System.out.println("Not Equal");
        //Output : Not Equal
```

3.
```
StringBuilder sb1 = new StringBuilder("SunBeam");
StringBuffer sb2 = new StringBuffer("SunBeam");
if( sb1 == sb2 )        //Compiler error
        System.out.println("Equal");
else
        System.out.println("Not Equal");
```

4.
```
StringBuilder sb1 = new StringBuilder("SunBeam");
StringBuffer sb2 = new StringBuffer("SunBeam");
if( sb1.equals(sb2) )
        System.out.println("Equal");
else
        System.out.println("Not Equal");
//Output : Not Equal
```

+ StringTokenizer: (Not recommended/legacy class) → use split() instead OR java.util.regex package

- java.util.StringTokenizer is a class.
- It implements Enumeration interface.
- The string tokenizer class allows an application to break a string into tokens.
- StringTokenizer is a legacy class that is retained for compatibility reasons although its use is discouraged in new code. It is recommended that anyone seeking this functionality use the split method of String or the java.util.regex package instead.
- Ctor's declared in StringTokenizer:

```
* public StringTokenizer(String str)
* public StringTokenizer(String str, String delim)
* public StringTokenizer(String str, String delim, boolean returnDelims)
```

- Methods:

* public int countTokens()
* public boolean hasMoreTokens()
* public String nextToken()
* public String nextToken(String delim)

- example:

```
String str = "www.sunbeaminfo.com";
String delim = ".";
StringTokenizer stk = new StringTokenizer(str,".");
while( stk.hasMoreTokens())
{
        String token = stk.nextToken();
        //TODO
}
```

+ Reflection:

- In Java, the process of analyzing and modifying all the capabilities of a class at runtime is calle Reflection.

- Reflection API in Java is used to manipulate class and its members which include fields, methods, constructor, etc. at runtime.

- One advantage of reflection API in java is, it can manipulate private members of the class too.

- The java.lang.reflect package provides many classes to implement reflection in java.

- Methods of the java.lang.Class class are used to gather the complete metadata of a particular class.

+ Application of Reflection: (debugger, intellisense)

- RAD tools use reflection to drag n drop components at runtime.

- To access values of private fields, debugger use reflection.

- To implement intellisense feature, IDE's use reflection.

- To create stub object, RMI use reflection.

- TO analyze members of the class, javap tool use reflection.

+ How to get complete information about a class?

1. using getClass() method:

```
Integer obj = new Integer(10);
Class<?> c=  obj.getClass();
```

2. using .class syntax:

```
Class<?> c = Number.class;
```

3. using Class.forName() method:

```
Class<?> c = Class.forName("java.lang.Object");
```

Methods declared in java.lang.Class:

1. public static Class<?> forName(String className) throws ClassNotFoundException
2. public Annotation[] getAnnotations()
3. public ClassLoader getClassLoader()
4. public Constructor<?>[] getConstructors()throws SecurityException
5. public Field getField(String name) throws NoSuchFieldException, SecurityException
6. public Field[] getFields() throws SecurityException
7. public Method getMethod(String name, Class<?>... parameterTypes);
8. public Method[] getMethods() throws SecurityException

9. public Class<?>[] getInterfaces()
10. public int getModifiers()
11. public String getName()
12. public Package getPackage()
13. public InputStream getResourceAsStream(String name)
14. public String getSimpleName()
15. public T newInstance() throws InstantiationException, IllegalAccessException

+ Classes declared in java.lang.reflect Package :

Following is a list of various Java classes in java.lang.package to implement reflection.

Field: This class is used to gather declarative information such as datatype, access modifier, name and value of a variable.

Method: This class is used to gather declarative information such as access modifier, return type, name, parameter types and exception type of a method.

Constructor: This class is used to gather declarative information such as access modifier, name and parameter types of a constructor.

Modifier: This class is used to gather information about a particular access modifier.

+ Middleware example:

```
class Convert
{
  public static Object changeType(String strValue, Parameter parameter )
  {
      switch( parameter.getType().getSimpleName())
      {
      case "int":
        return Integer.parseInt(strValue);
      }
      return null;
  }
}
public class Program
{
```

```
public static void main(String[] args)
{
    try(Scanner sc = new Scanner(System.in))
    {
      System.out.print("F.Q. Class Name        :        ");
      String className = sc.nextLine();
      Class<?> c = Class.forName(className);
      System.out.print("Method Name    :        ");
      String methodName = sc.nextLine();
      for( Method method : c.getMethods())
      {
          if( method.getName().equalsIgnoreCase(methodName))
          {
            Parameter[] parameters = method.getParameters();
            Object[] arguments = new Object[ parameters.length ];
            for( int index = 0; index < parameters.length; ++ index )
            {
                System.out.println(parameters[ index ].getName()+" : ");
                arguments[ index ] = Convert.changeType(sc.nextLine(), parameters[ index ]);
            }
            Object result = method.invoke(c.newInstance(), arguments);
            System.out.println("Result :        "+result);
          }
      }
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
} }
```

+ Metadata:

- Data about data is called as metadata.
- Metadata has may uses. Here are some of them are:
  * Metadata removes need for header and library files when compiling.
  * IDE's intellisense feature parses metadata to tell us what methods type offers.
  * Metadata allows an object fields to be serialized into a memory block, remoted to another machine and then de-serialized , recreating the object and its state on the remote machine.
  * Metadata allows the garbage collector to track lifetime of objects.

+ Annotations:

- Using annotations we can add metadata programatically.
- Annotations have no direct effect on the operation of the code they annotate.
- Annotations have a number of uses, among them:
  * Information for the compiler: Annotations can be used by the compiler to detect errors or suppress warnings.
  * Compile-time and deployment-time processing: Software tools can process annotation information to generate code, XML files, and so forth.
  * Runtime processing â€" Some annotations are available to be examined at runtime.
  * Many annotations replace comments in code.

+ Annotations Basics:

- The at sign character (@) indicates to the compiler that what follows is an annotation.

  e.g @Override

  public String toString { ... }

- The annotation can include elements, which can be named or unnamed, and there are values for those elements:

  @SuppressWarnings(value = "unchecked")

  void myMethod() { ... }

  If there is just one element named value, then the name can be omitted, as in:

  @SuppressWarnings("unchecked")

  void myMethod() { ... }

- If the annotation has no elements, then the parentheses can be omitted.
- If the annotations have the same type, then this is called a repeating annotation:

```
@Author(name = "Jane Doe")
@Author(name = "John Smith")
class MyClass { ... }
```

+ Types of Annotation

- Marker Annotation
- Single-Value Annotation
- Multi-Value Annotation

+ Declaring an Annotation Type:

- The annotation type definition looks similar to an interface definition where the keyword interface is preceded by the at sign (@)

```
@interface ClassPreamble
{
  String author();
  String date();
  int currentRevision() default 1;
  String lastModified() default "N/A";
  String lastModifiedBy() default "N/A";
  // Note use of array
  String[] reviewers();
}
```

```
- @ClassPreamble (
  author = "Sandeep Kulange",
  date = "7/02/2017",
  currentRevision = 6,
  lastModified = "15/07/2016",
  lastModifiedBy = "Smita Kadam",
  // Note array notation
```

```
reviewers = {"Nikhil", "Ritika", "Sangita"}

)
public class Generation3List extends Generation2List
{    }
```

- To make the information in @ClassPreamble appear in Javadoc-generated documentation, you must annotate the @ClassPreamble definition with the @Documented annotation

```
@Documented
@interface ClassPreamble
{       }
```

+ **Annotations That Apply to Other Annotations**:

- @Retention
- @Documented
- @Target
- @Inherited
- @Repeatable

- @Retention: It is used to specify to what level annotation will be available.

RetentionPolicy	Availability
SOURCE	refers to the source code, discarded during compilation.
CLASS	refers to the .class file, available to compiler but not to JVM.
RUNTIME	refers to the runtime, available to java compiler and JVM .

- @Documented: It is used to signal to the JavaDoc tool that your custom annotation should be visible in the JavaDoc for classes using your custom annotation.

- @Traget: This meta annotation says that this annotation type is applicable for only the element (ElementType) listed. Possible values for ElementType are:

1. CONSTRUCTOR
2. FIELD
3. LOCAL_VARIABLE
4. METHOD
5. PACKAGE
6. PARAMETER
7. TYPE

- @Inherited: It signals that a custom Java annotation used in a class should be inherited by subclasses inheriting from that class.

- @Repeatable: It is java 8 annotation. There are some situations where you want to apply the same annotation to a declaration or type use.

```
@Alert(role="Manager")
@Alert(role="Administrator")
public class UnauthorizedAccessException extends SecurityException
{ ... }
```

+ Enum :

- If we want to give name to the literals then we should use Enum.
- Enum is reference type in java.
- All enums are implicitly extended from java.lang.Enum class which is abstract.
- Methods declared in java.lang.Enum class:
    * public final String name()
    * public final int ordinal()
    * public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)
    * public final Class<E> getDeclaringClass()

+ Enum Declaration :

- enum Color

```
{
    RED, BLUE, GREEN;
}
```

- After compilation enum looks like as follows:

```
final class Color extends java.lang.Enum<Color>
 {
        public static final Color RED;

        public static final Color GREEN;

        public static final Color BLUE;

        public static Color[] values();

        public static Color valueOf( String color );
 }
```

- Since every enum is implicitly final class we cannot extend enum.
- Compiler adds values and valueOf method at compile time in enum.
- As shown in code above enum members are objects of enum.

- Let us see how to access it:

```
Color[] colors = Color.values();
for (Color color : colors)
{
   String name = color.name();
   int ordinal = color.ordinal();
   System.out.printf("%-15s%-5d\n",name, ordinal);
}
```

- Now Let us see how to give name to the literals:

```
enum Day
{
   SUN("SunDay",1),MON("MonDay",2),TUES("TuesDay",3);
   private String name;
   private int number;
   private Day(String name,int number)
   {
        this.name = name;
        this.number = number;
   }
   public String getName()
   {
        return name;
   }
   public int getNumber()
   {
        return number;
   }
}
```

- Let us see how to access it:

```
Day[] days = Day.values();
for (Day day : days)
{
    String name = day.name();
    int ordinal = day.ordinal();
    String dayName = day.getName();
    int dayNumber = day.getNumber();
}
```

- we can override methods in enum.

Java Notes - Chapter 18 - Concurrency

+ Concurrency:

- It is minor pillar of oops.
- A large subset of programming problems can be solved using sequential programming. For some problems, however, it becomes convenient or even essential to execute several parts of program in parallel.
- Parallel programming can produce great improvements in program execution speed.

Process of executing multiple task parallely is called concurrency.

+ Multitasking:

- An ability of any operating system to execute single task at a time is called single tasking.
- An ability of any operating system to execute multiple task at a time is called multitasking.
- Multitasking can be process based/thread based multi-tasking.

+ Process Based Multitasking: slower faster

- Program in execution is called process / running instance of a program is called process.
- More about Process:
  - * Each process run in separate address space.
  - * Each process has its own set of resources.
  - * A process contains at least one thread.
  - * Creating and disposing process is relatively time consuming task.
- When CPU executes multiple processes, it first save state of process into process control block (PCB) and then switches to another process. It is called context switch. But context switch is heavy hence process based multi-tasking is called heavy weight multitasking.

(PCB)

+ Thread Based Multitasking:

- Light weight process is called thread. In other words it is a seperate path of execution which runs independantly.
- If we want to utilize harware resources (CPU) efficiently then we should use thread.
- More about thread:
  - * Threads belonging to the same process share the process's address space and code.
  - * Thread creation is very echonomical.
- Since thread gets executed inside same process they do not require PCB. i.e context switching is not required. So thread based multitasking is faster than process based multitasking.

+ MultiThreading:

- If we create application using single thread then it is called as single threaded application and if we create application using multiple threads then it is called as multithreaded application.
- Java is multithreaded programming language.
- Java threading is based on the low level pthreads approach which comes from C.

Java Notes - Chapter 18 - Concurrency

- Java's multithreading is preemptive, which means that a scheduling mechanism provides time slices for each thread and context switching to the another thread, so that each one is given a reasonable amount of time to drive its task.

- If JVM starts execution of java application then it also starts execution of two threads:

1) * Main Thread
    # It is user thread.
    # It is responsible for invoking main method.

2) * Garbage Collector
    # It is daemon thread.
    # It is responsible for reclaming memory of unused object.

- If we want to manipulate threads in java then we should use types declared in java.lang package.

+ Thread creation in Java:

1) Runnable Interface -
2) Thread Class. -

- In java, we can create thread using two ways:
    1. By implementing java.lang.Runnable interface.) ← interface
    2. By extending java.lang.Thread class.) ← class .
- Runnable is functional interface which contains "void run()" method.
- Thread is a class which implements Runnable interface.
- Members of Thread class:

```
-----------------------------------------------------
Nested Type
-----------------------------------------------------
class Thread
{
    public static enum State
    {

NEW,RUNNABLE,BLOCKED,WAITING,TIMED_WAITING,TERMINATED
    }
}
-----------------------------------------------------
Fields
-----------------------------------------------------
public static final int MIN_PRIORITY = 1
public static final int NORM_PRIORITY = 5
public static final int MAX_PRIORITY = 10
-----------------------------------------------------
```

Constructor's

---

public Thread()
Thread(String name)
Thread(Runnable target)
Thread(Runnable target, String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)

Method's

---

public static Thread currentThread();
public final String getName()
public final void setName(String name)
public final int getPriority()
public final void setPriority(int newPriority)
public Thread.State getState()
public final ThreadGroup getThreadGroup()
public final boolean isAlive()
public final boolean isDaemon()
public final void setDaemon(boolean on);
public final void join() throws InterruptedException
public static void sleep(long millis) throws InterruptedException
public static void yield()
public void start()
public final void suspend() //@Deprecated
public final void resume() //@Deprecated
public final void stop() //@Deprecated
public void destroy() //@Deprecated

+ Thread creation using Runnable interface:

```
class CThread implements Runnable
{
  private Thread thread;
  public CThread( String name )
  {
```

```
        this.thread = new Thread(this, name);
        this.thread.start();
    }
    @Override
    public void run()
    {
        //TODO : Business logic
    }
}
public class Program
{
    public static void main(String[] args)
    {
        new CThread("User Thread#1");
    }
}
```

- Thread creation using Thread class:

```
class CThread extends Thread
{
    public CThread( String name )
    {
        super( name );
        this.start();
    }
    @Override
    public void run()
    {
        //TODO : Business logic
    }
}
public class Program
{
    public static void main(String[] args)
    {
        new CThread("User Thread#1");
```

```
    }
}
```

use Interface → if class already extending another class
use class → if class not extending anything(class)

+ Difference between process based and thread based multitasking:

- If class do not extends any class then we can create thread by extending Thread class.
- If class extends any class then we can create thread by implementing Runnable interface.
- If class extends Thread then sub class can ommit from threading.
- If class implements Runnable then all sub classes must participate in threading.

+ Relation between th.start() and run() method:

thread.start() do not call run method rather it register's the thread with operating system via JVM and then run method gets called on object which implements either Runnable interface or extends Thread class.

+ Types of thread:

1. User Thread:
   - User thread is also called as non daemon thread.
   - New thread created is by default treated as user thread.
   - Life span / execution of user thread is do not depends on execution of daemon thread.
2. Daemon Thread:
   - Daemon thread is also called as background thread.
   - using th.setDaemon(true) we can convert user thread into daemon thread.
   - Life span / execution of daemon thread is depends on execution of user thread.

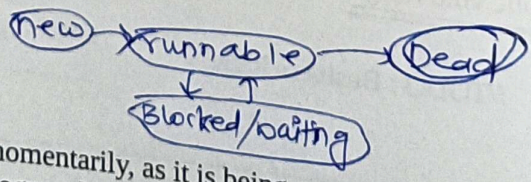
+ Thread termination:

Thread can terminate due to following reason:

1. if run method executes successfully.
2. during execution of run if any exception occurs
3. during execution of run, if jvm encounters return statement.

+ Thread Life Cycle:

A thread can be in any one of four states:



1. New:

A thread remains in this state only momentarily, as it is being created. The thread is not yet running. When thread is in new state, the program has not started executing code inside of it.

2. Runnable:

Once you invoke start method, the thread is in runnable state.

This means that a thread can be run when the time-slicing mechanism has CPU cycles available for thread. Thus thread might or might not be running at any moment but

there is nothing to prevent it from being run if the scheduler can arrange it. That is, it's not dead or block.

3. Blocked and Waiting:

The thread can be run, but something prevents it. While a thread is in blocked state, the scheduler will simply skip it and not give any CPU time. Until thread reenters the runnable state it won't perform any operations.

A task can become blocked for the following reason:

- You have put the task to sleep by calling Thread.sleep(...) method.
- You have suspended the execution of thread with wait
- The task is waiting for some I/O to complete.
- The task is trying to call synchronized method on another object, and that object's lock is not available because it has already been acquired by another task.

4. Dead:

A thread in dead or terminated state is no longer schedulable and will not receive any CPU time. Its task is completed and it is no longer runnable.

A thread can terminate due to following reason:

- It dies a natural death because the run method exist normally.
- It dies abruptly because an uncaught exception terminates the run method.

Thread Priorities:

- In java programming language every thread has a priority. Thread Priority talks about how much time should spent for thread.
- By default, a thread inherits priority of the thread that constructed it.
- We can set the priority to any value between MIN_PRIORITY and MAX_PRIORITY.
- We can increase or decrease priority of thread using setPriority() method.

```
e.g th.setPriority( Thread.NORM_PRIORITY + 3 );
```

- If the priority is not in the range MIN_PRIORITY to MAX_PRIORITY then jvm throws IllegalArgumentException.
- Whenever thread scheduler has a chance to pick a new thread, it prefers thread with higher priority. However thread priorities are highly system dependent. When virtual machine relies on thread implementation of the host platform, the java thread priorities are mapped to the priority levels of the host platform, which may have more or fewer thread priority levels.

- For example, Windows has 7 priority levels. Some of the java priorities will map to the same OS level. In Oracle JVM for linux, thread priorities are ignored all together- all threads have same priority.

+ Yielding:

- If you know that you've accomplished what you need to during one pass through a loop in your run() method, you can give a hint to the thread scheduling mechanism that you have done enough and that some other task might as well have the CPU. This hint takes the form of the yield() method.
- When you call yield(),you are suggesting that other threads of the same priority might be run.

+ Joining Thread:

- One thread may call join on another thread to wait for the second thread to complete before proceeding.
- You may also call join() with timeout argument so that if the target thread doesn't finish in that period of time, the call to join returns anyway.

+ Critical Section:

- Sometimes, you only want to prevent multiple thread access to part of the code inside method instead of entire method. This section of a code is called critical section.

+ Resource Locking / UnLocking:

- When multiple threads try to access same object at the same time, it is called as "race condition".
- This may result in corrupting object state and hence will get unexpected results.
- To avoid this Java provides sync primitive known as "monitor".
- Each java object is associated with a monitor object. The monitor can be locked or unlocked by the thread.
- If monitor is already locked, another thread trying lock it will be blocked until monitor is unlocked
- The thread who had locked monitor, is said to be owner of monitor and only that thread can unlock the monitor.
- Java provides "synchronized" keyword to deal with monitors.
  - synchronized block:

* example:

```
void someMethod()
{
    // ...
    synchronized(obj)
    {
        //TODO
    }
}
```

- When certain part of the method should be executed by single thread at a time, we can use "synchronized" block.
- When a thread begin execution of the sync block, it will lock the monitor associated with given object (obj) and continue execute within that block.
- Meanwhile if another thread try to execute same sync block, it will try to lock the object monitor again.
- Since object is already locked, the second thread will be blocked, until first thread release the lock (at the end of sync block).
- When first thread release the lock, the waiting second thread will acquire it and continue to execute the block.

- Synchronized Method:

* When non-static synchronized method is invoked by the thread, it will lock the monitor associate with current object ("this"). When method completes, thread will release the lock.
* When static synchronized method is invoked by the thread, it will lock the monitor object associated with "ClassName.class" object.
* This ensure that two threads cannot execute same synchronized method on the same object.

example:

```
class Account
{
	private double balance;
	// ...
	public synchronized void withdraw(double amt)
	{
		// get cur balance from db
		// balance = balance - amt;
		// update new balance into db
	}
	public synchronized void deposit(double amt)
	{
		// get cur balance from db
		// balance = balance + amt;
		// update new balance into db
	}
	public double getBalance()
	{
		// get cur balance from db
		// return balance;
```

```
        }
    }
```

* a1.withdraw(...) and a1.withdraw(...) executed by two different threads at the same time will not execute parallely. (As "a1" object is locked by thread1 and thread2 must wait for it).

* a1.withdraw(...) and a1.deposit(...) executed by two different threads at the same time will not execute parallely. (As "a1" object is locked by thread1 and thread2 must wait for it).

* a1.withdraw(...) and a2.withdraw(...) executed by two different threads at the same time can execute parallely. As lock is acquired by two threads on two different objects.

* a1.withdraw(...) and a2.getBalance(...) executed by two different threads at the same time can execute parallely. Because getBalance() method is not synchronized and hence the thread is neither trying lock the object nor will get blocked.

* If all methods in a class are synchronized, only one thread can perform any operation on object of that class. Such class is called as "synchronized" / "thread-safe" class. E.g. StringBuffer, Vector, Hashtable, etc.

+ Thread Synchronization:

- When multiple threads on same monitor try to communicate with each other then it is called inter thread communication.
- Inter thread communication implies synchronization.
- using wait(), notify() and notifyAll() methods we can achieve synchronoization.
- Since we perform synchronization and notification mechanism on objects monitor, wait / notify / notiifyAll belongs to java.lang.Object.
- Consider the following example

```
class Program
{
    void someMethod()throws InterruptedException
    {
        String str = new String("Hello");
        synchronized( this )
        {
            str.wait(); //Exception : IllegalMonitorStateException
        }
    }
}
```

- In multiprocessor environment, if we want to sync data between the threads then we should declare data as volatile.

+ Deadlock:

- It is possible for one task to get stuck waiting for another task, which in turns waits for another task and so on, until the chain leads back to task waiting on the first one. You get continuous loop of tasks waiting on each other and no one can move. This is called deadlock.

- To avoid dead lock programmer should write logic.

Java Notes – Chapter 21 – Inside the Java Virtual Machine

The Java Virtual Machine: The Java virtual machine is called "virtual" because it is an abstract computer defined by a specification.

The Lifetime of a Java Virtual Machine: A runtime instance of the Java virtual machine has a clear mission in life: to run one Java application. When a Java application starts, a runtime instance is born. When the application completes, the instance dies. If you start three Java applications at the same time, on the same computer, using the same concrete implementation, you'll get three Java virtual machine instances. Each Java application runs inside its own Java virtual machine.

Inside the Java virtual machine, threads come in two flavors: daemon and non- daemon. A daemon thread is ordinarily a thread used by the virtual machine itself, such as a thread that performs garbage collection. The application, however, can mark any threads it creates as daemon threads. The initial thread of an application--the one that begins at main()--is a non- daemon thread.

A Java application continues to execute (the virtual machine instance continues to live) as long as any non-daemon threads are still running. When all non-daemon threads of a Java application terminate, the virtual machine instance will exit. If permitted by the security manager, the application can also cause its own demise by invoking the exit() method of class Runtime or System.

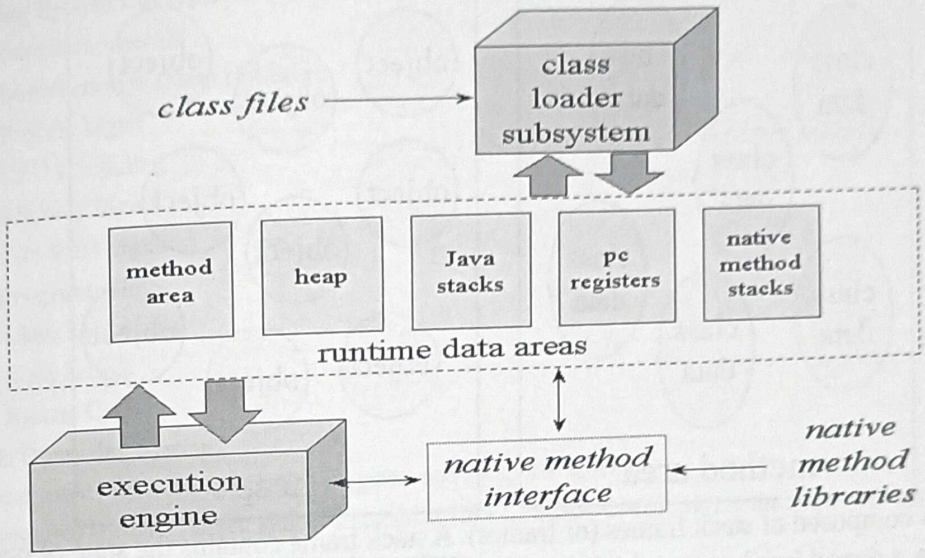
The Architecture of the Java Virtual Machine:

Block diagram of the Java virtual machine that includes the major

subsystems and memory areas described in the specification. Each Java virtual machine has a class loader subsystem: a mechanism for loading

types (classes and interfaces) given fully qualified names. Each Java virtual machine also has an execution engine: a mechanism responsible for executing the instructions contained in the methods of loaded classes.

When a Java virtual machine runs a program, it needs memory to store many things, including bytecodes and other information it extracts from loaded class files, objects the program instantiates, parameters to methods, return values, local variables, and intermediate results of computations. The Java virtual machine organizes the memory it needs to execute a program into several runtime data areas.

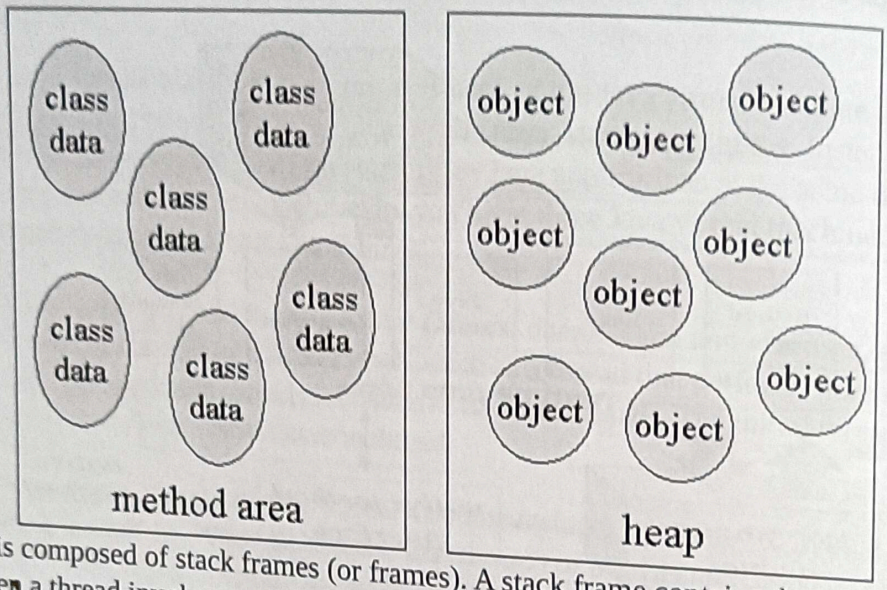


Although the same runtime data areas exist in some form in every Java virtual machine implementation, their specification is quite abstract. Many decisions about the structural details of the runtime data areas are left to the designers of individual implementations.

Different implementations of the virtual machine can have very different memory constraints. Some implementations may have a lot of memory in which to work, others may have very little. Some implementations may be able to take advantage of virtual memory, others may not. The abstract nature of the specification of the runtime data areas helps make it easier to implement the Java virtual machine on a wide variety of computers and devices.

Some runtime data areas are shared among all of an application's threads and others are unique to individual threads. Each instance of the Java virtual machine has one method area and one heap. These areas are shared by all threads running inside the virtual machine. When the virtual machine loads a class file, it parses information about a type from the binary data contained in the class file. It places this type information into the method area. As the program runs, the virtual machine places all objects the program instantiates onto the heap.

As each new thread comes into existence, it gets its own pc register (program counter) and Java stack. If the thread is executing a Java method (not a native method), the value of the pc register indicates the next instruction to execute. A thread's Java stack stores the state of Java (not native) method invocations for the thread. The state of a Java method invocation includes its local variables, the parameters with which it was invoked, its return value (if any), and intermediate calculations. The state of native method invocations is stored in an implementation-dependent way in native method stacks, as well as possibly in registers or other implementation-dependent memory areas.



The Java stack is composed of stack frames (or frames). A stack frame contains the state of one Java method invocation. When a thread invokes a method, the Java virtual machine pushes a new frame onto that thread's Java stack. When the method completes, the virtual machine pops and discards the frame for that method.

The Java virtual machine has no registers to hold intermediate data values. The instruction set uses the Java stack for storage of intermediate data values. This approach was taken by Java's designers to keep the Java virtual machine's instruction set compact and to facilitate implementation on architectures with few or irregular general purpose registers. In addition, the stack-based architecture of the Java virtual machine's instruction set facilitates the code optimization work done by just-in-time and dynamic compilers that operate at run-time in some virtual machine implementations.