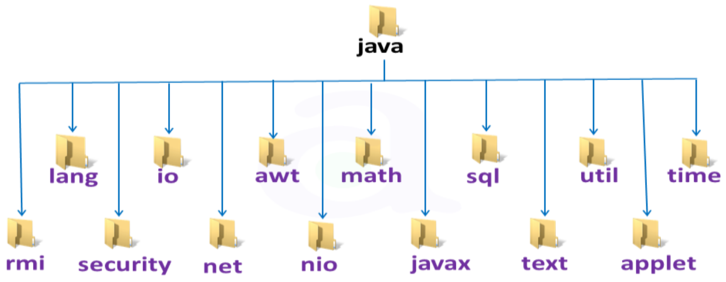
Java

1. **Overview**
2. **Java Basics**
3. **Java Oops concepts**
4. **Java Packages Detail**
5. **Java Advance concepts**

Java -Packages

1. **java.applet**:Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
2. **java.awt:**Contains all of the classes for creating user interfaces and for painting graphics and images.
3. **java.beans:** Contains classes related to developing beans -- components based on the JavaBeans™ architecture.
4. **java.io:** Provides for system input and output through data streams, serialization and the file system.
5. **java.lang:** Provides classes that are fundamental to the design of the Java programming language.
6. **java.math:** Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal).
7. **java.net:** Provides the classes for implementing networking applications.
8. **java.nio:** Defines buffers, which are containers for data, and provides an overview of the other NIO packages such as representing connections to entities that are capable of performing I/O operations, such as files and sockets; defines selectors, for multiplexed, non-blocking I/O operations.
9. **java.rmi:** Provides the RMI package.
10. **java.security:** Provides the classes and interfaces for the security framework.
11. **java.sql:** Provides the API for accessing and processing data stored in a data source (usually a relational database) using the JavaTM programming language.
12. **java.text:** Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages.
13. **java.time:** The main API for dates, times, instants, and durations.
14. **java.util:** Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).
15. **javax:**The javax prefix is used by the Java programming language for a package of standard Java extensions.It generally deals with servlets,and communication among ui and backendss.
16. **org:**The org package prefix is mostly used by non-profit organizations or for open source code, such as apache, w3c, etc,
    In-general **org** allows application developers to make use of security services like authentication, data integrity and data confidentiality from a variety of underlying security mechanisms like Kerberos, using a unified API.



1. Overview: (java developed by Sun Microsystems: 1995)

Java is a general-purpose programming language that is class-based, object-oriented, and designed to have few dependencies as possible. It is intended to let application developers **write once, run anywhere** meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of the underlying computer architecture.

Why java	Applications of java
• **Object Oriented –** In Java, everything is an Object. Java can be easily extended since it is based on the Object model. • **Platform Independent –** unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on. • **Simple –** Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master. • **Secure –** With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption. • **Architecture-neutral –** Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system. • **Portable –** Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset. • **Robust –** Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.	• **Multithreaded –** With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly. • **Interpreted –** Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process. • **High Performance –** with the use of Just-In-Time compilers, Java enables high performance. • **Distributed –** Java is designed for the distributed environment of the internet. • **Dynamic –** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

Java Architecture:

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

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

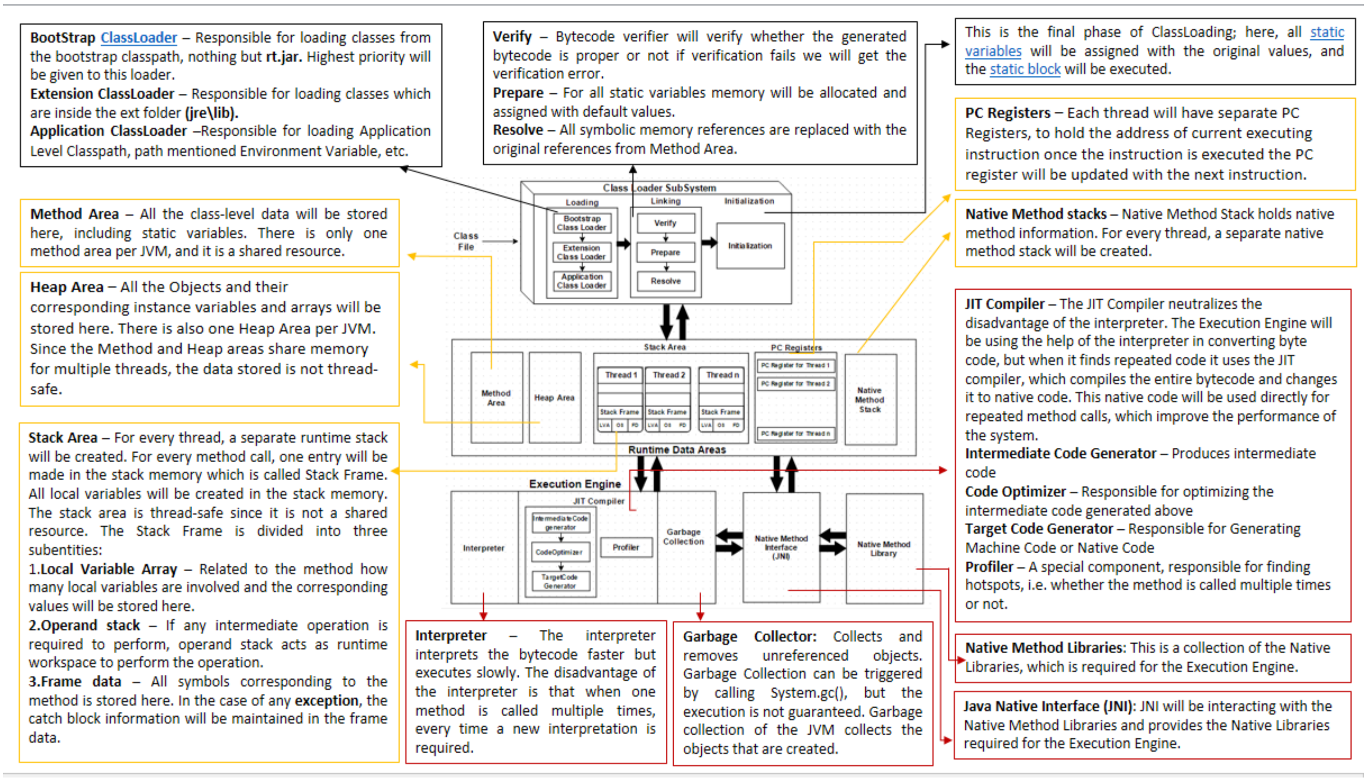
JRE – JRE is the implementation of **JVM** and is defined as a software package that provides Java class libraries, along with Java Virtual Machine (**JVM**), and other components to only run(not develop) applications written in Java programming.

JRE consists of the following components:

- **Deployment technologies**, including deployment, Java Web Start and Java Plug-in.
- **User interface toolkits,** including Abstract Window Toolkit (AWT), Swing, Java 2D, Accessibility, Image I/O, Print Service, Sound, drag and drop (DnD) and input methods.
- **Integration libraries**, including Interface Definition Language (IDL), Java Database Connectivity (JDBC), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), Remote Method Invocation Over Internet Inter-Orb Protocol (RMI-IIOP) and scripting.
- **Other base libraries,** including international support, input/output (I/O), extension mechanism, Beans, Java Management Extensions (JMX), Java Native Interface (JNI), Math, Networking, Override Mechanism, Security, Serialization and Java for XML Processing (XML JAXP).
- **Lang and util base libraries**, including lang and util, management, versioning, zip, instrument, reflection, Collections, Concurrency Utilities, Java Archive (JAR), Logging, Preferences API, Ref Objects and Regular Expressions.

JVM: Whatever Java program you run using JRE (.class files) goes into JVM and JVM is responsible for process executable class files

JVM Architecture



BootStrap ClassLoader – Responsible for loading classes from the bootstrap classpath, nothing but **rt.jar.** Highest priority will be given to this loader.
Extension ClassLoader – Responsible for loading classes which are inside the ext folder **(jre\lib).**
Application ClassLoader –Responsible for loading Application Level Classpath, path mentioned Environment Variable, etc.

Verify – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.
Prepare – For all static variables memory will be allocated and assigned with default values.
Resolve – All symbolic memory references are replaced with the original references from Method Area.

This is the final phase of ClassLoading; here, all static variables will be assigned with the original values, and the static block will be executed.

Method Area – All the class-level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.

Heap Area – All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe.

Stack Area – For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called Stack Frame. All local variables will be created in the stack memory. The stack area is thread-safe since it is not a shared resource. The Stack Frame is divided into three subentities:
1.Local Variable Array – Related to the method how many local variables are involved and the corresponding values will be stored here.
2.Operand stack – If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.
3.Frame data – All symbols corresponding to the method is stored here. In the case of any **exception,** the catch block information will be maintained in the frame data.

PC Registers – Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.

Native Method stacks – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

JIT Compiler – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.
Intermediate Code Generator – Produces intermediate code
Code Optimizer – Responsible for optimizing the intermediate code generated above
Target Code Generator – Responsible for Generating Machine Code or Native Code
Profiler – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.

Interpreter – The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.

Garbage Collector: Collects and removes unreferenced objects. Garbage Collection can be triggered by calling System.gc(), but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

Native Method Libraries: This is a collection of the Native Libraries, which is required for the Execution Engine.

Java Native Interface (JNI): JNI will be interacting with the Native Method Libraries and provides the Native Libraries required for the Execution Engine.

Runtime Data Access (JVM)(Stack Area maintain stacks of multiple threads):

The JVM allows an application to have multiple threads of execution running concurrently. In the Hotspot JVM, there is a direct mapping between a **Java Thread** and a **native operating system Thread**. After preparing all of the state for a Java thread such as thread-local storage, allocation buffers, synchronization objects, stacks and the program counter, the native thread is created. The operating system is therefore responsible for scheduling all threads and dispatching them to any available CPU. Once the native thread has initialized it invokes the run() method in the Java thread. When the run() method returns, uncaught exceptions are handled, then the native thread confirms if the JVM needs to be terminated as a result of the thread terminating (i.e. is it the last non-deamon thread).

JVM System Threads

If you use java console it is possible to see there are numerous threads running in the background. These background threads run in addition to the main thread, which is created as part of invoking public static void main(String[]).

JVM Threads	descriptions
VM thread	This thread waits for operations to complete that require the JVM to reach at a safe-point, It maintains garbage collectors, thread stack dump, thread suspension and locking revocation.
Periodic task thread	used to schedule execution of periodic operations
GC threads	These threads support the different types of garbage collection activities that occur in the JVM
Compiler threads	These threads compile byte code to native code at runtime Signal dispatcher thread -This thread receives signals sent to the JVM process and handle them inside the JVM by calling the appropriate JVM methods.

JVM Memory (divided into 5 **different** parts)

1. Heap 2.Stack. 3. Program Counter 4.Register 5.Native Method Stack.

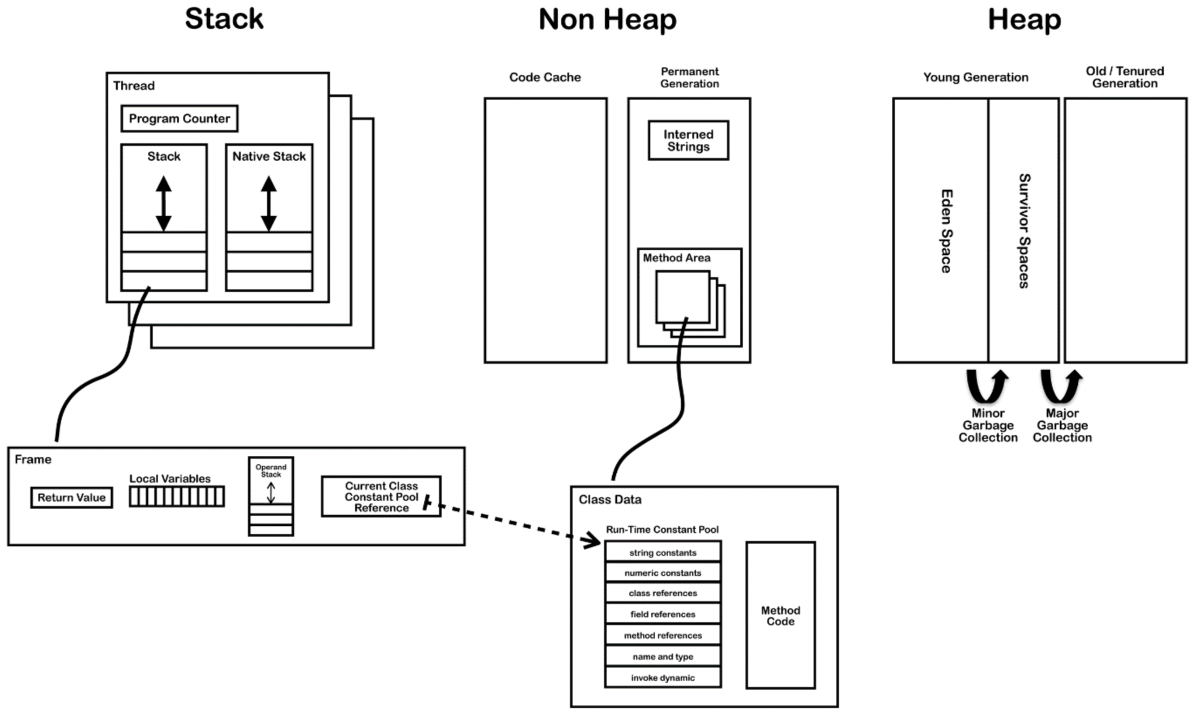
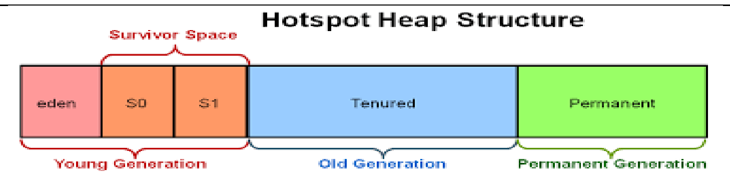


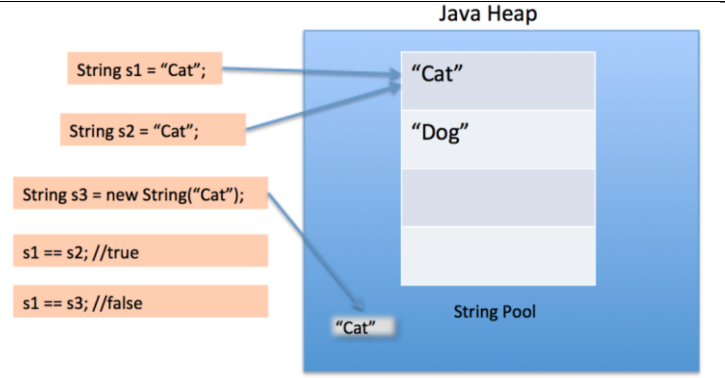
Fig-(JVM Memory)

Heap memory

It is **created** when JVM starts-up and all-type of memory is a part of shared memory which shared with threads)

Stack is used for static memory allocation and **Heap** for dynamic memory allocation, both stored **in the** computer's RAM .The Heap is used to allocate **class**, **instances** and **arrays** at runtime. Arrays and objects can never be stored on the stack because a frame is not designed to change in size after it has been created. The frame only stores references that point to objects or arrays on the heap. Unlike primitive variables and references in the local variable array (in each frame) objects are always stored on the heap so they are not removed when a method ends. Instead objects are only removed by the garbage collector.

Heap is divided into three sections(To support garbage collection the):	 Hotspot Heap Structure Survivor Space eden S0 S1 Tenured Permanent Young Generation Old Generation Permanent Generation
Young GenerationPermanent Generation) → Often split between Eden and Survivor	*New objects and arrays are created into the young generation *Minor garbage collection will operate in the young generation. Objects, that are still alive, will be moved from the eden space to the survivor space.
Old Generation (also called Tenured Generation)	Major garbage collection, which typically causes the application threads to pause, will move objects between generations. Objects, that are still alive, will be moved from the young generation to the old (tenured) generation.
Permanent Generation	**Permanent Generation** contains the application metadata required by the JVM to describe the classes and methods used in the application

JVM allocates some memory specially meant for **string literals**. This part of the heap memory is called **constant pool** (in general s**tring constants pool)**. Several types of data is stored in the constant pool including
• numeric literals
• string literals
• class references
• field references
• method references
When we use double quotes to create a String, it first looks for String with the same value in the String pool, if found it just returns the reference else it creates a new String in the pool and then returns the reference.
However using *new* operator, we force String class to create a new String object in heap space. We can use **intern())** method to put it into the pool or refer to another String object from the string pool having the same value.
	 Java Heap
	String s1 = "Cat";
	String s2 = "Cat";
	String s3 = new String("Cat");
	s1 == s2; //true
	s1 == s3; //false
	"Cat"
	"Dog"
	"Cat"
	String Pool
	Ex: String str = new String("Cat");
	In the above statement, either 1 or 2 string will be created. If there is already a string literal "Cat" in the pool, then only one string "str" will be created in the pool. If there is no string literal "Cat" in the pool, then it will be first created in the pool and then in the heap space, so a total of 2 string objects will be created

Heap dump generation:

During the peak load, an application may become slower and the **memory** consumption might be more. To troubleshoot the **memory** consumption, we require **Heap Dump**. If you have not specified the HeapDumpPath, then the JVM **generates** the file where the **JAVA** process is running.

Non-Heap Memory

Objects that are logically considered as part of the JVM mechanics are not created on the Heap.

Non heap contains	
1.Permanent Generation that contains	• the method area • interned strings
2.Code Cache used for compilation and storage of methods that have been compiled to native code by the JIT compiler	

2. Java Basics

Object: -

Object is an instance of class which holds state, behaviour and identity (identity maintained by jvm) of that class. **[Class Object** in java is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.]

(Java mainly deals with Objects, class, methods ,variable and interface)

Ways of Object instantiation of a class	Example.

1. **Using new** Operator	public class A **implements** Cloneable
	{
	String str="hello";
	public Object clone()**throws** CloneNotSupportedException{
	return super.clone();
	}
	public static void main(String args[])
	{
	A obj=new A(); //1
	A obj=A.**class**.newInstance(); //2
	Constructor<A> obj =A.**class**.getConstructor(); //3
	A obj = obj.clone(); //4
	}
	}

2. Using **Class.newInstance()** method [**java.lang.Class** package.]	
3. Using **newInstance()** method of constructor [**java.lang.reflect.Constructor**]	
4. Using **Object.clone()** method [java **clone()** method creates a copy of an existing object. Java.lang.**Object.clone**()]	

5.Using object **Serialization** and **Deserialization**

(The Object class does not implement Serializable interface because we may not want to serialize all the objects(jvm is using several classes internally which is not required to serialize) **,** and also for some security aspects, server connection info will cause security loophole **)**

→**Serialization** is a mechanism of converting the state of an object into a byte stream. so that we can transfer it over the network (using JPA and RMI etc).

→**Deserialization** is the reverse process where the byte stream is used to recreate the actual Java object in memory.

Cases of **Serialization** and **Deserialization**	**class** Demo **implements** Serializable
	{
	public int i;
	public String s;
	public transient String s2; //it will not serialize
	
	public Demo(**int** i, String s) //default constructor
	{
	this.i = i;
	this.s = s;
	}
	}
	public **class Test**
	{
	public static void main(String[] args) throws IOException
	{
	Demo **object** = new Demo(8, "Testing serialization");
	String filename = "Demofile.ser";
	/*-----------------Serialization----------*/
	
	FileOutputStream file = new FileOutputStream(filename);
	ObjectOutputStream out = new ObjectOutputStream(file);
	out.writeObject(object); //serialize object
	out.close(); //closes the ObjectOutputStream
	file.close(); //closes the file
	Demo **obj** = null;
	/*-----------------Deserialization--------*/
	FileInputStream file = new FileInputStream(filename);
	ObjectInputStream is = new ObjectInputStream(file);
	obj = (Demo)is.readObject(); //deserialize object
	is.close(); //closes the ObjectInputStream
	file.close(); //closes the file
	} }

1. If superclass is serializable then subclass is automatically serializable	
2.If a superclass is not serializable then subclass can still be serialized [At the time of serialization, if any instance variable is inheriting from non-serializable superclass, then JVM ignores original value of that instance variable and save default value to the file.] [At the time of de-serialization, if any non-serializable superclass is present, then JVM will execute instance control flow in the superclass. To execute instance control flow in a class, JVM will always invoke default(no-arg) constructor of that class. So every non-serializable superclass must necessarily contain default constructor, otherwise we will get runtime-exception]	
3.If the superclass is serializable but we don't want the subclass to be serialized [There is no direct way to prevent subclass from serialization in java. One possible way by which a programmer can achieve this is by implementing the *writeObject()* and *readObject()* methods in the subclass and needs to throw *NotSerializableException* from these methods.]	

class B extends A →class A is serializable
{
private void writeObject(ObjectOutputStream out) throws IOException
{
throw new NotSerializableException();
}
 	
private void readObject(ObjectInputStream in) throws IOException
{
throw new NotSerializableException();
}
}	

4.The Transient and Static Fields Do Not Get Serialized	
Note: Difference between Serialization and Externalization in Java. ... In **Externalization**, programmer will have complete control in **serializing** the object. **Serializable** is a marker interface so it doesn't contain any methods. Externalizable contain two methods i.e writeExternal() and readExternal().	

Class: It is a template that describes the behaviour and state of the object type.

Types of Classes	Example.	

1.Concreate class
[Any normal class which does not have any abstract method.**]**
2.Abstract class
[A class declared with abstract keyword and have zero or more abstract methods are known as abstract class.**]**
3.final class
[A class declared with the final keyword is a final class and it cannot be inherited by another class, ex- java.lang.**System**.**]**
4.static class
[Static classes are nested classes means a class declared within another class as a static member is called a static class.**]**
5. Anonymous inner class
An **inner class** declared without a **class** name is known as an **anonymous inner class**. In case of **anonymous inner classes**, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a **class** or an interface
	// -------1 concreate class**------------**
	class BaseClass {
	void Display() {
	System.out.print("test.");
	}
	}
	// ------------2 Abstract class**---------**
	abstract class **Animal {**
	public abstract void sound();
	}
	//-----------3 final class**-----------**
	final class **BaseClass {**
	void Display() {
	System.out.print("test.");
	}
	}
	// ----------------4 static inner class------------
	class A {
	static int s; // static variable
	static void met(int a, int b) { // static method
	s = a + b;
	}
	static class NestedClass { // static class
	static { // static block
	
	System.out.println("static block ");
	}
	}
	}
		//---------5. Anonymous inner class----
		→It can be achieve either by interface or abstract method.
		
		interface Age
		{
		int x = 21;
		void getAge();
		}
		
		class AnonymousDemo
		{
		public static void main(String[] args)
		{
		Age oj1 = new Age() {
		@Override
		public void getAge() {
		// printing age
		System.out.print("Age is "+x);
		}
		};
		oj1.getAge();
		}
		}

Static vs final		
BASIS FOR COMPARISON	**static keyword**	**final keyword**
Applicable	Static keyword is applicable to variables, methods , block and nested static class,	Final keyword is applicable to variables, methods and class.
Initialization	It is not compulsory to initialize the static variable at the time of its declaration.	It is compulsory to initialize the final variable at the time of its declaration.
Modification	The static variable can be reinitialized.	The final variable can not be reinitialized.
Methods	• Static methods cannot be overridden but can be inherited because they are not dispatched on the object instance at runtime • A static method can only call other static methods only. • A static method can access static data only. • A static method can not be referred to “this” or “super” in any conditions. • A static method can be accessed with the class name.	Final methods cannot be Overridden by subclass.
Class	• Static class can-not be inherited. • Java does have the concept of nested static class. The outermost class can not be made static whereas the innermost class can be made static. • A static nested class can not access the non-static member of the outer class. • It can only access the static members of the outer class.	A final class can not be inherited by any class.
Block	Static block is used to initialize the static variables.	Final keyword supports no such block.

Method: It is a behaviour of class in which all logical or functional action will be taken.

Types of Methods	Example.

1.Instance Method :- Instance method are methods which require an object of its class to be created before it can be called.	**class A**{
	public void **test**(){ →**instance method**
	s.o.p(‘calling instance method test’);
	}
	public **static** void test_static(){ →**static method**
	s.o.p(‘ static method with return type void’);
	}
	public **static A** test_Factory(){ →**static factory method**
	return new A();
	}
	}
	class B{
	B(){ →**default empty constructor of class B**
	A a1=new A();
	a1.test();
	A.test_static(); // or a1.test_static()
	a1=A.test_Factory(); // or a1.test_Factory()
	}
	}

2.Static Method :- • **Static methods** are the methods in Java that can be called without creating an object of class. They are referenced by the class name itself or reference to the Object of that class. • **Factory Method :-** A class can provide a public **static factory method**, which is simply a **static method** that returns an instance of the class.	

Variable:

It is a state of a class**.**

Types of variables	Example.

1. Local Variables
[A variable defined within a block or method or constructor of a class.]
	class A{
	int i; →**instance variable**
	static int j; **->static variable**
	static int b(){
	int x=10; → **local variable**
	i=10; // cannot be found ,
	A a= new A();
	a.i=10; // can be found in a's instance.
	A.j=20; // static variable call via class reference
	}
	}

2. Instance Variables **[**Instance variables are declared in a class, but outside a method, constructor or any block. Instance variables can be accessed directly by calling the variable name inside the class.or via fully qualified name ObjectReference.VariableName in static block.]	
3. Static Variables [Class variables also known as **static variables** are declared with the static keyword in a class, but outside a method, constructor or a block.]	

Interface:

An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface. It is used to achieve the abstraction in java.

Important points about interface

- We can’t create instance of interface but we can make reference of it that refers to the Object of its implementing class.
- A class can implement more than one interface.
- An interface can extends another interface or interfaces (more than one interface)
- A class that implements interface must implements all the methods in interface.
- All Interface Method can have either “abstract”, “static” or “default” modifier. Interface methods can have a body if static or default modifier is used against the method. And all the fields are public, static, and final.
- It is used to achieve multiple inheritance.
- It is used to achieve loose coupling.
- *After Java 9 changes, Interfaces look a lot like Abstract Class, still, there are some differences.
  - An abstract class can have variables with different modifiers which is not constant
  - Methods in Abstract can have the different signature than just private or public

```
public interface Test1{
        void method();                  // or public abstract void method
        public abstract void method1();
        public default void method2() {
                        // body
        }
        public static void method3() {  // private or private static methods added in java 9 calling as Test1.method3()
                       //body
        }
}

Public class Test2 implements Test1{

          Public void method(){}
          Public void method1(){}   // methods implementation

        Public static void main(String[] args){
         Test2 t2=new Test2();    // creation of class object and calling implemented method
          t2.method();
        or
        Test1 t1=new Test2();     // reference of class  to interface; (it is generally used for loose coupling)
        t1.method();

        }
}
```

Types of interfaces

1.Marker Interface : A Marker interface is an interface without any methods or fields declaration, means it is an empty interface.

```
Public interface Tesa{}      //ex. public interface Serializable, public interface Cloneable
```

2.functional Interface : A functional interface is an interface hava only one method. Lambda expression works on functional interfaces to replace anonymous classes.(Ex:- Runnable ,callable)

@FunctionalInterface → **annotation used in java 8 to test it is functional or not.**

```
Public interface test{
 Void test();
}
```

3.Nested Interface : if an interface resides inside of interface or class is called nested interface

```
class Test
{
   protected interface Yes
   {
      void show();
   }
}

class Testing implements Test.Yes
{
   public void show()
   {
      System.out.println(" test interface");
   }
}
```

```
interface Test
{
   protected interface Yes
   {
      void show();
   }
}

class Testing implements Test.Yes
{
   public void show()
   {
      System.out.println("test  interface");
   }
}
```

3.Java OOPS

OOPs is a programming paradigm based on the concept of “objects” that contain data and methods. The primary purpose of object-oriented programming is to increase the flexibility and maintainability of programs.

OOPS featues	
Abstraction	Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user.
Encapsulation	Encapsulation simply means binding object state(fields) and behaviour(methods) together. If you are creating class, you are doing encapsulation. Encapsulated class never share private member from outside.
Inheritance	The process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order. Or It is a deal with other class to populate their features based on certain rules of inheritance.
Polymorphism	Polymorphism is one of the OOPs feature that allows us to perform a single action in different ways. Type of polymorphism

- **static or compile time polymorphism** (Method Overloading)

  Whenever an object is bound with their functionality at the compile-time, this is known as the compile-time polymorphism. At compile-time, java knows which method to call by checking the method signatures. So this is called compile-time polymorphism or static or early binding. Compile-time polymorphism is achieved through method overloading.

```
public class Test{
   public static int add(int a, int b)       // we can take non static method
   {
      return a + b;
   }
   public static double add(
      double a, double b)
   {
      return a + b;
   }
   public static void main(String args[])
   {
      System.out.println(add(2, 3));       // 5
      System.out.println(add(2.0, 3.0));  // 5.0
   }
}
```

- **Runtime polymorphism** (Method Overriding)

  Whenever an object is bound with the functionality at run time, this is known as runtime polymorphism. The runtime polymorphism can be achieved by method overriding. Java virtual machine determines the proper method to call at the runtime, not at the compile time. It is also called dynamic or late binding. Method overriding says child class has the same method as declared in the parent class. It means if child class provides the specific implementation of the method that has been provided by one of its parent class then it is known as method overriding.

```
class Test {
    public void method()
    {
        System.out.println("Method 1");
    }
}
```

```
public class GFG extends Test {

    // Overriding the parent method
    public void method()
    {
        System.out.println("Method 2");
    }
    public static void main(String args[])
    {
        Test test = new GFG();   // parent class can take reference of child class
        test.method();               // Method 2
    }
}
```

Rules of Overriding

1.Overriding and Access-Modifiers

The access modifier for an overriding method can allow more, but not less .A protected instance method in the super-class can be made public but not private.

2.Final methods can not be overridden

→It will throw a compile time error

3. Static methods can not be overridden(Method Overriding vs Method Hiding)

When you define a static method with same signature as a static method in base class, it is known as method hiding.

As per overriding rules parent which holding the reference to child should call   Child static overridden method. Since static method can not be overridden, it calls Parent's static method.

4. Private methods can not be overridden

5. The overriding method must have same return type (or subtype)

From Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child's return type should be sub-type of parent's return type. This phenomena is known as **covariant return type**.

6. Invoking overridden method of parent from sub-class → super.methodName();

7. We can't Override a constructor due to same name as class name and having no return type.

8. Overriding and Exception-Handling

- **Rule#1 :** If the super-class overridden method does not throw any exception, **then** subclass overriding method can only throws the unchecked exception, throwing checked exception will lead to compile-time error.

```
class Parent {
  void m1(){
       System.out.println("From parent m1()");
  }

}

class Child extends Parent {
    // no issue while throwing unchecked exception
    @Override
    void m1() throws ArithmetiException
    {
        System.out.println("From child m1()");
    }
// compile time error
    @Override
    void m1() throws Exception
    {
        System.out.println("From child1 m1()");
    }
}
```

```
class Parent {
    // 1  access modifier can have heigher visibility
    protected void m2()
    {
        System.out.println("From parent m2()");
    }

   //3  it will be hidden in subclass
    static void m3()
    {
        System.out.println("Prent static m3()");
    }
    //4  private methods are not overridden
    private void m1()
    {
        System.out.println("From parent m1()");
    }
   //5 coveriant return type
    Parent show(){
    return this();  // it will return parent class obj
   }

  //6 invoking from child
  void test(){
    System.out.println("parent method")
  }

}
------------------------child-------------

class Child extends Parent {
 //1. m2() is more accessible protected→public
    @Override → Annotation to check super class has this method or not
    public void m2()
    {
        System.out.println("From child m2()");
    }

//3. if parent have a reference to child , then parent will call //their own static m3() if it is overridden in child
static void m3()
    {
        System.out.println("child static m1()");
    }

//4. private  m1() method will be unique to Child class
    private void m3()
    {
        System.out.println("From child m3()");
    }

//5 coveriant return type
Child show(){
return this();   // it will return child class obj
}

 //6 invoking from child
  void test(){
    super.test();
    System.out.println("child method")
  }

}
```

- **Rule#2 :** If the super-class overridden method does throws an exception, subclass overriding method can only throw same, subclass exception.

```
class Parent {
   void m1() throws RuntimeException{
         System.out.println("From parent m1()");
   }
}

class Child extends Parent {
    // no issue while throwing same exception
    @Override
    void m1() throws RuntimeException
    {
       System.out.println("From child m1()");
    }
// compile time error
    @Override
    void m1() throws Exception
    {
        System.out.println("From child1 m1()");
    }
}
```

```
class Main {
    public static void main(String[] args)
    {
        Parent obj1 = new Parent();
        obj1.m2();
        Parent obj2 = new Child();
        obj2.m2();
        obj2.show();  // it will return child obj;
    }
}
```

4.Java package

Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Package names and directory structure are closely related. Path of parent directory is present in CLASSPATH so we can easily able to access classes through parents CLASSPATH. Packages are used for:

- Preventing naming conflicts. For example there can be two classes having same name in different packages.
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier.
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

Packages are of two type→ **user defined** (as per the requirement of project) and **Built-in packages** (default packages in java)

Built-in Packages

These packages consist of a large number of classes which are a part of Java API.Some of the commonly used built-in packages are:
1) **java.io:** Contains classed for supporting input / output operations etc.
2) **java.lang:** Contains language support classes(e.g classed which defines primitive data types, math operations).
3) **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.

Package java.lang

Provides classes that are fundamental to the design of the Java programming language.

Understand below Hierarchy (ex :)
Integer extends (**Number** extends **Object**)→ integer extends Number->Number extends Object
[Hence **Integer class** holds **Number** and **Object** class properties]

Class Hierarchy

- java.lang.**Object**
    - java.lang.**Boolean** (implements java.lang.Comparable<T>, java.io.Serializable)
    - java.lang.**Character** (implements java.lang.Comparable<T>, java.io.Serializable)
    - java.lang.**Character.Subset**
        - java.lang.**Character.UnicodeBlock**
    - java.lang.**Class**<T> (implements java.lang.reflect.AnnotatedElement, java.lang.reflect.GenericDeclaration, java.io.Serializable, java.lang.reflect.Type)
    - java.lang.**ClassLoader**
    - java.lang.**ClassValue**<T>
    - java.lang.**Compiler**
    - java.lang.**Enum**<E> (implements java.lang.Comparable<T>, java.io.Serializable)
    - java.lang.**Math**
    - java.lang.**Number** (implements java.io.Serializable)
        - java.lang.**Byte** (implements java.lang.Comparable<T>)
        - java.lang.**Double** (implements java.lang.Comparable<T>)
        - java.lang.**Float** (implements java.lang.Comparable<T>)
        - java.lang.**Integer** (implements java.lang.Comparable<T>)
        - java.lang.**Long** (implements java.lang.Comparable<T>)
        - java.lang.**Short** (implements java.lang.Comparable<T>)
    - java.lang.**Package** (implements java.lang.reflect.AnnotatedElement)
    - java.security.**Permission** (implements java.security.Guard, java.io.Serializable)
        - java.security.**BasicPermission** (implements java.io.Serializable)
            - java.lang.**RuntimePermission**
    - java.lang.**Process**
    - java.lang.**ProcessBuilder**
    - java.lang.**ProcessBuilder.Redirect**
    - java.lang.**Runtime**
    - java.lang.**SecurityManager**
    - java.lang.**StackTraceElement** (implements java.io.Serializable)
    - java.lang.**StrictMath**
    - java.lang.**String** (implements java.lang.CharSequence, java.lang.Comparable<T>, java.io.Serializable)
    - java.lang.**StringBuffer** (implements java.lang.CharSequence, java.io.Serializable)

- java.lang.**StringBuilder** (implements java.lang.CharSequence, java.io.Serializable)
- java.lang.**System**
- java.lang.**Thread** (implements java.lang.Runnable)
- java.lang.**ThreadGroup** (implements java.lang.Thread.UncaughtExceptionHandler)
- java.lang.**ThreadLocal**<T>
  - java.lang.**InheritableThreadLocal**<T>
- java.lang.**Throwable** (implements java.io.Serializable)
  - java.lang.**Error**
    - java.lang.**AssertionError**
    - java.lang.**LinkageError**
      - java.lang.**BootstrapMethodError**
      - java.lang.**ClassCircularityError**
      - java.lang.**ClassFormatError**
        - java.lang.**UnsupportedClassVersionError**
      - java.lang.**ExceptionInInitializerError**
      - java.lang.**IncompatibleClassChangeError**
        - java.lang.**AbstractMethodError**
        - java.lang.**IllegalAccessError**
        - java.lang.**InstantiationError**
        - java.lang.**NoSuchFieldError**
        - java.lang.**NoSuchMethodError**
      - java.lang.**NoClassDefFoundError**
      - java.lang.**UnsatisfiedLinkError**
      - java.lang.**VerifyError**
    - java.lang.**ThreadDeath**
    - java.lang.**VirtualMachineError**
      - java.lang.**InternalError**
      - java.lang.**OutOfMemoryError**
      - java.lang.**StackOverflowError**
      - java.lang.**UnknownError**
  - java.lang.**Exception**
    - java.lang.**CloneNotSupportedException**
    - java.lang.**InterruptedException**
    - java.lang.**ReflectiveOperationException**
      - java.lang.**ClassNotFoundException**
      - java.lang.**IllegalAccessException**
      - java.lang.**InstantiationException**
      - java.lang.**NoSuchFieldException**
      - java.lang.**NoSuchMethodException**
    - java.lang.**RuntimeException**
      - java.lang.**ArithmeticException**
      - java.lang.**ArrayStoreException**
      - java.lang.**ClassCastException**
      - java.lang.**EnumConstantNotPresentException**
      - java.lang.**IllegalArgumentException**
        - java.lang.**IllegalThreadStateException**
        - java.lang.**NumberFormatException**
      - java.lang.**IllegalMonitorStateException**
      - java.lang.**IllegalStateException**
      - java.lang.**IndexOutOfBoundsException**
        - java.lang.**ArrayIndexOutOfBoundsException**
        - java.lang.**StringIndexOutOfBoundsException**
      - java.lang.**NegativeArraySizeException**
      - java.lang.**NullPointerException**
      - java.lang.**SecurityException**
      - java.lang.**TypeNotPresentException**
      - java.lang.**UnsupportedOperationException**
- java.lang.**Void**

Interface Hierarchy

- java.lang.**Appendable**
- java.lang.**AutoCloseable**
- java.lang.**CharSequence**
- java.lang.**Cloneable**
- java.lang.**Comparable**<T>
- java.lang.**Iterable**<T>
- java.lang.**Readable**
- java.lang.**Runnable**
- java.lang.**Thread.UncaughtExceptionHandler** →Thread class contains inner interface [static interface Thread.UncaughtExceptionHandler]

Annotation Type Hierarchy

- java.lang.**SuppressWarnings** (implements java.lang.annotation.Annotation)
- java.lang.**Override** (implements java.lang.annotation.Annotation) -->
- java.lang.**SafeVarargs** (implements java.lang.annotation.Annotation)
- java.lang.**Deprecated** (implements java.lang.annotation.Annotation)
- java.lang.**FunctionalInterface** (implements java.lang.annotation.Annotation)

Enum Hierarchy

- java.lang.**Object**
  - java.lang.**Enum**<E> (implements java.lang.Comparable<T>, java.io.Serializable)
    - java.lang.**Thread.State**
    - java.lang.**ProcessBuilder.Redirect.Type**
    - java.lang.**Character.UnicodeScript**

java.lang.Object

Class Object is the root of the class hierarchy. Every class has Object as a superclass.

Methods of Object class	
protected Object clone()	Creates and returns a copy of this object.
protected void finalize()	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?> getClass()	Returns the runtime class of this Object.
int hashCode()	Returns a hash code value for the object.
boolean equals(Object obj)	Indicates whether some other object is "equal to" this one.
String toString()	Returns a string representation of the object.
void wait()	Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void wait(long timeout)	Wait for specified time
void wait(long timeout, int nanos)	Wait for specified time
void notify()	Wakes up a single thread that is waiting on this object's monitor.
void notifyAll()	Wakes up all threads that are waiting on this object's monitor.

java.lang.Process: →it is an abstract class implemented by ProcessBuilder class

The **ProcessBuilder**.**start()** and Runtime.exec methods create a native process and return an instance of a subclass of Process that can be used to control the process and obtain information about it. The class Process provides methods for performing input from the process, performing output to the process, waiting for the process to complete, checking the exit status of the process, and destroying (killing) the process.
The methods that create processes may not work well for special processes on certain native platforms, such as native windowing processes, daemon processes, Win16/DOS processes on Microsoft Windows, or shell scripts.

By default, the created subprocess does not have its own terminal or console. All its standard I/O (i.e. stdin, stdout, stderr) operations will be redirected to the parent process, where they can be accessed via the streams obtained using the methods **getOutputStream(), getInputStream(), and getErrorStream()**. The parent process uses these streams to feed input to and get output from the subprocess. Because some native platforms only provide limited buffer size for standard input and output streams, failure to promptly write the input stream or read the output stream of the subprocess may cause the subprocess to block, or even deadlock.
The ProcessBuilder class manages several operations related to start process, redirect process, redirectErrorStream and so on.And hence contains several method related to operations required.

Static import in java

In Java, static import concept is introduced in 1.5 version. With the help of static import, we can access the static members of a class directly without class name or any object.

**class A{
public static void main(String[] args)
{
System.out.println(Math.sqrt(4));
System.out.println(Math.pow(2, 2));
System.out.println(Math.abs(6.3));
}
}**
	**import static java.lang.Math.*;
	class A{
	public static void main(String[] args)
	{
	System.out.println(sqrt(4));
	System.out.println(pow(2, 2));
	System.out.println(abs(6.3));
	}
	}**

String vs StringBuffer vs String Builder

- Objects of String are immutable, and objects of StringBuffer and StringBuilder are mutable.
- StringBuffer and StringBuilder are similar, but StringBuilder is faster and preferred over StringBuffer for single threaded program. If thread safety is needed, then StringBuffer is used.

String,StringBuffer,StringBuilder method →do Yourself

String Constructor

String()	String str = new String();
String("string literal");	String str = new String("FlowerBrackets");
String(String original)	
String(StringBuffer buffer)	
String(StringBuilder builder)	

String(byte[] byte_arr, Charset char_set)	byte[] bArr = {99, 114, 108, 102, 102};
	Charset ch = Charset.defaultCharset();
	String str = new String(bArr, ch);
	System.out.println(str); // crlff

String(byte[] byte_arr, String char_set_name)	byte[] bArr = {99, 114, 108, 102, 102};
	String str = new String(bArr, "US-ASCII");
	System.out.println(str); // crlff

String(byte arr[], int start_index, int length)	byte[] arr = {99, 114, 108, 102, 102};
	String str = new String(arr, 1, 3);
	System.out.println(str); // rlf

String(byte[] byte_arr, int start_index, int length, Charset char_set)	byte[] bArr = {99, 114, 108, 102, 102};
	Charset ch = Charset.defaultCharset();
	String str = new String(bArr, 1, 3, ch); //rlf

String(byte[] byteArr, int startIndex, int length, String charsetName)	byte bArr[] = {99, 114, 108, 102, 102}; String str = new String(bArr, 1, 4, "US-ASCII"); / /rlff
String(char ch[], int start, int count)	char ch[] = {'h', 'e', 'l', 'l', 'o'}; String str = new String(ch, 1, 3); //ell
String(int[] codePoints, int offset, int count)	int uniCode[] = {99, 114, 108, 102, 102}; String str = new String(uniCode, 1, 3); // rlf

StringBuffer constructor	**StringBuilder constructor**	
StringBuffer()	**StringBuilder()**	Default buffer size 16

StringBuffer(CharSequence seq)	**StringBuilder (CharSequence seq)**	char ch[] = {'h', 'e', 'l', 'l', 'o'};
		StringBuffer str = new StringBuffer (ch);

StringBuffer(int capacity)	**StringBuilder (int capacity)**	Constructs a string buffer with no characters in it and the specified initial capacity.
StringBuffer(String str)	**StringBuilder (String str)**	new StringBuffer(new String("string Buffer"))

Thread, ThreadGroup, ThreadLocal

A thread is a light-weight smallest part of a process that can run concurrently with the other parts(other threads) of the same process. Threads are independent because they all have separate path of execution that's the reason if an exception occurs in one thread, it doesn't affect the execution of other threads. All threads of a process share the common memory. **The process of executing multiple threads simultaneously is known as multithreading.** Let's summarize the discussion in points:

1. The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to maximum utilize the CPU time. A multithreaded program contains two or more parts that can run concurrently. Each such part of a program called thread.

2. Threads are lightweight sub-processes, they share the common memory space. In Multithreaded environment, programs that are benefited from multithreading, utilize the maximum CPU time so that the idle time can be kept to minimum.

3. A thread can be in one of the following states:

NEW – A thread that has not yet started is in this state.
RUNNABLE – A thread executing in the Java virtual machine is in this state.
BLOCKED – A thread that is blocked waiting for a monitor lock is in this state.
WAITING – A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
TIMED_WAITING – A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
TERMINATED – A thread that has exited is in this state.

A thread can be in only one state at a given point in time.

There are two ways to create thread in java;

- Implement the Runnable interface (java.lang.Runnable) [Preferred →Extending the Thread class means that the subclass cannot extend any other class, whereas a class implementing the Runnable interface has this option.]
- By Extending the Thread class (java.lang.Thread) [We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.]

Thread.Start()→fist it will register thread in scheduler→it will call run method->start execution.
→If we will not override run method then it will call default empty run method of thread class.

Initializing thread

Extending Thread	**Implementing Runnable**

public class **MyThread** extend **Thread** {
@Override
public void **run**(){
System.out.println(Thread.currentthread().getname());
}
}
	public class **MyThread** implements **Runnable**{
	@Override
	public void **run**(){
	System.out.println(Thread.currentthread().getname());
	}
	}

Public class **Test**{
Public static void main(){
MyThread myThread=new MyThread();
myThread.start(); →1.new Thread Created
}
}
	Public class **Test**{
	Public static void main(){
	MyThread myThread=new MyThread();
	Thread t=new Thread(myThread);
	t.start(); →3.new Thread Created
	}
	}

Public class **Test**{
Public static void main(){
Thread myThread=new MyThread();
myThread.start(); →2.new Thread Created
}
}
	Public class **Test**{
	Public static void main(){
	Runnable myThread=new MyThread();
	Thread t=new Thread(myThread);
	t.start(); →4.new Thread Created
	}
	}

User **Annonimous inner**(**lambda**) class when you want to execute thread separately without any class.	

Thread thread=new Thread(new Runnable(){
@Override
public void run(){
System.out.println(Thread.currentthread().getname());
}
});
thread.start();→5 new Thread Created
	Thread thread=new Thread(()=>System.out.println(
	Thread.currentthread().getname())
	);
	thread.start();→6 new Thread Created

Common thread methods

static int	activeCount()	Returns an estimate of the number of active threads in the current thread's thread group and its subgroups.
Void	checkAccess()	Determines if the currently running thread has permission to modify this thread.
protected Object	clone()	Throws CloneNotSupportedException as a Thread can not be meaningfully cloned.
static Thread	currentThread()	Returns a reference to the currently executing thread object.
static void	dumpStack()	Prints a stack trace of the current thread to the standard error stream.
static int	enumerate(Thread[] tarray)	Copies into the specified array every active thread in the current thread's thread group and its subgroups.
static Map<Thread,StackTraceElement[]>	getAllStackTraces()	Returns a map of stack traces for all live threads.
ClassLoader	getContextClassLoader( )	Returns the context ClassLoader for this Thread.
static Thread.UncaughtExceptionHandler	getDefaultUncaughtExceptionHandler()	Returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.
long	getId()	Returns the identifier of this Thread.
String	getName()	Returns this thread's name.
int	getPriority()	Returns this thread's priority.
StackTraceElement[]	getStackTrace()	Returns an array of stack trace elements representing the stack dump of this thread.
Thread.State	getState()	Returns the state of this thread.
ThreadGroup	getThreadGroup()	Returns the thread group to which this thread belongs.
Thread.UncaughtExceptionHandler	getUncaughtExceptionHandler()	Returns the handler invoked when this thread abruptly terminates due to an uncaught exception.
static boolean	holdsLock(Object obj)	Returns true if and only if the current thread holds the monitor lock on the specified object.
void	interrupt()	Interrupts this thread.
static boolean	interrupted()	Tests whether the current thread has been interrupted.
boolean	isAlive()	Tests if this thread is alive.
boolean	isDaemon()	Tests if this thread is a daemon thread.
boolean	isInterrupted()	Tests whether this thread has been interrupted.
void	join()	Waits until other thread get completed.
void	join(long millis)	Waits at most millis milliseconds for this thread to die.
void	join(long millis, int nanos)	Waits at most millis milliseconds plus nanos nanoseconds for this thread to die.
void	run()	If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns.
void	setContext ClassLoader(ClassLoader cl)	Sets the context ClassLoader for this Thread.
void	setDaemon(boolean on)	Marks this thread as either a daemon thread or a user thread.
static void	setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)	Set the default handler invoked when a thread abruptly terminates due to an uncaught exception, and no other handler has been defined for that thread.
void	setName(String name)	Changes the name of this thread to be equal to the argument name.
void	setPriority(int newPriority)	Changes the priority of this thread.
void	setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)	Set the handler invoked when this thread abruptly terminates due to an uncaught exception.
static void	sleep(long millis)	Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
static void	sleep(long millis, int nanos)	Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds plus the specified number of nanoseconds, subject to the precision and accuracy of system timers and schedulers.
void	start()	Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

String	**toString()**	**Returns a string representation of this thread, including the thread's name, priority, and thread group.**
static void	**yield()**	**Pause execution and give cpu to other high priority thread, if priority of other thread will be lower than current thread, then paused thread will continue. if thread have same priority jvm randomly select which thread will execute.**
void	**stop(Throwable obj)**	**Deprecated. (causes dead-lock)**
void	**suspend()**	**Deprecated. (causes dead-lock)**
void	**resume()**	**Deprecated. (causes dead-lock)**
void	**destroy()**	**Deprecated. (causes dead-lock)**
int	**countStackFrames()**	**Deprecated. (causes dead-lock)**
void	**stop()**	**Deprecated. (causes dead-lock)**

Constructors

1	Thread()
2	Thread(**String** name)
3	Thread(**Runnable** target)
4	Thread(**Runnable** target,**String** name)
5	Thread(**ThreadGroup** group, **Runnable** target)
6	Thread(**ThreadGroup** group, **Runnable** target,**String** name)
7	Thread(**ThreadGroup** group, **Runnable** target, **String** name, **long** stackSize)
8	Thread(**ThreadGroup** group, **String** name)

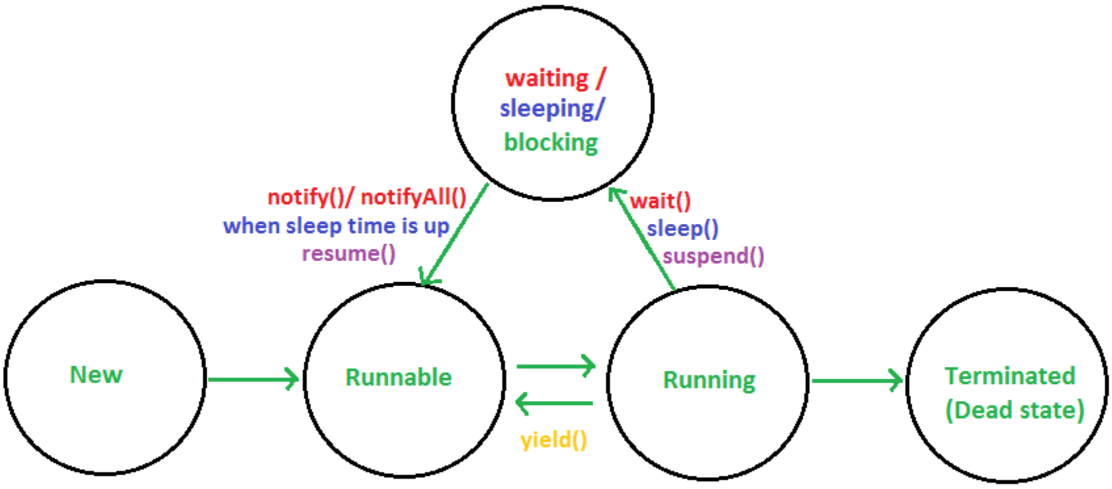


Fig. THREAD STATES

join() example	**yield() example**

class MyThread extends Thread{
 public void run(){
    for(;;){
       try(){
       Thread.sleep(2000);→sleep
       }catch(InteruptedException){
        // when main thread interrupt child
       }
    }
 }}
-----------------------Test-------------
class Test{
  public static void main(String args){
    MyThread mythread=new MyThread();
    t.start();
    t.join(); →main thread wait till child complete
  // t.interrupt() →main thread interrupt child
 }
}
	class MyThread extends Thread{
	 public void run(){
	   for(;;){
	     Thread.yield(); →give cpu to high priority thread.
	   }
	 }
	}
	-------------Test------
	class Test{
	  public static void main(String args){
	    MyThread mythread=new MyThread();
	    t.start();
	  }
	}

```
class MyThread extends Thread{
Static Thread mt;
 public void run(){
            for(;;){
             try(){
                  →wait for completing main thread
                   mt.join(); →main thread referance
                }catch(InteruptedException){}
            }
 }
}
```

```
-----------------------Test-------------
class Test{
  public static void main(String args){
            MyThread.mt=Thread.currentThread();
            MyThread mythread=new MyThread();
            t.start();
  }
}
```

Note:if both thread have join method waiting for each other dead-lock will occurre.
Thread.currentThread().join→Deadloack

Java concurrency :

Our systems can do more than one thing at a time. They assume that they can continue to work in a word processor, while other applications download files, manage the print queue, and stream audio. Even a single application is often expected to do more than one thing at a time. For example, that streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display. Software that can do such things is known as ***concurrent*** software.

The basics of Java concurrency:

- **Creating and Starting Java Threads**
- **Race Conditions and Critical Sections**
- **Thread Safety and Shared Resources**
- **Thread Safety and Immutability**
- **Java Memory Model**
- **Java Synchronized Blocks**
- **Java Volatile Keyword**
- **Java ThreadLocal**
- **Java Thread Signaling**

Typical problems in Java concurrency:

- **Deadlock**
- **Deadlock Prevention**
- **Starvation and Fairness**
- **Nested Monitor Lockout**
- **Slipped Conditions**

Java concurrency constructs that help against the issues above:

- **Locks in Java**
- **Read / Write Locks in Java**
- **Reentrance Lockout**
- **Semaphores**
- **Blocking Queues**
- **Thread Pools**
- **Compare and Swap**

Java Concurrency Utilities (java.util.concurrent):

- **Java Concurrency Utilities - java.util.concurrent**

1.The basics of Java concurrency:

Race Conditions and Critical Sections

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data.

Critical section

When more than one threads access a same code segment that segment is known as critical section. Critical section contains shared variables or resources which are needed to be synchronized to maintain consistency of data variable.

synchronized Method (block whole method)	Synchronized block (block some part)

public class TwoSums {
 	
    private int sum1 = 0;
    private int sum2 = 0;
 	
    public synchronized void add(int val1, int val2){
            this.sum1 += val1;
            this.sum2 += val2;
    }
}
	public class TwoSums {
	
	    private int sum1 = 0;
	    private int sum2 = 0;
	
	    public void add(int val1, int val2){
	      // ……….
	       synchronized(this){
	           this.sum1 += val1;
	           this.sum2 += val2;
	       }
	     // ………
	    }
	}

Thread Safety and Shared Resources

Code that is safe to call by multiple threads simultaneously is called *thread safe* .Race condition only occur when multiple threads update shared resources. Therefore it is important to know what resources Java threads share when executing.

1.Local variable

Local variables are stored in each thread's own stack. That means that local variables are never shared between threads. That also means that all local primitive variables are thread safe.

```
public void someMethod(){
  long threadSafeInt = 0;
  threadSafeInt++;
}
```

---- ------------------

2.Local Object References

Local references to objects are a bit different. The reference itself is not shared. The object referenced however, is not stored in each threads's local stack. All objects are stored in the shared heap.

```
public void someMethod(){
  LocalObject localObject = new LocalObject();
  localObject.setValue("value1");
  localObject.callMethod(); →call non shared method
 method2(localObject);→pass to non-shared method
}
```

Each thread executing the method will create its own `LocalObject` instance and assign it to the `localObject` reference. Therefore the use of the `LocalObject` here is thread safe.

Thread safety may break if you explicitly share local instance variable with other thread.

3.Object Member Variables

Object member variables (fields) are stored on the heap along with the object. Therefore, if two threads call a method on the same object instance and this method updates object member variables, the method is not thread safe. Here is an example of a method that is not thread safe:

```
public class NotThreadSafe{
    StringBuilder builder = new StringBuilder();

    public add(String text){
        this.builder.append(text);
    }
}

public class MyRunnable implements Runnable{
  NotThreadSafe instance = null;

  public MyRunnable(NotThreadSafe instance){
    this.instance = instance;
  }

  public void run(){
    this.instance.add("some text");
  }
}

public class Test{
public static void main(string[] ags){
NotThreadSafe sharedInstance = new
NotThreadSafe();
//race condition with same obj
new Thread(new
MyRunnable(sharedInstance)).start();
new Thread(new
MyRunnable(sharedInstance)).start();

// no race condition both thread have separate
instance
new Thread(new MyRunnable(new
NotThreadSafe)).start();
new Thread(new MyRunnable(new
NotThreadSafe)).start();
}
}
```

#The Thread Control Escape Rule

If a resource is created, used and disposed within the control of the same thread,and never escapes the control of this thread,the use of that resource is thread safe.

Resources can be any shared resource like an object, array, file, database connection, socket etc. In Java you do not always explicitly dispose objects, so "disposed" means losing or null'ing the reference to the object.

Even if the use of an object is thread safe, if that object points to a shared resource like a file or database, your application as a whole may not be thread safe.

Thread Safety and Immutability

Race condition occur only if multiple threads are accessing the same resource, **and** one or more of the threads **write** to the resource. If multiple threads read the same resource **race conditions** do not occur.

We can make sure that objects shared between threads are never updated by any of the threads by making the shared objects immutable, and thereby thread safe.

Immutable object is thread safe	**The Reference is not Thread Safe!**

public class **ImmutableValue**{
private int value = 0;
 	
public **ImmutableValue**(int value){
this.value = value;
}
 	
public int getValue(){
return this.value;
}
 	
// return new object after operation
public ImmutableValue add(int x){
return new ImmutableValue(this.value + x);
}
 	
}
 	
→immutable class doesn't have setter method
	public class **Calculator**{
	private ImmutableValue currentValue = null;
	
	public ImmutableValue getValue(){
	return currentValue;
	}
	
	public void setValue(ImmutableValue newValue){
	this.currentValue = newValue;
	}
	
	// return same object after operation
	
	public void add(int newValue){
	this.currentValue = this.currentValue.add(newValue);
	}
	}
	
	→Here reference of immutable class is not thread safe.

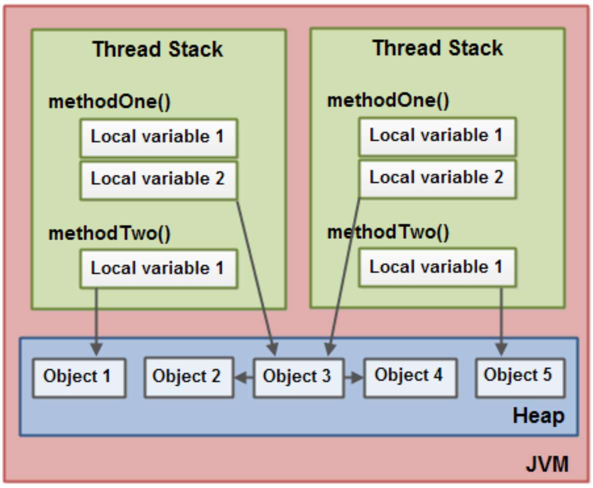
Java Memory Model

The Java memory model specifies how the Java virtual machine works with the computer's memory (RAM). The Java virtual machine is a model of a whole computer so this model naturally includes a memory model - AKA the Java memory model.

The Internal Java Memory Model

The Java memory model used internally in the JVM divides memory between thread stacks and the heap. This diagram illustrates the Java memory model from a logic perspective:

Each thread running in the Java virtual machine has its own thread stack.



The thread stack contains information about what methods the thread has called to reach the current point of execution. I will refer to this as the "call stack". As the thread executes its code, the call stack changes.

The thread stack also contains all local variables for each method being executed. A thread can only access it's own thread stack. Local variables created by a thread are invisible to other threads . Even if two threads are executing the exact same code, the two threads will still create the local variables of that code in each their own thread stack. Thus, each thread has its own version of each local variable.

All local variables of primitive types (boolean, byte, short, char, int, long, float, double) are fully stored on the thread stack and are thus not visible to other threads. One thread may pass a copy of a pritimive variable to another thread, but it cannot share the primitive local variable itself.

A local variable may also be a reference to an object. In that case the reference (the local variable) is stored on the thread stack, but the object itself if stored on the

An object may contain methods and these methods may contain local variables. These local variables are also stored on the thread stack, even if the object the method belongs to is stored on the heap.

The heap contains all objects created in your Java application, regardless of what thread created the object. This includes the object versions of the primitive types (e.g. `Byte`, `Integer`, `Long` etc.). It does not matter if an object was created and assigned to a local variable, or created as a member variable of another object, the object is still stored on the heap.

An object's member variables are stored on the heap along with the object itself. That is true both when the member variable is of a primitive type, and if it is a reference to an object.

Static class variables are also stored on the heap along with the class definition.

Objects on the heap can be accessed by all threads that have a reference to the object. When a thread has access to an object, it can also get access to that object's member variables.

If two threads call a method on the same object at the same time, they will both have access to the object's member variables, but each thread will have its own copy of the local variables.

EX-> Notice how the shared object (Object 3) has a reference to Object 2 and Object 4 as member variables .Via these member variable references in Object 3 the two threads can access Object 2 and Object 4.

Two threads have a set of local variables. One of the local variables (`Local Variable 2`) point to a shared object on the heap (Object 3). The two threads each have a different reference to the same object. Their references are local variables and are thus stored in each thread's thread stack (on each). The two different references point to the same object on the heap, though.

So, what kind of Java code could lead to the above memory graph? Well, code as simple as the code below:

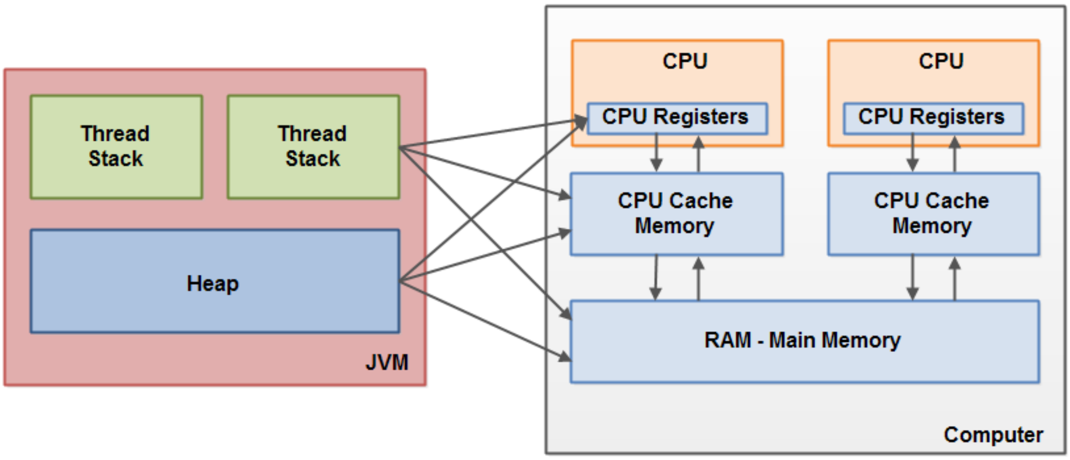
```
public class MyRunnable implements Runnable() {

    public void run() {
        methodOne();
    }

    public void methodOne() {
        int localVariable1 = 45;

        MySharedObject localVariable2 =
            MySharedObject.sharedInstance;

        //... do more with local variables.

        methodTwo();
    }

    public void methodTwo() {
        Integer localVariable1 = new Integer(99);

        //... do more with local variable.
    }
}
```

```
public class MySharedObject {

   //static variable pointing to instance of MySharedObject

    public static final MySharedObject
sharedInstance =
        new MySharedObject();

    //member variables pointing to two objects on the heap

    public Integer object2 = new Integer(22);
    public Integer object4 = new Integer(44);

    public long member1 = 12345;
    public long member2 = 67890;
}

------------------  main class------------
MySharedObject counter = new MySharedObject ();
Thread  threadA = new MyRunnable (counter);
Thread  threadB = new MyRunnable (counter);

threadA.start();
threadB.start();
```

If two threads were executing the run() method then the diagram shown earlier would be the outcome.

Bridging The Gap Between The Java Memory Model And The Hardware Memory Architecture

As already mentioned, the Java memory model and the hardware memory architecture are different. The hardware memory architecture does not distinguish between thread stacks and heap. On the hardware, both the thread stack and the heap are located in main memory. Parts of the thread stacks and heap may sometimes be present in CPU caches and in internal CPU registers. This is illustrated in this diagram:



When objects and variables can be stored in various different memory areas in the computer, certain problems may occur. The two main problems are:

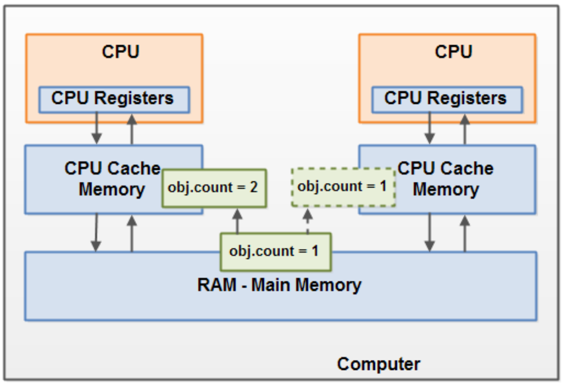
- Visibility of thread updates (writes) to shared variables.
- Race conditions when reading, checking and writing shared variables.

Both of these problems will be explained in the following sections.

Visibility of Shared Objects

If two or more threads are sharing an object, without the proper use of either volatile declarations or synchronization, updates to the shared object made by one thread may not be visible to other threads.

Imagine that the shared object is initially stored in main memory. A thread running on CPU one then reads the shared object into its CPU cache. There it makes a change to the shared object. As long as the CPU cache has not been flushed back to main memory, the changed version of the shared object is not visible to threads running on other CPUs. This way each thread may end up with its own copy of the shared object, each copy sitting in a different CPU cache

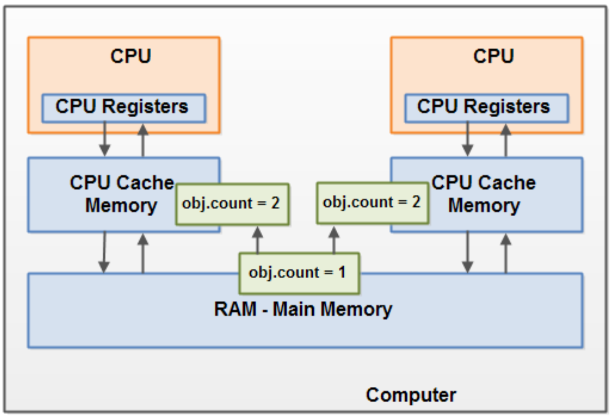


To solve this problem you can use **Java's volatile keyword**. The volatile keyword can make sure that a given variable is read directly from main memory, and always written back to main memory when updated.

Race Conditions

If two or more threads share an object, and more than one thread updates variables in that shared object, **race conditions** may occur.

Imagine two threads want to update common shared resource in different CPU cache and If these increments had been carried out sequentially, the variable count would be incremented twice and had the original value + 2 written back to main memory.



To solve this problem you can use a **Java synchronized block**. A synchronized block guarantees that only one thread can enter a given critical section of the code at any given time. Synchronized blocks also guarantee that all variables accessed inside the synchronized block will be read in from main memory, and when the thread exits the synchronized block, all updated variables will be flushed back to main memory again, regardless of whether the variable is declared volatile or not.

Java Synchronized Blocks

A *Java synchronized block* marks a method or a block of code as *synchronized*. A synchronized block in Java can only be executed a single thread at a time (depending on how you use it). Java synchronized blocks can thus be used to avoid **race conditions**.

The `synchronized` keyword can be used to mark four different types of blocks:

1. Instance methods
2. Static methods
3. Code blocks inside instance methods
4. Code blocks inside static methods

1.Synchronized Instance Methods

```
public class MyCounter {
  private int count = 0;
  public synchronized void add(int value){
      this.count += value;
  }
}
```

Notice the use of the synchronized keyword in the add() method declaration. This tells Java that the method is synchronized.

Only one thread per instance can execute inside a synchronized instance method

2.Synchronized Static methods

```
public static MyStaticCounter{
  private static int count = 0;
  public static synchronized void add(int value){
      count += value;
  }
}
```

here the synchronized keyword tells Java that the add() method is synchronized.

Synchronized static methods are synchronized on the class instance and only one thread per instance will allow.

3.Synchronized Blocks in Instance Methods

You do not have to synchronize a whole method. Sometimes it is preferable to synchronize only part of a method. Java synchronized blocks inside methods makes this possible.

```
public void add(int value){
   //….
    synchronized(this){
       this.count += value;
    }
  //…
  }
```

4.Synchronized Blocks in Static Methods

Synchronized blocks can also be used inside of static methods. Here are the same two examples from the previous section as static methods.

```
 public class MyClass {

    public static synchronized void log1(String msg1,
String msg2){
       log.writeln(msg1);
       log.writeln(msg2);
    }

    public static void log2(String msg1, String msg2){
       synchronized(MyClass.class){
          log.writeln(msg1);
          log.writeln(msg2);
       }
    }
  }
```

Java Concurrency Utilities

The synchronized mechanism was Java's first mechanism for synchronizing access to objects shared by multiple threads. The synchronized mechanism isn't very advanced though. That is why Java 5 got a whole set of **concurrency utility classes** to help developers implement more fine grained concurrency control than what you get with synchronized.(java.util.Concurrency)

Synchronized Blocks in Lambda Expressions

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

    Consumer<String> func = (String param) -> {
      synchronized(SynchronizedExample.class) {
        System.out.println(  Thread.currentThread().getName() + " step 1: " + param);

        try {
          Thread.sleep( (long) (Math.random() * 1000));
        } catch (InterruptedException e) {
          e.printStackTrace();
        }

        System.out.println( Thread.currentThread().getName() +   " step 2: " + param);
      }
    };

    Thread thread1 = new Thread(() -> {  func.accept("Parameter");  }, "Thread 1");

    Thread thread2 = new Thread(() -> {
        func.accept("Parameter");
    }, "Thread 2");

    thread1.start();
    thread2.start();
  }
}
```

Synchronized and Data Visibility	When a thread enters a synchronized block it will refresh the values of all variables visible to the thread. When a thread exits a synchronized block all changes to variables visible to the thread will be committed to main memory. This is similar to how the volatile keyword works.
Synchronized and Instruction Reordering	See below
What Objects to Synchronize On	Primitive type or their Wrapper class should not be synchronized , it will return same object to every thread.
Synchronized Block Limitations and Alternatives	
Synchronized Block Performance Overhead	
Synchronized Block Reentrance	

Synchronized and Instruction Reordering

The Java compiler and Java Virtual Machine are allowed to reorder instructions in your code to make them execute faster, typically by enabling the reordered instructions to be executed in parallel by the CPU.

→Synchronized block or volatile key word solves the reordering problem.

Synchronizing the shared data access across threads achieves two things:

- Mutual exclusion of two thread execution over synchronized block. This achieves atomicity. A threads is not able to see incomplete writes (involving multiple variables) of other threads.
- Flushing of local thread memory to main memory at the time when the lock is released (at the end of synchronized block).

Reordering example:-

Example:-1

```
public class ReorderExample {
  private int a = 2;
  private boolean flg = false;

  public void method1() {
      a = 1;
      flg = true;
  }

  public void method2() {
      if (flg) {
          //2 might be printed out on some JVM/machines
          System.out.println("a = " + a);
      }
  }

  public static void main(String[] args) {
      for (int i = 0; i < 100; i++) {
          ReorderExample reorderExample = new
ReorderExample();
          Thread thread1 = new Thread(() -> {
              reorderExample.method1();
          });
          Thread thread2 = new Thread(() -> {
              reorderExample.method2();
          });
          thread1.start();
          thread2.start();
      }
  }
}
```

→Thread 1 changed a=1, flg=true but it may be in CPU cache.
→Thread 2 may get flg=true any time and a=2 as previous cache.(or any combination)

Example:-2

```
public class ReorderPrevention {
  private int a = 2;
  private boolean flg = false;

  public synchronized void method1() {
      a = 1;
      flg = true;
  }

  public synchronized void method2() {
      if (flg) {
          System.out.println("a = " + a);
      }
  }

  public static void main(String[] args) {
      for (int i = 0; i < 100; i++) {
          ReorderPrevention reorderExample = new
ReorderPrevention();
          Thread thread1 = new Thread(() -> {
              reorderExample.method1();
          });
          Thread thread2 = new Thread(() -> {
              reorderExample.method2();
          });
          thread1.start();
          thread2.start();
      }
  }
}
```

→Thread 1 changed a=1,flg=true and before releasing block it changes in main memory, all thread cache updates current value.
→Thread 2 will get updated value and prints currect result

What Objects to Synchronize On

You can actually choose any object to synchronize on, but it is recommended that you do not synchronize on String objects, or any primitive type wrapper objects, as the compiler might optimize those, so that you are using the same instances in different places in your code where you thought you were using different instance. Look at this example:

```
synchronized("Hey") {
  //do something in here.
}
```

If you have more than one synchronized block that is synchronized on the literal String value "Hey", then the compiler might actually use the same String object behind the scenes. The result being, that both of these two synchronized blocks are then synchronized on the same object. That might not be the behaviour you were looking for.

The same can be true for using primitive type wrapper objects. Look at this example:

```
synchronized(Integer.valueOf(1)) {
  //do something in here.
}
```

If you call Integer.valueOf(1) multiple times, it might actually return the same wrapper object instance for the same input parameter values. That means, that if you are synchronizing multiple blocks on the same primitive wrapper object and then you risk that those synchronized blocks all get synchronized on the same object. That might also not be the behaviour you were looking for.

To be on the safe side, synchronize on this - or on a new Object() . Those are not cached or reused internally by the Java compiler, Java VM or Java libraries.

Synchronized Block Limitations and Alternatives

Synchronized blocks in Java have several limitations. For instance, a synchronized block in Java only allows a single thread to enter at a time. However, what if two threads just wanted to read a shared value, and not update it? That might be safe to allow. As alternative to a synchronized block you could guard the code with a **Read / Write Lock** which as more advanced locking semantics than a synchronized block. Java actually comes with a built in **ReadWriteLock** class you can use.

What if you want to allow N threads to enter a synchronized block, and not just one? You could use a **Semaphore** to achieve that behaviour. Java actually comes with a built-in **Java Semaphore** class you can use.

Synchronized blocks do not guarantee in what order threads waiting to enter them are granted access to the synchronized block. What if you need to guarantee that threads trying to enter a synchronized block get access in the exact sequence they requested access to it? You need to implement **Fairness** yourself.

What if you just have one thread writing to a shared variable, and other threads only reading that variable? Then you might be able to just use a **volatile variable** without any synchronization around.

Synchronized Block Performance Overhead

There is a small performance overhead associated with entering and exiting a synchronized block in Java. As Jave have evolved this performance overhead has gone down, but there is still a small price to pay.

The performance overhead of entering and exiting a synchronized block is mostly something to worry about if you enter and exit a synchronized block lots of times within a tight loop or so.

Also, try not to have larger synchronized blocks than necessary. In other words, only synchronize the operations that are really necessary to synchronize - to avoid blocking other threads from executing operations that do not have to be synchronized. Only the absolutely necessary instructions in synchronized blocks. That should increase parallelism of your code.

Synchronized Block Reentrance

Once a thread has entered a synchronized block the thread is said to "hold the lock" on the monitoring object the synchronized block is synchronized on. If the thread calls another method which calls back to the first method with the synchronized block inside, the thread holding the lock can reenter the synchronized block. It is not blocked just because a thread (itself) is holding the lock. Only if a differen thread is holding the lock. Look at this example:

```
public class MyClass {

  List<String> elements = new ArrayList<String>();

  public void count() {
    if(elements.size() == 0) {
        return 0;
    }
    synchronized(this) {
       elements.remove();
       return 1 + count();
    }
  }
}
```

Forget for a moment that the above way of counting the elements of a list makes no sense at all. Just focus on how inside the synchronized block inside the count() method calls the count() method recursively. Thus, the thread calling count() may eventually enter the same synchronized block multiple times. This is allowed. This is possible.

As an alternative to a synchronized block you could also use one of the many atomic data types found in the **java.util.concurrent package**. For instance, the **AtomicLong** or **AtomicReference** or one of the others.

Java Volatile Keyword

The Java volatile keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

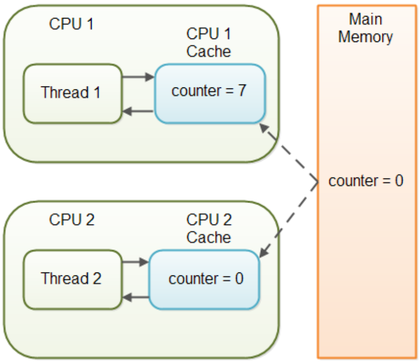
Variable Visibility Problems

The Java volatile keyword guarantees visibility of changes to variables across threads.

In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU cache while working on them, for performance reasons. If your computer contains more than one CPU, each thread may run on a different CPU. That means, that each thread may copy the variables into the CPU cache of different CPUs. This is illustrated here:

```
public class SharedObject {
    public int counter = 0; //non volatile
}
```

→ The updates of one thread are not visible to other threads.



```
public class SharedObject {
    public volatile int counter = 0; // volatile
}
```

// the updated value will be visible to all thread.

Full volatile Visibility Guarantee

Actually, the visibility guarantee of Java volatile goes beyond the volatile variable itself.

- If Thread A writes to a volatile variable and Thread B subsequently reads the same volatile variable, then all variables visible to Thread A before writing the volatile variable, will also be visible to Thread B after it has read the volatile variable. The volatile keyword is guaranteed to work on 32 bit and 64 variables.
- **If Thread A reads a volatile variable, then all variables visible to Thread A when reading the volatile variable will also be re-read from main memory.**
- **It may affect the performance as it will work directly to main memory instead of cpu cache.**

```
public class MyClass {
    private int years;
    private int months
    private volatile int days;

    public void update(int years, int months, int days){
        this.years  = years;
        this.months = months;
        this.days   = days;
→All thread which is reading volatile variable will update all thread variable mentioned above it
→ because all variable will got updated in main memory after variable reading.

        this.years  = years+1; → After the volatile variable ,which is non-volatile may not be visible to all
thread.
        this.months = months+1;

    }
}
```

java.util.ThreadLocal

The *Java ThreadLocal* class enables you to create variables that can only be read and written by the same thread. Thus, even if two threads are executing the same code, and the code has a reference to the same ThreadLocal variable, the two threads cannot see each other's ThreadLocal variables. Thus, the Java ThreadLocal class provides a simple way to make code **thread safe** that would not otherwise be so.

Creating a ThreadLocal • set,get,remove	see example below is generic → private ThreadLocal threadLocal = new ThreadLocal();
Generic ThreadLocal	(See example below it is generic one)

Initial ThreadLocal Value
→Override initialValue()
→Provide a Supplier Implementation
 	
// The second method for specifying an initial value for a Java ThreadLocal variable is to use its static factory method withInitial(Supplier) passing a Supplier interface implementation as parameter.
	//overriding value
	private ThreadLocal myThreadLocal = new ThreadLocal<String>() {
	    @Override protected String initialValue() {
	        return String.valueOf(System.currentTimeMillis());
	    }
	}; Or
	ThreadLocal<String> threadLocal = ThreadLocal.withInitial(new Supplier<String>() {
	    @Override
	    public String get() {
	        return String.valueOf(System.currentTimeMillis());
	    }
	}); Or
	ThreadLocal threadLocal3 = ThreadLocal.withInitial(
	        () -> String.valueOf(System.currentTimeMillis()));

InheritableThreadLocal	The InheritableThreadLocal class is a subclass of ThreadLocal. Instead of each thread having its own value inside a ThreadLocal, the InheritableThreadLocal grants access to values to a thread and all child threads created by that thread.

```
public class ThreadLocalExample {

    public static void main(String[] args) {
        MyRunnable sharedRunnableInstance = new MyRunnable();

        Thread thread1 = new Thread(sharedRunnableInstance);
        Thread thread2 = new Thread(sharedRunnableInstance);

        thread1.start();
        thread2.start();

        thread1.join(); //wait for thread 1 to terminate
        thread2.join(); //wait for thread 2 to terminate
    }
}
public class MyRunnable implements Runnable {
    private ThreadLocal<Integer> threadLocal = new ThreadLocal<Integer>();

    @Override
    public void run() {

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
        }
        threadLocal.set( (int) (Math.random() * 100D) ); → set ThreadLocal
        System.out.println((Integer)threadLocal.get());   →get ThreadLocal value;
        threadLocal.remove();                              → remove threadLocal
    }
}
```

Inner Thread communication

→Threads can communicate with each other by using **wait(),notify(), notifyAll()** methods, the thread which is executing child thread for update operation is responsible to call **wait()** method , after calling **wait()** method thread will enter into waiting state. Child thread which is performing update operation is responsible to call **notify()** or **notifyAll()** method after completion of update operation. The waiting Thread will receive notification and it will start further execution.

→To call wait(),notify(),notifyAll() method on any object , thread should be inside synchronised area otherwise Runtime-Exception will be there **(IllegalMonitoStateException**)

→if a thread calls wait() method of any object it will immediately releases the lock of the particular object and entered into waiting state.

→ if a thread calls notify() method of any object it may or may not release that object immediately.
→Every wait() method throws **InterruptedException** which is **checked exception** ,we should handle these exception otherwise we will get **compile time error.**

```
Public Class MyThread extends
Thread{
  Int total=0;
  Public void run(){               //(3)

    //----------more  code above --------
    Synchronised(this){
      S.O.P(“child thread started”);
      For(int i=0;i<20;i++){
            total=total+I;
       }
     this.notify(); // (4)
    }
//----------more code below--------


  }
}
```

```
Public class Test{
   Public static void main(string[] args){
        MyThread mt=new MyThread();
        mt.start();

       Synchronized(mt){
          S.O.P(“main Thread calling wait()”);   // (1)

           try(){
                 mt.wait();    //(2)      → waiting for notify from chield
          }
         Catch(InterruptedException e){
                      // --------
         }

          S.O.P(“child thread give notification”+ mt.total );   //(5)

        }
   }

}
```

→ after getting notification from child CPU will give time to any of the thread for further normal execution.

Daemon Thread

Daemon thread is a low priority thread (in context of JVM) that runs in background to perform tasks such as garbage collection (gc) etc., they do not prevent the JVM from exiting (even if the daemon thread itself is running) when all the user threads (non-daemon threads) finish their execution. JVM terminates itself when all user threads (non-daemon threads) finish their execution, JVM does not care whether Daemon thread is running or not, if JVM finds running daemon thread (upon completion of user threads), it terminates the thread and after that shutdown itself.

Properties of Daemon threads:

A newly created thread inherits the daemon status of its parent. That’s the reason all threads created inside main method (child threads of main thread) are non-daemon by default, because main thread is non-daemon. However you can make a user thread to Daemon by using setDaemon() method of thread class.

Just a quick note on main thread: When the JVM starts, it creates a thread called “Main”. Your program will run on this thread, unless you create additional threads yourself. The first thing the “Main” thread does is to look for your static void main (String args[]) method and invoke it. That is the entry-point to your program. If you create additional threads in the main method those threads would be the child threads of main thread.

```
public class MyThread extends Thread {
    Public void run() {
        For(int i = 0; i < 20; i++) {
            System.out.println(“child thread”);
            try() {
                Thread.sleep(2000);
            }
            Catch(InterruptedException)
        }
    }
}
```

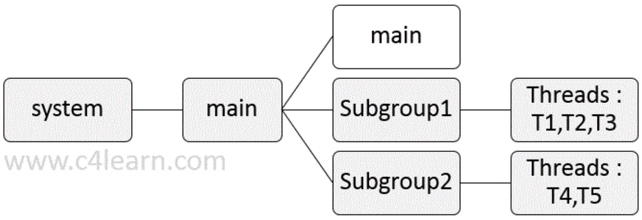
```
public class Test{
    public static void maiin(String[] args){
        MyThread t=new MyThread();
        t.setDaemon(true);              →setDaemon() mehod
        t.start();
        System.out.println(t.isDaemon());  →isDaemon()
method

    }
}
```

→1. When we will not make a thread , daemon thread ,then after completion of main thread ,child thread will be running.

→2. If we will make a thread daemon ,after completion of parent thread daemon will automatically ends.

Thread Group

→ Based on functionality we can group threads into a single unit which is nothing but thread group. i.e Thread group contains a group of threads. In addition to thread ,thread group can also contain sub-groups.
→Every thread in java belongs to some group , main thread belongs to main group.Every thread group is the child group of system group either directly or indirectly .Hence system group is the root for all thread group in java.
→System group contains several system level thread like (finalizer,reference handler,signal dispatcher,clock,attach listener,garbage collector.)



Advantage:-

Thread group allow to perform common operations like setMaxPriority,setMinPriority and destroy all producer thread.

Every thread in java belongs to some Thread Group.

Thread.currentThread().getThreadGroup().getNames();	We can get current thread group name **Main thread→main threadGruop→ main**
Thread.currentThread().getThreadGroup().getParent(). getNames();	**We can get parent thread group from current thread group** **Main thread→main threadGruop→system thread group→ system**

Constructor of ThreadGroup
1.ThreadGroup g=new ThreadGroup(String gName);
2.ThreadGroup g=new ThreadGroup(ThreadGroup pg ,String gName);

Imp methods	
1.String getName()	
2.Int getMaxpriority()	
3.void setMaxPriority(int p)	// default maxPriority(10)
4.ThreadGroup getParent()	
5.void list()	// shows threadGroup info .
6.int activeCount()	// returns no of active thread in current threadGroup
7.int activeGroupCount()	//return no of threadGroup within current threadGroup
8.int enumerate(Thread[] t)	
9.int enumerate(ThreadGroup g)	
10.boolean isDaemon()	// check threadGroup is daemon of not
11.void setDaemon(Boolean b)	
12.void interrupt()	// interrupt all waiting or sleeping threads of threadGroup
13.void destroy()	// destroy threadGroup and it's subGroup

→Threads in the thread group that already have higher priority won't be affected but for newly added threads this max priority is set.

```
class MyThread extends Thread{
    MyThread(ThreadGroup g,String name){
        super(g,name);
    }
    public void run(){
        system.out.println("child thread");
        try(){
            Thread.sleep(200);
        }
        catch(InterruptedException){}
    }
}
```

```
System→main→parentGroup→ChildGroup
                        →childThread1
                        →childThread2
```

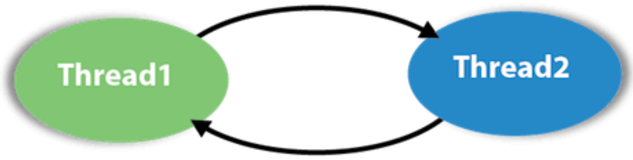
```
class ThreadGroupDemo{
    public static void main(String[] args ){
        ThreadGroup pg=new ThreadGroup("ParentGroup");
        ThreadGroup cg=new ThreadGroup(pg,"childGroup");
       MyThread t1=new Mythread(pg,"child thread1");  //priority 5
        Pg.setPriority(3);
        MyThread t2=new Mythread(pg,"child thread2");
//priority 3
        t1.start();
        t2.start();
        s.o.p(pg.activeCount());   //2 waitign thread
        s.o.p(pg.activeCount());   //1 child group

    }
}
```

Typical problems in Java concurrency:

Deadlock

Thread Deadlock

Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



```
public class Demo {
  public static void main(String[] args) {
    final String resource1 = "ratan jaiswal";
    final String resource2 = "vimal jaiswal";
    // t1 tries to lock resource1 then resource2
    Thread t1 = new Thread() {
      public void run() {
          synchronized (resource1) {
           System.out.println("Thread 1: locked resource 1");
           try { Thread.sleep(100);} catch (Exception e) {}
           synchronized (resource2) {
            System.out.println("Thread 1: locked resource 2");
           }
         }
      }
    };

    // t2 tries to lock resource2 then resource1
    Thread t2 = new Thread() {
      public void run() {
        synchronized (resource2) {
          System.out.println("Thread 2: locked resource 2");
          try { Thread.sleep(100);} catch (Exception e) {}
          synchronized (resource1) {
            System.out.println("Thread 2: locked resource 1");
          }
        }
      }
    };

    t1.start();
    t2.start();
  }
}
```

Database deadlock

A more complicated situation in which deadlocks can occur, is a database transaction. A database transaction may consist of many SQL update requests. When a record is updated during a transaction, that record is locked for updates from other transactions, until the first transaction completes.

Deadlock Prevention

Lock Ordering

Deadlock occurs when multiple threads need the same locks but obtain them in different order.
If you make sure that all locks are always taken in the same order by any thread, deadlocks cannot occur.

Lock Timeout

Another deadlock prevention mechanism is to put a timeout on lock attempts meaning a thread trying to obtain a lock will only try for so long before giving up.

Deadlock Detection

Deadlock detection is a heavier deadlock prevention mechanism aimed at cases in which lock ordering isn't possible, and lock timeout isn't feasible.
Every time a thread takes a lock it is noted in a data structure (map, graph etc.) of threads and locks. Additionally, whenever a thread requests a lock this is also noted in this data structure.

Starvation and Fairness

Low priority thread will never get CPU for execution or blocked for other execution, endless waiting for CPU of a thread is called starvation. The solution of starvation is ‘fairness’ concept.

Fairness in Java (with Synchronized)

```
public class Synchronizer{
  public synchronized void doSynchronized(){
    //do a lot of work which takes a long time
  }
}
```

Increase Fairness in Java (with Lock)

```
public class Synchronizer{
  Lock lock = new Lock();
  public void doSynchronized() throws InterruptedException{
    this.lock.lock();
      //critical section, do a lot of work which takes a long time
    this.lock.unlock();
  }
}
```

A Fair Lock

Below is shown the previous Lock class turned into a fair lock called FairLock. every thread calling lock() is now queued, and only the first thread in the queue is allowed to lock the FairLock instance, if it is unlocked. All other threads are parked waiting until they reach the top of the queue.

```
public class FairLock {
    private boolean  isLocked       = false;
    private Thread    lockingThread  = null;
    private List<QueueObject> waitingThreads =
            new ArrayList<QueueObject>();

  public void lock() throws InterruptedException{
    QueueObject queueObject   = new QueueObject();
    boolean     isLockedForThisThread = true;
    synchronized(this){
        waitingThreads.add(queueObject);
    }

    while(isLockedForThisThread){
      synchronized(this){
        isLockedForThisThread =
            isLocked || waitingThreads.get(0) != queueObject;
        if(!isLockedForThisThread){
          isLocked = true;
           waitingThreads.remove(queueObject);
           lockingThread = Thread.currentThread();
           return;
         }
      }
      try{
        queueObject.doWait();
      }catch(InterruptedException e){
        synchronized(this) {
                waitingThreads.remove(queueObject);
 }
        throw e;
      }
    }
  }
  public synchronized void unlock(){
    if(this.lockingThread != Thread.currentThread()){
      throw new IllegalMonitorStateException(
        "Calling thread has not locked this lock");
    }
    isLocked      = false;
    lockingThread = null;
    if(waitingThreads.size() > 0){
      waitingThreads.get(0).doNotify();
    }
  }
}
```

```
public class QueueObject {

  private boolean isNotified = false;

  public synchronized void doWait() throws InterruptedException {
    while(!isNotified){
        this.wait();
    }
    this.isNotified = false;
  }

  public synchronized void doNotify() {
    this.isNotified = true;
    this.notify();
  }

  public boolean equals(Object o) {
    return this == o;
  }
}
```

→

First you might notice that the lock() method is no longer declared synchronized. Instead only the blocks necessary to synchronize are nested inside synchronized blocks.

FairLock creates a new instance of QueueObject and enqueue it for each thread calling lock(). The thread calling unlock() will take the top QueueObject in the queue and call doNotify() on it, to awaken the thread waiting on that object. This way only one waiting thread is awakened at a time, rather than all waiting threads. This part is what governs the fairness of the FairLock.

Slipped Conditions

Slipped Condition is a special type of race condition that can occur in a multithreaded application. In this, a thread is suspended after reading a condition and before performing the activities related to it. It rarely occurs, however, one must look for it if the outcome is not as expected.

Example: Suppose there are two thread A and thread B which want to process a string S. Firstly, thread A is started, it checks if there are any more characters left to process, initially the whole string is available for processing, so the condition is true. Now, thread A is suspended and thread B starts. It again checks the condition, which evaluates to true and then processes the whole string S. Now, when thread A again starts execution, the string S is completely processed by this time and hence an error occurs. This is known as a slipped condition.

```
class Resource {
    static final String string = "Hello";
    static int pointerPosition = 0;
}
```

```
// Thread to process the whole String
class ReadingThread extends Thread {
    @Override
    public void run()
    {
        System.out.println("Thread2 trying to "
                          + "process the string");
        while (Resource.pointerPosition
               < Resource.string.length()) {

            Resource.pointerPosition++;
        }
    }
}
```

```
class SlippedThread extends Thread {
    @Override
    public void run()
    {
        if (Resource.pointerPosition !=
Resource.string.length()) {
            try {
                synchronized (this)
                {
                    wait(500);
                }
            }

            catch (InterruptedException e) {
                System.out.println(e);
            }

            try {

S.O.P(Resouce.string.charAt(Resource.pointerPosition));
                }
            }

            catch (StringIndexOutOfBoundsException e) {
                System.out.println("\nNo more character left or
sliped");
            }
        }
    }
}
```

```
public class Main {
    public static void main(String[] args)
    {
        ReadingThread readingThread  = new ReadingThread();
        SlippedThread slippedThread   = new SlippedThread();

        slippedThread.start();   → this will not able to count further because other thread processed all count.
        readingThread.start();   → this will processed all
    }
}
```

Java.util.concurrent package

Problems with traditional collection

Sr.No.	Key	Traditional Collections	Concurrent Collections
1	Thread Safety	Most of the classic classes in Java Collections such as Array List, Linked List, Hash Map etc. are not synchronized and are not thread safe in multi-threading environment.	On other hand Java introduces same classes in Concurrent Collections with implement synchronization in them which not only make these classes as Synchronized but also thread safe in nature.
2	Locking Mechanism	We have some synchronized classes in traditional collections as well such as Vector and Stack but these classes uses lock over whole collection which reduces performance and speed of execution.	On other hand concurrent collections introduces concept of partial locking where it locks only part of collection in case of multi-threading environment which improves the performance and speed of collections in such environment.
3	Runtime Exception	In case of traditional collections if we try to modify a collection through separate thread during collection iteration then we got Runtime Exception ConcurrentModificationException.	On other hand one would not get such exception if deals with the concurrent collections i.e. concurrent collections allows modification in collection during its iteration.
4	Preference	Due to reason mentioned in above points traditional collections are not preferred in multi-threading environment.	On other hand Concurrent collections are primarily preferred in multi-threading environment.
5	Introduction in market	Traditional collections are type of legacy collection in Java and are introduced before concurrent collections.	While concurrent collections are introduced in JDK 1.5 i.e. are introduced after traditional collections.

Package java.util.concurrent Description

Utility classes commonly useful in concurrent programming. This package includes a few small standardized extensible frameworks, as well as some classes that provide useful functionality and are otherwise tedious or difficult to implement. Here are brief descriptions of the main components. See also the java.util.concurrent.locks and java.util.concurrent.atomic packages.

<table>
<tr><th colspan="2">Concurrent Package overview</th></tr>
<tr><td><b>1.Executors Interfaces</b>.
<b>Executor</b> is a simple standardized interface for defining custom thread-like subsystems, including thread pools, asynchronous I/O, and lightweight task frameworks.
<b>ExecutorService</b> provides a more complete asynchronous task execution framework. An ExecutorService manages queuing and scheduling of tasks, and allows controlled shutdown.
<b>ScheduledExecutorService</b> subinterface and associated interfaces add support for delayed and periodic task execution.
<b>ExecutorServices</b> provide methods arranging asynchronous execution of any function expressed as Callable, the result-bearing analog of Runnable.
<b>Future</b> returns the results of a function, allows determination of whether execution has completed, and provides a means to cancel execution.
<b>RunnableFuture</b> is a Future that possesses a run method that upon execution, sets its results.</td><td><b>Implementations</b>.
<b>ThreadPoolExecutor</b> and <b>ScheduledThreadPoolExecutor</b> Classes provide tunable, flexible thread pools.
<b>Executors</b> class provides factory methods for the most common kinds and configurations of Executors, as well as a few utility methods for using them.
<b>FutureTask</b> providing a common extensible implementation of <b>Futures</b>, and <b>ExecutorCompletionService</b>, that assists in coordinating the processing of groups of asynchronous tasks.
Class <b>ForkJoinPool</b> provides an Executor primarily designed for processing instances of ForkJoinTask and its subclasses. These classes employ a work-stealing scheduler that attains high throughput for tasks conforming to restrictions that often hold in computation-intensive parallel processing.</td></tr>
<tr><td colspan="2"><b>Queues</b>
The ConcurrentLinkedQueue class supplies an efficient scalable thread-safe non-blocking FIFO queue. The ConcurrentLinkedDeque class is similar, but additionally supports the Deque interface.</td></tr>
<tr><td colspan="2"><b>Timing</b>
The TimeUnit class provides multiple granularities (including nanoseconds) for specifying and controlling time-out based operations.</td></tr>
<tr><td colspan="2"><b>Concurrent Collections</b>
Besides Queues, this package supplies Collection implementations designed for use in multithreaded contexts: ConcurrentHashMap, ConcurrentSkipListMap, ConcurrentSkipListSet, CopyOnWriteArrayList, and CopyOnWriteArraySet.</td></tr>
</table>

Class Hierarchy

- java.lang.**Object**
    - java.util.**AbstractCollection**<E> (implements java.util.Collection<E>)
        - java.util.**AbstractQueue**<E> (implements java.util.Queue<E>)
            - java.util.concurrent.**ArrayBlockingQueue**<E> (implements java.util.concurrent.BlockingQueue<E>, java.io.Serializable)
            - java.util.concurrent.**ConcurrentLinkedQueue**<E> (implements java.util.Queue<E>, java.io.Serializable)
            - java.util.concurrent.**DelayQueue**<E> (implements java.util.concurrent.BlockingQueue<E>)
            - java.util.concurrent.**LinkedBlockingDeque**<E> (implements java.util.concurrent.BlockingDeque<E>, java.io.Serializable)
            - java.util.concurrent.**LinkedBlockingQueue**<E> (implements java.util.concurrent.BlockingQueue<E>, java.io.Serializable)
            - java.util.concurrent.**LinkedTransferQueue**<E> (implements java.io.Serializable, java.util.concurrent.TransferQueue<E>)
            - java.util.concurrent.**PriorityBlockingQueue**<E> (implements java.util.concurrent.BlockingQueue<E>, java.io.Serializable)
            - java.util.concurrent.**SynchronousQueue**<E> (implements java.util.concurrent.BlockingQueue<E>, java.io.Serializable)
        - java.util.**AbstractSet**<E> (implements java.util.Set<E>)
            - java.util.concurrent.**ConcurrentSkipListSet**<E> (implements java.lang.Cloneable, java.util.NavigableSet<E>, java.io.Serializable)
            - java.util.concurrent.**CopyOnWriteArraySet**<E> (implements java.io.Serializable)
        - java.util.concurrent.**ConcurrentLinkedDeque**<E> (implements java.util.Deque<E>, java.io.Serializable)
    - java.util.concurrent.**AbstractExecutorService** (implements java.util.concurrent.ExecutorService)
        - java.util.concurrent.**ForkJoinPool**
        - java.util.concurrent.**ThreadPoolExecutor**
            - java.util.concurrent.**ScheduledThreadPoolExecutor** (implements java.util.concurrent.ScheduledExecutorService)
    - java.util.**AbstractMap**<K,V> (implements java.util.Map<K,V>)
        - java.util.concurrent.**ConcurrentHashMap**<K,V> (implements java.util.concurrent.ConcurrentMap<K,V>, java.io.Serializable)
        - java.util.concurrent.**ConcurrentSkipListMap**<K,V> (implements java.lang.Cloneable, java.util.concurrent.ConcurrentNavigableMap<K,V>, java.io.Serializable)
    - java.util.concurrent.**CompletableFuture**<T> (implements java.util.concurrent.CompletionStage<T>, java.util.concurrent.Future<V>)
    - java.util.concurrent.**ConcurrentHashMap.KeySetView**<K,V> (implements java.io.Serializable, java.util.Set<E>)
    - java.util.concurrent.**CopyOnWriteArrayList**<E> (implements java.lang.Cloneable, java.util.List<E>, java.util.RandomAccess, java.io.Serializable)
    - java.util.concurrent.**CountDownLatch**
    - java.util.concurrent.**CyclicBarrier**
    - java.util.concurrent.**Exchanger**<V>
    - java.util.concurrent.**ExecutorCompletionService**<V> (implements java.util.concurrent.CompletionService<V>)
    - java.util.concurrent.**Executors**
    - java.util.concurrent.**ForkJoinTask**<V> (implements java.util.concurrent.Future<V>, java.io.Serializable)
        - java.util.concurrent.**CountedCompleter**<T>
        - java.util.concurrent.**RecursiveAction**
        - java.util.concurrent.**RecursiveTask**<V>
    - java.util.concurrent.**FutureTask**<V> (implements java.util.concurrent.RunnableFuture<V>)
    - java.util.concurrent.**Phaser**
    - java.util.**Random** (implements java.io.Serializable)
        - java.util.concurrent.**ThreadLocalRandom**
    - java.util.concurrent.**Semaphore** (implements java.io.Serializable)
    - java.lang.**Thread** (implements java.lang.Runnable)
        - java.util.concurrent.**ForkJoinWorkerThread**
    - java.util.concurrent.**ThreadPoolExecutor.AbortPolicy** (implements java.util.concurrent.RejectedExecutionHandler)
    - java.util.concurrent.**ThreadPoolExecutor.CallerRunsPolicy** (implements java.util.concurrent.RejectedExecutionHandler)
    - java.util.concurrent.**ThreadPoolExecutor.DiscardOldestPolicy** (implements java.util.concurrent.RejectedExecutionHandler)
    - java.util.concurrent.**ThreadPoolExecutor.DiscardPolicy** (implements java.util.concurrent.RejectedExecutionHandler)
    - java.lang.**Throwable** (implements java.io.Serializable)
        - java.lang.**Exception**
            - java.util.concurrent.**BrokenBarrierException**
            - java.util.concurrent.**ExecutionException**

- java.lang.**RuntimeException**
  - java.util.concurrent.**CompletionException**
  - java.lang.**IllegalStateException**
    - java.util.concurrent.**CancellationException**
  - java.util.concurrent.**RejectedExecutionException**
- java.util.concurrent.**TimeoutException**

Interface Hierarchy

- java.util.concurrent.**Callable**<V>
- java.lang.**Comparable**<T>
  - java.util.concurrent.**Delayed**
    - java.util.concurrent.**ScheduledFuture**<V> (also extends java.util.concurrent.Future<V>)
      - java.util.concurrent.**RunnableScheduledFuture**<V> (also extends java.util.concurrent.RunnableFuture<V>)
- java.util.concurrent.**CompletableFuture.AsynchronousCompletionTask**
- java.util.concurrent.**CompletionService**<V>
- java.util.concurrent.**CompletionStage**<T>
- java.util.concurrent.**Executor**
  - java.util.concurrent.**ExecutorService**
    - java.util.concurrent.**ScheduledExecutorService**
- java.util.concurrent.**ForkJoinPool.ForkJoinWorkerThreadFactory**
- java.util.concurrent.**ForkJoinPool.ManagedBlocker**
- java.util.concurrent.**Future**<V>
  - java.util.concurrent.**RunnableFuture**<V> (also extends java.lang.Runnable)
    - java.util.concurrent.**RunnableScheduledFuture**<V> (also extends java.util.concurrent.ScheduledFuture<V>)
  - java.util.concurrent.**ScheduledFuture**<V> (also extends java.util.concurrent.Delayed)
    - java.util.concurrent.**RunnableScheduledFuture**<V> (also extends java.util.concurrent.RunnableFuture<V>)
- java.lang.**Iterable**<T>
  - java.util.**Collection**<E>
    - java.util.**Queue**<E>
      - java.util.concurrent.**BlockingQueue**<E>
        - java.util.concurrent.**BlockingDeque**<E> (also extends java.util.Deque<E>)
        - java.util.concurrent.**TransferQueue**<E>
      - java.util.**Deque**<E>
        - java.util.concurrent.**BlockingDeque**<E> (also extends java.util.concurrent.BlockingQueue<E>)
- java.util.**Map**<K,V>
  - java.util.concurrent.**ConcurrentMap**<K,V>
    - java.util.concurrent.**ConcurrentNavigableMap**<K,V> (also extends java.util.NavigableMap<K,V>)
  - java.util.**SortedMap**<K,V>
    - java.util.**NavigableMap**<K,V>
      - java.util.concurrent.**ConcurrentNavigableMap**<K,V> (also extends java.util.concurrent.ConcurrentMap<K,V>)
- java.util.concurrent.**RejectedExecutionHandler**
- java.lang.**Runnable**
  - java.util.concurrent.**RunnableFuture**<V> (also extends java.util.concurrent.Future<V>)
    - java.util.concurrent.**RunnableScheduledFuture**<V> (also extends java.util.concurrent.ScheduledFuture<V>)
- java.util.concurrent.**ThreadFactory**

Hierarchy For Package java.util.concurrent.locks

Class Hierarchy

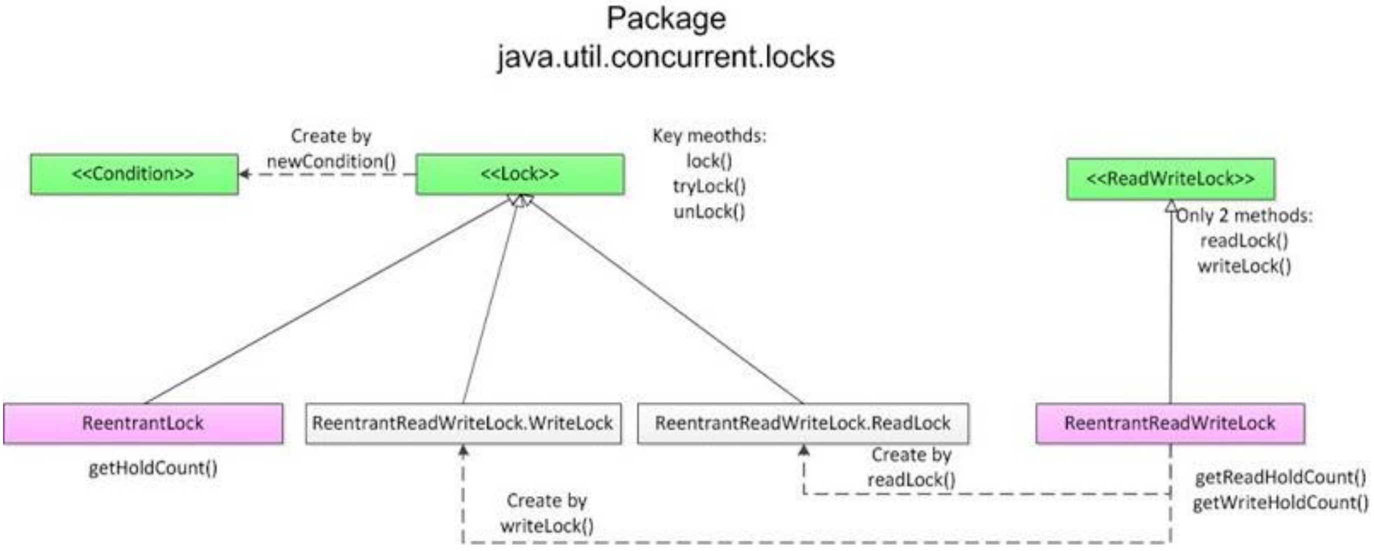
- java.lang.**Object**
  - java.util.concurrent.locks.**AbstractOwnableSynchronizer** (implements java.io.Serializable)
    - java.util.concurrent.locks.**AbstractQueuedLongSynchronizer** (implements java.io.Serializable)
    - java.util.concurrent.locks.**AbstractQueuedSynchronizer** (implements java.io.Serializable)
  - java.util.concurrent.locks.**AbstractQueuedLongSynchronizer.ConditionObject** (implements java.util.concurrent.locks.Condition, java.io.Serializable)
  - java.util.concurrent.locks.**AbstractQueuedSynchronizer.ConditionObject** (implements java.util.concurrent.locks.Condition, java.io.Serializable)
  - java.util.concurrent.locks.**LockSupport**
  - java.util.concurrent.locks.**ReentrantLock** (implements java.util.concurrent.locks.Lock, java.io.Serializable)
  - java.util.concurrent.locks.**ReentrantReadWriteLock** (implements java.util.concurrent.locks.ReadWriteLock, java.io.Serializable)
  - java.util.concurrent.locks.**ReentrantReadWriteLock.ReadLock** (implements java.util.concurrent.locks.Lock, java.io.Serializable)
  - java.util.concurrent.locks.**ReentrantReadWriteLock.WriteLock** (implements java.util.concurrent.locks.Lock, java.io.Serializable)

Interface Hierarchy

- java.util.concurrent.locks.**Condition**
- java.util.concurrent.locks.**Lock**
- java.util.concurrent.locks.**ReadWriteLock**

Some important interfaces and classes in Java Lock API are:

<table>
<tr><td><b>Lock</b>:</td><td>This is the base interface for Lock API. It provides all the features of synchronized keyword with additional ways to create different Conditions for locking, providing timeout for thread to wait for lock. Some of the important methods are lock() to acquire the lock, unlock() to release the lock, tryLock() to wait for lock for a certain period of time, newCondition() to create the Condition etc.</td></tr>
<tr><td><b>Condition</b>:</td><td>Condition objects are similar to Object wait-notify model with additional feature to create different sets of wait. A Condition object is always created by Lock object. Some of the important methods are await() that is similar to wait() and signal(), signalAll() that is similar to notify() and notifyAll() methods.</td></tr>
<tr><td><b>ReentrantLock</b>:</td><td>This is the most widely used implementation class of Lock interface. This class implements the Lock interface in similar way as synchronized keyword. Apart from Lock interface implementation, ReentrantLock contains some utility methods to get the thread holding the lock, threads waiting to acquire the lock etc.</td></tr>
<tr><td colspan="2">synchronized block are reentrant in nature i.e if a thread has lock on the monitor object and if another synchronized block requires to have the lock on the same monitor object then thread can enter that code block. I think this is the reason for the class name to be ReentrantLock.</td></tr>
</table>



ReentrancLock Methods	
Lock Interface methed	**Class method**
Void lock()	int get HoldCount()
Boolean tryLock()	boolean isHeldByCurrentThread()
Void tryLoc(long l,TimeUnit t)	int getQueLength()
Void lockInteruptibly()	Collection getQueuedThread()
Void lock()	boolean hasQueuedThread()
	boolean isFair()
	Thread getOwner

Example:

```
public class Test{

public synchronized foo(){
   //do something
   bar();
  }

  public synchronized bar(){
   //do some more
  }
}
```

If a thread enters foo(), it has the lock on Test object, so when it tries to execute bar() method, the thread is allowed to execute bar() method since it’s already holding the lock on the Test object i.e same as synchronized(this).

Java Lock (with synchronised)

Let’s say we have a Resource class with some operation where we want it to be thread-safe and some methods where thread safety is not required.

```
Public class Resource{

            public void doSomething(){
             //do some operation, DB read, write etc
             }

             public void doLogging(){
                         //logging, no need for thread safety
             }
}

public class LockExample implements Runnable{
             private Resource resource;
             public LockExample (Resource r){
                         this.resource = r;
             }

             @Override
             public void run() {
                         synchronized (resource) {
                                     resource.doSomething();
                         }
                         resource.doLogging();
             }
}
```

```
Pubic class test{
  Public static void main(String[] args)
 {
    Resoutce R=new Resoutce();

      LockExample l1=new LockExample(R);
      LockExample l2new LockExample(R);

L1.start();
L1.start();

  }
}
```

// note: we can not able to control lock if ,
Object enter into synchronized block.
Other thread may wait for long time();

Or Throws exception if fails in any point of time.

Java Lock (concurrentLock)

```
Public class Resource{

            public void doSomething(){
             //do some operation, DB read, write etc
             }

             public void doLogging(){
                         //logging, no need for thread safety
             }
}

public class ConcurrencyLock implements Runnable{

             private Resource resource;
             private Lock lock;

             public ConcurrencyLockExample(Resource r){
                         this.resource = r;
                         this.lock = new ReentrantLock();
             }

             @Override
public void run() {
        try {
             if(lock.tryLock(10, TimeUnit.SECONDS)){
                                     resource.doSomething();
                }
             } catch (InterruptedException e) {
                                     e.printStackTrace();
             }finally{
                                     //release lock
                                     lock.unlock();
                         }
                         resource.doLogging();
             }//run

}
```

```
Pubic class test{
   Public static void main(String[] args)
  {
     Resoutce R=new Resoutce();

       ConcurrencyLock l1=new ConcurrencyLock (R);
       ConcurrencyLock l2new ConcurrencyLock (R);

L1.start();
L1.start();

  }
}
```

→note:

using tryLock() method to make sure my thread waits only for definite time and if it’s not getting the lock on object, it’s just logging and exiting. Another important point to note is the use of try-finally block to make sure lock is released even if doSomething() method call throws any exception.

Java Lock vs synchronized

- Based on above details and program, we can easily conclude following differences between Java Lock and synchronization.
- Java Lock API provides more visibility and options for locking, unlike synchronized where a thread might end up waiting indefinitely for the lock, we can use tryLock() to make sure thread waits for specific time only.
- Synchronization code is much cleaner and easy to maintain whereas with Lock we are forced to have try-finally block to make sure Lock is released even if some exception is thrown between lock() and unlock() method calls.
- synchronization blocks or methods can cover only one method whereas we can acquire the lock in one method and release it in another method with Lock API.
- synchronized keyword doesn’t provide fairness whereas we can set fairness to true while creating ReentrantLock object so that longest waiting thread gets the lock first.
- We can create different conditions for Lock and different thread can await() for different conditions.

Producer and consumer problem with Lock, conditional lock(explicit lock) and Queue;

```
class Producer extends Thread {
    ProducerConsumerImpl pc;

    public Producer(ProducerConsumerImpl
sharedObject) {
        super("PRODUCER");
        this.pc = sharedObject;
    }

    @Override
    public void run() {
        try {
            pc.put();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class Consumer extends Thread {
    ProducerConsumerImpl pc;

    public Consumer(ProducerConsumerImpl
sharedObject) {
        super("CONSUMER");
        this.pc = sharedObject;
    }

    @Override
    public void run() {
        try {
            pc.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}


-----------------------------
public class Test {

    public static void main(String[] args) {

        ProducerConsumerImpl sharedObject = new

ProducerConsumerImpl();

        // creating producer and consumer threads
        Producer p = new Producer(sharedObject);
        Consumer c = new Consumer(sharedObject);

        // starting producer and consumer threads
        p.start();
        c.start();
    }

}
```

Output:-

```
CONSUMER : Buffer is empty, waiting
PRODUCER added 279133501 into queue
PRODUCER : Signalling that buffer is no more empty now
CONSUMER consumed 279133501 from queue
CONSUMER : Signalling that buffer may be empty now
```

```
class ProducerConsumerImpl {

    private static final int CAPACITY = 10;
    private final Queue queue = new LinkedList<>();
    private final Random theRandom = new Random();

    // lock and condition variables

    private final Lock lock = new ReentrantLock();
    private final Condition bufferNotFull = lock.newCondition();
    private final Condition bufferNotEmpty = lock.newCondition();

    public void put() throws InterruptedException {
        lock.lock();
        try {
            while (queue.size() == CAPACITY) {
                System.out.println(" Buffer is full, waiting");
                bufferNotEmpty.await();
            }

            int number = theRandom.nextInt();
            boolean isAdded = queue.offer(number);
            if (isAdded) {

     // signal form consumer thread
              System.out.println("added”+value+“into queue”);
                bufferNotFull.signalAll();
            }
        } finally {
            lock.unlock();
        }
    }

    public void get() throws InterruptedException {
        lock.lock();
        try {
            while (queue.size() == 0) {
                System.out.println(" Buffer is empty, waiting");
                bufferNotFull.await();
            }

            Integer value = queue.poll();
            if (value != null) {

              System.out.println("comsumed”+value+  “from queue");
                bufferNotEmpty.signalAll();
            }

        } finally {
            lock.unlock();
        }
    }
}
```

Thread Pools in Java

Background

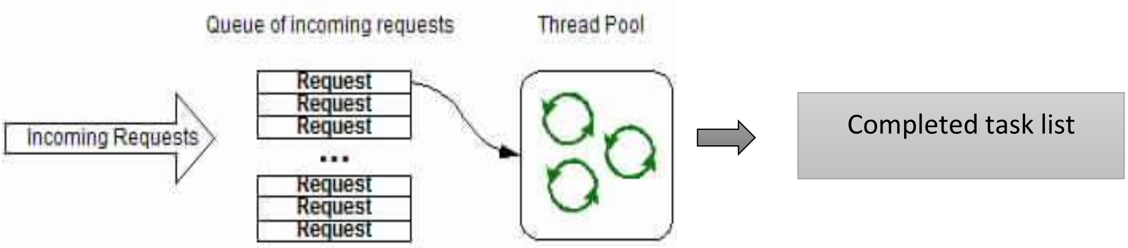
Server Programs such as database and web servers repeatedly execute requests from multiple clients and these are oriented around processing a large number of short tasks. An approach for building a server application would be to create a new thread each time a request arrives and service this new request in the newly created thread. While this approach seems simple to implement, it has significant disadvantages. A server that creates a new thread for every request would spend more time and consume more system resources in creating and destroying threads than processing actual requests.

Since active threads consume system resources, a JVM creating too many threads at the same time can cause the system to run out of memory. This necessitates the need to limit the number of threads being created.

ThreadPool

A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing.

- Java provides the Executor framework which is centered around the Executor interface, its sub-interface – **ExecutorService** and the class-**ThreadPoolExecutor**, which implements both of these interfaces. By using the executor, one only has to implement the Runnable objects and send them to the executor to execute.
- They allow you to take advantage of threading, but focus on the tasks that you want the thread to perform, instead of thread mechanics.
- To use thread pools, we first create a object of ExecutorService and pass a set of tasks to it. ThreadPoolExecutor class allows to set the core and maximum pool size.The runnables that are run by a particular thread are executed sequentially.



Thread pool example:-

```
class Task implements Runnable
{
    private String name;

    public Task(String s)
    {
        name = s;
    }

    public void run()
    {
        try
        {
            for (int i = 0; i<=5; i++)
            {
             System.out.println("thread
running"+name);
                Thread.sleep(1000);
            }
            System.out.println(name+" complete");
        }

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

```
public class Test
{
     // Maximum number of threads in thread pool
    static final int MAX_T = 3;

    public static void main(String[] args)
    {
        // creates five tasks
        Runnable r1 = new Task("task 1");
        Runnable r2 = new Task("task 2");
        Runnable r3 = new Task("task 3");
        Runnable r4 = new Task("task 4");
        Runnable r5 = new Task("task 5");

        ExecutorService pool =
Executors.newFixedThreadPool(MAX_T);
        pool.execute(r1);
        pool.execute(r2);
        pool.execute(r3);
        pool.execute(r4);
        pool.execute(r5);

        // pool shutdown
        pool.shutdown();
    }
}
```

Callable and Future in Java

The need for Callable

There are two ways of creating threads – one by extending the Thread class and other by creating a thread with a Runnable. However, one feature lacking in Runnable is that we cannot make a thread return result when it terminates, i.e. when run() completes. For supporting this feature, the Callable interface is present in Java.

Callable vs Runnable

- For implementing Runnable, the run() method needs to be implemented which does not return anything, while for a Callable, the call() method needs to be implemented which returns a result on completion. Note that a thread can't be created with a Callable, it can only be created with a Runnable.
- Another difference is that the call() method can throw an exception whereas run() cannot.

```
public Object call() throws Exception; →callable method
```

Future

When the call() method completes, answer must be stored in an object known to the main thread, so that the main thread can know about the result that the thread returned. How will the program store and obtain this result later? For this, a Future object can be used. Think of a Future as an object that holds the result – it may not hold it right now, but it will do so in the future (once the Callable returns). Thus, a Future is basically one way the main thread can keep track of the progress and result from other threads

Observe that Callable and Future do two different things – Callable is similar to Runnable, in that it encapsulates a task that is meant to run on another thread, whereas a Future is used to store a result obtained from a different thread. In fact, the Future can be made to work with Runnable as well, which is something that will become clear when Executors come into the picture.

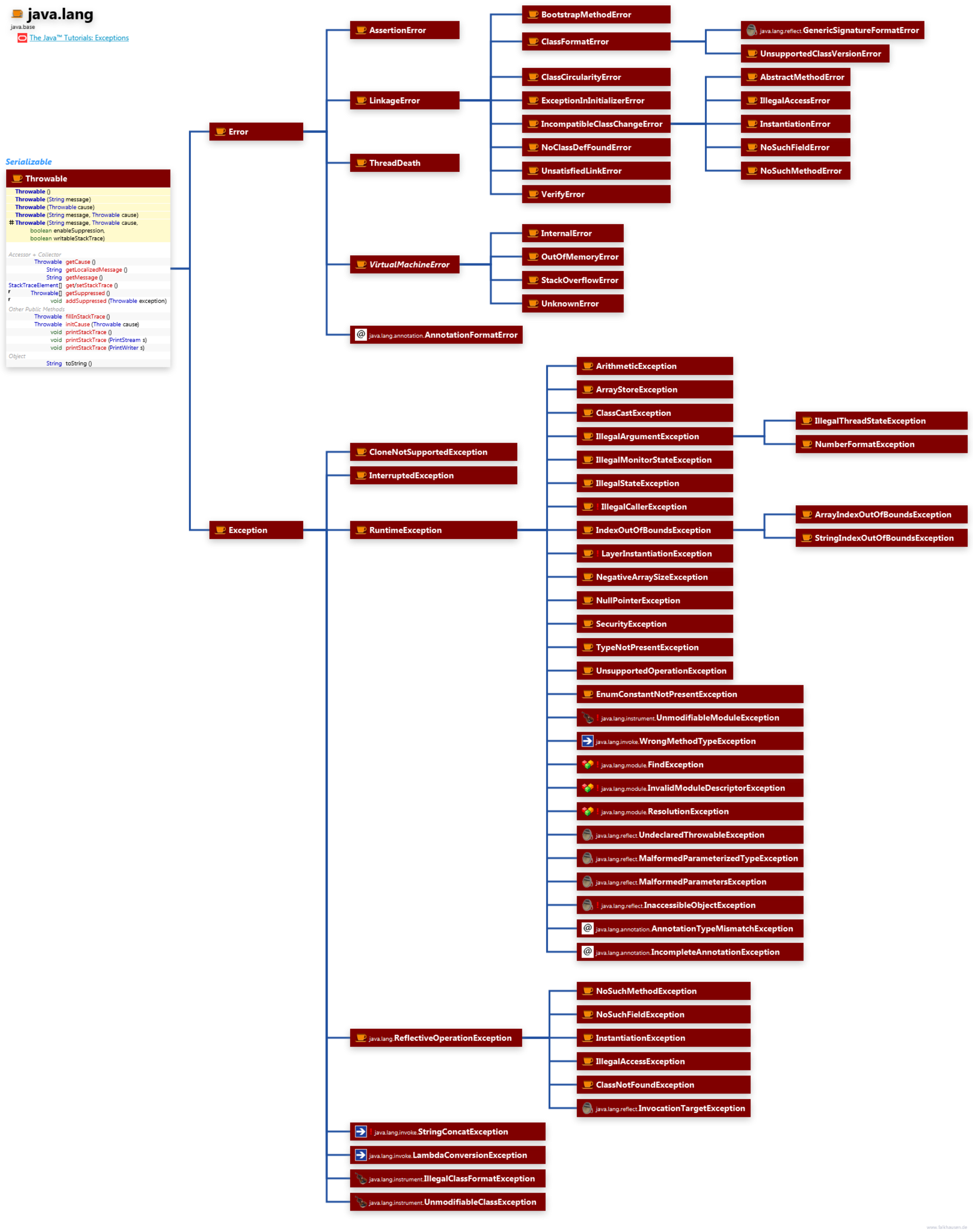
- **public boolean cancel(boolean mayInterrupt):** Used to stop the task. It stops the task if it has not started. If it has started, it interrupts the task only if mayInterrupt is true.
- **public Object get() throws InterruptedException, ExecutionException:** Used to get the result of the task. If the task is complete, it returns the result immediately, otherwise it waits till the task is complete and then returns the result.
- **public boolean isDone():** Returns true if the task is complete and false otherwise

```
class CallableExample implements Callable
{

  public Object call() throws Exception
  {
    Random generator = new Random();
    Integer randomNumber = generator.nextInt(5);

        Thread.sleep(randomNumber * 1000);
        return randomNumber;
 }

}
```

```
public class CallableFutureTest
{
  public static void main(String[] args) throws Exception
  {

    FutureTask[] randomNumberTasks = new FutureTask[5];

    for (int i = 0; i < 5; i++)
    {
      Callable callable = new CallableExample();

      // Create the FutureTask with Callable
     randomNumberTasks[i] = new FutureTask(callable);

      Thread t = new Thread(randomNumberTasks[i]);
      t.start();
    }

    for (int i = 0; i < 5; i++)
    {
      // As it implements Future, we can call get()
      S.o.p(randomNumberTasks[i].get());


    }
  }
}
```

Java Throwable

The **Throwable** class is the superclass of all errors and exceptions in the **Java** language. Only objects that are instances of this class (or one of its subclasses) are thrown by the **Java** Virtual Machine or can be thrown by the **Java** throw statement.


java.lang
java.base
The Java™ Tutorials: Exceptions
Serializable
Throwable
Throwable ()
Throwable (String message)
Throwable (Throwable cause)
Throwable (String message, Throwable cause)
# Throwable (String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace)
Accessor + Collector
Throwable getCause ()
String getLocalizedMessage ()
String getMessage ()
StackTraceElement[] get/setStackTrace ()
F Throwable[] getSuppressed ()
F void addSuppressed (Throwable exception)
Other Public Methods
Throwable fillInStackTrace ()
Throwable initCause (Throwable cause)
void printStackTrace ()
void printStackTrace (PrintStream s)
void printStackTrace (PrintWriter s)
Object
String toString ()
Error
AssertionError
LinkageError
ThreadDeath
VirtualMachineError
java.lang.annotation.AnnotationFormatError
BootstrapMethodError
ClassFormatError
ClassCircularityError
ExceptionInInitializerError
IncompatibleClassChangeError
NoClassDefFoundError
UnsatisfiedLinkError
VerifyError
java.lang.reflect.GenericSignatureFormatError
UnsupportedClassVersionError
AbstractMethodError
IllegalAccessError
InstantiationError
NoSuchFieldError
NoSuchMethodError
InternalError
OutOfMemoryError
StackOverflowError
UnknownError
Exception
CloneNotSupportedException
InterruptedException
RuntimeException
ArithmeticException
ArrayStoreException
ClassCastException
IllegalArgumentException
IllegalMonitorStateException
IllegalStateException
IllegalCallerException
IndexOutOfBoundsException
LayerInstantiationException
NegativeArraySizeException
NullPointerException
SecurityException
TypeNotPresentException
UnsupportedOperationException
EnumConstantNotPresentException
java.lang.instrument.UnmodifiableModuleException
java.lang.invoke.WrongMethodTypeException
java.lang.module.FindException
java.lang.module.InvalidModuleDescriptorException
java.lang.module.ResolutionException
java.lang.reflect.UndeclaredThrowableException
java.lang.reflect.MalformedParameterizedTypeException
java.lang.reflect.MalformedParametersException
java.lang.reflect.InaccessibleObjectException
java.lang.annotation.AnnotationTypeMismatchException
java.lang.annotation.IncompleteAnnotationException
IllegalThreadStateException
NumberFormatException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException
java.lang.ReflectiveOperationException
NoSuchMethodException
NoSuchFieldException
InstantiationException
IllegalAccessException
ClassNotFoundException
java.lang.reflect.InvocationTargetException
java.lang.invoke.StringConcatException
java.lang.invoke.LambdaConversionException
java.lang.instrument.IllegalClassFormatException
java.lang.instrument.UnmodifiableClassException
www.falkhausen.de

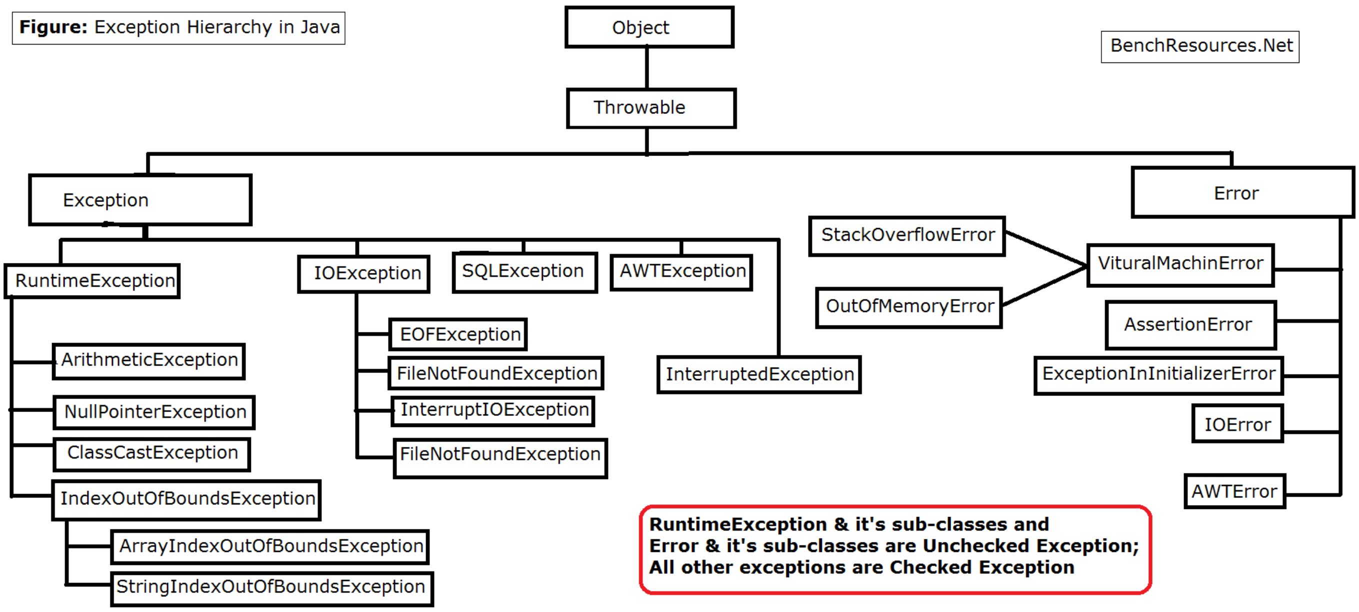


Figure: Exception Hierarchy in Java

RuntimeException & it's sub-classes and Error & it's sub-classes are Unchecked Exception; All other exceptions are Checked Exception

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception.

Throw exception

```
------------------------------------
public class TestThrow1{
   static void validate(int age){
     if(age<18)
      throw new ArithmeticException("not valid");
     else
      System.out.println("welcome to vote");
   }
   public static void main(String args[]){
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

Throw custom exception

```
------------------------------------
class InvalidAgeException extends Exception{
 InvalidAgeException(String s){
  super(s);
 }
}

---------------------
class TestCustomException1{
   static void validate(int age)throws InvalidAgeException{
     if(age<18)
      throw new InvalidAgeException("not valid");
     else
      System.out.println("welcome to vote");
   }
   public static void main(String args[]){
      try{
      validate(13);
      }catch(Exception m){System.out.println("Exception occured: "+m);}
        System.out.println("rest of the code...");
  }
}
```

Java.util package

Class Hierarchy

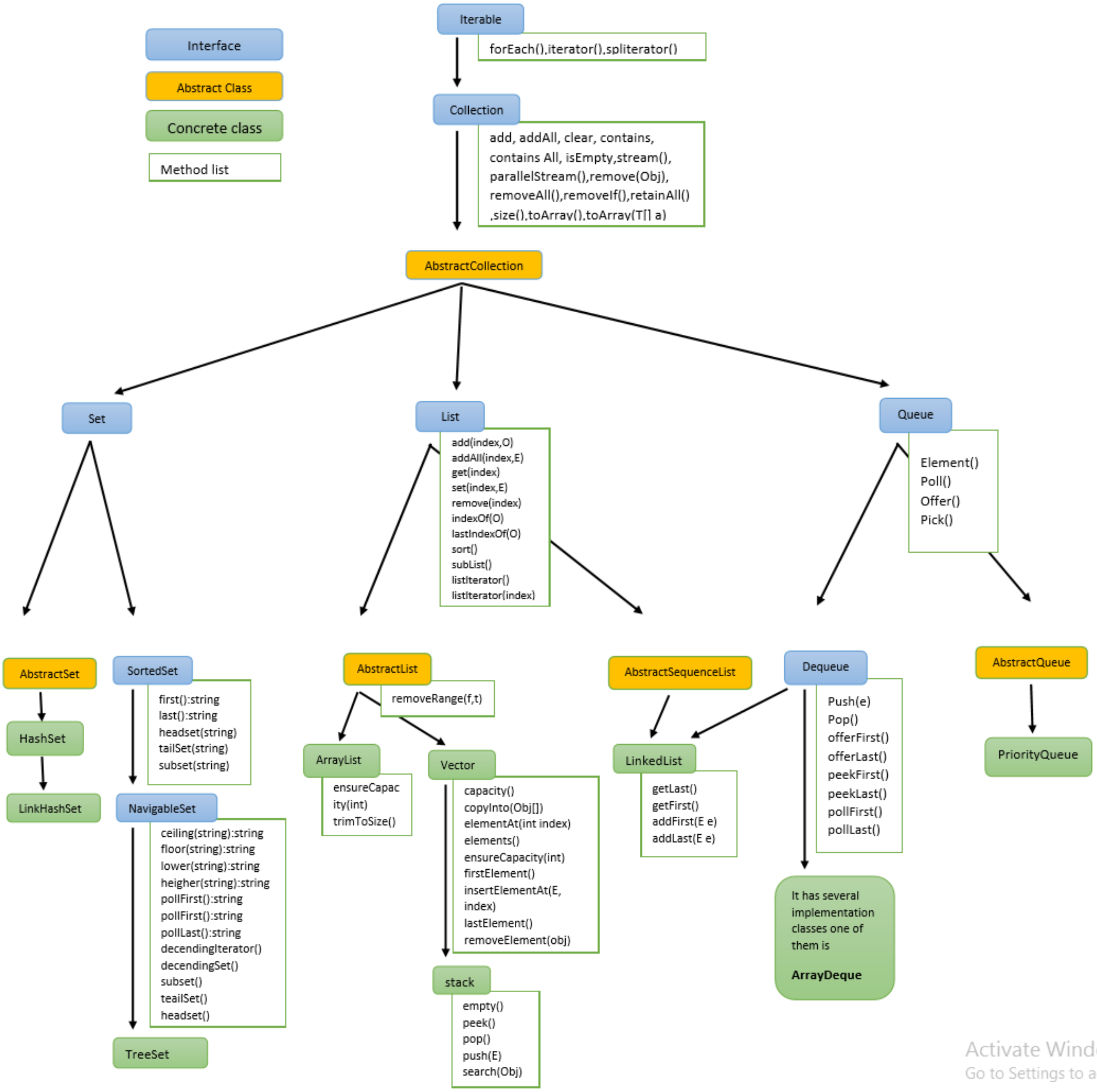
- java.lang.**Object**
    - java.util.**AbstractCollection**<E> (implements java.util.Collection<E>)
        - java.util.**AbstractList**<E> (implements java.util.List<E>)
            - java.util.**AbstractSequentialList**<E>
                - java.util.**LinkedList**<E> (implements java.lang.Cloneable, java.util.Deque<E>, java.util.List<E>, java.io.Serializable)
            - java.util.**ArrayList**<E> (implements java.lang.Cloneable, java.util.List<E>, java.util.RandomAccess, java.io.Serializable)
            - java.util.**Vector**<E> (implements java.lang.Cloneable, java.util.List<E>, java.util.RandomAccess, java.io.Serializable)
                - java.util.**Stack**<E>
        - java.util.**AbstractQueue**<E> (implements java.util.Queue<E>)
            - java.util.**PriorityQueue**<E> (implements java.io.Serializable)
        - java.util.**AbstractSet**<E> (implements java.util.Set<E>)
            - java.util.**EnumSet**<E> (implements java.lang.Cloneable, java.io.Serializable)
            - java.util.**HashSet**<E> (implements java.lang.Cloneable, java.io.Serializable, java.util.Set<E>)
                - java.util.**LinkedHashSet**<E> (implements java.lang.Cloneable, java.io.Serializable, java.util.Set<E>)
            - java.util.**TreeSet**<E> (implements java.lang.Cloneable, java.util.NavigableSet<E>, java.io.Serializable)
        - java.util.**ArrayDeque**<E> (implements java.lang.Cloneable, java.util.Deque<E>, java.io.Serializable)
    - java.util.**AbstractMap**<K,V> (implements java.util.Map<K,V>)
        - java.util.**EnumMap**<K,V> (implements java.lang.Cloneable, java.io.Serializable)
        - java.util.**HashMap**<K,V> (implements java.lang.Cloneable, java.util.Map<K,V>, java.io.Serializable)
            - java.util.**LinkedHashMap**<K,V> (implements java.util.Map<K,V>)
        - java.util.**IdentityHashMap**<K,V> (implements java.lang.Cloneable, java.util.Map<K,V>, java.io.Serializable)
        - java.util.**TreeMap**<K,V> (implements java.lang.Cloneable, java.util.NavigableMap<K,V>, java.io.Serializable)
        - java.util.**WeakHashMap**<K,V> (implements java.util.Map<K,V>)
    - java.util.**AbstractMap.SimpleEntry**<K,V> (implements java.util.Map.Entry<K,V>, java.io.Serializable)
    - java.util.**AbstractMap.SimpleImmutableEntry**<K,V> (implements java.util.Map.Entry<K,V>, java.io.Serializable)
    - java.util.**Arrays**
    - java.util.**Base64**
    - java.util.**Base64.Decoder**
    - java.util.**Base64.Encoder**
    - java.util.**BitSet** (implements java.lang.Cloneable, java.io.Serializable)
    - java.util.**Calendar** (implements java.lang.Cloneable, java.lang.Comparable<T>, java.io.Serializable)
        - java.util.**GregorianCalendar**
    - java.util.**Calendar.Builder**
    - java.util.**Collections**
    - java.util.**Currency** (implements java.io.Serializable)
    - java.util.**Date** (implements java.lang.Cloneable, java.lang.Comparable<T>, java.io.Serializable)
    - java.util.**Dictionary**<K,V>
        - java.util.**Hashtable**<K,V> (implements java.lang.Cloneable, java.util.Map<K,V>, java.io.Serializable)
            - java.util.**Properties**
    - java.util.**DoubleSummaryStatistics** (implements java.util.function.DoubleConsumer)
    - java.util.**EventListenerProxy**<T> (implements java.util.EventListener)
    - java.util.**EventObject** (implements java.io.Serializable)
    - java.util.**FormattableFlags**
    - java.util.**Formatter** (implements java.io.Closeable, java.io.Flushable)
    - java.util.**IntSummaryStatistics** (implements java.util.function.IntConsumer)
    - java.util.**Locale** (implements java.lang.Cloneable, java.io.Serializable)
    - java.util.**Locale.Builder**
    - java.util.**Locale.LanguageRange**
    - java.util.**LongSummaryStatistics** (implements java.util.function.IntConsumer, java.util.function.LongConsumer)
    - java.util.**Objects**
    - java.util.**Observable**
    - java.util.**Optional**<T>
    - java.util.**OptionalDouble**

  - java.util.**OptionalInt**
  - java.util.**OptionalLong**
  - java.security.**Permission** (implements java.security.Guard, java.io.Serializable)
    - java.security.**BasicPermission** (implements java.io.Serializable)
      - java.util.**PropertyPermission**
  - java.util.**Random** (implements java.io.Serializable)
  - java.util.**ResourceBundle**
    - java.util.**ListResourceBundle**
    - java.util.**PropertyResourceBundle**
  - java.util.**ResourceBundle.Control**
  - java.util.**Scanner** (implements java.io.Closeable, java.util.Iterator<E>)
  - java.util.**ServiceLoader**<S> (implements java.lang.Iterable<T>)
  - java.util.**Spliterators**
  - java.util.**Spliterators.AbstractDoubleSpliterator** (implements java.util.Spliterator.OfDouble)
  - java.util.**Spliterators.AbstractIntSpliterator** (implements java.util.Spliterator.OfInt)
  - java.util.**Spliterators.AbstractLongSpliterator** (implements java.util.Spliterator.OfLong)
  - java.util.**Spliterators.AbstractSpliterator**<T> (implements java.util.Spliterator<T>)
  - java.util.**SplittableRandom**
  - java.util.**StringJoiner**
  - java.util.**StringTokenizer** (implements java.util.Enumeration<E>)
  - java.util.**Timer**
  - java.util.**TimerTask** (implements java.lang.Runnable)
  - java.util.**TimeZone** (implements java.lang.Cloneable, java.io.Serializable)
    - java.util.**SimpleTimeZone**
  - java.util.**UUID** (implements java.lang.Comparable<T>, java.io.Serializable)

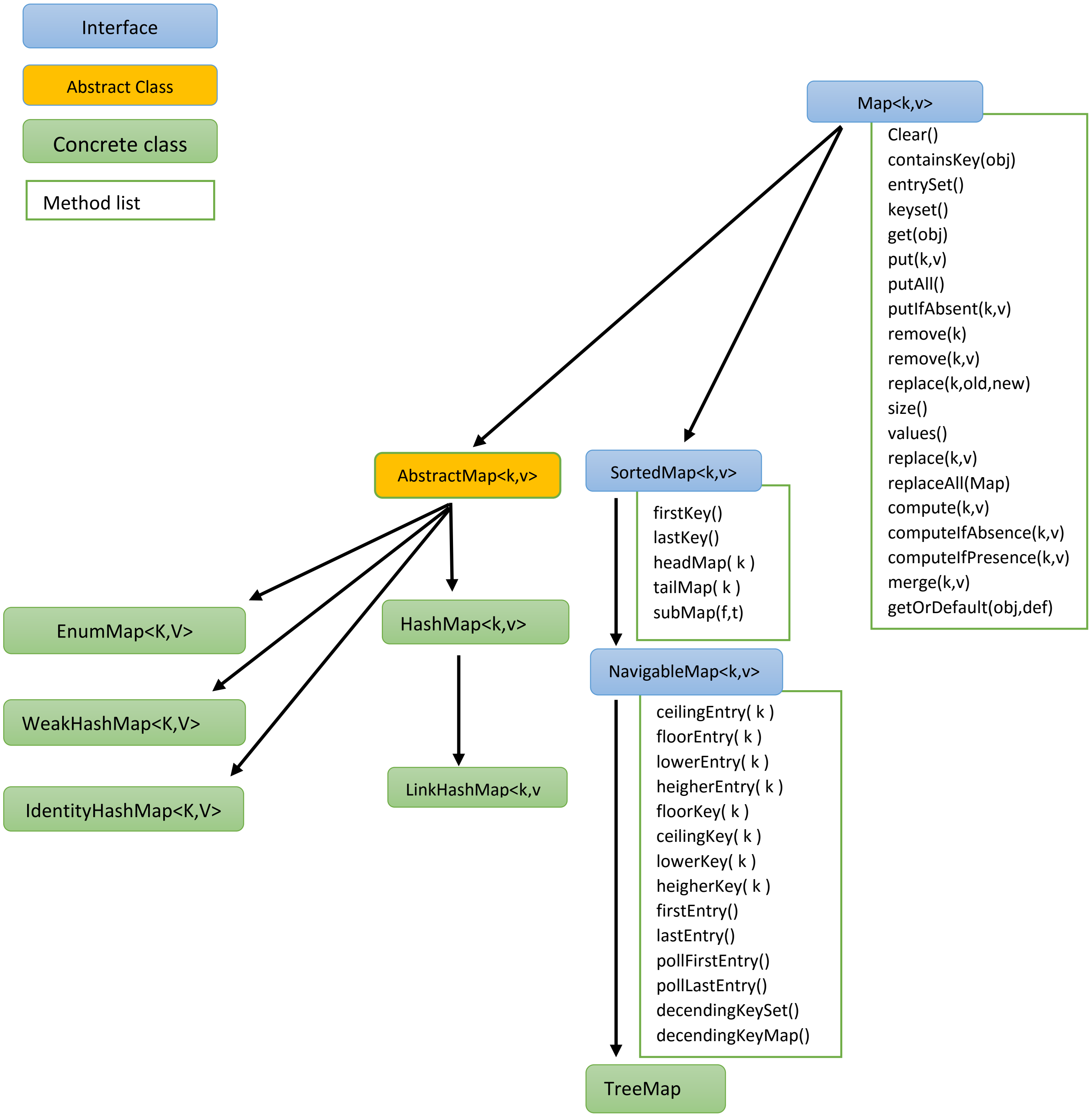
Interface Hierarchy

- java.util.**Comparator**<T>
- java.util.**Enumeration**<E>
- java.util.**EventListener**
- java.util.**Formattable**
- java.lang.**Iterable**<T>
  - java.util.**Collection**<E>
    - java.util.**List**<E>
    - java.util.**Queue**<E>
      - java.util.**Deque**<E>
    - java.util.**Set**<E>
      - java.util.**SortedSet**<E>
        - java.util.**NavigableSet**<E>
- java.util.**Iterator**<E>
  - java.util.**ListIterator**<E>
  - java.util.**PrimitiveIterator**<T,T_CONS>
    - java.util.**PrimitiveIterator.OfDouble**
    - java.util.**PrimitiveIterator.OfInt**
    - java.util.**PrimitiveIterator.OfLong**
- java.util.**Map**<K,V>
  - java.util.**SortedMap**<K,V>
    - java.util.**NavigableMap**<K,V>
- java.util.**Map.Entry**<K,V>
- java.util.**Observer**
- java.util.**RandomAccess**
- java.util.**Spliterator**<T>
  - java.util.**Spliterator.OfPrimitive**<T,T_CONS,T_SPLITR>
    - java.util.**Spliterator.OfDouble**
    - java.util.**Spliterator.OfInt**
    - java.util.**Spliterator.OfLong**

JAVA Collection



Java Map



Interface

Abstract Class

Concrete class

Method list

Map<k,v>

Clear()
containsKey(obj)
entrySet()
keyset()
get(obj)
put(k,v)
putAll()
putIfAbsent(k,v)
remove(k)
remove(k,v)
replace(k,old,new)
size()
values()
replace(k,v)
replaceAll(Map)
compute(k,v)
computeIfAbsence(k,v)
computeIfPresence(k,v)
merge(k,v)
getOrDefault(obj,def)

AbstractMap<k,v>

SortedMap<k,v>

firstKey()
lastKey()
headMap(k)
tailMap(k)
subMap(f,t)

EnumMap<K,V>

HashMap<k,v>

WeakHashMap<K,V>

IdentityHashMap<K,V>

LinkHashMap<k,v

NavigableMap<k,v>

ceilingEntry(k)
floorEntry(k)
lowerEntry(k)
heigherEntry(k)
floorKey(k)
ceilingKey(k)
lowerKey(k)
heigherKey(k)
firstEntry()
lastEntry()
pollFirstEntry()
pollLastEntry()
decendingKeySet()
decendingKeyMap()

TreeMap

Collection Example

```
                                   ---------List Example-----

List<String> list = Arrays.asList("Lars", "Simon");
       or
List<String> anotherList = new ArrayList<>();
anotherList.add("Lars");
anotherList.add("Simon");

list.forEach(System.out::println);                    // print with method references

      or
for (String string : list) {
      System.out.println(string);
}

// some methods

l1.sort(null);   // natural sorting
l2.sort((s1, s2) -> s1.compareToIgnoreCase(s2));    //sorting using lambda expression
l1.removeIf(s-> s.toLowerCase().contains("x”)      // remove containing element from list

                                   ---------map Example---

Map<String, String> map = new HashMap<>();
map.put("iPhone", "Created by Apple");
map.put("Android", "Mobile");

// some methods
map.remove("Android");
 int k = map.getOrDefault("Android", "default");
 Integer calculatedVaue = map.computeIfAbsent("Java", it -> 0);
 String[] strings = map.keySet().toArray(new String[map.keySet().size()]);    // map keys in string array[]
 List<String> list = new ArrayList<String>(map.keySet());                          // map keys in list

                                   --------ways of iterating map--------
1. iterator
                          Iterator<Map.Entry<String, String>> itr = map.entrySet().iterator();
                               while(itr.hasNext())
                               {
                                    Map.Entry<String, String> entry = itr.next();
                                    System.out.println("Key = " + entry.getKey() +
                                                    ", Value = " + entry.getValue());
                               }
2.forEach
                           map.forEach((k, v) -> System.out.printf("%s %s%n", k, v));

3.entrySet
                          for (Map.Entry<String,String> entry : map.entrySet())
                                 System.out.println("Key = " + entry.getKey() +
                                            ", Value = " + entry.getValue());
                            }
4.keySet
                          for (String name : map.keySet())  {
                                   System.out.println("key: " + name+"Value"+map.get(key));
                          }

5.getValues
                               for (String url : map.values())
                                 System.out.println("value: " + url);
                            }
```

Java.util.Collections

Collections class in java is a useful utility class to work with collections in java. The java.util.Collections class directly extends the Object class and exclusively consists of the static methods that operate on Collections or return them.

Collections class contains polymorphic algorithms that operate on collections and “wrappers”

Collections class fields

Collections class contains 3 fields: EMPTY_LIST, EMPTY_SET, EMPTY_MAP, which can be used to get immutable empty List, Map and Set respectively.

```
List<Object > list=Collections.EMPTY.LIST  // return an empty list
```

boolean addAll(Collection c, T... elements)	List fruits = new ArrayList();
	Collections.addAll(fruits, "Apples", "Oranges", "Banana");
	fruits.forEach(System.out::println);

void sort(List list, Comparator c)	Collections.sort(fruits); // natural ordering
	Collections.sort(fruits,null); // natural ordering
	Collections.sort(fruits, Comparator.reverseOrder());
	fruits.forEach(System.out::println);

Queue asLifoQueue(Deque deque)	Deque deque = new LinkedList();
	deque.addFirst("Apples");
	deque.add("Oranges");
	deque.addLast("Bananas");
	
	Queue queue = **Collections.asLifoQueue(deque);**
	System.out.println(queue.poll());
	System.out.println(queue.poll());
	System.out.println(queue.poll());

int binarySearch(List<? extends Comparable> list, T key)	**Collections.binarySearch(fruits, "Banana");** //return 1 if exist else –1

Collection checkedCollection(Collection c, Class type)	List list = new ArrayList();
	Collections.addAll(list, "one", "two", "three", "four");
	Collection checkedList = Collections.checkedCollection(list, String.class);
	//we can add any type of element to list
	list.add(10);
	//we cannot add any type of elements to chkList, doing so
	//throws ClassCastException
	checkedList.add(10);

void copy(List dest, List src)	**Collections.copy(list, fruits);**
	list.forEach(System.out::println);

boolean disjoint(Collection c1, Collection c2)	**Collections.disjoint(list, fruits)** // return true if no element found common
void fill(List list, T obj)	Collections.fill(list, "apple"); // fill list with ‘apple’
int frequency(Collection c, Object o)	**Collections.frequency(list,”apple”)** // it will return count of same element in list – 4,5..

int indexOfSubList(List source, List target)	List fruitsSubList1 = new ArrayList();
	Collections.addAll(fruitsSubList1, "Oranges", "Banana");
	Collections.indexOfSubList(fruits, fruitsSubList1)
	// get index of sublist in a list ,if not found return -1

Collection<T> synchronizedCollection(Collection<T> c)	**Collection<String> synchronizedCollection =**
	  Collections.synchronizedCollection(fruits);
	List<String> synchronizedList =
	Collections.synchronizedList(fruits);

Also, There are **synchronizedMap**, **synchronizedSet**, **synchronizedSortedSet** as well as **synchronizedSortedMap** methods available which do the similar job. These are the frequently used methods, apart from which we also have other methods like **newSetFromMap**, **replaceAll**, **swap**, **reverse**, etc. Note that all the methods of this class throw a **NullPointerException** if the collections or class objects provided to them are null.	

Java.util.Arrays

The **java.util.Arrays** class contains a static factory that allows arrays to be viewed as lists.

- This class contains various static methods for manipulating arrays (such as sorting and searching).
- The methods in this class throw a NullPointerException if the specified array reference is null.

```
int[] baseArray = { 2, 4, 3, 7, 21, 9, 98, 76, 74 };
List<Integer> integerList = Arrays.asList(baseArray);  0r Arrays.asList(2,3,5,8..);
                         int idx =Arrays.binarySearch(baseArray, 21);
List<Integer> integerList = Arrays.sort(baseArray, 1, 6);
         int[] copyOfArray = Arrays.copyOf(baseArray, 11); // default size of new array 11
int[] copyOfRangeArray  = Arrays.copyOfRange(baseArray, 5, 8);

int[] fillArray = new int[5];
                                   Arrays.fill(fillArray, 1); //fill all index with 1
                                   Arrays.fill(fillArray, 1, 7, 3); // fill 1-7 index with 3
         System.out.println(Arrays.toString());
```

Note:- There are several Overloaded method ,above list are unique one out of them.

Sorting using external comparator

```
class Sortbyroll implements
Comparator<Student>
{
   public int compare(Student a, Student b)
   {
      return a.rollno - b.rollno;
   }
}
```

```
ArrayList<Student> ar = new
ArrayList<Student>();
ar.add(new Student(111, "bbbb", "london"));
ar.add(new Student(131, "aaaa", "nyc"));
Collections.sort(ar, new Sortbyroll());
```

Or using lambda exp

```
List<Integer> list = Arrays.asList(5,4,3,7,2,1);

// custom comparator
Collections.sort(list, (o1, o2) -> (o1>o2 ? -1 : (o1==o2 ? 0 : 1)));
//print list with method reference
list.forEach(System.out::println);
```

Java.util.Observable

java.util.Observable is used to create subclasses that other parts of the program can observe. When an object of such subclass undergoes a change, observing classes are notified. The update( ) method is called when an observer is notified of a change.

An object that is being observed must follow two simple rules :

- If it is changed, it must call setChanged( ) method.
- When it is ready to notify observers of this change, it must call notifyObservers( ) method. This causes the update( ) method in the observing object(s) to be called.

Methods of Observable :-
addObserver(Observer observer) ,setChanged(), clearChanged(),notifyObservers(),notifyObservers(Object obj),countObservers( ) ,deleteObserver(Observer observer),deleteObservers()

Example:-

```
// First observer
class FirstNewsReader implements Observer {

    public void update(Observable obj, Object arg) {
        System.out.println("FirstNewsReader got The
news:"+(String)arg);
    }

}

//Second Observer
class SecondNewsReader implements Observer {

    public void update(Observable obj, Object arg) {
        System.out.println("SecondNewsReader got The
news:"+(String)arg);
    }

}
 -------------------
```

Output :-

```
SecondNewsReader got The news:News 1
FirstNewsReader got The news:News 1
SecondNewsReader got The news:News 2
FirstNewsReader got The news:News 2
SecondNewsReader got The news:News 3
FirstNewsReader got The news:News 3
```

```
// This is the class being observed.
class News extends Observable {
    void news() {
        String[] news = {"News 1", "News 2", "News 3"};
        for(String s: news){
                setChanged();  //set change
                 // notifyObservers()
               notifyObservers(s); //notify observers
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("Error Occurred.");
            }
        }
    }
}
//-------------------------------
class ObserverObservableDemo {
    public static void main(String args[]) {
        News observedNews = new News();
        FirstNewsReader observer1= new FirstNewsReader();
       SecondNewsReader observer2 = new SecondNewsReader();
        observedNews.addObserver(observer1);
        observedNews.addObserver(observer2);
       //  count = observedNews.countObservers();
       //  observedNews.deleteObserver(observer1);
        observedNews.news();
    }
}
```

Functional interface supplier and consumer:-

```
@FunctionalInterface
public interface Supplier<T>{
T get();
}
```

Example-1 use of supplier

```
String s=new String(new Supplier<String>() {
    @Override
    public String get() {
        return String.valueOf(System.currentTimeMillis());
    }
});
```

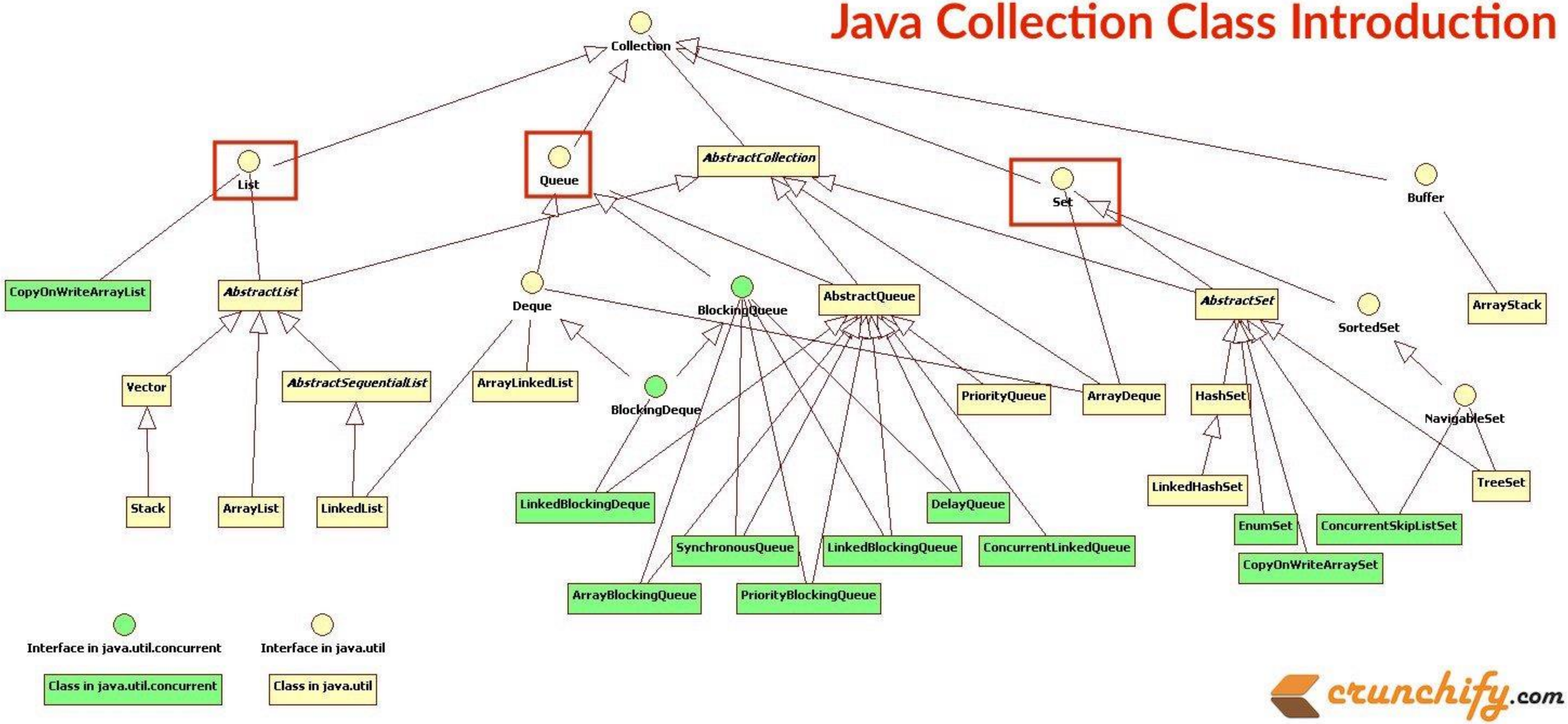
```
String s=new String(
       ()->  return String.valueOf(System.currentTimeMillis());
  );
```

```
@FunctionalInterface
public interface Consumer<T>{
void       accept(T t);
default Consumer<T> andThen(Consumer<? super T> after);
}
```

Example-2 use of consumer

```
        Consumer<String> display = a -> S.o.p (a);  // Consumer to display a String
        display.accept(s);  // Implement display using accept()

      // Consumer to multiply 2 to every integer of a list
        Consumer<List<Integer> > modify = list ->
                                                    {
                                                      for (int i = 0; i < list.size(); i++)
                                                          list.set(i, 2 * list.get(i));
                                                    };

        // Consumer to display a list of integers
        Consumer<List<Integer> >    dispList = list -> list.stream().forEach(a -> S.o.p(a + " "));
        List<Integer> list = new ArrayList<Integer>();
        list.add(2);
        list.add(1);
        list.add(3);

        modify.andThen(dispList).accept(list);  // using addThen()
```

Java Collection Class Introduction



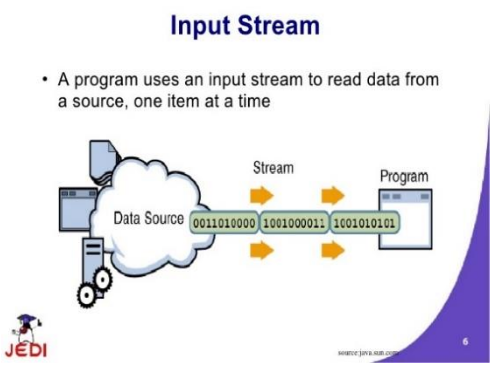
JAVA.IO

Java IO is an API that comes with Java which is targeted at reading and writing data (input and output). Most applications need to process some input and produce some output based on that input. For instance, read data from a file or over network, and write to a file or write a response back over the network.

Java NIO - The Alternative IO API

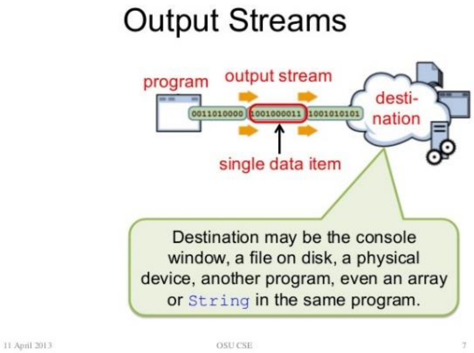
Java also contains another IO API called Java NIO. It contains classes that does much of the same as the Java IO and Java Networking APIs, but Java NIO can work in non-blocking mode. Non-blocking IO can in some situations give a big performance boost over blocking IO.

	Byte Based		Character Based	
	Input	Output	Input	Output
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
Files	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWriter
Pipes	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
Filtering	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
Parsing	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
Strings			StringReader	StringWriter
Data	DataInputStream	DataOutputStream		
Data - Formatted		PrintStream		PrintWriter
Objects	ObjectInputStream	ObjectOutputStream		
Utilities	SequenceInputStream			



Input Stream

- A program uses an input stream to read data from a source, one item at a time



Output Streams

Destination may be the console window, a file on disk, a physical device, another program, even an array or `String` in the same program.

OutputStream

Java InputStream's are used for writing byte based data, one byte at a time

```
OutputStream fout= new
FileOutputStream("D:\\testout.txt");

String s="Welcome to java.";
byte b[]=s.getBytes();//converting string into byte array
      try{
             fout.write(b);
             fout.close();
             System.out.println("success...");
      }
       catch(Exception e){System.out.println(e);}
```

Testout.txt-welcome to java

InputSteam

Java InputStream's are used for reading byte based data, one byte at a time.

```
InputStream fin  = new
FileInputStream("D:\\testout.txt");

 int i=0;
try{
         while((fin.read())!=-1){
          System.out.print((char) fin.read());
     fin.close();

}
 catch(Exception e){System.out.println(e);}
```

Output: welcome to java

ByteArrayOutputStream

Java ByteArrayOutputStream class is used to write common data into multiple files. In this stream, the data is written into a byte array which can be written to multiple streams later.

```
FileOutputStream fout1=new FileOutputStream("D:\\f1.txt");
 FileOutputStream fout2=new FileOutputStream("D:\\f2.txt");

String s="Welcome to java.";
byte b[]=s.getBytes();

ByteArrayOutputStream bout=new ByteArrayOutputStream();
      bout.write(b);
      bout.writeTo(fout1);     // write in file one
      bout.writeTo(fout2);     // write on file two

      bout.flush();
      bout.close();
```

Testout.txt-welcome to java

ByteArrayInputStream

it can be used to read byte array as input

```
byte[] buf = { 35, 36, 37, 38 };
ByteArrayInputStream byt = new ByteArrayInputStream(buf);
                         Or from file
ByteArrayInputStream bin = new
ByteArrayInputStream("D:\\f1.txt");

    int k = 0;
    while ((k = byt.read()) != -1) {
      //Conversion of a byte into character
      char ch = (char) k;
      System.out.println(Special character is: " + ch);
    }
```

Output: welcome to java

OutputStreamWriter

OutputStreamWriter is a class which is used to convert character stream to byte stream, the characters are encoded into byte using a specified charset.

```
try {
     OutputStream outputStream = new FileOutputStream("file.txt");
     Writer outputStreamWriter = new OutputStreamWriter(outputStream);

     outputStreamWriter.write("welcome to java");

       outputStreamWriter.close();
       } catch (Exception e) {
           e.getMessage();
   }
```

Testout.txt-welcome to java

InputStreamReader

An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset.

```
try  {
            InputStream stream = new FileInputStream("file.txt");
            Reader reader = new InputStreamReader(stream);
            int data = reader.read();
            while (data != -1) {
                System.out.print((char) data);
                data = reader.read();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
```

Output: welcome to java

5.Advance java

Core java	Advance java
To develop general purpose applications.	To develop online application and mobile application.
Without Core Java no one can develop any advanced java applications.	Where as advanced java only deals with some specialization like Database, DOM(web), networking etc.
OOP, data types, operators, functions, loops, exception handling, threading etc.	Apart from the core java parts it has some specific sections like database connectivity, web services, servlets etc.
It uses only one tier architecture that is why it is called as ‘stand alone’ application.	It uses two tier architecture i.e. client side architecture and server side or backend architecture.
Core java programming covers the swings, socket, awt, thread concept, collection object and classess.	Advance java is used for web based application and enterprise applic

-: Interview Questions:-