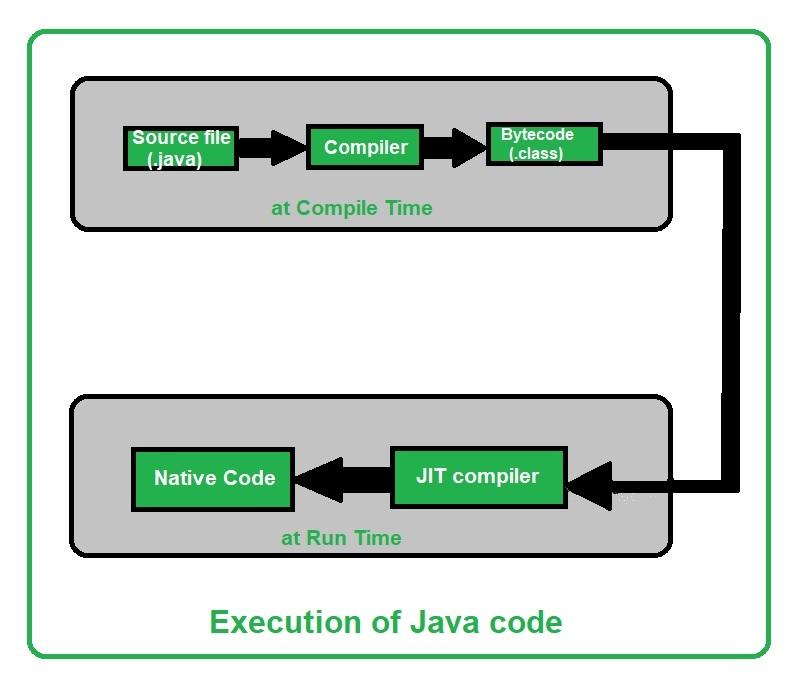
fast, reliable and secure

desktop to web applications, scientific supercomputers to gaming consoles, cell phones to the Internet, Java is used in every nook and corner.

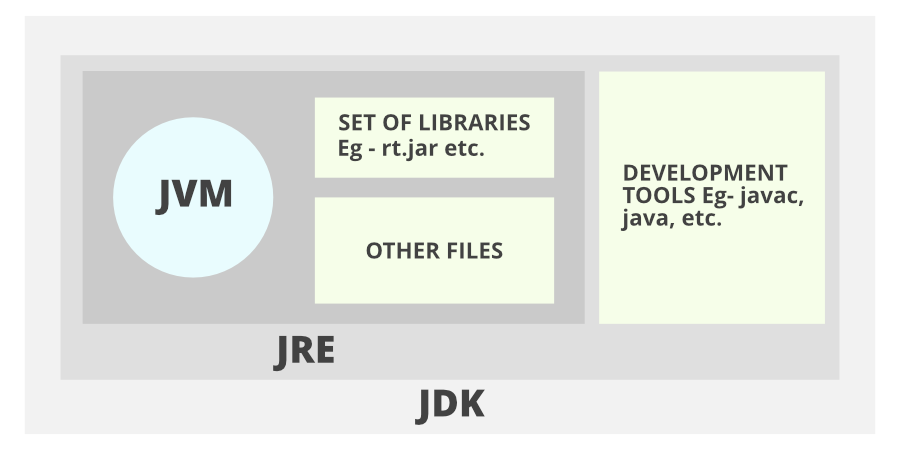
Has removed explicit pointers, operator overloading etc.

automatic garbage collector

**platform-independent, object-oriented, robust, multithreaded, does not require any preprocessor**



The C++ compiler compiles and converts the source code into the machine code, so c++ is faster than java



This is doc comment /\*\* documentation \*/

* **Access Modifiers:** default, public, protected, private
* **Non-access Modifiers:** final, abstract, strictfp.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Access Modifier | Within Class | Within Package | Outside Package by subclass only | Outside Package |
| Private | Y | N | N | N |
| Default | Y | Y | N | N |
| Protected | Y | Y | Y | N |
| Public | Y | Y | Y | Y |

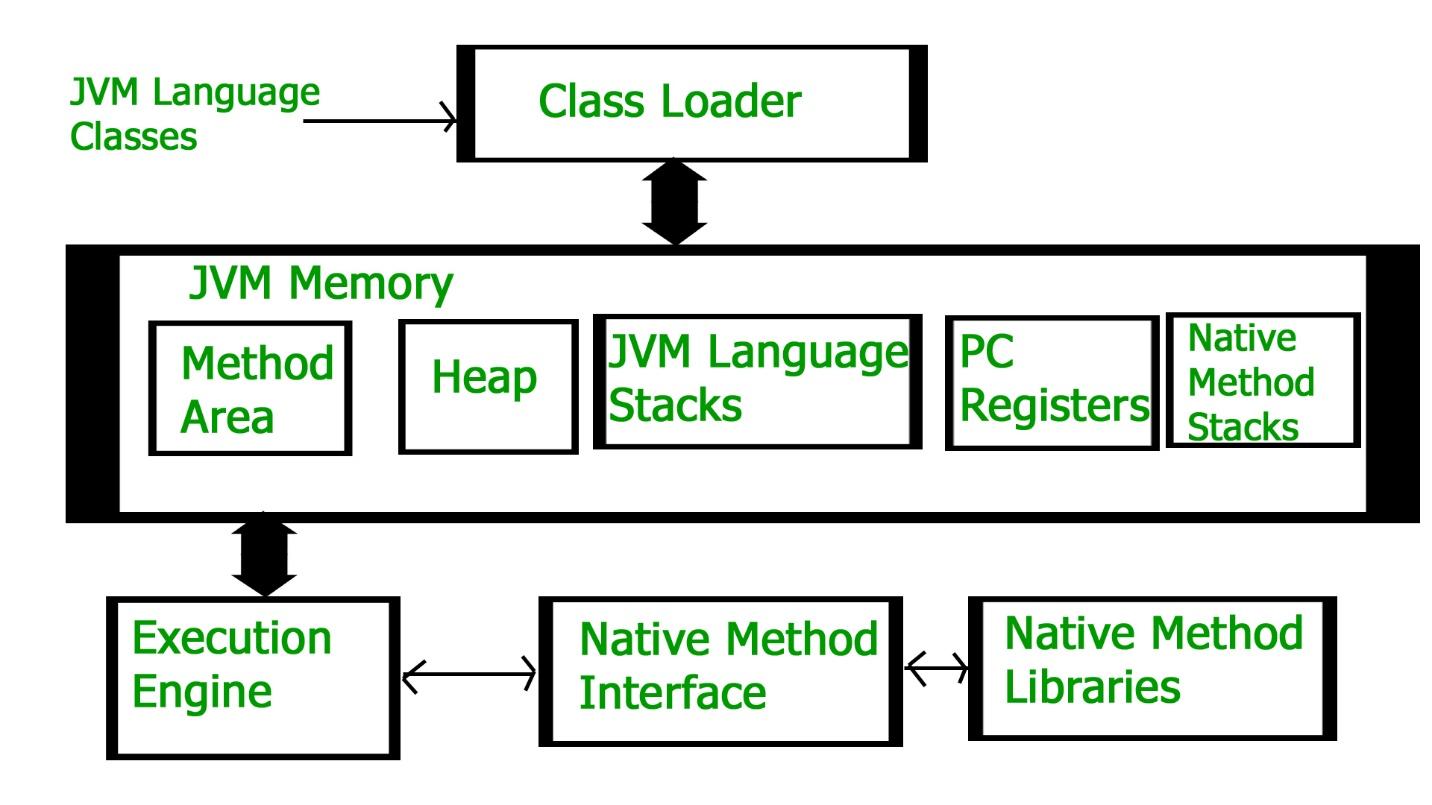
**JDK** (Java Development Kit), **develop and execute(run)**, has Development Tools (to provide an environment to develop your java programs) and JRE (to execute your java program).

**JRE - only run (not develop),** used by those who only want to run Java programs that are end-users of your system

[**JVM**](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/) (interpreter)- contained or inbuilt in both JDK and JRE.

Responsible for executing the java program line by line

calls the **main** method

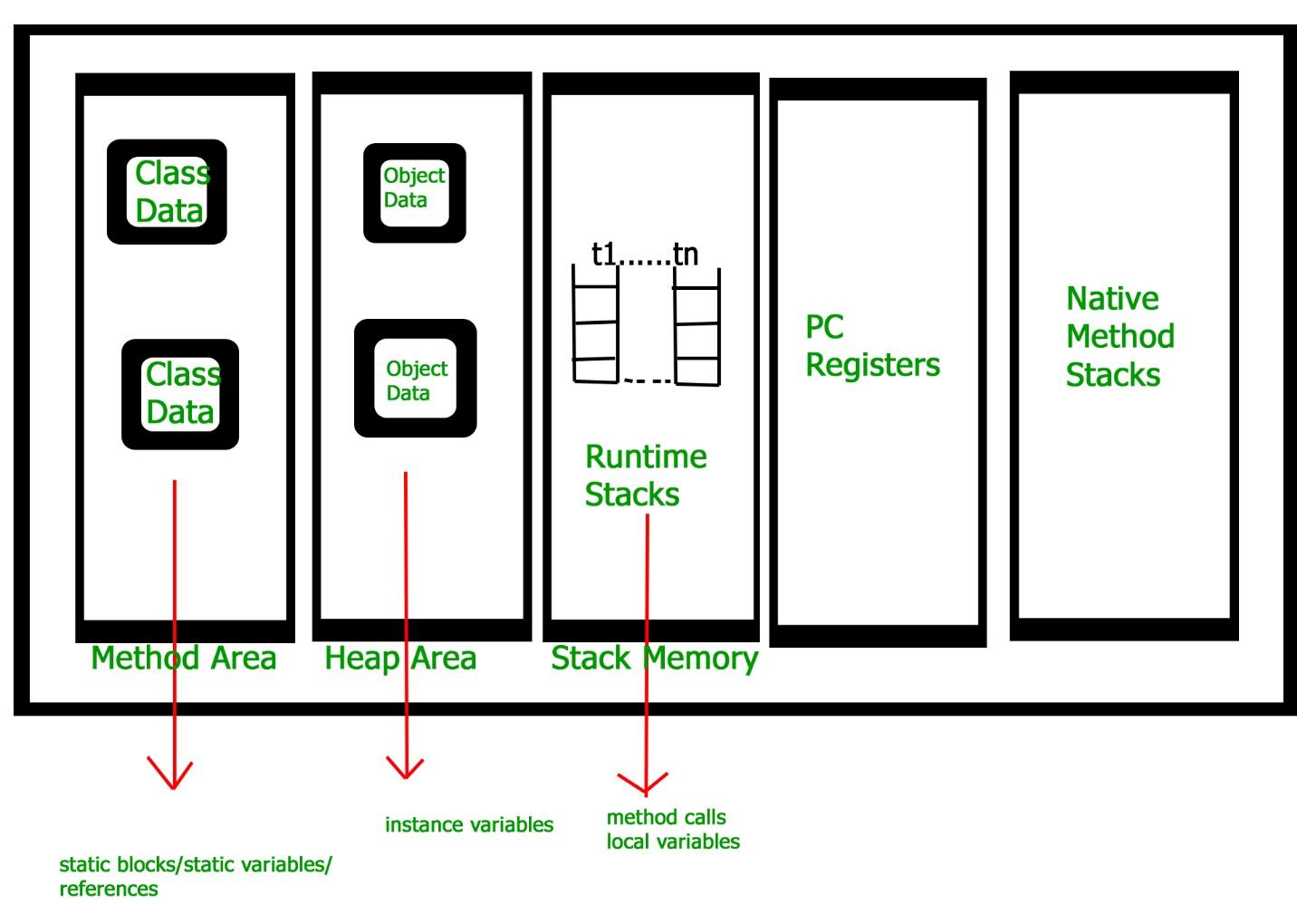


After loading the “*.class”* file, JVM creates an object of type Class to represent this file in the heap memory.

For every loaded “*.class”* file, only **one** object of the class is created

**JVM Memory**

1. **Method area:** In the method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.
2. **Heap area:** Information of all objects is stored in the heap area. There is also one Heap Area per JVM. It is also a shared resource.
3. **Stack area:** For every thread, JVM creates one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which stores methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminates, its run-time stack will be destroyed by JVM. It is not a shared resource.
4. **PC Registers:** Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers.
5. **Native method stacks:** For every thread, a separate native stack is created. It stores native method information.



**Execution Engine**

Execution engine executes the “*.class”* (bytecode). It reads the byte-code line by line, uses data and information present in various memory area and executes instructions. It can be classified into three parts:

* *Interpreter*: It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
* *Just-In-Time Compiler (JIT)*: It is used to increase the efficiency of an interpreter. It compiles the entire bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part so re-interpretation is not required, thus efficiency is improved.

GC (garbage collector)

free heap memory by destroying **unreachable objects.**

implementation lives in the JVM.

1. **Minor or incremental Garbage Collection:**It is said to have occurred when unreachable objects in the young generation heap memory are removed.
2. **Major or Full Garbage Collection:** It is said to have occurred when the objects that survived the minor garbage collection are copied into the old generation or permanent generation heap memory are removed. When compared to the young generation, garbage collection happens less frequently in the old generation.

There are generally four ways to make an object eligible for garbage collection.

* 1. Nullifying the reference variable
  2. Re-assigning the reference variable
  3. An object created inside the method
  4. Island of Isolation

object will be destroyed only whenever JVM runs the Garbage Collector program

request JVM to run Garbage Collector

***System.gc()***

A variable declared inside pair of brackets “{” and “}” in a method has scope within the brackets only.

**Package**

encapsulate a group of classes, sub packages and interfaces

data encapsulation

named in reverse order of domain names, i.e., org.geeksforgeeks.practice.

members of a subpackage have no access privileges, i.e., they are considered

as different package for protected and default access specifiers.

java.util.ArrayList newList = new java.util.ArrayList(); //creating object without import statement

Static import

allows members ( fields and methods ) defined in a class as public **static** to be used in Java code without specifying the class in which the field is defined.

import static java.lang.System.\*;

out.println("GeeksforGeeks"); // in main method no need to use System.out

**Handling name conflicts**

import java.util.Date;

import java.sql.\*; //it has date class too

use a full package name every time we declare a new object of that class

java.util.Date deadLine = new java.util.Date();

java.sql.Date today = new java.sql.Date();

A Label is used to identifies a block of code.   
Syntax:

label:

{

statement1;

statement2;

statement3;

.

.

}

first : {

    second : {

    third : {

        System.out.println("Before the break statement");

        if (t)

            break second;

        System.out.println("This won't execute.");

    }

        System.out.println("This won't execute.");

    }

        System.out.println("This is after second block.");

    }

Output :

Before the break statement.

This is after the second block.

Arrays

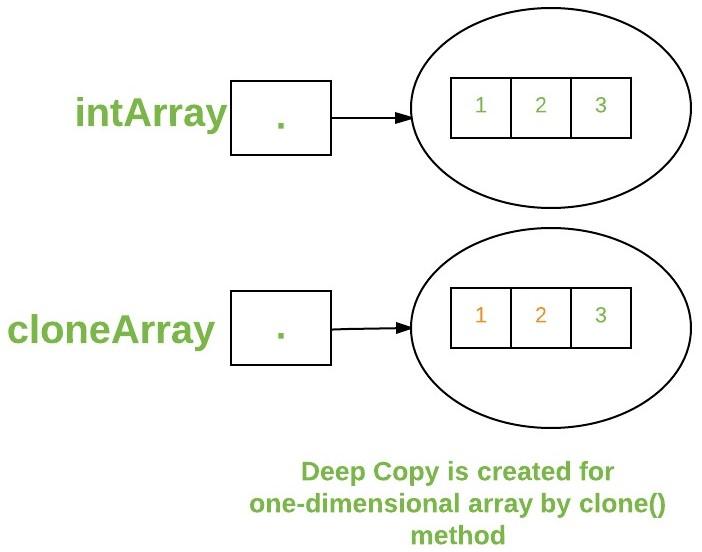
Dynamically allocated.

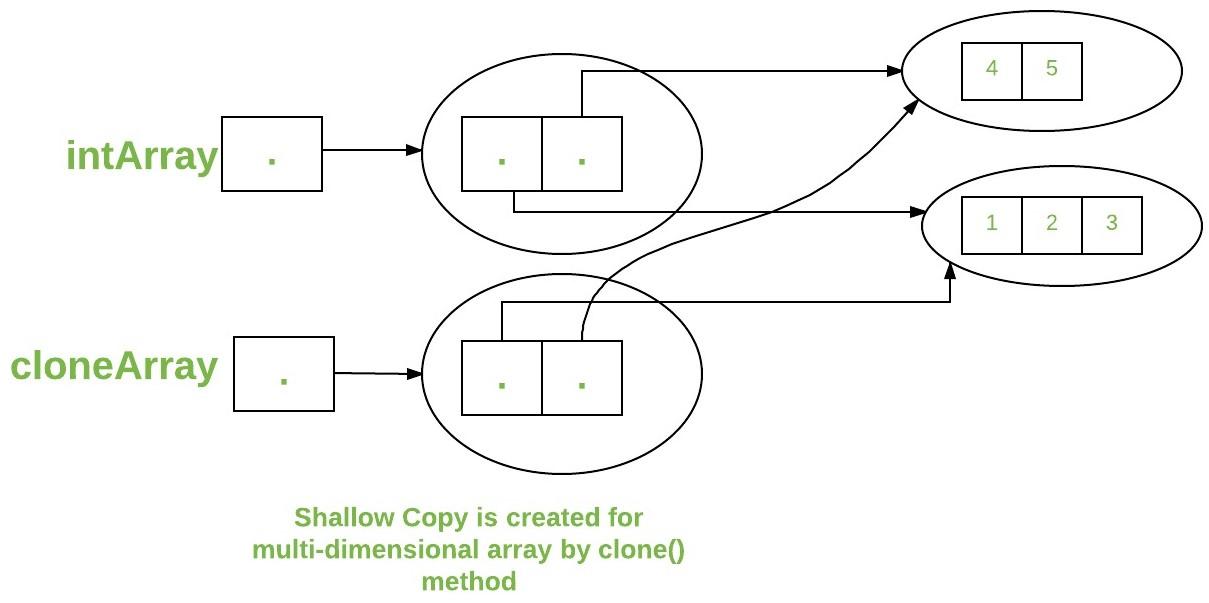
primitive data types are stored in contiguous memory locations.

class objects are stored in heap segments.

int[] intArray ={ 1,2,3,4,5,6,7,8,9,10 }; //valid

All the members inherited from class Object; the only method of Object that is not inherited is its [clone](https://www.geeksforgeeks.org/clone-method-in-java-2/) method.





int[][][] arr = { { { 1, 2 }, { 3, 4 } }, { { 5, 6 }, { 7, 8 } } };

**Jagged Array**

Arrays having rows with different number of colomns

Syntax: data\_type array\_name[][] = new data\_type[n][]; //n: no. of rows

array\_name[] = new data\_type[n1] //n1= no. of columns in row-1

array\_name[] = new data\_type[n2] //n2= no. of columns in row-2

array\_name[] = new data\_type[n3] //n3= no. of columns in row-3

.

.

.

array\_name[] = new data\_type[nk] //nk=no. of columns in row-n

Example :

int arr\_name[][] = new int[][] {

new int[] {10, 20, 30 ,40},

new int[] {50, 60, 70, 80, 90, 100},

new int[] {110, 120}

};

# Default Array Values in Java

| S. No. | Datatype | Default Value |
| --- | --- | --- |
| 1 | boolean | false |
| 2 | int | 0 |
| 3 | double | 0.0 |
| 4 | String | null |
| 5 | User-Defined Type | null |

Strings

Strings in Java are Objects that are backed internally by a char array

Immutable because arrays are immutable (cannot grow)

String constant pool / String pool

 The cache which stores these string instances

Creation of string:

1. As a literal - String str = "Geeks";

Literals are created in String constant pool

If we try to create another literal with same value

String str2 = "Geeks";

Str and str2 will be pointing to same “Geeks”

While a new string is created as a literal, the JVM checks the String Constant Pool. If the string does not exist, then a new string instance is created and placed in a pool. If the string exists, then it will not create a new object. Rather, it will return the reference to the same instance

1. Using new - String str = new String("Geeks");

allocated they are assigned a new memory location in heap.

will not be added to String constant pool.

If we create a string using new, JVM will create a new string object in the normal heap area even if the same string object is present in the string pool.

If you want to store this string in the constant pool then you will need to “intern” it.

String internedString = str.intern();

// this will add the string to string constant pool.

preferred to use String literals as it allows JVM to optimize memory allocation.

Illustration

class StringStorage {

    public static void main(String args[])

    {

        String s1 = "TAT";

        String s2 = "TAT";

        String s3 = new String("TAT");

        String s4 = new String("TAT");

        System.out.println(s1);

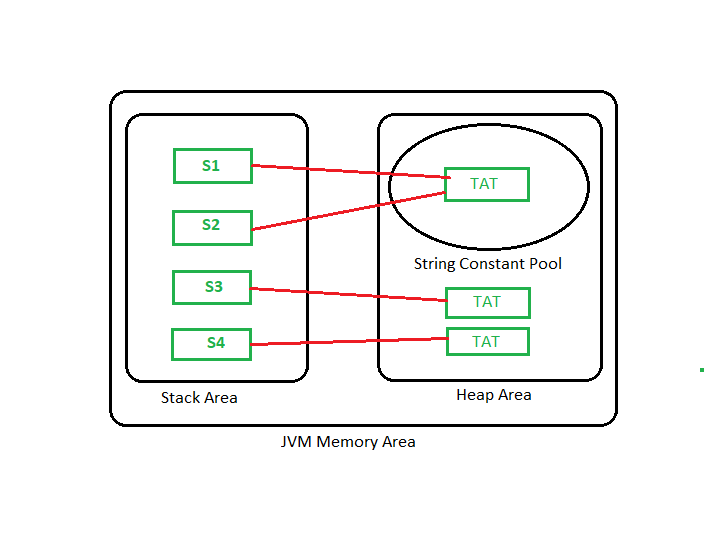
        System.out.println(s2);

        System.out.println(s3);

        System.out.println(s4);

    }

}



All objects in Java are stored in a heap.

reference variable to the object is stored in the stack area or they can be contained in other objects which puts them in the heap area also.

**StringBuffer**

growable and writable character sequences.

Multithreading, thread safe, guarantees synchronization

StringBuffer s = **new** StringBuffer(); //  It reserves room for 16 characters without reallocation

StringBuffer s = **new** StringBuffer(20); //  It reserves room for 20 characters

StringBuffer s = **new** StringBuffer("GeeksforGeeks"); //  It reserves room for “GeeksforGeeks” + 16 more characters

**StringBuilder**

no guarantee of synchronization, not thread safe

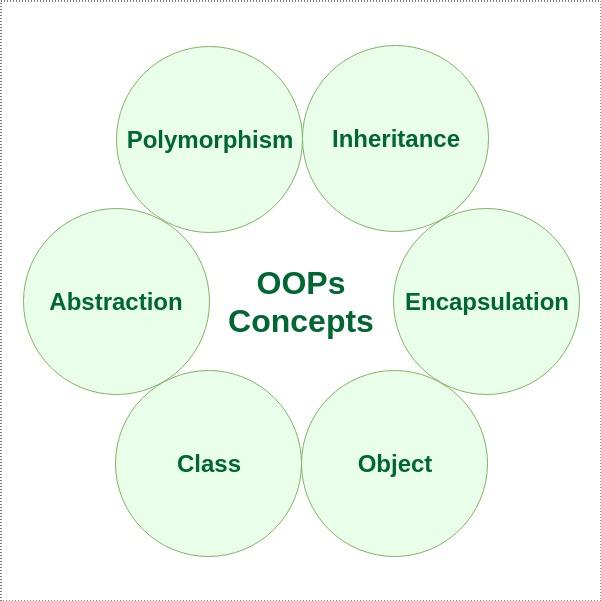
high in performance compared to String buffer.

**Appendable**

* Interface implemented by all classes that support appending CharacterSequence (StringBuilder, StringBuffer)
* Appendable appendable = new StringBuilder();
* Manipulate the character sequence
* append(CharSequence csq)
* append(CharSequence csq, int start, int end)
* append(char c)

**CharacterSequence**

* Interface implemented by all classes that support appending character sequence
* Get data about the sequence
* length()
* charAt(int index)
* subSequence(int start, int end)
* toString()
* CharSequence ch = “Hello”;
* CharSequence ch = new StringBuilder(“Hello”)



* [Polymorphism](https://www.geeksforgeeks.org/polymorphism-in-java/)
  + Compile-time polymorphism
  + Runtime polymorphism
* abstraction is achieved by [interfaces](https://www.geeksforgeeks.org/interfaces-in-java/) and [abstract classes](https://www.geeksforgeeks.org/abstract-classes-in-java/).

We can achieve 100% abstraction using interfaces.

* Encapsulation can be achieved by declaring all the variables in a class as private and writing public methods in the class to set and get the values of the variables.

### Ways to create an object of a class

**Using new keyword**

**Using Class.forName(String className) method**

// creating object of public class Test

// consider class Test present in *com.p1* package

Test obj = (Test)Class.forName("com.p1.Test").newInstance();

**Using clone() method**

Test t2 = (Test)t1.clone();

### Anonymous objects

objects that are instantiated but are not stored in a reference variable.

used for immediate method calling.

will be destroyed after method calling.

**Rules to Name a Method**

* the method name must be a **verb** and start with a **lowercase** letter.
* If the method name has more than two words, the first name must be a verb followed by an adjective or noun.
* In the multi-word method name, the first letter of each word must be in **uppercase** except the first word. For example, findSum, computeMax, setX and getX.

A method returns to the code that invoked it when:

* It completes all the statements in the method
* It reaches a return statement
* Throws an exception

#### Memory Allocation for Methods Calls

implemented through a stack.

 Whenever a method is called a stack frame is created within the stack area and after that, the arguments passed to, the local variables and value to be returned by this called method are stored in this stack frame and when execution of the called method is finished, the allocated stack frame would be deleted.

There is a stack pointer register that tracks the top of the stack which is adjusted accordingly.

# null error in Java

public class Test

{

    public void fun(Integer i) //method 1

    {

        System.out.println("fun(Integer ) ");

    }

    public void fun(String name) //method 2

    {

        System.out.println("fun(String ) ");

    }

    public static void main(String [] args)

    {

        Test mv = new Test();

        // The below line causes error

        mv.fun(null);

//The below set of lines won’t

Integer arg = null;

mv.fun(arg); //will invoke method 2 because null will be treated as string type due to method overloading resolution

    }

}

mv.fun(null) causes error as null matches with both function calls

But if method 1 is like -> public void fun(Object i)

Then,

Integer arg = null;

mv.fun(arg); // will invoke method 1

# Overload

# cannot overload just by return type

# cannot if differ only by static keyword

can be done by changing:

1. The number of parameters in two methods.
2. The data types of the parameters of methods.
3. The Order of the parameters of methods.

# Overload static methods

# cannot overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is the same).

public class Test {

    public static void foo() {

        System.out.println("Test.foo() called ");

    }

    public void foo() { // Compiler Error: cannot redefine foo()

        System.out.println("Test.foo(int) called ");

    }

    public static void main(String args[]) {

        Test.foo();

    }

}

**Can we Override static methods in java?**

No

If a derived class defines a static method with the same signature as a static method in the base class, the method in the derived class is hidden by the method in the base class.

class Base {

public static void display() { //m1

        System.out.println("Static or class method from Base");

    }

     public void print()  { //m2

         System.out.println("Non-static or Instance method from Base");

    }

}

class Derived extends Base {

    public static void display() { //m3

         System.out.println("Static or class method from Derived");

    }

    public void print() { //m4

         System.out.println("Non-static or Instance method from Derived");

   }

}

public class Test {

    public static void main(String args[ ])  {

       Base obj1 = new Derived();

       obj1.display();  //m1 is called

       obj1.print();     //m4 is called

    }

}

* 1. For class (or static) methods, the method according to the type of reference is called, not according to the object being referred, which means method call is decided at compile time.
  2. For instance (or non-static) methods, the method is called according to the type of object being referred, not according to the type of reference, which means method calls is decided at run time.
  3. An instance method cannot override a static method, and a static method cannot hide an instance method. For example, the following program has two compiler errors.

**4)** In a subclass (or Derived Class), we can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass methods — they are new methods, unique to the subclass.

class Base {

    // Static method in base class which will be hidden in subclass

    public static void display() {

        System.out.println("Static or class method from Base");

    }

     // Non-static method which will be overridden in derived class

     public void print()  {

         System.out.println("Non-static or Instance method from Base");

    }

}

class Derived extends Base {

     (Causes Compiler Error, saying not able to override Base’s display())

    public void display() {

        System.out.println("Non-static method from Derived");

    }

     (Causes Compiler Error, saying not able to override Base’s print())

    public static void print() {

        System.out.println("Static method from Derived");

   }

}

 in an interface are *public* even if we do not specify *public*with method names. Also, data fields are *public static final* even if we do not mention it with fields names. Therefore, data fields must be initialized.

# main() Method

JVM looks for the main method when it starts executing a Java program.

If we change the signature of the method, the program compiles but does not execute.

class GeeksforGeeks {

    private static void main(String[] args) //private

    {

        System.out.println("I am a Geek");

    }

}

Error: Main method not found in class, please define the main method as:

public static void main(String[] args)

So,

1)Must be public in order for it to be globally available

2)Must be static so that JVM can invoke it without instantiating the class.

3)Must be void because

The Java program is not even a process of Operating System directly. There is no direct interaction between the Java program and Operating System. There is no direct allocation of resources to the Java program directly, or the Java program does not occupy any place in the process table. Whom should it return an exit status to, then? This is why the main method of Java is designed not to return an int or exit status.

final public static void main(String[] args) //possible

public synchronized static void main(String[] args) //possible

* **Overloading Main method:**We can overload main() with different types of parameters.

class Test

{

    public static void main(String[] args)

    {

        System.out.println("Main Method String Array");

    }

    public static void main(int[] args)

    {

        System.out.println("Main Method int Array");

    }

}

**Output:**

Main Method String Array

public class Test {

    // Normal main()

    public static void main(String[] args) {

        System.out.println("Hi Geek (from main)");

        Test.main("Geek");

    }

    // Overloaded main methods

    public static void main(String arg1) {

        System.out.println("Hi, " + arg1);

        Test.main("Dear Geek","My Geek");

    }

    public static void main(String arg1, String arg2) {

        System.out.println("Hi, " + arg1 + ", " + arg2);

    }

}

**Output:**

Hi Geek (from main)

Hi, Geek

Hi, Dear Geek, My Geek

* **Inheritance of Main method:**JVM Executes the main() without any errors.

class A

{

    public static void main(String[] args)

    {

        System.out.println("Main Method Parent");

    }

}

class B extends A

{

}

**Output:**

Main Method Parent

**Method Hiding of main(), but not Overriding:**Since main() is static, derived class main() hides the base class main.

class A

{

    public static void main(String[] args)

    {

        System.out.println("Main Method Parent");

    }

}

class B extends A

{

    public static void main(String[] args)

    {

        System.out.println("Main Method Child");

    }

}

**Output:**

Main Method Parent

A **static block** in Java is a group of statements that gets executed only once when the class is loaded into the memory by ClassLoader, It is also known as a static initialization block, and it goes into the stack memory.

class StaticBlock {

static

{

System.out.println(

"This class can be executed without main");

System.exit(0);

}

}

# Can we override private methods in Java?

No

class Outer {

     private String msg = "GeeksforGeeks";

     private void fun() {

          System.out.println("Outer fun()");

     }

     class Inner extends Outer {

         private void fun()  {

               System.out.println("Accessing Private Member of Outer: " + msg);

         }

     }

     public static void main(String args[])  {

          Outer o = new Outer();

          Inner  i   = o.new Inner();

          i.fun();

          o = i;

          o.fun();

     }

}

Output:

Accessing Private Member of Outer: GeeksforGeeks

Outer fun()

method fun() has not been overridden.

***private methods are bonded during compile time and it is the type of the reference variable – not the type of object that it refers to – that determines what method to be called.***

private methods may be performance-wise better (compared to non-private and non-final methods) due to static binding.

**Need of Wrapper Classes**

convert primitive data types into objects.

classes in java.util package handles only objects

Collection framework, such as [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/) and [Vector](https://www.geeksforgeeks.org/vector-vs-arraylist-java/), store only objects

object is needed to support synchronization in multithreading.

[*valueOf()*](https://www.geeksforgeeks.org/data-conversion-using-valueof-method-java/), *[parseInt()](https://www.geeksforgeeks.org/string-to-integer-in-java-parseint/)*, *[toString()](https://www.geeksforgeeks.org/integer-tostring-in-java/)* can be used

# Constructors

called when an object of a class is created

*java compiler creates a default constructor if your class doesn’t have a constructor.*

called only once at the time of Object creation while method(s) can be called any number of times.

constructor in Java can not be abstract, final, static, or Synchronized.

Super() is the first line in constructor if u specify or not

Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

if we write a constructor with arguments or no arguments then the compiler does not create a default constructor.

*Default constructor provides the default values to the object like 0, null, etc. depending on the type.*

*constructor returns the current class instance*

# Constructor Chaining

**Within same class**:  using **this()**

**From base class:**using **super()**

**this()** expression should always be the first line of the constructor.

**Init block / Instance initialization block / IIB**:

gets executed for every constructor call (default or parameterized)

executed before any constructor

If there are more than one blocks, they are executed in the order in which they are defined within the same class

class Temp

{

    // block to be executed first

    {

        System.out.println("init");

    }

    Temp()

    {

        System.out.println("default");

    }

    Temp(int x)

    {

        System.out.println(x);

    }

    // block to be executed after the first block

    // which has been defined above.

    {

        System.out.println("second");

    }

    public static void main(String args[])

    {

        new Temp();

        new Temp(10);

    }

}

**Output :**

init

second

default

init

second

10

compiler executes parents class’s IIB before executing current class’s IIBs.

class B {

    B() { System.out.println("B-Constructor Called"); }

    {

        System.out.println("B-IIB block");

    }

}

// Child class

class A extends B {

    A()

    {

        System.out.println("A-Constructor Called");

    }

    A(int x)

    {

        System.out.println("A-Constructor Called :" + x);

    }

    {

        System.out.println("A-IIB block");

    }

    public static void main(String[] args)

    {

        A a = new A(10);

    }

}

**Output**

B-IIB block

B-Constructor Called

A-IIB block

A-Constructor Called : 10

# Copy Constructor

Complex(Complex c)

    {

        System.out.println("Copy constructor called");

        re = c.re;

        im = c.im;

    }

Complex c1 = new Complex(10, 15);

Complex c2 = new Complex(c1); //copy constructor is called

Complex c3 = c2; //Following doesn't involve a copy constructor call as non-primitive variables are just references.

***Singleton class***

limits the number of objects of that class to one.

Used in Networking and Database Connectivity.

Constructor is private

class MySingleton

{

    static MySingleton instance = null;

    public int x = 10;

    // private constructor can't be accessed outside the class

    private MySingleton() {  }

    // Factory method to provide the users with instances

    static public MySingleton getInstance()

    {

        if (instance == null)

             instance = new MySingleton();

        return instance;

    }

}

// Driver Class

class Main

{

   public static void main(String args[])

   {

       MySingleton a = MySingleton.getInstance();

       MySingleton b = MySingleton.getInstance();

       a.x = a.x + 10;

       System.out.println("Value of a.x = " + a.x);

       System.out.println("Value of b.x = " + b.x);

   }

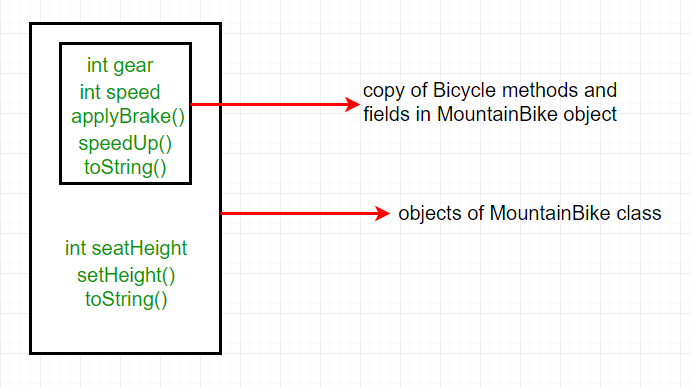
}

Output:

Value of a.x = 20

Value of b.x = 20

# Inheritance



Constructors are not members, so they are not inherited

We can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus [hiding](https://www.geeksforgeeks.org/g-fact-63/) it.

interface PI1 {

    // Default method

    default void show()

    {

        System.out.println("Default PI1");

    }

}

// Interface 2

interface PI2 {

    // Default method

    default void show()

    {

        System.out.println("Default PI2");

    }

}

class TestClass implements PI1, PI2 {

    // Overriding default show method

    public void show()

    {

        // Using super keyword to call the show

        // method of PI1 interface

        PI1.super.show();

        // Using super keyword to call the show

        // method of PI2 interface

        PI2.super.show();

    }

    public static void main(String args[])

    {

        TestClass d = new TestClass();

        d.show();

    }

}

**Output**

Default PI1

Default PI2

**Types of polymorphism**

In Java polymorphism is mainly divided into two types:

* Compile-time Polymorphism
* Runtime Polymorphism

**Type 1:**Compile-time polymorphism

function overloading

**Type 2:**[Runtime polymorphism](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/)

 Dynamic Method Dispatch

Overriding

# Overriding

# Poor performance

# Argument list should be same in method overriding.

# variables are not overridden

class A

{

    int x = 10;

}

// class B

class B extends A

{

    int x = 20;

}

// Driver class

public class Test

{

    public static void main(String args[])

    {

        A a = new B(); // object of type B

        // Data member of class A will be accessed

        System.out.println(a.x);

    }

}

Output:

10

# Since variables are not overridden, so the statement “a.x” will always refer to data member of super class.

* private, final and static methods and variables uses static binding and bonded by compiler while overridden methods are bonded during runtime based upon type of runtime object

# Overriding and Access-Modifiers: The [access modifier](https://www.geeksforgeeks.org/access-modifiers-java/) for an overriding method can allow more, but not less, access than the overridden method

For example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.

**Final methods cannot be overridden**

**Static methods cannot 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](https://www.geeksforgeeks.org/can-we-overload-or-override-static-methods-in-java/).

**Private methods can not be overridden :** [Private methods](https://www.geeksforgeeks.org/can-override-private-methods-java/) cannot be overridden as they are bonded during compile time.

The presence of synchronized/strictfp modifier with method have no effect on the rules of overriding, i.e. it’s possible that a synchronized/strictfp method can override a non synchronized/strictfp one and vice-versa.

# Abstraction

1. An abstract class may or may not have all abstract methods. Some of them can be concrete methods
2. Any class that contains one or more abstract methods must also be declared with an abstract keyword.
3. There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the [*new operator*](https://www.geeksforgeeks.org/new-operator-java/).
4. We can have an abstract class without any abstract method.
5. We can define static methods in an abstract class
6. If the**Child class** is unable to provide implementation to all abstract methods of the**Parent class**then we should declare that **Child class as abstract**so that the next level Child class should provide implementation to the remaining abstract method
7. *Abstract classes can also have****final****methods*

# Interface

# Interface can have only abstract methods.

# Variables declared in a Java interface are public, static, and final by default.

# Members of a Java interface are public and abstract by default.

* If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

# Interface in a class

# We can assign public, protected or private also

import java.util.\*;

class Test

{

    protected interface Yes

    {

        void show();

    }

}

class Testing implements Test.Yes

{

    public void show()

    {

        System.out.println("show method of interface");

    }

}

class A

{

    public static void main(String[] args)

    {

        Test.Yes obj;

        Testing t = new Testing();

        obj=t;

        obj.show();

    }

}

**Output**

show method of interface

**Interface in another Interface**

interface Test

{

   protected interface Yes //protected

   {

      void show();

   }

}

class Testing implements Test.Yes

{

   public void show()

   {

      System.out.println("show method of interface");

   }

}

class A

{

   public static void main(String[] args)

   {

     Test.Yes obj;

     Testing t = new Testing();

     obj = t;

     obj.show();

   }

}

**Output**

illegal combination of modifiers: public and protected

protected interface Yes

# Marker interface

empty interface (no field or methods)

Ex: Serializable, Cloneable and Remote interface

public interface Serializable

{

// nothing here

}

**Cloneable interface** :

present in java.lang package.

There is a method clone() in [Object](https://www.geeksforgeeks.org/object-class-in-java/) class.

A class that implements the Cloneable interface indicates that it is legal for clone() method to make a field-for-field copy of instances of that class. 

Invoking Object’s clone method on an instance of the class that does not implement the Cloneable interface results in an exception CloneNotSupportedException being thrown.

By convention, classes that implement this interface should override Object.clone() method.

class A implements Cloneable

{

    int i;

    String s;

    public A(int i,String s)

    {

        this.i = i;

        this.s = s;

    }

    // Overriding clone() method by simply calling Object class clone() method.

    @Override

    protected Object clone() throws CloneNotSupportedException

    {

        return super.clone();

    }

}

public class Test

{

    public static void main(String[] args)

        throws CloneNotSupportedException

    {

        A a = new A(20, "GeeksForGeeks");

        // cloning 'a' and holding

        // new cloned object reference in b

        // down-casting as clone() return type is Object

        A b = (A)a.clone();

        System.out.println(b.i);

        System.out.println(b.s);

    }

}

Output:

20

GeeksForGeeks

**Serializable interface** :

used to make an object eligible for saving its state into a file.

**Comparator interface** :

used to order the objects of user-defined classes.

**Syntax:**

public int compare(Object obj1, Object obj2):

Internally the Sort method does call Compare method of the classes it is sorting.

class Student {

    int rollno;

    String name, address;

    public Student(int rollno, String name, String address)

    {

        this.rollno = rollno;

        this.name = name;

        this.address = address;

    }

    public String toString()

    {

        return this.rollno + " " + this.name + " "

            + this.address;

    }

}

class Sortbyroll implements Comparator<Student> { ////

    public int compare(Student a, Student b)

    {

        return a.rollno - b.rollno;

    }

}

class Sortbyname implements Comparator<Student> {

    public int compare(Student a, Student b)

    {

        return a.name.compareTo(b.name);

    }

}

class GFG {

    public static void main(String[] args)

    {

        ArrayList<Student> ar = new ArrayList<Student>();

        ar.add(new Student(111, "Mayank", "london"));

        ar.add(new Student(131, "Anshul", "nyc"));

        ar.add(new Student(121, "Solanki", "jaipur"));

        ar.add(new Student(101, "Aggarwal", "Hongkong"));

        // Sorting student entries by roll number

        Collections.sort(ar, new Sortbyroll());

        for (int i = 0; i < ar.size(); i++)

            System.out.println(ar.get(i));

        // Sorting student entries by name

        Collections.sort(ar, new Sortbyname());

        // // Again iterating over entries to print them

        for (int i = 0; i < ar.size(); i++)

            System.out.println(ar.get(i));

    }

}

**Output**

101 Aggarwal Hongkong //sorted by roll no

111 Mayank london

121 Solanki jaipur

131 Anshul nyc

101 Aggarwal Hongkong //sorted by name

131 Anshul nyc

111 Mayank london

121 Solanki jaipur

For sorting based on two fields

static class CustomerSortingComparator

        implements Comparator<Student> {

        // Method 1

        // To compare customers

        @Override

        public int compare(Student customer1,

                           Student customer2)

        {

            // Comparing customers

            int NameCompare = customer1.getName().compareTo(

                customer2.getName());

            int AgeCompare = customer1.getAge().compareTo(

                customer2.getAge());

            // 2nd level comparison

            return (NameCompare == 0) ? AgeCompare

                                      : NameCompare;

        }

    }

**Output**

Customer{Name=Ajay, Age=22}

Customer{Name=Ajay, Age=27}

Customer{Name=Ajay, Age=29}

Customer{Name=Simran, Age=37}

Customer{Name=Sneha, Age=22}

Customer{Name=Sneha, Age=23}

# THIRD RULES

# Interface

# There are 2 interfaces I and J. Both have a default method called show() implemented in them.

# Class A implements I and J. Class A does not implement show()

# Then the object of Class A can’t call show().

# If class A has also implemented show() , then the object can call show()

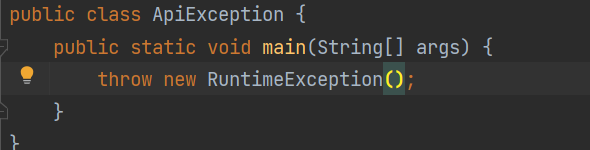
If class A has not implemented show(), but only interface I implemented show(), then the object can call show().

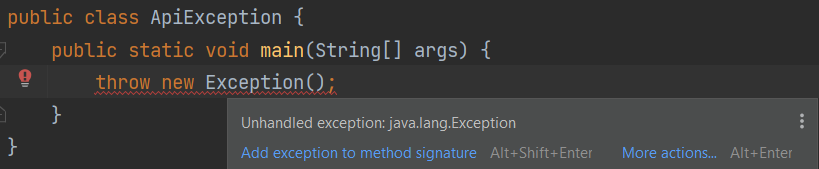
# Exception

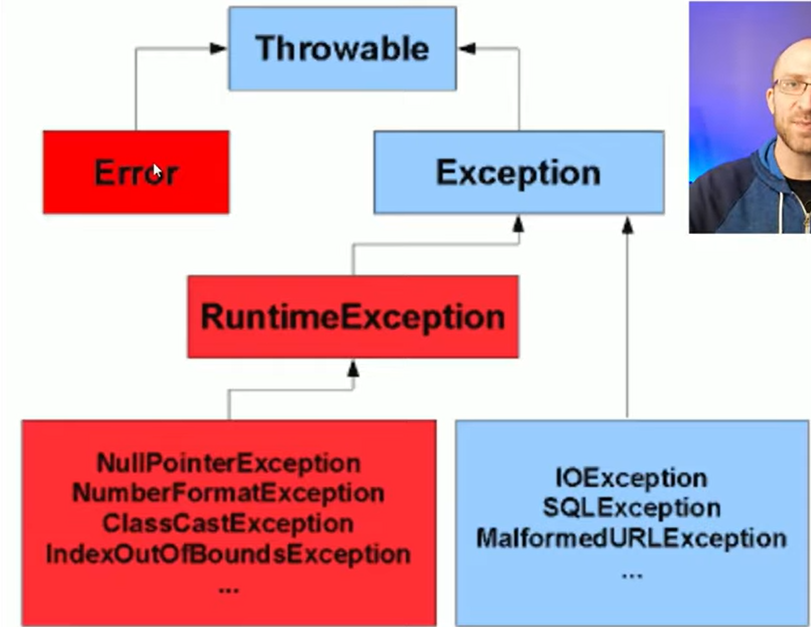
**Types**

Checked - Need to handle with try catch blocks, else compiler will show error in compile time and will not let you execute the program

Unchecked - No error is thrown at compile time







Blue - checked exceptions

Red - Unchecked exceptions

# 

# try - throws exception, once exception is thrown, the code below the statements will never get executed

# try with resources - throws exception if can not load resources inside the try(a) {b} cath(E) {c}. If a fails, only c gets executed.

# catch - catches exception

# finally - gets executed always

psvm(){

try{

System.out.println(Integer.parseInt(“pants”)); //exception is thrown

System.out.println(“After exception is thrown”); //never gets executed since exception is thrown

}

catch(Exception e){

System.out.println(“in catch block”);

}

finally{  
 System.out.println(“in finally block”);

}

System.out.println(“inside main()”);

}

**Output:**

in catch block

in finally block

inside main()

psvm(){

try{

return 3;

}

catch(Exception e){

return 4;

}

finally{  
 return 5;

}

}

**Output:**

5 will be returned always since the code in finally block gets executed at the end always

Try with resources

try (FileInputStream fileIn = new FileInputStream("employee.ser");

ObjectInputStream in = new ObjectInputStream(fileIn)) {

Employee employee = (Employee) in.readObject();

System.out.println("Object deserialized: " + employee);

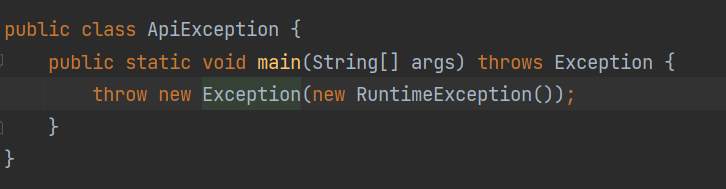
} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

**Throwable**

contains the exception cause, which is an object of Throwable, that contains information about the exception



In the above code new RunTimeException() is the cause object

**Logging**

**Types**

log.info

log.debug

log.warn

log.error

log.info(“entered to process id : ” + id)

-bad , since it is costly to concatenate strings and log

If(log.isDebugEnabled()){

log.info(“entered to process id : ” + id)

}

-better

But the above can also be written as

log.info(“entered to process id : {}” , id)

multi variable example :

log.info("entered to process id: {}, name: {}, age: {}", id, name, age);

**Java 8 features**

* 1. **Functional Interfaces and Lambda Expressions**

|  |  |
| --- | --- |
| **Predicate<T>** | Takes one argument and returns a boolean |

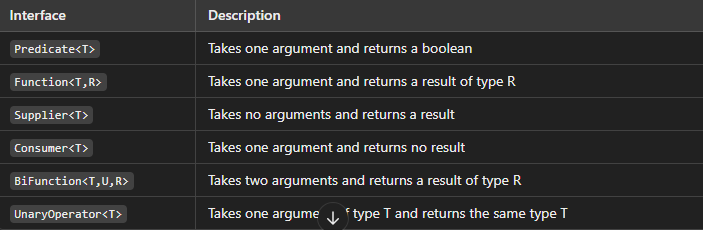
|  |  |
| --- | --- |
| **Function<T,R>** | Takes one argument and returns a result of type R |

|  |  |
| --- | --- |
| **Supplier<T>** | Takes no arguments and returns a result |

|  |  |
| --- | --- |
| **Consumer<T>** | Takes one argument and returns no result |

|  |  |
| --- | --- |
| **BiFunction<T,U,R>** | Takes two arguments and returns a result of type R |

|  |  |
| --- | --- |
| **UnaryOperator<T>** | Takes one argument of type T and returns the same type T |

****

**Functional Interface (**@FunctionalInterface**)**

* contains only one abstract method
* can have multiple default or static methods

**Lambda Expressions**

* direct implementation of functional interface

Runnable r = () -> System.out.println("Hello, Lambda!");

**Implement your own interface**

@FunctionalInterface

interface Calculator

{

int calculate(int a, int b);

}

Calculator add = (a, b) -> a + b;

System.out.println(add.calculate(5, 3));

// Output: 8

**Method References**

**1.** **Static Method Reference**: ClassName::staticMethod

Consumer<List<String>> sortList = Collections::sort;

**2.** **Instance Method Reference of a Particular Object**: instance::methodName

Consumer<String> printer = System.out::println;

**3.** **Instance Method Reference of an Arbitrary Object of a Particular Type**: ClassName::methodName

Function<String, Integer> strLength = String::length;

**4.** **Constructor Reference**: ClassName::new

Supplier<List<String>> listSupplier = ArrayList::new;

**Higher-Order Functions**

* pass lambdas as parameters or return them from methods

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.forEach(name -> System.out.println(name));

**2. Streams API**

* sequence of elements supporting sequential and parallel aggregate operations
* not a data structure but a high-level abstraction for expressing computation on data.

**Key Characteristics of Streams**:

* **Non-Storage**: Streams do not store elements but derive them from a source (like collections, arrays, or I/O channels).
* **Laziness**: Intermediate operations on streams are lazy, meaning they are not executed until a terminal operation is invoked.
* **Immutability**: Streams do not modify the source; instead, they return new streams or results.
* **Parallelism**: Streams can be processed in parallel without additional code complexity by using parallelStream().

**Higher-Order Functions**

List<integer> numbers = Arrays.asList(1,2,3,4);

numbers.stream()

**//Intermediate operations (returns a stream, gets invoked only when a terminal operation is called)**

.filter(no -> no < 4)

.map(n -> n \* n)

.sorted()

. distinct()

**//Terminal operations**

.forEach(System.out::println); //returns nothing

.collect(Collectors.toList()); //returns a list

.reduce(0, (a, b) -> a + b); //returns an object (in this case integer sum)

.count(); //returns a boolean

.anyMatch(name -> name.equals("John")); //returns a Boolean (same for allMatch, noneMatch)

.removeIf(Predicate<? super T> filter) //removes the items in the original List itself \*\*\*\*

**Collectors**

List<String> result = names.stream().collect(Collectors.toList());

Set<String> uniqueNames = names.stream().collect(Collectors.toSet());

String result = names.stream().collect(Collectors.joining(", "));

.collect(Collectors.joining("," , "[" , "]")); //”[Alice,Bob,John]”

Map<String, Integer> nameLengthMap = names.stream().collect(Collectors.toMap(name -> name, name -> name.length()));

Map<Integer, List<String>> groupedByLength = Stream.of("Alice", "Bob", "Charlie", "David") .collect(Collectors.groupingBy(String::length));

// {3=[Bob], 5=[Alice, David], 7=[Charlie]}

Map<Integer, Long> countByLength = Stream.of("Alice", "Bob", "Charlie", "David") .collect(Collectors.groupingBy(String::length, Collectors.counting()));

// {3=1, 5=2, 7=1}

**Best Practices**:

* Avoid modifying the source data structure during stream operations.
* Use method references for cleaner code.
* Prefer sequential streams for small datasets, as parallel streams have overhead.

**Best Practices for Using Parallel Streams**

1. **Use Parallel Streams for Large Datasets**: For small datasets, the overhead of managing multiple threads can outweigh the benefits. Parallel streams are most effective when working with large collections.
2. **Ensure Thread-Safety**: Avoid modifying shared mutable state within parallel streams. Prefer immutable data structures or thread-safe collections.
3. **Profile Before Use**: Before switching to parallel streams, profile your application to ensure that the benefits outweigh the potential drawbacks.
4. **Avoid Parallel Streams for Order-Dependent Tasks**: For tasks that rely on the ordering of elements (e.g., sorting or sequential processing), stick with sequential streams.
5. **Tune the Fork/Join Pool**: The number of threads in the **Fork/Join Pool** can be adjusted using System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "desired\_thread\_count").

**Map Enhancements**

Map<String, Integer> ageMap = Map.of("Alice", 25, "Bob", 30, "Charlie", 35);

ageMap.forEach((name, age) -> System.out.println(name + " is " + age + " years old"));

ageMap.putIfAbsent("Alice", 25); // Adds "Alice" only if not present

ageMap.computeIfPresent("Alice", (name, age) -> age + 1); // Increases Alice's age by 1

ageMap.computeIfAbsent("David", name -> 40); // Adds "David" with the age 40

ageMap.merge("Alice", 5, Integer::sum); // Adds 5 to Alice's existing age

**Comparator Enhancements**

List<Person> people = Arrays.asList(new Person("Alice", 25), new Person("Bob", 30)); people.sort(Comparator.comparing(Person::getAge));

people.sort(Comparator.comparing(Person::getAge).thenComparing(Person::getName));

people.sort(Comparator.comparing(Person::getAge).reversed());

names.sort(Comparator.nullsFirst(Comparator.naturalOrder())); // null comes first

**3. Optional**

**-** to avoid NullPointerException

String name = getName(); // getName() may return null

if (name != null) {

System.out.println(name.toUpperCase());

} else {

System.out.println("Name is not available");

}

Can be written as

Optional<String> name = Optional.ofNullable(getName());

name.ifPresent(n -> System.out.println(n.toUpperCase()));

**How to create**

Optional<String> name = Optional.of("John"); //Returns an Optional with a non-null value.

= Optional.ofNullable(getName()); //Returns an Optional that may contain a null value. If the value is null, it returns an empty Optional

= Optional.empty(); //Returns an empty Optional

**Methods**

opt.isPresent() //returns Boolean

opt.ifPresent(Consumer<? super T> action)

opt.get() // Returns the value if present, but throws NoSuchElementException if the value is absent. Use this method cautiously as it can still lead to runtime exceptions if not checked properly.

opt.orElse(T other) // Returns the value if present; otherwise, returns the provided default value.

opt.orElseGet(Supplier<? extends T> other) // T is generated only if value is absent.

opt.filter(Predicate <T>) //returns Optional only if value matches Predicate

opt.map(Function<T, R>) //Returns Optional of type R

**Best Practices for Using Optional**

1. **Avoid using Optional.get()**: Use get() only if you are certain the Optional is non-empty. Prefer orElse(), orElseGet(), or ifPresent().
2. **Use Optional for return types, not fields or method parameters**: Optional is designed to represent an absent value in return types. It should not be used for class fields or method parameters.
3. **Combine Optional with Streams**: Optional’s methods like map() and flatMap() are powerful when combined with streams for processing data pipelines.
4. **Prefer orElseGet() for expensive operations**: Use orElseGet() if the default value involves a costly operation, as it will only be invoked if the value is absent.

# Multithreading

Process

* Executing instance of a program
* Has its own stack and heap memory

threads

* light-weight processes
* smallest unit of process
* lives inside process
* shares resources like memory

Concurrency

* Multiple tasks are being handled at the same time. Not necessarily at the same moment. More like switching in between tasks, but only handling one at a time. But it switches so fast, it seems like it is happening at the same time

**Parallelism**

* Actual simultaneous execution of tasks on multiple processors or cores. Handling multiple tasks at a same time.

**Benefits of Multithreading:**

1. **Improved Performance**: Tasks can be split and run in parallel, especially in multi-core systems.
2. **Resource Sharing**: Threads within the same process share the same memory space, which reduces the overhead of managing separate memory.
3. **Responsiveness**: In applications (especially UIs), heavy tasks like file I/O or long computations can run on separate threads, allowing the main thread (like a UI thread) to remain responsive.
4. **Efficient Resource Utilization**: Threads allow the CPU to utilize idle time better by switching between tasks when one thread is waiting for I/O operations.

can be created by:

1. **Extending the Thread class**

class MyThread extends Thread {

@Override

public void run() {

System.out.println("Thread running: " + Thread.currentThread().getName());

}

}

public class Main {

public static void main(String[] args) {

MyThread thread = new MyThread();

thread.start(); // Start the thread, which calls the run() method

}

}

1. **Implementing the Runnable Interface**

class MyRunnable implements Runnable {

@Override

public void run() {

System.out.println("Runnable running: " + Thread.currentThread().getName());

}

}

public class Main {

public static void main(String[] args) {

Thread thread = new Thread(new MyRunnable());

thread.start(); // Start the thread

}

}

**ExecutorService**

* Helps to create n no. of thread and divide the tasks between them.
* Easier to manage threads instead of using for loop
* part of the java.util.concurrent
* makes it easier to run tasks using a pool of threads.

public class Main {

public static void main(String[] args) {

// Create a thread pool with 3 threads

ExecutorService executor = Executors.newFixedThreadPool(3);

// Submit tasks to the thread pool

for (int i = 0; i < 4; i++) {

executor.submit(() -> {

System.out.println("task ” + i + “ executed by thread: " + Thread.currentThread().getName());

});

}

// Shut down the executor once all tasks are complete

executor.shutdown();

}

}

executor.shutdown(); // Graceful shutdown

It stops accepting new tasks but allows existing tasks to complete.

**Output :**

ExecutorService executor = Executors.newSingleThreadExecutor();

task 0 executed by pool-1-thread-1

task 1 executed by pool-1-thread-1

task 2 executed by pool-1-thread-1

//tasks will be divided between n threads, in this case 2

ExecutorService executor = Executors.newFixedThreadPool(2);

task 0 executed by pool-1-thread-1

task 1 executed by pool-1-thread-2

task 2 executed by pool-1-thread-1

//for each task a new thread will be created if all threads are occupied

//tasks are pushed to synchronous queue not blocking queue

ExecutorService executor = Executors.newCachedThreadPool();

task 0 executed by pool-1-thread-1

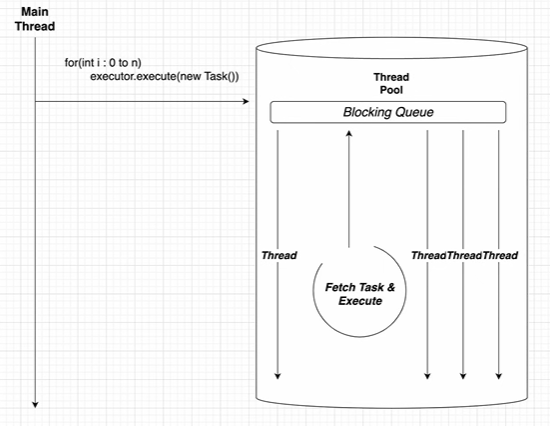
task 1 executed by pool-1-thread-2

task 2 executed by pool-1-thread-3

//calls the new Task() for every 2 seconds infinitely

ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

Executor.scheduleAtFixedRate(new Task() , 1000 , 2000 , TimeUnit.MILLISECONDS);



* All tasks will be pushed to Blocking Queue
* Threads will pick tasks from queue and execute

 **FixedThreadPoolExecutor**

 **CachedThreadPool**: Dynamically manages threads based on demand.

 **SingleThreadExecutor**: Executes tasks sequentially in a single thread.

 **ScheduledExecutor**:

**B. Controlling Thread Lifecycle**

* **Start**: Call the start() method to begin thread execution.
* **Sleep**: Thread.sleep(milliseconds) pauses the thread for a specified time.
* **Join**: thread.join() allows one thread to wait for the completion of another.

Example for join()

thread1.join(); // Main thread waits for thread1 to finish

D. **Handling Thread Safety**

**Synchronized Blocks/Methods**: Prevents multiple threads from accessing critical sections simultaneously.

Example :

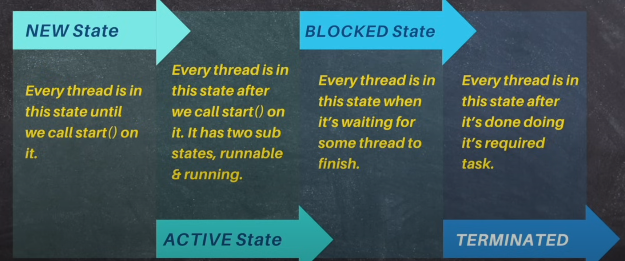
synchronized(this) {

// Critical section

}

**Best Practices for Managing Threads**

**Youtube**

****

**Join() :**

**public class Main {**

**public static void main(String[] args) throws InterruptedException {**

**Thread one = new Thread(() -> {**

**for (int i = 0; i < 5; i++) {**

**System.out.println( "Thread 1 : " + i);**

**}**

**});**

**Thread two = new Thread(() -> {**

**for (int i = 0; i < 5; i++) {**

**System.out.println( "Thread 2 : " + i);**

**}**

**});**

**System.out.println("Before....");**

**one.start();**

**two.start();**

**one.join(); //case 1 : if one.join() not present, line 1 would have come in between**

**two.join(); //case 2 : same as above**

**System.out.println("After...."); //line 1**

**}**

**}**

**Output :**

**Before....**

**Thread 2 : 0**

**Thread 2 : 1**

**Thread 1 : 0**

**Thread 2 : 2**

**Thread 1 : 1**

**Thread 1 : 2**

**Thread 1 : 3**

**Thread 1 : 4**

**//After…. (If case 2: two.join() not present)**

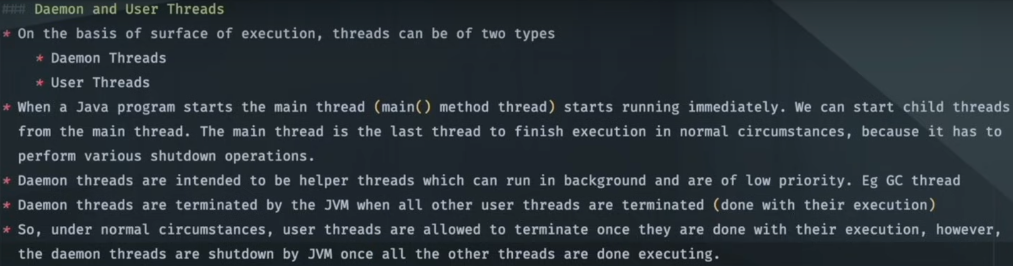
**Thread 2 : 3**

**Thread 2 : 4**

**After....**

**childThread.join() tells the main thread to continue execution only after childThread completes**

**Daemon Threads:**

****

**public class Daemon {  
 public static void main(String[] args) {  
 Thread daemonT = new Thread(() -> {  
 try {  
 for (int i = 0; i < 100; i++) {  
 Thread.*sleep*(1000);  
  
 System.*out*.println( "daemonT Running....");  
 }  
 } catch (InterruptedException e) {throw new RuntimeException(e);}  
  
 });  
  
 daemonT.setDaemon(true); //stops the daemonT as soon as userT completes  
  
 Thread userT = new Thread(() -> {  
 try {  
 Thread.*sleep*(5000);  
  
 System.*out*.println( "userT Running....");  
 } catch (InterruptedException e) {throw new RuntimeException(e);}  
 });  
  
 userT.start();  
 daemonT.start();  
  
  
 }  
}**

**Output :**

**daemonT Running....**

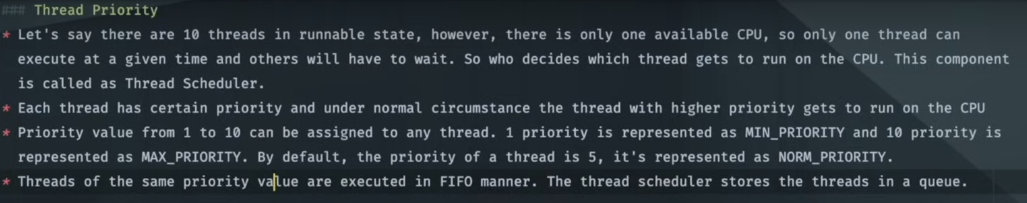
**daemonT Running....**

**daemonT Running....**

**daemonT Running....**

**userT Running....**

**Thread priority / Thread scheduler :**

****

**Thread.currentThread().setPriority(Thread.MAX\_PRIORITY)**

**Synchronized block :**

**public class Synchronised {  
  
 private static int *counter* = 0;  
  
 public static void main(String[] args) throws InterruptedException {  
  
 Thread one = new Thread(() -> {  
 for (int i = 0; i < 10000; i++) {  
 *increment*(); //case 1: instead of increment if there was counter++;  
 }  
 });  
  
 Thread two = new Thread(() -> {  
 for (int i = 0; i < 10000; i++) {  
 *increment*();  
 }  
 });  
  
 one.start();  
 two.start();  
 one.join();  
 two.join();  
  
 System.*out*.println(*counter*);  
 }  
  
 private static synchronized void increment() {  
 *counter*++;  
 }  
}**

**Output :**

**20000**

**synchronized lets only one thread access the code at a time**

**if case 1: then thread two would access the counter along with thread one and cause a different value of counter at the end**

**that is if both threads access counter at the same time then, both could take counter value, increment it and then replace it.**

**At the end of two increment, counter value would have only increased by 1**

**How synchronized keyword works :**

**As soon as a thread enters the method or object it locks the object (monitor lock/ intrinsic lock).  
  
When the method is locked no other thread can enter(access) the method/object. When other threads try to access a locked method, it goes to a blocked state.**

**When the thread completes execution, it releases the lock and other threads can enter.**

**Issues with synchronization:**

* 1. **On method level synchronization, entire code inside method gets locked.**
  2. **Also the class members gets locked. Therefore other unsynchronized methods also will be locked, not allowing for other threads to access the unsynchronized methods. Therefore only one thread will be able to access the class object at a time**

**Overcome by using locks and synchronized block instead of method level synchronization**

**public class SyncBlock {  
  
 private static int *counter1* = 0;  
 private static int *counter2* = 0;  
  
 private static final Object *lock1* = new Object();  
 private static final Object *lock2* = new Object();  
  
 public static void main(String[] args) throws InterruptedException {  
  
 Thread one = new Thread(() -> {  
 for (int i = 0; i < 10000; i++) {  
 *increment1*(); }  
 });  
  
 Thread two = new Thread(() -> {  
 for (int i = 0; i < 10000; i++) {  
 *increment2*();  
 }  
 });  
  
 one.start();  
 two.start();  
 one.join();  
 two.join();  
  
 System.*out*.println(*counter1* + "--" + *counter2*);  
 }  
  
 private static void increment1() {  
 synchronized (*lock1*){  
 *counter1*++;  
 }  
 }  
 private static void increment2() {  
 synchronized (*lock2*){  
 *counter2*++;  
 }  
 }  
  
// if lock1 not present, when t1 is in increment1(), no other threads will be able to get inside increment2(). Because normally method level locks will acquire class level. This will cause bottleneck. Because why should t2 wait to access increment2() when t1 is accessing increment1(). They both are independent operations in this case right.**

**// private static synchronized void increment1() {  
// counter1++;  
// }  
  
// private static synchronized void increment2() {  
// counter2++;  
// }  
  
}**

**Output : 10000 – 10000**

**Wait() & notify() :**

**Wait – locks(suspends/pauses) the current object thread. notify() resumes the thread.**

**public class WaitAndNotify {  
  
 private static final Object *lock* = new Object();  
  
 public static void main(String[] args) throws InterruptedException {  
  
 Thread one = new Thread(() -> {  
 try {  
 *one*();  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 });  
  
 Thread two = new Thread(() -> {  
 try {  
 *two*();  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 });  
  
 one.start();  
 two.start();  
  
 }  
  
 private static void one() throws InterruptedException {  
 synchronized (*lock*) {  
 System.*out*.println("from one BEFORE wait()");  
 *lock*.wait();  
 System.*out*.println("from one AFTER wait()");  
 }  
 }  
  
 private static void two() throws InterruptedException {  
 synchronized (*lock*) {  
 System.*out*.println("from two BEFORE wait()");  
 *lock*.notify();  
 System.*out*.println("from two AFTER wait()");  
 }  
 }  
}**

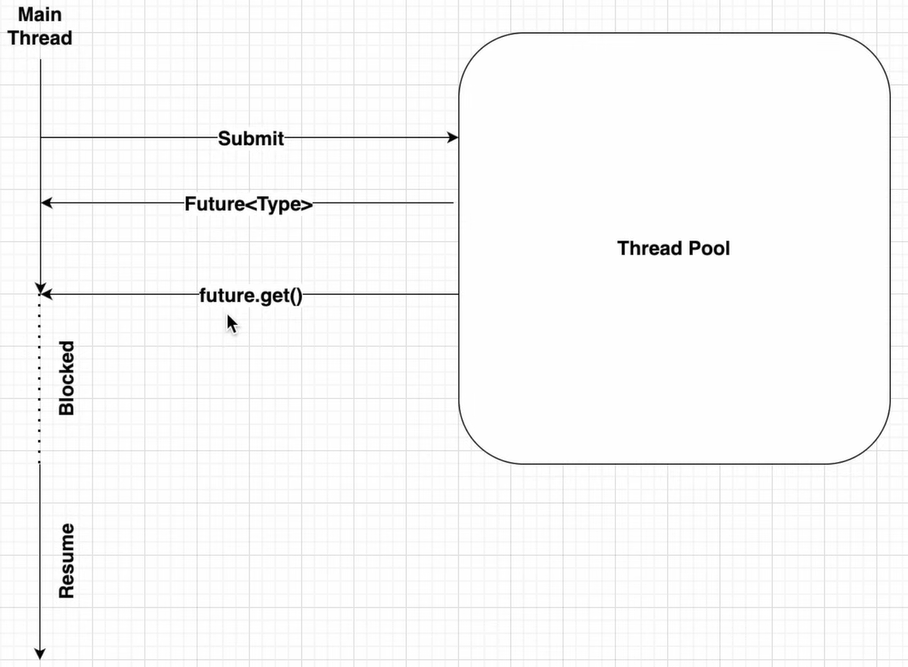
**from one BEFORE wait()**

**from two BEFORE wait()**

**from two AFTER wait()**

**from one AFTER wait()**

**FUTURE AND CALLABLE**

****

**public class CallableExample {  
  
 public static void main(String[] args) throws InterruptedException, ExecutionException {  
 try (ExecutorService executorService = Executors.*newFixedThreadPool*(2)) {  
  
 Future<Integer> result = executorService.submit(new ReturnValueTask());  
 System.*out*.println(result.get()); //line 1  
 System.*out*.print("Main thread execution completed! "); //line 2  
 }  
 }  
}  
  
class ReturnValueTask implements Callable<Integer> {  
 @Override  
 public Integer call() throws Exception {  
 Thread.*sleep*(5000);  
 return 12;  
 }  
}**

**Callable**

* **Interface**
* **Same as Runnable but can return value**

**It is a blocking operation, therefore all other executions wait until result is obtained.**

**line 2 gets executed only after line 1, that is after 5 seconds**

**Output :**

**12**

**Main thread execution completed!**

**COLLECTIONS in multithreading**

* **List, HashMap are all inconsistent and not threadsafe**

**List<Integer> list = Collections.synchronizedList(new ArrayList ( ) ) ;**

**Feature/downside**

* **Allows only single thread to access the collection (because it uses only 1 lock to synchronize the collection)**
* **Therefore overhead**

**COUNTDOWNLATCH**

* **Sophisticated join()**
* **latch.countDown(); //reduces the count of the latch, when it becomes 0, main thread continues**

**public class CountDownLatchDemo {  
  
 public static void main(String[] args) throws InterruptedException, ExecutionException {  
 int latchCount = 3;  
 CountDownLatch latch = new CountDownLatch(latchCount);  
  
 new Thread(new Chef("Chef A" , "Pizza" , latch)).start();  
 new Thread(new Chef("Chef B" , "Burger" , latch)).start();  
 new Thread(new Chef("Chef C" , "Salad" , latch)).start();  
  
 latch.await();  
  
 System.*out*.println("All dishes ready");  
 }  
}  
  
class Chef implements Runnable {  
  
 private final String name;  
 private final String dish;  
 private final CountDownLatch latch;  
  
 Chef(String name, String dish, CountDownLatch latch) {  
 this.name = name;  
 this.dish = dish;  
 this.latch = latch;  
 }  
  
 @Override  
 public void run() {  
 try {  
 System.*out*.println(name + " is preparing " + dish);  
 Thread.*sleep*(2000);  
 System.*out*.println(name + " has finished " + dish);  
  
 latch.countDown(); //reduces the count of the latch, when it becomes 0, main thread continues  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
}**

**BLOCKING QUEUE**

* **Has only certain no. of items in queue**
* **Used to simulate producer consumer mechanism**

**static BlockingQueue<Integer> taskQueue = new ArrayBlockingQueue(QUEUE\_CAPACITY);**

**inside producerThread -> *taskQueue*.put(i);**

**inside consumerOneThread -> taskQueue.take();**

**inside consumerTwoThread -> taskQueue.take();**

**consumerOneThread and consumerTwoThread take the values from the taskQueue.**

**Only after values are reduced from taskQueue, *taskQueue*.put(i); can be called**

**ConcurrentHashMap**

**Map<String , String> map = new ConcurrentHashMap();**

**Put elements**

* 1. **Key is hashed to determine segment**

**Segment -> like a small HashMap in the larger ConcurrentHashMap**

* 1. **Lock is acquired for that segment**
  2. **Element added**
  3. **Lock is released after addition**

**Get elements**

1. **Element is searched**

**Advantages :**

* **Thread safe**
* **HashMap in multithreaded environment -> data corruption**
  + **t1 & t2 put at the same time in a cell**
    - **leads to data corruption**
    - **infinite loop**

**Disdvantages :**

* **Slow due to synchronization**

**CyclicBarrier**

* **private static final CyclicBarrier *barrier* = new CyclicBarrier(*5* , () -> System.*out*.println("Tour Guide starts speaking...."));**
* **barrier has internal counter with value 5**
* **barrier.await() reduces counter value**
* **Only lets other thread continue after counter gets to 0, i.e only after all threads reach barrier.await() , all threads can continue.**
* **Like a checkpoint**

**Exchanger**

* **Used to exchange data between Threads**
* **exchange()**
  + **takes t1’s object as input and exchanges with t2’s exchange object and returns the received t2’s object**

**CopyOnWriteArrayList**

* **list = new CopyOnWriteArrayList()**
* **everything same as ArrayList()**
* **No need to synchronization**
* **Behaviour**
  + **Creates copy of original array when read or written**
* **When to use?**
  + **Read >>>> write**
  + **Thread safe iteration in a multithreaded environment**

**Read Write Lock**

* **Multiple thread can read the resource at a time, but only one can write at a time**
* **Read lock used by N threads, write lock is used by only 1 thread**

**Deadlock**

* **Thread waiting for each other and therefore waiting infinitely**

**How to detect Deadlocks**

* **Use this code in main()**

**new Thread(() -> {  
 ThreadMXBean mxBean = ManagementFactory.*getThreadMXBean*( ) ;  
 while (true) {  
 long[] threadIds = mxBean.findDeadlockedThreads();  
 if (threadIds != null) {  
 System.*out*.println("Deadlock detected!");  
 ThreadInfo[] threadInfo = mxBean.getThreadInfo(threadIds);  
 for (long threadId : threadIds) {  
 System.*out*.print("Thread with ID " + threadId + " is in Deadlock");  
 }  
 break;  
 }  
 try {  
 Thread.*sleep*(2000);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
}).start();**

**How to prevent Deadlocks**

* **Use Timeouts**
* **Global ordering of locks (lock a should be acquired before lock b)**
* **Avoid nested locks**
* **Use Thread safe alternatives**

**Atomic variable**

* **Thread safe (i.e read,modify,write is a single step in atomic variables, therefore no inconsistency)**
* **No need to synchronize or locks**

**Semaphore**

* **Allows only N no. of threads to run at a time**
* **private Semaphore semaphore = new Semaphore(3);**

**Mutex**

* **mutual exclusion**
* **allowing only one thread to access a code at a time**
* **fancy term for LOCK and synchronization**

**Fork Join Pool**

* **recursion on steroids, each call executes on a new Thread and then joins**
* **similar to ExecutorService, difference is FJP can create subtasks**

**Assert**

**public static void main(String[] args) {**

**int a = 3, b = 5;**

**assert a > b : "a must be greater";**

**System.out.println("Success");**

**}**

**Output :**

**Exception in thread "main" java.lang.AssertionError: Invalid snake: head must be greater than tail at Test.main(Test.java:5)**

* **Used for debugging only**
  + **Not for production, as it can be disabled, therefore exceptions are preferred**

**In Junit (SpringFramework)**

* **Assert.notNull()**
* **Assert.isNull()**

**Generics**

**class Demo <T> { //<T extends Animal & Serializable & Cloneable>  
 T thing;  
  
 Demo(T thing) {  
 this.thing = thing;  
 }  
  
 public void printer(T thing) {  
 System.*out*.println(thing.toString());  
 }  
  
  
 public static void main(String[] args) {  
 Demo<Integer> demo = new Demo<>(65);  
 demo.printer(demo.thing);  
  
 *print*("Hi" , 11);  
 }**

**public static <L,M,N> N print (L first , M second) {  
 System.*out*.println(first + " " + second);  
 return null;  
 }  
  
 public static void printList (List<?> first) { // List<? extends Animal>  
 System.*out*.println(first.size());  
 }  
}**

**//<T extends Animal & Serializable & Cloneable>**

**T can only be of type Animal(class)**

**While including interfaces, classes should come first**

**public static void printList (List<?> first) { }**

* **Can give any type of list**

**Design Patterns**

* **Flexibility (No need to change client code)**
* **Decoupling (Client depends only on interface)**
* **Reusability**
* **Encapsulation (Centralizing logic)**
* **Scalability (Easily add new implementations)**
* **Consistency**

**Singleton**

* **Single object / class**
* **Ex: DB connection object, Logger**
* **Member and methods are static**
* **Private constructor**

**class Singleton  
{  
 static Singleton *instance* = null;  
 public int x = 10;  
  
 // private constructor can't be accessed outside the class  
 private Singleton() { }  
  
 // Factory method to provide the users with instances  
 static public Singleton getInstance()  
 {  
 if (*instance* == null)  
 *instance* = new Singleton();  
  
 return *instance*;  
 }  
}**

**// Driver Class  
class Main  
{  
 public static void main(String args[])  
 {  
 Singleton a = Singleton.*getInstance*();  
 Singleton b = Singleton.*getInstance*();  
 a.x = a.x + 10;  
 System.*out*.println("Value of a.x = " + a.x);  
 System.*out*.println("Value of b.x = " + b.x);  
 }  
}**

**For multithreading**

**static public Singleton getInstance()  
{  
 if (*instance* == null) {  
  
 //static method, therefore class level locking  
 synchronized (Singleton.class) {  
 if (*instance* == null) {  
 *instance* = new Singleton();  
 }  
 }  
 }  
 return *instance*;  
}**

**How to break pattern**

* **Using reflections (gets hold of private constructor)**
* **Through serialization of singleton object and deserializing the same**

**Factory**

* **To instantiate objects without specifying class**
* **Object created by factory method**
* **Factory Class: Contains the logic to create objects.**
* **Product Interface: Defines the type of object to be created.**
* **Concrete Products: Implement the product interface.**
* **Dynamic Object Creation: The exact type of object to be created depends on dynamic conditions or inputs.**
* **Simplifying Object Instantiation: When object creation logic becomes complex, moving it into a factory improves maintainability.**
* **Decoupling Object Instantiation: The client code depends only on the interface or abstract class, not on specific implementations.**

**Common Use Cases:**

* **Logging Frameworks: The logger type (e.g., console, file) can be chosen dynamically.**
* **Parser Creation: Creating parsers for different file formats like XML, JSON, or CSV.**
* **GUI Frameworks: Factories provide UI components like buttons or text fields dynamically based on user preferences or configurations.**
* **Game Development: Creating different weapons, characters, or levels based on the game scenario.**

**Abstract Factory**

* **To create family of related products**
* **When system needs to be independent of the way products are created**
* **Families of Related Objects: You need to create groups of related or dependent objects without exposing their concrete classes.**
* **Consistent Object Families: Ensures that objects created together are compatible.**
* **Platform-Specific Implementations: Abstract factory provides a way to implement multiple variations for different environments.**

**Common Use Cases:**

* **Cross-Platform GUI Development: Different UI components (buttons, scrollbars, text fields) for platforms like Windows, Mac, or Linux.**
* **Database Connectivity: Providing different database drivers (e.g., MySQL, Oracle, SQL Server) for applications.**
* **Cloud Provider SDKs: Creating objects for AWS, Azure, or GCP services where each has its own configurations and APIs.**
* **Payment Gateways: Generating objects for different gateways (e.g., Stripe, PayPal) with consistent interfaces.**

**Builder**

* **When client wants to initiate only certain members of class**
* **Immutable objects after creation**
* **Readability, flexibility**
* **Avoid complex constructor**
* **Ex: StringBuilder, StringBuffer, configuring HTTP requests**

**Strategy**

* **Composition of object**
* **Switch between implementations at runtime**
* **Flexibility (No need to change client code)**
* **Decoupling (Client depends only on interface)**
* **Ex : Choosing between payments , card, gpay..**

**Template**

* **Base class provides the general workflow**
* **defines the skeleton of an algorithm in a base class but allows subclasses to override specific steps without changing the overall structure.**

**Structure:**

* **Abstract Class (Template Class): Contains the template method and some default implementations.**
* **Concrete Subclasses: Override specific steps of the algorithm.**

**Facade**

* **provides a unified interface to a complex subsystem**
* **reduce dependencies between the client and the subsystem classes.**
* **improve code readability and usability.**
* **Adds another layer, limited flexibility, decoupling**
* **Ex : JDBC Api (has different DBs), SLF4J**

**Structure:**

* **Facade: A high-level interface that provides a simplified way to access the subsystem.**
* **Subsystem Classes: The underlying complex system that contains multiple classes or methods.**
* **Client: The code that interacts with the facade instead of directly accessing the subsystem.**

**Facade**

* **avoid a permanent binding between abstraction and its implementation.**
* **have multiple dimensions of variation (e.g., shape and color in a graphics system).**
* **When changes to the implementation should not affect the abstraction or vice versa.**
* **Great choice when abstraction and implementation need to evolve independently while maintaining flexibility.**

**Structure:**

* **Abstraction: Defines the high-level control logic and maintains a reference to the implementor.**
* **Implementor Interface: Defines the low-level interface for implementation classes.**
* **Concrete Implementors: Provide specific implementations of the implementor interface.**
* **Refined Abstraction: Extends the abstraction and interacts with the implementor.**