**lambda**

A lambda expression is an inline code that implements a functional interface without creating a concrete or anonymous class. A lambda expression is basically an anonymous method.

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface.

Java lambda expression is treated as a function, so compiler does not create .class file.

Adv:

-To provide the implementation of Functional interface.

-Less coding.

Fewer Lines of Code − One of the most benefits of a lambda expression is to reduce the amount of code. We know that lambda expressions can be used only with a functional interface. For instance, Runnable is a functional interface, so we can easily apply lambda expressions.

Sequential and Parallel execution support by passing behavior as an argument in methods − By using Stream API in Java 8, the functions are passed to collection methods. Now it is the responsibility of collection for processing the elements either in a sequential or parallel manner.

Higher Efficiency − By using Stream API and lambda expressions, we can achieve higher efficiency (parallel execution) in case of bulk operations on collections. Also, lambda expression helps in achieving the internal iteration of collections rather than external iteration.

() -> {

//Body of no parameter lambda

}

eg. Drawable d2=()->{

System.out.println("Drawing "+width);

};

eg. list.forEach(

(n)->System.out.println(n)

);

eg. // using lambda to filter data

Stream<Product> filtered\_data = list.stream().filter(p -> p.price > 20000);

// using lambda to iterate through collection

filtered\_data.forEach(

product -> System.out.println(product.name+": "+product.price)

);

**functional interface**

A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit.

A functional interface can have any number of default methods. Runnable, ActionListener, Comparable are some of the examples of functional interfaces.

Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

@FunctionalInterface annotation is used to ensure that the functional interface can’t have more than one abstract method.

eg. public static void main(String args[])

{

// create anonymous inner class object

new Thread(new Runnable()

{

@Override

public void run()

{

System.out.println("New thread created");

}

}).start();

}

**Default Methods**

Java 8 introduces the “Default Method” or (Defender methods) feature, which allows the developer to add new methods to the interfaces without breaking their existing implementation. It provides the flexibility to allow interface to define implementation which will use as the default in a situation where a concrete class fails to provide an implementation for that method.

eg.

public interface oldInterface {

public void existingMethod();

default public void newDefaultMethod() {

System.out.println("New default method"

" is added in interface");

}

}

Default methods enable us to add new functionality to existing interfaces without breaking older implementation of these interfaces.

When we extend an interface that contains a default method, we can perform following:

-Not override the default method and will inherit the default method.

-Override the default method similar to other methods we override in subclass.

-Redeclare default method as abstract, which forces subclass to override it.

After introducing Default Method, it seems that interfaces and abstract classes are the same. However, they are still a different concept in Java 8.

The abstract class can define constructors. They are more structured and can have a state associated with them. While in contrast, default method can be implemented only in the terms of invoking other interface methods, with no reference to a particular implementation's state. Hence, both are used for different purposes and choosing between two really depends on the scenario context.

**Virtual Function**

The programmers coming from c++ background to Java normally think that where is the Virtual function? In Java there is no keyword named “virtual“.

In object-oriented programming, a virtual function or virtual method is a function or method whose behavior can be overridden within an inheriting class by a function with the same signature to provide the polymorphic behavior.

Therefore, according to definition, every non-static method in JAVA is by default virtual method except final and private methods. The methods which cannot be inherited for polymorphic behavior is not a virtual method.

eg.

public class Animal

{

public void eat()

{

System.out.println("I eat like a generic Animal.");

}

}

public class Wolf extends Animal

{

@Override

public void eat()

{

System.out.println("I eat like a wolf!");

}

}

public class Fish extends Animal

{

@Override

public void eat()

{

System.out.println("I eat like a fish!");

}

}

Abstract class is nothing but the pure virtual method equivalent to C++ in Java.

Question: why we say that static method is not a virtual method in Java?

Answer: static method is bound to the class itself, so calling the static method from class name or object does not provide the polymorphic behavior to the static method. We can override the static method however it will not give the advantage of the polymorphism.

**Pure Virtual Function**

A pure virtual function or pure virtual method is a virtual function that is required to be implemented by a derived class if the derived class is not abstract. Classes containing pure virtual methods are termed "abstract" and they cannot be instantiated directly. A subclass of an abstract class can only be instantiated directly if all inherited pure virtual methods have been implemented by that class or a parent class. Pure virtual methods typically have a declaration (signature) and no definition (implementation).

As an example, an abstract base class MathSymbol may provide a pure virtual function doOperation(), and derived classes Plus and Minus implement doOperation() to provide concrete implementations. Implementing doOperation() would not make sense in the MathSymbol class, as MathSymbol is an abstract concept whose behavior is defined solely for each given kind (subclass) of MathSymbol. Similarly, a given subclass of MathSymbol would not be complete without an implementation of doOperation().

Although pure virtual methods typically have no implementation in the class that declares them, pure virtual methods in C++ are permitted to contain an implementation in their declaring class, providing fallback or default behavior that a derived class can delegate to, if appropriate.

Pure virtual functions can also be used where the method declarations are being used to define an interface - similar to what the interface keyword in Java explicitly specifies. In such a use, derived classes will supply all implementations. In such a design pattern, the abstract class which serves as an interface will contain only pure virtual functions, but no data members or ordinary methods. In C++, using such purely abstract classes as interfaces works because C++ supports multiple inheritance. However, because many OOP languages do not support multiple inheritance, they often provide a separate interface mechanism. An example is the Java programming language.

**Constructors**

Constructors are used to initialize the object’s state. Like methods, a constructor also contains collection of statements (i.e. instructions) that are executed at time of Object creation. So, constructors are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).

Each time an object is created using new() keyword at least one constructor (it could be default constructor) is invoked to assign initial values to the data members of the same class.

Constructor(s) of a class must have same name as the class name in which it resides.

A constructor in Java cannot be abstract, final, static and Synchronized.

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

There are two type of constructor in Java:

No-argument constructor: A constructor that has no parameter is known as default constructor. If we don’t define a constructor in a class, then compiler creates default constructor (with no arguments) for the class. And 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.

Parameterized Constructor: A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with your own values, then use a parameterized constructor.

How constructors are different from methods in Java?

-Constructor(s) must have the same name as the class within which it defined while it is not necessary for the method in java.

-Constructor(s) do not return any type while method(s) have the return type or void if does not return any value.

-Constructor is called only once at the time of Object creation while method(s) can be called any numbers of time.

**Copy Constructor**

Like C++, Java also supports copy constructor. But, unlike C++, Java doesn’t create a default copy constructor if you don’t write your own.

Generally, the copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

That's helpful when we want to copy a complex object that has several fields, or when we want to make a deep copy of an existing object.

eg.

class Complex {

private double re, im;

// A normal parametrized constructor

public Complex(double re, double im) {

this.re = re;

this.im = im;

}

// copy constructor

Complex(Complex c) {

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

re = c.re;

im = c.im;

}

}

public static void main(String[] args) {

Complex c1 = new Complex(10, 15);

// Following involves a copy constructor call

Complex c2 = new Complex(c1);

// Note that following doesn't involve a copy constructor call as

// non-primitive variables are just references.

Complex c3 = c2;

System.out.println(c2); // toString() of c2 is called here

}

**Constructor Chaining**

Constructor chaining is the process of calling one constructor from another constructor with respect to current object.

Constructor chaining can be done in two ways:

Within same class: It can be done using this() keyword for constructors in same class

From base class: by using super() keyword to call constructor from the base class.

For super():

Constructor chaining occurs through inheritance. A sub class constructor’s task is to call super class’s constructor first. This ensures that creation of sub class’s object starts with the initialization of the data members of the super class. There could be any numbers of classes in inheritance chain. Every constructor calls up the chain till class at the top is reached.

This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.

eg.

class Temp

{

// default constructor 1

// default constructor will call another constructor

// using this keyword from same class

Temp()

{

// calls constructor 2

this(5);

System.out.println("The Default constructor");

}

// parameterized constructor 2

Temp(int x)

{

// calls constructor 3

this(5, 15);

System.out.println(x);

}

// parameterized constructor 3

Temp(int x, int y)

{

System.out.println(x \* y);

}

public static void main(String args[])

{

// invokes default constructor first

new Temp();

}

}

Alternative method: using Init block:

When we want certain common resources to be executed with every constructor we can put the code in the init block. Init block is always executed before any constructor, whenever a constructor is used for creating a new object.

eg.

class Temp

{

// block to be executed before any constructor.

{

System.out.println("init block");

}

// no-arg constructor

Temp()

{

System.out.println("default");

}

}

**Instance Initialization Block (init block)**

In a Java program, operations can be performed on methods, constructors and initialization blocks. Instance Initialization Blocks or IIB are used to initialize instance variables. IIBs are executed before constructors. They run each time when object of the class is created.

Initialization blocks are executed whenever the class is initialized and before constructors are invoked.

They are typically placed above the constructors within braces.

It is not at all necessary to include them in your classes.

eg.

class GfG

{

// Instance Initialization Block

{

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

}

// Constructor of GfG class

GfG()

{

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

}

public static void main(String[] args)

{

GfG a = new GfG();

}

}

We can also have multiple IIBs in a single class. If compiler finds multiple IIBs, then they all are executed from top to bottom i.e. the IIB which is written at top will be executed first.

You can have IIBs in parent class also. Instance initialization block code runs immediately after the call to super() in a constructor. The compiler executes parent class’s IIB before executing current class’s IIBs.

Instance Initialization Blocks run every time a new instance is created.

Initialization Blocks run in the order they appear in the program

The Instance Initialization Block is invoked after the parent class constructor is invoked (i.e. after super() constructor call)

**Destructors**

Because Java is a garbage collected language you cannot predict when (or even if) an object will be destroyed. Hence there is no direct equivalent of a destructor.

There is an inherited method called finalize, but this is called entirely at the discretion of the garbage collector. So, for classes that need to explicitly tidy up, the convention is to define a close method and use finalize only for sanity checking (i.e. if close has not been called do it now and log an error).

A destructor is a special method that gets called automatically as soon as the life-cycle of an object is finished. A destructor is called to de-allocate and free memory. The following tasks get executed when a destructor is called.

Releasing the release locks

Closing all the database connections or files

Releasing all the network resources

Other Housekeeping tasks

Recovering the heap space allocated during the lifetime of an object

Destructors in Java also known as finalizers are non-deterministic. The allocation and release of memory are implicitly handled by the garbage collector in Java.

Finalizers in Java have to be implicitly invoked since their invocation is not guaranteed

**Garbage Collector**

A garbage collector is a program that runs on the Java virtual machine to recover the memory by deleting the objects which are no longer in use or have finished their life-cycle. An object is said to be eligible for garbage collection if and only if the object is unreachable.

Let’s try to understand how garbage collection works in Java:

Garbage collection is mainly the process of marking or identifying the unreachable objects and deleting them to free the memory. The implementation lives in the JVM, the only requirement is that it should meet the JVM specifications

**Finalizers**

deprecated in Java9

The java.lang.Object.finalize() is called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

A subclass overrides the finalize method to dispose of system resources or to perform cleanup

Following is the declaration for java.lang.Object.finalize() method: - protected void finalize()

eg.

public static void main(String[] args) {

try {

// create a new ObjectDemo object

ObjectDemo cal = new ObjectDemo();

// print current time

System.out.println("" + cal.getTime());

// finalize cal

System.out.println("Finalizing...");

cal.finalize();

System.out.println("Finalized.");

} catch (Throwable ex) {

ex.printStackTrace();

}

}

**Polymorphism**

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Real life example of polymorphism: A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person possess different behavior in different situations. This is called polymorphism.

Polymorphism is considered as one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, so it means many forms.

In Java polymorphism is mainly divided into two types:

-Compile time Polymorphism

-Runtime Polymorphism

Compile time polymorphism: It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

Method Overloading: When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

eg. static int Multiply(int a, int b)

{

return a \* b;

}

// Method with the same name but 2 double parameter

static double Multiply(double a, double b)

{

return a \* b;

}

static int Multiply(int a, int b)

{

return a \* b;

}

// Method with the same name but 3 parameter

static int Multiply(int a, int b, int c)

{

return a \* b \* c;

}

Operator Overloading: Java also provide option to overload operators. For example, we can make the operator (‘+’) for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So, a single operator ‘+’ when placed between integer operands, adds them and when placed between string operands, concatenates them.

In java, only “+” operator can be overloaded:

To add integers

To concatenate strings

Runtime polymorphism: It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

Method overriding, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

eg.

class Parent {

void Print()

{

System.out.println("parent class");

}

}

class subclass1 extends Parent {

void Print()

{

System.out.println("subclass1");

}

}

**HashMap need to Override Hashcode() and equals() method**

Although there are lots of materials are available on internet and API document about the necessity of the overriding the hashcode() and equals() method in Java but lots of new developers still not able to understand the necessity of hashcode() method.

In this article, I will try to explain step by step the need of overriding hashcode() method in Java.

Few Thump rules:

-If two objects are same then they must return same value in hashcode() and equals() method whenever invoked.

-It is not necessary that two different objects must have different hashcode values. it might be possible that they share common hash bucket.

JVM assigns unique hashcode value to each object when they are created in memory and if developers don’t override the hashcode method then there is no way the two object returns same hashcode value.

As the question comes in your mind that equals() method is used to compare objects that they are having same value or not but why should we override the hashcode method ?

The answer to the question is for the hash technique-based data structures like HashMap and HashTable.

As you can see in above diagram that every object is placed in Hash bucket depending on the hashcode they have. It is not necessary that every different object must have different hashcode. hashcode is used to narrow the search result. When we try to insert any key in HashMap first it checks whether any other object present with same hashcode and if yes then it checks for the equals() method. If two objects are same then HashMap will not add that key instead it will replace the old value by new one.

package com.G2.Collections;

import java.util.HashMap;

class Movie {

private String name, actor;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getActor() {

return actor;

}

public void setActor(String actor) {

this.actor = actor;

}

public int getReleaseYr() {

return releaseYr;

}

public void setReleaseYr(int releaseYr) {

this.releaseYr = releaseYr;

}

private int releaseYr;

}

public class HashMapDemo {

public static void main(String[] args) {

Movie m = new Movie();

m.setActor("Akshay");

m.setName("Thank You");

m.setReleaseYr(2011);

Movie m1 = new Movie();

m1.setActor("Akshay");

m1.setName("Khiladi");

m1.setReleaseYr(1993);

Movie m2 = new Movie();

m2.setActor("Akshay");

m2.setName("Taskvir");

m2.setReleaseYr(2010);

Movie m3 = new Movie();

m3.setActor("Akshay");

m3.setName("Taskvir");

m3.setReleaseYr(2010);

HashMap<Movie, String> map = new HashMap<Movie, String>();

map.put(m, "ThankYou");

map.put(m1, "Khiladi");

map.put(m2, "Tasvir");

map.put(m3, "Duplicate Tasvir");

//Iterate over HashMap

for (Movie mm : map.keySet()) {

System.out.println(map.get(mm).toString());

}

Movie m4 = new Movie();

m4.setActor("Akshay");

m4.setName("Taskvir");

m4.setReleaseYr(2010);

We are trying to retrieve m2, by creating object m4 with exact values as of m2, However Hashcode method is not implemented and that why we are not able to get Object m2 \*/

if(map.get(m4) == null ){

System.out.println("----------------");

System.out.println("Object not found");

System.out.println("----------------");

}else{

System.out.println(map.get(m4).toString());

}

}

}

Output:

Khiladi

Tasvir

ThankYou

Duplicate Tasvir

—————-

Object not found

—————-

As you can see in above program:

Duplicate objects are added in Hashmap as a key (Because we have not overridden the hashcode and equals method)

We are not able to get back object from map (Because hashcode is not implemented)

Same program with equals and hashcode implementation:

package com.G2.Collections;

import java.util.HashMap;

class Movie {

private String name, actor;

@Override

public boolean equals(Object o) {

Movie m = (Movie) o;

return m.actor.equals(this.actor) && m.name.equals(this.name) && m.releaseYr == this.releaseYr;

}

@Override

public int hashCode() {

return actor.hashCode() + name.hashCode() + releaseYr;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getActor() {

return actor;

}

public void setActor(String actor) {

this.actor = actor;

}

public int getReleaseYr() {

return releaseYr;

}

public void setReleaseYr(int releaseYr) {

this.releaseYr = releaseYr;

}

private int releaseYr;

}

public class HashMapDemo {

public static void main(String[] args) {

Movie m = new Movie();

m.setActor("Akshay");

m.setName("Thank You");

m.setReleaseYr(2011);

Movie m1 = new Movie();

m1.setActor("Akshay");

m1.setName("Khiladi");

m1.setReleaseYr(1993);

Movie m2 = new Movie();

m2.setActor("Akshay");

m2.setName("Taskvir");

m2.setReleaseYr(2010);

Movie m3 = new Movie();

m3.setActor("Akshay");

m3.setName("Taskvir");

m3.setReleaseYr(2010);

HashMap<Movie, String> map = new HashMap<Movie, String>();

map.put(m, "ThankYou");

map.put(m1, "Khiladi");

map.put(m2, "Tasvir");

map.put(m3, "Duplicate Tasvir");

// Iterate over HashMap

for (Movie mm : map.keySet()) {

System.out.println(map.get(mm).toString());

}

Movie m4 = new Movie();

m4.setActor("Akshay");

m4.setName("Taskvir");

m4.setReleaseYr(2010);

if (map.get(m4) == null) {

System.out.println("----------------");

System.out.println("Object not found");

System.out.println("----------------");

} else {

System.out.println("----------------");

System.out.println(map.get(m4).toString());

System.out.println("----------------");

}

}

}

Output:

Khiladi

Duplicate Tasvir

ThankYou

—————-

Duplicate Tasvir

—————-

As you can see :

Duplicate Keys are not added instead there values are replaced.

Now the object is retrieved from the Map.

**Immutable Classes**

Immutable class means that once an object is created, we cannot change its content. In Java, all the wrapper classes (like Integer, Boolean, Byte, Short) and String class is immutable. We can create our own immutable class as well.

Following are the requirements:

-The class must be declared as final (So that child classes can’t be created)

-Data members in the class must be declared as final (So that we can’t change the value of it after object creation)

-A parameterized constructor

-Getter method for all the variables in it

-No setters (To not have the option to change the value of the instance variable)

eg.

public final class Student

{

final String name;

final int regNo;

final Age age;

public Student(String name, int regNo, Age age) {

this.name = name;

this.regNo = regNo;

// Objects should be deep cloned

Age cloneAge = new Age();

cloneAge.setDay(age.getDay());

cloneAge.setMonth(age.getMonth());

cloneAge.setYear(age.getYear());

this.age = cloneAge;

}

public String getName() {

return name;

}

public int getRegNo() {

return regNo;

}

public Age getAge() {

Age cloneAge = new Age();

cloneAge.setDay(this.age.getDay());

cloneAge.setMonth(this.age.getMonth());

cloneAge.setYear(this.age.getYear());

return cloneAge;

}

}

Immutable classes provide a lot of advantages especially when used correctly in a multi-threaded environment. The only disadvantage is that they consume more memory than the traditional class since upon each modification of them a new object is created in the memory. But, a developer should not overestimate the memory consumption as its negligible compared to the advantages provided by these types of classes.

**Abstraction**

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc. in the car. This is what abstraction is.

-An abstract class is a class that is declared with abstract keyword.

-An abstract method is a method that is declared without an implementation.

-An abstract class may or may not have all abstract methods. Some of them can be concrete methods

-A method defined abstract must always be redefined in the subclass, thus making overriding compulsory OR either make subclass itself abstract.

-Any class that contains one or more abstract methods must also be declared with abstract keyword.

-There can be no object of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator.

-An abstract class can have parametrized constructors and default constructor is always present in an abstract class.

-In java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

eg.

abstract class Shape

{

String color;

// these are abstract methods

abstract double area();

public abstract String toString();

// abstract class can have constructor

public Shape(String color) {

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

this.color = color;

}

// this is a concrete method

public String getColor() {

return color;

}

}

class Circle extends Shape

{

double radius;

public Circle(String color,double radius) {

// calling Shape constructor

super(color);

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

this.radius = radius;

}

@Override

double area() {

return Math.PI \* Math.pow(radius, 2);

}

@Override

public String toString() {

return "Circle color is " + super.color +

"and area is : " + area();

}

}

class Rectangle extends Shape{

double length;

double width;

public Rectangle(String color,double length,double width) {

// calling Shape constructor

super(color);

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

this.length = length;

this.width = width;

}

@Override

double area() {

return length\*width;

}

@Override

public String toString() {

return "Rectangle color is " + super.color +

"and area is : " + area();

}

}

**Encapsulation**

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Other way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.

Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.

As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.

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

You can also have validators in setter methods to validate the incoming data.

eg.

public class Encapsulate

{

//private variables declared

//these can only be accessed by

//public methods of class

private String geekName;

private int geekRoll;

private int geekAge;

//get method for age to access

//private variable geekAge

public int getAge()

{

return geekAge;

}

//get method for name to access

//private variable geekName

public String getName()

{

return geekName;

}

//get method for roll to access

//private variable geekRoll

public int getRoll()

{

return geekRoll;

}

//set method for age to access

//private variable geekage

public void setAge( int newAge)

{

geekAge = newAge;

}

//set method for name to access

//private variable geekName

public void setName(String newName)

{

geekName = newName;

}

//set method for roll to access

//private variable geekRoll

public void setRoll( int newRoll)

{

geekRoll = newRoll;

}

}

Advantages of Encapsulation:

Data Hiding: The user will have no idea about the inner implementation of the class. It will not be visible to the user that how the class is storing values in the variables. He only knows that we are passing the values to a setter method and variables are getting initialized with that value.

Increased Flexibility: We can make the variables of the class as read-only or write-only depending on our requirement. If we wish to make the variables as read-only then we have to omit the setter methods like setName(), setAge() etc. from the above program or if we wish to make the variables as write-only then we have to omit the get methods like getName(), getAge() etc. from the above program

Reusability: Encapsulation also improves the re-usability and easy to change with new requirements.

Testing code is easy: Encapsulated code is easy to test for unit testing.

**Encapsulation vs Abstraction**

Encapsulation is data hiding (information hiding) while Abstraction is detail hiding (implementation hiding).

While encapsulation groups together data and methods that act upon the data, data abstraction deals with exposing the interface to the user and hiding the details of implementation.

Adv:

-It reduces the complexity of viewing the things.

-Avoids code duplication and increases reusability.

-Helps to increase security of an application or program as only important details are provided to the user.

**Inheritance**

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in java by which one class is allow to inherit the features (fields and methods) of another class.

Important terminology:

Super Class: The class whose features are inherited is known as super class (or a base class or a parent class).

Sub Class: The class that inherits the other class is known as sub class (or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

Reusability: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

The keyword used for inheritance is extends.

Example: In below example of inheritance, class Bicycle is a base class, class MountainBike is a derived class which extends Bicycle class and class Test is a driver class to run program.

class Bicycle {

// the Bicycle class has two fields

public int gear;

public int speed;

// the Bicycle class has one constructor

public Bicycle(int gear, int speed) {

this.gear = gear;

this.speed = speed;

}

// the Bicycle class has three methods

public void applyBrake(int decrement) {

speed -= decrement;

}

public void speedUp(int increment) {

speed += increment;

}

}

// derived class

class MountainBike extends Bicycle {

// the MountainBike subclass adds one more field

public int seatHeight;

// the MountainBike subclass has one constructor

public MountainBike(int gear,int speed, int startHeight) {

// invoking base-class(Bicycle) constructor

super(gear, speed);

seatHeight = startHeight;

}

// the MountainBike subclass adds one more method

public void setHeight(int newValue) {

seatHeight = newValue;

}

}

In above program, when an object of MountainBike class is created, a copy of the all methods and fields of the superclass acquire memory in this object. That is why, by using the object of the subclass we can also access the members of a superclass.

Please note that during inheritance only object of subclass is created, not the superclass.

In practice, inheritance and polymorphism are used together in java to achieve fast performance and readability of code.

Types of Inheritance in Java:

Below are the different types of inheritance which is supported by Java.

Single Inheritance: In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.

Multilevel Inheritance: In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent’s members.

Hierarchical Inheritance: In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class. In below image, the class A serves as a base class for the derived class B,C and D.

Multiple Inheritance (Through Interfaces): In Multiple inheritance, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support multiple inheritance with classes. In java, we can achieve multiple inheritance only through Interfaces. In image below, Class C is derived from interface A and B.

Important facts about inheritance in Java:

Default superclass: Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object class.

Superclass can only be one: A superclass can have any number of subclasses. But a subclass can have only one superclass. This is because Java does not support multiple inheritance with classes. Although with interfaces, multiple inheritance is supported by java.

Inheriting Constructors: A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

Private member inheritance: A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods (like getters and setters) for accessing its private fields, these can also be used by the subclass.

What all can be done in a Subclass?

In sub-classes we can inherit members as is, replace them, hide them, or supplement them with new members:

-The inherited fields can be used directly, just like any other fields.

-We can declare new fields in the subclass that are not in the superclass.

-The inherited methods can be used directly as they are.

-We can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it (as in example above, toString() method is overridden).

-We can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.

-We can declare new methods in the subclass that are not in the superclass.

-We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

**Method Hiding**

While inheritance, we can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.

In Java, if the name of a derived class static function is the same as a base class static function then the base class static function shadows (or conceals) the derived class static function. For example, the following Java code prints “A.fun()”

Note: Static method is a class property, so if a static method is called from a class name or object having a class container then the method of that class is called not the object’s method.

eg.

// Parent class

class A {

static void fun() {

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

}

}

// B is inheriting A

// Base class

class B extends A {

static void fun()

{

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

}

}

// Driver Method

public class Main {

public static void main(String args[]) {

A a = new B();

a.fun(); // prints A.fun();

// B a = new B();

// a.fun(); // prints B.fun()

// the variable type decides the method

// being invoked, not the assigned object type

}

}

If we make both A.fun() and B.fun() as non-static then the above program would print “B.fun()”. While both methods are static types, the variable type decides the method being invoked, not the assigned object type

**Interfaces**

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

Interfaces specify what a class must do and not how. It is the blueprint of the class.

An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So, it specifies a set of methods that the class has to implement.

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.

A Java library example is, Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement interface use implements keyword.

Why do we use interface?

-It is used to achieve total abstraction.

-Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.

-It is also used to achieve loose coupling.

Interfaces are used to implement abstraction. So, the question arises why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.

eg.

interface Player {

final int id = 10;

int move();

}

Prior to JDK 8, interface could not define implementation. We can now add default implementation for interface methods. This default implementation has special use and does not affect the intention behind interfaces.

Suppose we need to add a new function in an existing interface. Obviously, the old code will not work as the classes have not implemented those new functions. So, with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

eg.

interface In1 {

final int a = 10;

default void display() {

System.out.println("hello");

}

}

// A class that implements the interface.

class TestClass implements In1 {

// Driver Code

public static void main (String[] args) {

TestClass t = new TestClass();

t.display();

}

}

Another feature that was added in JDK 8 is that we can now define static methods in interfaces which can be called independently without an object. Note: these methods are not inherited.

interface In1 {

final int a = 10;

static void display() {

System.out.println("hello");

}

}

// A class that implements the interface.

class TestClass implements In1 {

// Driver Code

public static void main (String[] args) {

In1.display();

}

}

Important points about interface or summary of article:

-We can’t create instance (interface can’t be instantiated) 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 extend another interface or interfaces (more than one interface).

-A class that implements interface must implements all the methods in interface.

-All the methods are public and abstract. And all the fields are public, static, and final.

-It is used to achieve multiple inheritance.

-It is used to achieve loose coupling.

New features added in interfaces in JDK 9:

From Java 9 onwards, interfaces can contain following also

-Static methods

-Private methods

-Private Static methods

**OOPS**

List of OOP Concepts in Java

There are four main OOP concepts in Java. These are:

Abstraction. Abstraction means using simple things to represent complexity. We all know how to turn the TV on, but we don’t need to know how it works in order to enjoy it. In Java, abstraction means simple things like objects, classes, and variables represent more complex underlying code and data. This is important because it lets avoid repeating the same work multiple times.

Encapsulation. This is the practice of keeping fields within a class private, then providing access to them via public methods. It’s a protective barrier that keeps the data and code safe within the class itself. This way, we can re-use objects like code components or variables without allowing open access to the data system-wide.

Inheritance. This is a special feature of Object-Oriented Programming in Java. It lets programmers create new classes that share some of the attributes of existing classes. This lets us build on previous work without reinventing the wheel.

Polymorphism. This Java OOP concept lets programmers use the same word to mean different things in different contexts. One form of polymorphism in Java is method overloading. That’s when different meanings are implied by the code itself. The other form is method overriding. That’s when the different meanings are implied by the values of the supplied variables. See more on this below.

OOP, concepts in Java work by letting programmers create components that can be re-used in different ways, but still maintain security.

Best Practices for OOP Concepts in Java:

Since the aim of OOP concepts in Java is to save time without sacrificing security and ease of use, the best practices are all oriented toward advancing that main goal.

DRY (Don’t Repeat Yourself). This is the core concept in Java. You should never have two blocks of identical code in two different places. Instead, have one method you use for different applications.

If you expect your Java code to change in the future, encapsulate it by making all variables and methods private at the outset. As the code changes, increase access to “protected” as needed, but not too public.

Single Responsibility. Another best practice for OOP concepts in Java is the Single Responsibility Principle. Simply put, a class should always have only one functionality. That way, it can be called and/or extended on its own when new uses arise for it, without causing coupling between different functionalities.

Open Closed Design. Make all methods and classes Closed for modification but Open for an extension. That way, tried and tested code can remain static but can be modified to perform new tasks as needed.

**Overloading**

Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters or both. Overloading is related to compile-time (or static) polymorphism.

What is the advantage?

We don’t have to create and remember different names for functions doing the same thing. For example, in our code, if overloading was not supported by Java, we would have to create method names like sum1, sum2, … or sum2Int, sum3Int, … etc.

-Can we overload methods on return type?

We cannot overload by return type. This behavior is same in C++. Refer this for details

However, Overloading methods on return type are possible in cases where the data type of the function being called is explicitly specified. Look at the examples below:

-Can we overload static methods?

The answer is ‘Yes’. We can have two or more static methods with same name, but differences in input parameters. For example, consider the following Java program. Refer this for details.

-Can we overload methods that differ only by static keyword?

We cannot overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same). See following Java program for example. Refer this for details.

-Can we overload main() in Java?

Like other static methods, we can overload main() in Java. Refer overloading main() in Java for more details.

**Overriding**

In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

Method overriding is one of the ways by which java achieve Run Time Polymorphism. The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

Rules:

Overriding and Access-Modifiers: The access modifier 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: If we don’t want a method to be overridden, we declare it as final. Please see Using final with Inheritance.

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.

The following table summarizes what happens when you define a method with the same signature as a method in a super-class.

Private methods cannot be overridden: Private methods cannot be overridden as they are bonded during compile time. Therefore, we can’t even override private methods in a subclass.

The overriding method must have same return type (or subtype): From Java 5.0 onwards it is possible to have different return type for an overriding method in child class, but child’s return type should be sub-type of parent’s return type. This phenomenon is known as covariant return type.

Invoking overridden method from sub-class: We can call parent class method in overriding method using super keyword.

Overriding and constructor: We cannot override constructor as parent and child class can never have constructor with same name (Constructor name must always be same as Class name).

Overriding and Exception-Handling: Below are two rules to note when overriding methods related to exception-handling.

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

Rule#2: If the super-class overridden method does throws an exception, subclass overriding method can only throw same, subclass exception. Throwing parent exception in Exception hierarchy will lead to compile time error. Also, there is no issue if subclass overridden method is not throwing any exception.

Overriding and abstract method: Abstract methods in an interface or abstract class are meant to be overridden in derived concrete classes otherwise a compile-time error will be thrown.

Overriding and synchronized/strictfp method: 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.

**Comparator**

Comparator interface is used to order the objects of user-defined classes. A comparator object is capable of comparing two objects of two different classes.

Unlike Comparable, Comparator is external to the element type we are comparing. It’s a separate class. We create multiple separate classes (that implement Comparator) to compare by different members.

Collections class has a second sort() method and it takes Comparator. The sort() method invokes the compare() to sort objects.

How does Collections.Sort() work?

Internally the Sort method does call Compare method of the classes it is sorting. To compare two elements, it asks “Which is greater?” Compare method returns -1, 0 or 1 to say if it is less than, equal, or greater to the other. It uses this result to then determine if they should be swapped for its sort.

eg.

class Sortbyroll implements Comparator<Student> {

// Used for sorting in ascending order of

// roll number

public int compare(Student a, Student b) {

return a.rollno - b.rollno;

}

}

class Sortbyname implements Comparator<Student> {

// Used for sorting in ascending order of

// roll name

public int compare(Student a, Student b) {

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

}

}

**Comparable**

A comparable object is capable of comparing itself with another object. The class itself must implements the java.lang.Comparable interface to compare its instances.

Consider a Movie class that has members like, rating, name, year. Suppose we wish to sort a list of Movies based on year of release. We can implement the Comparable interface with the Movie class, and we override the method compareTo() of Comparable interface.

eg.

class Movie implements Comparable<Movie> {

private double rating;

private String name;

private int year;

// Used to sort movies by year

public int compareTo(Movie m) {

return this.year - m.year;

}

// Constructor

public Movie(String nm, double rt, int yr) {

this.name = nm;

this.rating = rt;

this.year = yr;

}

// Getter methods for accessing private data

public double getRating() { return rating; }

public String getName() { return name; }

public int getYear() { return year; }

}

Now, suppose we want sort movies by their rating and names also. When we make a collection element comparable (by having it implement Comparable), we get only one chance to implement the compareTo() method. The solution is using Comparator.

Comparable is meant for objects with natural ordering which means the object itself must know how it is to be ordered. For example, Roll Numbers of students. Whereas, Comparator interface sorting is done through a separate class.

Logically, Comparable interface compares “this” reference with the object specified and Comparator in Java compares two different class objects provided.

If any class implements Comparable interface in Java then collection of that object either List or Array can be sorted automatically by using Collections.sort() or Arrays.sort() method and objects will be sorted based on their natural order defined by CompareTo method.

**Collections**

In Java, a separate framework named the “Collection Framework” has been defined in JDK 1.2 which holds all the collection classes and interface in it.

The Collection interface (java.util.Collection) and Map interface (java.util.Map) are the two main “root” interfaces of Java collection classes.

Adv:

Consistent API: The API has a basic set of interfaces like Collection, Set, List, or Map, all the classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have some common set of methods.

Reduces programming effort: A programmer doesn’t have to worry about the design of the Collection but rather he can focus on its best use in his program. Therefore, the basic concept of Object-oriented programming (i.e.) abstraction has been successfully implemented.

Increases program speed and quality: Increases performance by providing high-performance implementations of useful data structures and algorithms because in this case, the programmer need not think of the best implementation of a specific data structure. He can simply use the best implementation to drastically boost the performance of his algorithm/program.

The collection framework contains multiple interfaces where every interface is used to store a specific type of data. The following are the interfaces present in the framework.

1. Iterable Interface: This is the root interface for the entire collection framework. The collection interface extends the iterable interface. Therefore, inherently, all the interfaces and classes implement this interface. The main functionality of this interface is to provide an iterator for the collections. Therefore, this interface contains only one abstract method which is the iterator. It returns the:

Iterator iterator();

2. Collection Interface: This interface implements the iterable interface and is implemented by all the classes in the collection framework. This interface contains all the basic methods which every collection has like adding the data into the collection, removing the data, clearing the data, etc. All these methods are implemented in this interface because these methods are implemented by all the classes irrespective of their style of implementation. And also, having these methods in this interface ensures that the names of the methods are universal for all the collections. Therefore, in short, we can say that this interface builds a foundation on which the collection classes are implemented.

3. List Interface: This is a child interface of the collection interface. This interface is dedicated to the data of the list type in which we can store all the ordered collection of the objects. This also allows duplicate data to be present in it. This list interface is implemented by various classes like ArrayList, Vector, Stack, etc. Since all the subclasses implement the list, we can instantiate a list object with any of these classes. For example,

List <T> al = new ArrayList<> ();

List <T> ll = new LinkedList<> ();

List <T> v = new Vector<> ();

Where T is the type of the object

The classes which implement the List interface are as follows:

ArrayList: ArrayList provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. The size of an ArrayList is increased automatically if the collection grows or shrinks if the objects are removed from the collection. Java ArrayList allows us to randomly access the list. ArrayList cannot be used for primitive types, like int, char, etc. We will need a wrapper class for such cases.

LinkedList: LinkedList class is an implementation of the LinkedList data structure which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node.

Vector: A vector provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. This is identical to ArrayList in terms of implementation. However, the primary difference between a vector and an ArrayList is that a Vector is synchronized and an ArrayList is non-synchronized.

Stack: Stack class models and implements the Stack data structure. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek. The class can also be referred to as the subclass of Vector.

Note: Stack is a subclass of Vector and a legacy class. It is **thread** **safe** which might be an overhead in an environment where thread safety is not needed. An alternate to Stack is to use ArrayDeque which is not thread safe and faster array implementation.

4. Queue Interface: As the name suggests, a queue interface maintains the FIFO (First In First Out) order similar to a real-world queue line. This interface is dedicated to storing all the elements where the order of the elements matter. For example, whenever we try to book a ticket, the tickets are sold at the first come first serve basis. Therefore, the person whose request arrives first into the queue gets the ticket. There are various classes like PriorityQueue, Deque, ArrayDeque, etc. Since all these subclasses implement the queue, we can instantiate a queue object with any of these classes. For example,

Queue <T> pq = new PriorityQueue<> ();

Queue <T> ad = new ArrayDeque<> ();

Where T is the type of the object.

The most frequently used implementation of the queue interface is the PriorityQueue.

Priority Queue: A PriorityQueue is used when the objects are supposed to be processed based on the priority. It is known that a queue follows the First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority and this class is used in these cases. The PriorityQueue is based on the priority heap. The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

5. Deque Interface: This is a very slight variation of the queue data structure. Deque, also known as a double-ended queue, is a data structure where we can add and remove the elements from both the ends of the queue. This interface extends the queue interface. The class which implements this interface is ArrayDeque. Since this class implements the deque, we can instantiate a deque object with this class. For example,

Deque<T> ad = new ArrayDeque<> ();

Where T is the type of the object.

The class which implements the deque interface is ArrayDeque.

ArrayDeque: ArrayDeque class which is implemented in the collection framework provides us with a way to apply resizable-array. This is a special kind of array that grows and allows users to add or remove an element from both sides of the queue. Array deques have no capacity restrictions and they grow as necessary to support usage.

6. Set Interface: A set is an unordered collection of objects in which duplicate values cannot be stored. This collection is used when we wish to avoid the duplication of the objects and wish to store only the unique objects. This set interface is implemented by various classes like **HashSet**, **TreeSet**, **LinkedHashSet**, etc. Since all the subclasses implement the set, we can instantiate a set object with any of these classes. For example,

Set<T> hs = new HashSet<> ();

Set<T> lhs = new LinkedHashSet<> ();

Set<T> ts = new TreeSet<> ();

Where T is the type of the object.

The following are the classes which implement the Set interface:

HashSet: The HashSet class is an inherent implementation of the hash table data structure. The objects that we insert into the HashSet do not guarantee to be inserted in the same order. The objects are inserted based on their hashcode. This class also allows the insertion of NULL elements.

LinkedHashSet: A LinkedHashSet is very similar to a HashSet. The difference is that this uses a doubly linked list to store the data and retains the ordering of the elements.

7. Sorted Set Interface: This interface is very similar to the set interface. The only difference is that this interface has extra methods that maintain the ordering of the elements. The sorted set interface extends the set interface and is used to handle the data which needs to be sorted. The class which implements this interface is TreeSet. Since this class implements the SortedSet, we can instantiate a SortedSet object with this class. For example,

SortedSet<T> ts = new TreeSet<> ();

Where T is the type of the object.

The class which implements the sorted set interface is TreeSet.

TreeSet: The TreeSet class uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface. It can also be ordered by a Comparator provided at set creation time, depending on which constructor is used. Uses Red Black BST, which self-balanced tree for ordering. Insertion is O(logn) and Extraction is O(logn)

8. Map Interface: A map is a data structure which supports the key-value pair mapping for the data. This interface doesn’t support duplicate keys because the same key cannot have multiple mappings. A map is useful if there is a data and we wish to perform operations on the basis of the key. This map interface is implemented by various classes like HashMap, TreeMap, SortedMap, etc. Since all the subclasses implement the map, we can instantiate a map object with any of these classes. For example,

Map<T> hm = new HashMap<> ();

Map<T> tm = new TreeMap<> ();

Map<T> sm = new SortedMap<> ();

Where T is the type of the object.

The frequently used implementation of a Map interface is a HashMap.

HashMap: HashMap provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs. To access a value in a HashMap, we must know its key. HashMap uses a technique called Hashing. Hashing is a technique of converting a large String to small String that represents the same String so that the indexing and search operations are faster.

LinkedHashMap: LinkedHashMap is just like HashMap with an additional feature of maintaining an order of elements inserted into it (uses double linked list)

TreeMap: It is also like hashmap but it maintains natural ordering of data, ie. Data is stored in sorted format. Uses Red black BST, which is a self-balanced tree.

Notes:

HashMap- not thread safe, allows one null

TreeMap- not thread safe, map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

ConcurrentHashMap- Thread safe, not allow null, concurrency level- 16, no locking while read, but locking while updation, Segment locking or bucket locking. Hence at a time 16 update operations can be performed by threads.

HashTable- Thread safe, not allow null, other features similar to HashMap

**HashTable vs ConcurrentHashMap**

Hashtable is belongs to the Collection framework; ConcurrentHashMap belongs to the Executor framework.

Hashtable uses single lock for whole data. ConcurrentHashMap uses multiple locks on segment level (16 by default) instead of object level i.e. whole Map.

ConcurrentHashMap locking is applied only for updates. In case of retrievals, it allows full concurrency, retrievals reflect the results of the most recently completed update operations. So, reads can happen very fast while writes are done with a lock.

ConcurrentHashMap doesn't throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it and does not allow null values.

ConcurrentHashMap returns Iterator, which fails-safe (i.e. iterator will make a copy of the internal data structure) on concurrent modification.

ConcurrentHashMap uses a database shards logic (Segment<K, V>[] segments) is known as Concurrency-Level, i.e. divides the data into shards(segments) than puts locks on each shard (segment) instead of putting a single lock for whole data (Map). The default value is 16

The following analogy helps you get understand the concept only (not logic)

Assume Hashtable and ConcurrentHashMap are two types of Homes.

Hashtable locks home's main door.

ConcurrentHashMap locks specific room door instead of main door.

Which is more efficient for threaded applications?

ConcurrentHashMap is more efficient for threaded applications.

HashTable implementation:

public class Hashtable<K, V> extends Dictionary<K, V> implements Map<K, V>, Cloneable, Serializable {

public synchronized V put(K key, V value) {

...

}

public synchronized V get(Object key) {

...

}

public synchronized boolean containsKey(Object key) {

...

}

public synchronized boolean containsValue(Object value) {

...

}

}

**HashTable vs Synchronized HashMap**

The only difference between Hashtable and Synchronized Map is that later is not a legacy and you can wrap any Map to create it's synchronized version by using Collections. synchronizedMap() method. On the other hand, ConcurrentHashMap is specially designed for concurrent use i.e. more than one thread.

crunchifyHashTableObject = new Hashtable<String, Integer>();

crunchifySynchronizedMapObject = Collections.synchronizedMap(new HashMap<String, Integer>());

crunchifyConcurrentHashMapObject = new ConcurrentHashMap<String, Integer>();

**Types of Linked List**

LL

DLL

CLL

**How ArrayList Works**

When you initialize an ArrayList, an array of size 10 (default capacity) is created and an element added to the ArrayList is actually added to this array. 10 is the default size and it can be passed as a parameter while initializing the ArrayList.

When adding a new element, if the array is full, then a new array of 50% more the initial size is created and the last array is copied to this new array so that now there are empty spaces for the new element to be added.

Since the underlying data-structure used is an array, it is fairly easy to add a new element to the ArrayList as it is added to the end of the list. When an element is to be added anywhere else, say the beginning, then all the elements shall have to move one position to the right to create an empty space at the beginning for the new element to be added. This process is time-consuming (linear-time). But the Advantage of ArrayList is that retrieving an element at any position is very fast (constant-time), as underlying it is simply using an array of objects.

**HashMap Collision**

The Java HashMap already handles collisions for you in this way. All you need to do is ensure you are overriding and implementing the key's hashCode() and equals() method.

Each hash code will map to a specific "bucket". Each bucket contains a linked list for the case of collisions.

The only way to avoid (or rather minimize) collisions is to create a hash function that creates the best possible distribution of values throughout the HashMap. Depending on the density of your HashMap and the quality of your hash code, collisions are almost inevitable, hence the need to override the two methods.

To override the two methods:

public class MyObject {

String var1;

int var2;

//...

public boolean equals(Object obj) {

if(obj == null) return false;

if(this == obj) return true; // Reference equality

if(!(obj instanceof MyObject)) return false;

MyObject myObj = MyObject(obj);

return (var1.equals(myObj.var1)) && (var2 == myObj.var2);

}

public int hashCode {

return var1.hashCode() ^ var2;

}

}

The collision only occurs if you use the same key." not really, it occurs when 2 different objects have the same hascode.

**Treefy Factor**

As of java 8, when the Entries in a linked list reaches 8 (MIN\_TREEIFY\_CAPACITY;), it converts the linked list to a Balanced Tree. This improved the performance a million times.

**HashMap Default Size & Load Factor**

hashmap default size = 16

load factor = 0.75

**Immutable Objects**

An immutable object is an object that will not change its internal state after creation. Immutable objects are very useful in multithreaded applications because they can be shared between threads without synchronization. Immutable objects are always thread safe.

An immutable object is an object whose internal state remains constant after it has been entirely created.

**String Immutable**

In Java, Strings are immutable. An obvious question that is quite prevalent in interviews is “Why Strings are designed as immutable in Java?”

The key benefits of keeping this class as immutable are caching, security, synchronization, and performance

Advantages of making string immutable:

-Pooling

-Syncronization

-Performance

-HashCode Caching

-Security

**String Pool**

The String is the most widely used data structure. Caching the String literals and reusing them saves a lot of heap space because different String variables refer to the same object in the String pool. String intern pool serves exactly this purpose.

Java String Pool is the special memory region where Strings are stored by the JVM. Since Strings are immutable in Java, the JVM optimizes the amount of memory allocated for them by storing only one copy of each literal String in the pool. This process is called interning:

String s1 = "Hello World";

String s2 = "Hello World";

assertThat(s1 == s2).isTrue();

Because of the presence of the String pool in the preceding example, two different variables are pointing to same String object from the pool, thus saving crucial memory resource.

**Checked Exception**

Checked: are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.

For example, consider the following Java program that opens file at location “C:\test\a.txt” and prints the first three lines of it. The program doesn’t compile, because the function main() uses FileReader() and FileReader() throws a checked exception FileNotFoundException. It also uses readLine() and close() methods, and these methods also throw checked exception IOException

To fix the above program, we either need to specify list of exceptions using throws, or we need to use try-catch block. We have used throws in the below program. Since FileNotFoundException is a subclass of IOException, we can just specify IOException in the throws list and make the above program compiler-error-free.

**Unchecked Exception**

Unchecked are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.

In Java exceptions under Error and RuntimeException classes are unchecked exceptions, everything else under throwable is checked.

Consider the following Java program. It compiles fine, but it throws ArithmeticException when run. The compiler allows it to compile, because ArithmeticException is an unchecked exception.

If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception

**Custom Exception**

The custom exception should extend RuntimeException if you want to make it unchecked else extend it with Exception.

With unchecked exceptions calling code method is not required to declare in its throws clause any subclasses of RuntimeException that might be thrown during the execution of the method but not caught.

As the calling method may not handle `RuntimeException``, one needs to be careful while throwing RuntimeException.

Runtime exceptions represent problems that are the result of a programming problem, and as such, the API client code cannot reasonably be expected to recover from them or to handle them in any way. Such problems include arithmetic exceptions, such as dividing by zero; pointer exceptions, such as trying to access an object through a null reference; and indexing exceptions, such as attempting to access an array element through an index that is too large or too small.

Runtime exceptions can occur anywhere in a program, and in a typical one they can be very numerous. Having to add runtime exceptions in every method declaration would reduce a program's clarity. Thus, the compiler does not require that you catch or specify runtime exceptions (although you can).

If you extend RuntimeException, you don't need to declare it in the throws clause (i.e. it's an unchecked exception). If you extend Exception, you do (it's a checked exception).

Some people argue that all exceptions should extend from RuntimeException, but if you want to force the user to handle the exception, you should extend Exception instead.

eg.

Java provides us facility to create our own exceptions which are basically derived classes of Exception. For example, MyException in below code extends the Exception class.

We pass the string to the constructor of the super class- Exception which is obtained using “getMessage()” function on the object created.

// A Class that represents use-defined expception

class MyException extends Exception {

public MyException(String s) {

// Call constructor of parent Exception

super(s);

}

}

// A Class that uses above MyException

public class Main {

// Driver Program

public static void main(String args[]) {

try{

// Throw an object of user defined exception

throw new MyException("GeeksGeeks");

} catch (MyException ex) {

System.out.println("Caught");

// Print the message from MyException object

System.out.println(ex.getMessage());

}

}

}

**Array vs ArrayList**

An array is basic functionality provided by Java. ArrayList is part of collection framework in Java. Therefore, array members are accessed using [], while ArrayList has a set of methods to access elements and modify them.

Array is a fixed size data structure while ArrayList is not. One need not to mention the size of Arraylist while creating its object. Even if we specify some initial capacity, we can add more elements.

Array can contain both primitive data types as well as objects of a class depending on the definition of the array. However, ArrayList only supports object entries, not the primitive data types.

Note: When we do arraylist.add(1); : it converts the primitive int data type into an Integer object.

Since ArrayList can’t be created for primitive data types, members of ArrayList are always references to objects at different memory locations (See this for details). Therefore, in ArrayList, the actual objects are never stored at contiguous locations. References of the actual objects are stored at contiguous locations.

In array, it depends whether the arrays are of primitive type or object type. In case of primitive types, actual values are contiguous locations, but in case of objects, allocation is similar to ArrayList.

Java ArrayList supports many additional operations like indexOf(), remove(), etc. These functions are not supported by Arrays.

**Streams API**

Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

The features of Java stream are –

A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.

Streams don’t change the original data structure, they only provide the result as per the pipelined methods.

Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Different Operations on Streams:-

Intermediate Operations:

map: The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.

List number = Arrays.asList(2,3,4,5);

List square = number.stream().map(x->x\*x).collect(Collectors.toList());

filter: The filter method is used to select elements as per the Predicate passed as argument.

List names = Arrays.asList("Reflection","Collection","Stream");

List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());

sorted: The sorted method is used to sort the stream.

List names = Arrays.asList("Reflection","Collection","Stream");

List result = names.stream().sorted().collect(Collectors.toList());

Terminal Operations:

collect: The collect method is used to return the result of the intermediate operations performed on the stream.

List number = Arrays.asList(2,3,4,5,3);

Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());

forEach: The forEach method is used to iterate through every element of the stream.

List number = Arrays.asList(2,3,4,5);

number.stream().map(x->x\*x).forEach(y->System.out.println(y));

reduce: The reduce method is used to reduce the elements of a stream to a single value.

The reduce method takes a BinaryOperator as a parameter.

List number = Arrays.asList(2,3,4,5);

int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

Important Points/Observations:

A stream consists of source followed by zero or more intermediate methods combined together (pipelined) and a terminal method to process the objects obtained from the source as per the methods described.

Stream is used to compute elements as per the pipelined methods without altering the original value of the object.

**Internal working of HashMap**

Hashing is a process of converting an object into integer form by using the method hashCode(). It’s necessary to write hashCode() method properly for better performance of HashMap. Here I am taking key of my own class so that I can override hashCode() method to show different scenarios.

hashCode() method:

hashCode() method is used to get the hash Code of an object. hashCode() method of object class returns the memory reference of object in integer form. Definition of hashCode() method is public native hashCode(). It indicates the implementation of hashCode() is native because there is not any direct method in java to fetch the reference of object. It is possible to provide your own implementation of hashCode().

In HashMap, hashCode() is used to calculate the bucket and therefore calculate the index.

equals() method:

equals method is used to check that 2 objects are equal or not. This method is provided by Object class. You can override this in your class to provide your own implementation.

HashMap uses equals() to compare the key whether they are equal or not. If equals() method return true, they are equal otherwise not equal.

Buckets:

A bucket is one element of HashMap array. It is used to store nodes. Two or more nodes can have the same bucket. In that case link list structure is used to connect the nodes. Buckets are different in capacity. A relation between bucket and capacity is as follows:

capacity = number of buckets \* load factor

A single bucket can have more than one nodes, it depends on hashCode() method. The better your hashCode() method is, the better your buckets will be utilized.

Index Calculation in Hashmap:

Hash code of key may be large enough to create an array. hash code generated may be in the range of integer and if we create arrays for such a range, then it will easily cause outOfMemoryException. So, we generate index to minimize the size of array. Basically, following operation is performed to calculate index.

index = hashCode(key) & (n-1).

where n is number of buckets or the size of array. In our example, I will consider n as default size that is 16.

Initially Empty hashMap: Here, the hashmap is size is taken as 16.

HashMap map = new HashMap();

Inserting Key-Value Pair: Putting one key-value pair in above HashMap

map.put(new Key("vishal"), 20);

Steps:

Calculate hash code of Key {“vishal”}. It will be generated as 118.

Calculate index by using index method it will be 6.

Create a node object as:

{

int hash = 118

// {"vishal"} is not a string but

// an object of class Key

Key key = {"vishal"}

Integer value = 20

Node next = null

}

Place this object at index 6, if no other object is presented there.

Inserting another Key-Value Pair: Now, putting other pair that is,

map.put(new Key("sachin"), 30);

Steps:

Calculate hashCode of Key {“sachin”}. It will be generated as 115.

Calculate index by using index method it will be 3.

Create a node object as:

{

int hash = 115

Key key = {"sachin"}

Integer value = 30

Node next = null

}

Place this object at index 3 if no other object is presented there.

In Case of collision: Now, putting another pair that is,

map.put(new Key("vaibhav"), 40);

Steps:

Calculate hash code of Key {“vaibhav”}. It will be generated as 118.

Calculate index by using index method it will be 6.

Create a node object as:

{

int hash = 118

Key key = {"vaibhav"}

Integer value = 40

Node next = null

}

Place this object at index 6 if no other object is presented there.

In this case a node object is found at the index 6 – this is a case of collision.

In that case, check via hashCode() and equals() method that if both the keys are same.

If keys are same, replace the value with current value.

Otherwise connect this node object to the previous node object via linked list and both are stored at index 6.

Using get method()

Now let’s try some get method to get a value. get(K key) method is used to get a value by its key. If you don’t know the key then it is not possible to fetch a value.

Fetch the data for key sachin:

map.get(new Key("sachin"));

Steps:

Calculate hash code of Key {“sachin”}. It will be generated as 115.

Calculate index by using index method it will be 3.

Go to index 3 of array and compare first element’s key with given key. If both are equals then return the value, otherwise check for next element if it exists.

In our case it is found as first element and returned value is 30.

Fetch the data for key vaibahv:

map.get(new Key("vaibhav"));

Steps:

Calculate hash code of Key {“vaibhav”}. It will be generated as 118.

Calculate index by using index method it will be 6.

Go to index 6 of array and compare first element’s key with given key. If both are equals then return the value, otherwise check for next element if it exists.

In our case it is not found as first element and next of node object is not null.

If next of node is null then return null.

If next of node is not null traverse to the second element and repeat the process 3 until key is not found or next is not null.

HashMap Changes in Java 8

As we know now that in case of hash collision entry objects are stored as a node in a linked-list and equals() method is used to compare keys. That comparison to find the correct key with in a linked-list is a linear operation so in a worst-case scenario the complexity becomes O(n).

To address this issue, Java 8 hash elements use balanced trees instead of linked lists after a certain threshold is reached. Which means HashMap starts with storing Entry objects in linked list but after the number of items in a hash becomes larger than a certain threshold, the hash will change from using a linked list to a balanced tree, which will improve the worst case performance from O(n) to O(log n).

Important Points

Time complexity is almost constant for put and get method until rehashing is not done.

In case of collision, i.e. index of two or more nodes are same, nodes are joined by link list i.e. second node is referenced by first node and third by second and so on.

If key given already exist in HashMap, the value is replaced with new value.

hash code of null key is 0.

When getting an object with its key, the linked list is traversed until the key matches or null is found on next field.

**Internal working of HashSet**

we can say that a set will never contain duplicate elements. But how in java Set interface implemented classes like HashSet, LinkedHashSet, TreeSet etc. achieve this uniqueness. In this post, we will discuss the hidden truth behind this uniqueness.

// predefined HashSet class

public class HashSet {

// A HashMap object

private transient HashMap map;

// A Dummy value(PRESENT) to associate with an Object in the Map

private static final Object PRESENT = new Object();

// default constructor of HashSet class

// It creates a HashMap by calling

// default constructor of HashMap class

public HashSet() {

map = new HashMap<>();

}

// add method

// it calls put() method on map object

// and then compares it's return value with null

public boolean add(E e) {

return map.put(e, PRESENT)==null;

}

// Other methods in Hash Set

}

Now as you can see that whenever we create a HashSet, it internally creates a HashMap and if we insert an element into this HashSet using add() method, it actually call put() method on internally created HashMap object with element you have specified as its key and constant Object called “PRESENT” as its value. So, we can say that a Set achieves uniqueness internally through HashMap. Now the whole story comes around how a HashMap and put() method internally works.

As we know in a HashMap each key is unique and when we call put(Key, Value) method, it returns the previous value associated with key, or null if there was no mapping for key. So in add() method we check the return value of map.put(key, value) method with null value.

If map.put(key, value) returns null, then the statement “map.put(e, PRESENT) == null” will return true and element is added to the HashSet(internally HashMap).

If map.put(key, value) returns old value of the key, then the statement “map.put(e, PRESENT) == null” will return false and element is not added to the HashSet(internally HashMap).

As LinkedHashSet extends HashSet, so it internally calls constructors of HashSet using super(). Similarly creating an object of TreeSet class internally creates object of Navigable Map as backing map.

**Semaphores**

A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

Working of semaphore:

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit.

If the semaphore’s count is greater than zero, then the thread acquires a permit, which causes the semaphore’s count to be decremented.

Otherwise, the thread will be blocked until a permit can be acquired.

When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore’s count to be incremented.

If there is another thread waiting for a permit, then that thread will acquire a permit at that time.

Java provide Semaphore class in java.util.concurrent package that implements this mechanism, so you don’t have to implement your own semaphores.

Constructors in Semaphore class: There are two constructors in Semaphore class.

-Semaphore(int num)

-Semaphore(int num, boolean how)

Here, num specifies the initial permit count. Thus, it specifies the number of threads that can access a shared resource at any one time. If it is one, then only one thread can access the resource at any one time. By default, all waiting threads are granted a permit in an undefined order. By setting how to true, you can ensure that waiting threads are granted a permit in the order in which they requested access.

Using Semaphores as Locks (preventing race condition):

We can use a semaphore to lock access to a resource, each thread that wants to use that resource must first call acquire( ) before accessing the resource to acquire the lock. When the thread is done with the resource, it must call release( ) to release lock. Here is an example that demonstrate this:

The output can be different in different executions of above program, but final value of count variable will always remain 0.

Explanation of above program: https://www.geeksforgeeks.org/semaphore-in-java/

The program uses a semaphore to control access to the count variable, which is a static variable within the Shared class. Shared.count is incremented five times by thread A and decremented five times by thread B. To prevent these two threads from accessing Shared.count at the same time, access is allowed only after a permit is acquired from the controlling semaphore. After access is complete, the permit is released. In this way, only one thread at a time will access Shared.count, as the output shows.

Notice the call to sleep( ) within run( ) method inside MyThread class. It is used to “prove” that accesses to Shared.count are synchronized by the semaphore. In run( ), the call to sleep( ) causes the invoking thread to pause between each access to Shared.count. This would normally enable the second thread to run. However, because of the semaphore, the second thread must wait until the first has released the permit, which happens only after all accesses by the first thread are complete. Thus, Shared.count is first incremented five times by thread A and then decremented five times by thread B. The increments and decrements are not intermixed at assembly code.

Without the use of the semaphore, accesses to Shared.count by both threads would have occurred simultaneously, and the increments and decrements would be intermixed. To confirm this, try commenting out the calls to acquire( ) and release( ). When you run the program, you will see that access to Shared.count is no longer synchronized, thus you will not always get count value 0.

Eg.

**class** FooBar1 {

**private** **int** n;

Semaphore s = **new** Semaphore(0);

Semaphore s2 = **new** Semaphore(1);

**public** FooBar1(**int** n) {

**this**.n = n;

}

**public** **void** foo(Runnable printFoo) **throws** InterruptedException {

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

s2.acquire();

printFoo.run();

s.release();

}

}

**public** **void** bar(Runnable printBar) **throws** InterruptedException {

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

s.acquire();

printBar.run();

s2.release();

}

}

}

Methods supported by semaphores:

void acquire() : This method acquires a permit, if one is available and returns immediately, reducing the number of available permits by one.If the current thread is interrupted while waiting for a permit then InterruptedException is thrown.

void acquire(int permits) : This method acquires the given number of permits, if they are available, and returns immediately, reducing the number of available permits by the given amount. If the current thread is interrupted while waiting for a permit then InterruptedException is thrown.

void release() : This method releases a permit, increasing the number of available permits by one. If any threads are trying to acquire a permit, then one is selected and given the permit that was just released.

void release(int permits) : This method releases the given number of permits, increasing the number of available permits by that amount. If any threads are trying to acquire permits, then one is selected and given the permits that were just released. If the number of available permits satisfies that thread’s request then that thread is (re)enabled for thread scheduling purposes; otherwise the thread will wait until sufficient permits are available.

boolean tryAcquire() : This method acquires a permit, if one is available and returns immediately, with the value true, reducing the number of available permits by one. If no permit is available then this method will return immediately with the value false.

boolean tryAcquire(int permits) : This method acquires the given number of permits, if they are available, and returns immediately, with the value true, reducing the number of available permits by the given amount.If insufficient permits are available then this method will return immediately with the value false.

**Mutex**

A mutex (or mutual exclusion) is the simplest type of synchronizer – it ensures that only one thread can execute the critical section of a computer program at a time.

To access a critical section, a thread acquires the mutex, then accesses the critical section, and finally releases the mutex. In the meantime, all other threads block till the mutex releases. As soon as a thread exits the critical section, another thread can enter the critical section.

A mutex is used for serial access to a resource while a semaphore limits access to a resource up to a set number. You can think of a mutex as a semaphore with an access count of 1. Whatever you set your semaphore count to, that many threads can access the resource before the resource is blocked.

While in case of a mutex only one thread can access a critical section, Semaphore allows a fixed number of threads to access a critical section. Therefore, we can also implement a mutex by setting the number of allowed threads in a Semaphore to one.

Let's now create thread-safe version of SequenceGenerator using Semaphore:

public class SequenceGeneratorUsingSemaphore extends SequenceGenerator {

private Semaphore mutex = new Semaphore(1);

@Override

public int getNextSequence() {

try {

mutex.acquire();

return super.getNextSequence();

} catch (InterruptedException e) {

// exception handling code

} finally {

mutex.release();

}

}

}

eg.

public class SequenceGeneratorUsingSynchronizedBlock extends SequenceGenerator {

private Object mutex = new Object();

@Override

public int getNextSequence() {

synchronized (mutex) {

return super.getNextSequence();

}

}

}

**ThreadPool**

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.

What is ThreadPool in Java?

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. Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.

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 an object of ExecutorService and pass a set of tasks to it. ThreadPoolExecutor class allows to set the core and maximum pool size. The runnable that are run by a particular thread are executed sequentially.

Executor Thread Pool Methods:

Method Description

newFixedThreadPool(int) Creates a fixed size thread pool.

newCachedThreadPool() Creates a thread pool that creates new

threads as needed, but will reuse previously

constructed threads when they are available

newSingleThreadExecutor() Creates a single thread.

In case of a fixed thread pool, if all threads are being currently run by the executor then the pending tasks are placed in a queue and are executed when a thread becomes idle.

eg.

import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

// Task class to be executed (Step 1)

class Task implements Runnable {

private String name;

public Task(String s) {

name = s;

}

// Prints task name and sleeps for 1s

// This Whole process is repeated 5 times

public void run() {

try {

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

if (i==0) {

Date d = new Date();

SimpleDateFormat ft = new SimpleDateFormat("hh:mm:ss");

System.out.println("Initialization Time for"

+ " task name - "+ name +" = " +ft.format(d));

//prints the initialization time for every task

} else {

Date d = new Date();

SimpleDateFormat ft = new SimpleDateFormat("hh:mm:ss");

System.out.println("Executing Time for task name - "+

name +" = " +ft.format(d));

// prints the execution time for every task

}

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");

// creates a thread pool with MAX\_T no. of

// threads as the fixed pool size(Step 2)

ExecutorService pool = Executors.newFixedThreadPool(MAX\_T);

// passes the Task objects to the pool to execute (Step 3)

pool.execute(r1);

pool.execute(r2);

pool.execute(r3);

pool.execute(r4);

pool.execute(r5);

// pool shutdown ( Step 4)

pool.shutdown();

}

}

One of the main advantages of using this approach is when you want to process 100 requests at a time, but do not want to create 100 Threads for the same, so as to reduce JVM overload. You can use this approach to create a ThreadPool of 10 Threads and you can submit 100 requests to this ThreadPool.

ThreadPool will create maximum of 10 threads to process 10 requests at a time. After process completion of any single Thread,

ThreadPool will internally allocate the 11th request to this Thread

and will keep on doing the same to all the remaining requests.

Risks in using Thread Pools:

Deadlock: While deadlock can occur in any multi-threaded program, thread pools introduce another case of deadlock, one in which all the executing threads are waiting for the results from the blocked threads waiting in the queue due to the unavailability of threads for execution.

Thread Leakage: Thread Leakage occurs if a thread is removed from the pool to execute a task but not returned to it when the task completed. As an example, if the thread throws an exception and pool class does not catch this exception, then the thread will simply exit, reducing the size of the thread pool by one. If this repeats many times, then the pool would eventually become empty and no threads would be available to execute other requests.

Resource Thrashing: If the thread pool size is very large then time is wasted in context switching between threads. Having more threads than the optimal number may cause starvation problem leading to resource thrashing as explained.

Important Points:

-Don’t queue tasks that concurrently wait for results from other tasks. This can lead to a situation of deadlock as described above.

-Be careful while using threads for a long-lived operation. It might result in the thread waiting forever and would eventually lead to resource leakage.

-The Thread Pool has to be ended explicitly at the end. If this is not done, then the program goes on executing and never ends. Call shutdown() on the pool to end the executor. If you try to send another task to the executor after shutdown, it will throw a RejectedExecutionException.

One needs to understand the tasks to effectively tune the thread pool. If the tasks are very contrasting then it makes sense to use different thread pools for different types of tasks so as to tune them properly.

You can restrict maximum number of threads that can run in JVM, reducing chances of JVM running out of memory.

If you need to implement your loop to create new threads for processing, using ThreadPool will help to process faster, as ThreadPool does not create new Threads after it reached its max limit.

After completion of Thread Processing, ThreadPool can use the same Thread to do another process (so saving the time and resources to create another Thread.)

Tuning Thread Pool

The optimum size of the thread pool depends on the number of processors available and the nature of the tasks. On a N processor system for a queue of only computation type processes, a maximum thread pool size of N or N+1 will achieve the maximum efficiency. But tasks may wait for I/O and in such a case we take into account the ratio of waiting time(W) and service time(S) for a request; resulting in a maximum pool size of N\*(1+ W/S) for maximum efficiency.

The thread pool is a useful tool for organizing server applications. It is quite straightforward in concept, but there are several issues to watch for when implementing and using one, such as deadlock, resource thrashing. Use of executor service makes it easier to implement.

**Method Level Synchronization vs Block Level Synchronization**

Method Synchronization:

Synchronized methods enable a simple strategy for preventing the thread interference and memory consistency errors. If an Object is visible to more than one threads, all reads or writes to that Object’s fields are done through the synchronized method.

It is not possible for two invocations for synchronized methods to interleave. If one thread is executing the synchronized method, all others thread that invoke synchronized method on the same Object will have to wait until first thread is done with the Object.

eg.

There can be two trains (more than two) which need to use same line at same time so there is chance of collision. Therefore, to avoid collision we need to synchronize the line in which multiple want to run.

Example: Synchronized access to getLine() method on the same Object

// Example that shows multiple threads

// can execute the same method but in

// synchronized way.

class Line {

// if multiple threads(trains) trying to access

// this synchronized method on the same Object

// but only one thread will be able

// to execute it at a time.

synchronized public void getLine() {

for (int i = 0; i < 3; i++) {

System.out.println(i);

Try {

Thread.sleep(400);

} catch (Exception e) {

System.out.println(e);

}

}

}

}

class Train extends Thread {

// Reference variable of type Line.

Line line;

Train(Line line) {

this.line = line;

}

@Override

public void run() {

line.getLine();

}

}

class GFG {

public static void main(String[] args) {

Line obj = new Line();

// we are creating two threads which share

// same Object.

Train train1 = new Train(obj);

Train train2 = new Train(obj);

// both threads start executing.

train1.start();

train2.start();

}

}

Block Synchronization:

If we only need to execute some subsequent lines of code not all lines (instructions) of code within a method, then we should synchronize only block of the code within which required instructions are exists.

For example, let’s suppose there is a method that contains 100 lines of code but there are only 10 lines (one after one) of code which contain critical section of code i.e. these lines can modify (change) the Object’s state. So, we only need to synchronize these 10 lines of code method to avoid any modification in state of the Object and to ensure that other threads can execute rest of the lines within the same method without any interruption.

eg.

import java.io.\*;

import java.util.\*;

public class Geek {

String name = "";

public int count = 0;

public void geekName(String geek, List<String> list) {

// Only one thread is permitted

// to change geek's name at a time.

synchronized(this) {

name = geek;

count++; // how many threads change geek's name.

}

// All other threads are permitted

// to add geek name into list.

list.add(geek);

}

}

class GFG {

public static void main (String[] args) {

Geek gk = new Geek();

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

gk.geekName("mohit", list);

System.out.println(gk.name);

}

}

When a thread enters into synchronized method or block, it acquires lock and once it completes its task and exits from the synchronized method, it releases the lock.

When thread enters into synchronized instance method or block, it acquires Object level lock and when it enters into synchronized static method or block it acquires class level lock.

Java synchronization will throw null pointer exception if Object used in synchronized block is null. For example, If in synchronized(instance), instance is null then it will throw null pointer exception.

In Java, wait(), notify() and notifyAll() are the important methods that are used in synchronization.

You cannot apply java synchronized keyword with the variables.

Don’t synchronize on the non-final field on synchronized block because the reference to the non-final field may change anytime and then different threads might synchronize on different objects i.e. no synchronization at all.

**Static method synchronization**

In simple words a static synchronized method will lock the class instead of the object, and it will lock the class because the keyword static means: "class instead of instance".

The keyword synchronized means that only one thread can access the method at a time.

And static synchronized mean: Only one thread can access the class at one time.

Suppose there are multiple static synchronized methods (m1, m2, m3, m4) in a class, and suppose one thread is accessing m1, then no other thread at the same time can access any other static synchronized methods.

eg.

class Table{

synchronized static void printTable(int n){

for(int i=1;i<=10;i++){

System.out.println(n\*i);

try{

Thread.sleep(400);

}catch(Exception e){}

}

}

}

A static synchronized method printTable(int n) in class Table is equivalent to the following declaration by using block synchronization

static void printTable(int n) {

synchronized (Table.class) { // Synchronized block on class A

// ...

}

}

**Notify vs NotifyAll**---Applies on objects

notifyAll: The java.lang.Object.notifyAll() wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.

The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object. The awakened threads will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened threads enjoy no reliable privilege or disadvantage in being the next thread to lock this object.

This method should only be called by a thread that is the owner of this object's monitor.

notify: The java.lang.Object.notify() wakes up a single thread that is waiting on this object's monitor. If many threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the wait methods.

This method should only be called by a thread that is the owner of this object's monitor. A thread becomes the owner of the object's monitor in one of three ways −

-By executing a synchronized instance method of that object.

-By executing the body of a synchronized statement that synchronizes on the object.

-For objects of type Class, by executing a synchronized static method of that class.

-Only one thread at a time can own an object's monitor.

**Singleton Design**

Singleton is a part of Gang of Four design pattern and it is categorized under creational design patterns

Singleton pattern is a design pattern which restricts a class to instantiate its multiple objects. It is nothing but a way of defining a class. Class is defined in such a way that only one instance of the class is created in the complete execution of a program or project. It is used where only a single instance of a class is required to control the action throughout the execution. A singleton class shouldn’t have multiple instances in any case and at any cost. Singleton classes are used for logging, driver objects, caching and thread pool, database connections.

An implementation of singleton class should have following properties:

It should have only one instance: This is done by providing an instance of the class from within the class. Outer classes or subclasses should be prevented to create the instance. This is done by making the constructor private in java so that no class can access the constructor and hence cannot instantiate it.

Instance should be globally accessible: Instance of singleton class should be globally accessible so that each class can use it. In Java, it is done by making the access-specifier of instance public

Initialization Types of Singleton

Singleton class can be instantiated by two methods:

Early initialization: In this method, class is initialized whether it is to be used or not. The main advantage of this method is its simplicity. You initiate the class at the time of class loading. Its drawback is that class is always initialized whether it is being used or not.

Lazy initialization: In this method, class in initialized only when it is required. It can save you from instantiating the class when you don’t need it. Generally, lazy initialization is used when we create a singleton class.

Examples of Singleton class:

java.lang.Runtime : Java provides a class Runtime in its lang package which is singleton in nature.

java.awt.Desktop : The Desktop class allows a Java application to launch associated applications registered on the native desktop to handle a URI or a file

Applications of Singleton classes:

Hardware interface access: The use of singleton depends on the requirements. Singleton classes are also used to prevent concurrent access of class.

Logger: Singleton classes are used in log file generations. Log files are created by the logger class object

Configuration File: This is another potential candidate for Singleton pattern because this has a performance benefit as it prevents multiple users to repeatedly access and read the configuration file or properties file

Cache: We can use the cache as a singleton object as it can have a global point of reference and for all future calls to the cache object the client application will use the in-memory object.

Method 1: Classic Implementation

// Classical Java implementation of singleton

// design pattern

class Singleton {

private static Singleton obj;

// private constructor to force use of

// getInstance() to create Singleton object

private Singleton() {}

public static Singleton getInstance() {

if (obj==null)

obj = new Singleton();

return obj;

}

}

Note that Singleton obj is not created until we need it and call getInstance() method. This is called lazy instantiation. The main problem with above method is that it is not thread safe.

Method 2: make getInstance() synchronized

// Thread Synchronized Java implementation of

// singleton design pattern

class Singleton

{

private static Singleton obj;

private Singleton() {}

// Only one thread can execute this at a time

public static synchronized Singleton getInstance()

{

if (obj==null)

obj = new Singleton();

return obj;

}

}

Here using synchronized makes sure that only one thread at a time can execute getInstance().

The main disadvantage of this is method is that using synchronized every time while creating the singleton object is expensive and may decrease the performance of your program. However if performance of getInstance() is not critical for your application this method provides a clean and simple solution.

Method 3: Eager Instantiation

// Static initializer-based Java implementation of

// singleton design pattern

class Singleton

{

private static Singleton obj = new Singleton();

private Singleton() {}

public static Singleton getInstance() {

return obj;

}

}

Here we have created instance of singleton in static initializer. JVM executes static initializer when the class is loaded and hence this is guaranteed to be thread safe. Use this method only when your singleton class is light and is used throughout the execution of your program.

Method 4 (Best): Use “Double Checked Locking”

If you notice carefully once an object is created synchronization is no longer useful because now obj will not be null and any sequence of operations will lead to consistent results.

So we will only acquire lock on the getInstance() once, when the obj is null. This way we only synchronize the first way through, just what we want.

// Double Checked Locking based Java implementation of

// singleton design pattern

class Singleton {

private volatile static Singleton obj;

private Singleton() {}

public static Singleton getInstance() {

if (obj == null) {

// To make thread safe

synchronized (Singleton.class) {

// check again as multiple threads

// can reach above step

if (obj==null)

obj = new Singleton();

}

}

return obj;

}

}

We have declared the obj volatile which ensures that multiple threads offer the obj variable correctly when it is being initialized to Singleton instance. This method drastically reduces the overhead of calling the synchronized method every time.

Method 5: Bill Pugh Singleton Implementation:

Prior to Java5, memory model had a lot of issues and above methods caused failure in certain scenarios in multithreaded environment. So, Bill Pugh suggested a concept of inner static classes to use for singleton.

// Java code for Bill Pugh Singleton Implementaion

public class GFG {

private GFG() {

// private constructor

}

// Inner class to provide instance of class

private static class BillPughSingleton {

private static final GFG INSTANCE = new GFG();

}

public static GFG getInstance() {

return BillPughSingleton.INSTANCE;

}

}

When the singleton class is loaded, inner class is not loaded and hence doesn’t create object when loading the class. Inner class is created only when getInstance() method is called. So, it may seem like eager initialization but it is lazy initialization.

This is the most widely used approach as it doesn’t use synchronization.

Eager initialization is easy to implement but it may cause resource and CPU time wastage. Use it only if cost of initializing a class is less in terms of resources or your program will always need the instance of class.

By using Static block in Eager initialization, we can provide exception handling and also can control over instance.

Using synchronized we can create singleton class in multi-threading environment also but it can cause slow performance, so we can use Double check locking mechanism.

Bill Pugh implementation is most widely used approach for singleton classes. Most developers prefer it because of its simplicity and advantages.

**Volatile keyword**

Using volatile is yet another way (like synchronized, atomic wrapper) of making class thread safe. Thread safe means that a method or class instance can be used by multiple threads at the same time without any problem.

Suppose that two threads are working on SharedObj. If two threads run on different processors each thread may have its own local copy of sharedVariable. If one thread modifies its value the change might not reflect in the original one in the main memory instantly. This depends on the write policy of cache. Now the other thread is not aware of the modified value which leads to data inconsistency.

volatile vs synchronized:

Before we move on let’s take a look at two important features of locks and synchronization.

Mutual Exclusion: It means that only one thread or process can execute a block of code (critical section) at a time.

Visibility: It means that changes made by one thread to shared data are visible to other threads.

Java’s synchronized keyword guarantees both mutual exclusion and visibility. If we make the blocks of threads that modifies the value of shared variable synchronized only one thread can enter the block and changes made by it will be reflected in the main memory. All other thread trying to enter the block at the same time will be blocked and put to sleep.

In some cases, we may only desire the visibility and not atomicity. Use of synchronized in such situation is an overkill and may cause scalability problems. Here volatile comes to the rescue. Volatile variables have the visibility features of synchronized but not the atomicity features. The values of volatile variable will never be cached and all writes and reads will be done to and from the main memory. However, use of volatile is limited to very restricted set of cases as most of the times atomicity is desired.

**Double Checked Locking**

Double-checked locking (also known as "double-checked locking optimization") is a software design pattern used to reduce the overhead of acquiring a lock by testing the locking criterion (the "lock hint") before acquiring the lock. Locking occurs only if the locking criterion check indicates that locking is required.

The pattern, when implemented in some language/hardware combinations, can be unsafe. At times, it can be considered an anti-pattern.

It is typically used to reduce locking overhead when implementing "lazy initialization" in a multi-threaded environment, especially as part of the Singleton pattern. Lazy initialization avoids initializing a value until the first time it is accessed.

**Inner Class**

In Java, it is also possible to nest classes (a class within a class). The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.

To access the inner class, create an object of the outer class, and then create an object of the inner class:

eg.

class OuterClass {

int x = 10;

class InnerClass {

int y = 5;

}

}

public class MyMainClass {

public static void main(String[] args) {

OuterClass myOuter = new OuterClass();

OuterClass.InnerClass myInner = myOuter.new InnerClass();

System.out.println(myInner.y + myOuter.x);

}

}

Private Inner Class

Unlike a "regular" class, an inner class can be private or protected. If you don't want outside objects to access the inner class, declare the class as private:

eg.

class OuterClass {

int x = 10;

private class InnerClass {

int y = 5;

}

}

public class MyMainClass {

public static void main(String[] args) {

OuterClass myOuter = new OuterClass();

OuterClass.InnerClass myInner = myOuter.new InnerClass();

System.out.println(myInner.y + myOuter.x);

}

} // gives error as can't access

Static Inner Class

An inner class can also be static, which means that you can access it without creating an object of the outer class:

eg.

class OuterClass {

int x = 10;

static class InnerClass {

int y = 5;

}

}

public class MyMainClass {

public static void main(String[] args) {

OuterClass.InnerClass myInner = new OuterClass.InnerClass();

System.out.println(myInner.y);

}

}

Access Outer Class from Inner Class

One advantage of inner classes, is that they can access attributes and methods of the outer class:

eg.

class OuterClass {

int x = 10;

class InnerClass {

public int myInnerMethod() {

return x;

}

}

}

public class MyMainClass {

public static void main(String[] args) {

OuterClass myOuter = new OuterClass();

OuterClass.InnerClass myInner = myOuter.new InnerClass();

System.out.println(myInner.myInnerMethod());

}

}

// Outputs 10

**Association**

Association refers to the relationship between multiple objects. It refers to how objects are related to each other and how they are using each other's functionality. Composition and aggregation are two types of association.

**Composition**

Composition in java is the design technique to implement has-a relationship in classes. We can use java inheritance or Object composition in java for code reuse.

The composition is the strong type of association. An association is said to composition if an Object owns another object and another object cannot exist without the owner object. Consider the case of Human having a heart. Here Human object contains the heart and heart cannot exist without Human.

eg.

public class Person {

//composition has-a relationship

private Job job;

public Person(){

this.job=new Job();

job.setSalary(1000L);

}

public long getSalary() {

return job.getSalary();

}

}

eg.

public class Car {

//engine is a mandatory part of the car

private final Engine engine;

public Car () {

engine = new Engine();

}

}

//Engine Object

class Engine {}

**Aggregation**

Aggregation is a weak association. An association is said to be aggregation if both Objects can exist independently. For example, a Team object and a Player object. The team contains multiple players but a player can exist without a team.

eg.

//Team

public class Team {

//players can be 0 or more

private List players;

public Car () {

players = new ArrayList();

}

}

//Player Object

class Player {}

**Throwable**

The Throwable class is the superclass of every error and exception in the Java language. Only objects that are one of the subclasses this class are thrown by any “Java Virtual Machine” or may be thrown by the Java throw statement. For the motives of checking of exceptions during compile-time, Throwable and any subclass of Throwable which is not also a subclass of either Error or RuntimeException are considered as checked exceptions.

Throwable class is the root class of Java Exception Hierarchy and is inherited by two subclasses:

1.Exception

2.Error

**Exception vs Error**

Error: An Error “indicates serious problems that a reasonable application should not try to catch.”

Both Errors and Exceptions are the subclasses of java.lang.Throwable class. Errors are the conditions which cannot get recovered by any handling techniques. It surely causes termination of the program abnormally. Errors belong to unchecked type and mostly occur at runtime. Some of the examples of errors are Out of memory error or a System crash error.

They are defined in java.lang.Error package. All errors in java are unchecked type.

Examples: java.lang.StackOverflowError, java.lang.OutOfMemoryError

Exceptions: An Exception “indicates conditions that a reasonable application might want to catch.”

Exceptions are the conditions that occur at runtime and may cause the termination of program. But they are recoverable using try, catch and throw keywords. Exceptions are divided into two catagories : checked and unchecked exceptions. Checked exceptions like IOException known to the compiler at compile time while unchecked exceptions like ArrayIndexOutOfBoundException known to the compiler at runtime. It is mostly caused by the program written by the programmer.

They are defined in java.lang.Exception package. Exceptions include both checked as well as unchecked type. Examples: Checked Exceptions : SQLException, IOException,

Unchecked Exceptions : ArrayIndexOutOfBoundException, NullPointerException, ArithmeticException.

**Anonymous Inner Class**

It is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain “extras” such as overloading methods of a class or interface, without having to actually subclass a class.

Anonymous inner classes are useful in writing implementation classes for listener interfaces in graphics programming.

An inner class declared without a class name is known as an anonymous inner class.

we declare and instantiate them at the same time.

In general, they are used whenever you need to override the method of a class or an interface.

public class Outer\_class {

public static void main(String args[]) {

AnonymousInner inner = new AnonymousInner() {

public void mymethod() {

System.out.println("This is an example of anonymous inner class");

}

};

inner.mymethod();

}

}

**Anonymous Objects**

we can use a method on an object without assigning it to any reference.

public class Tester {

public String message(){

return "Hello World!";

}

public static void main(String[] args) {

System.out.println(new Tester().message());

}

}

**Iterator**

Iterator is an interface which is made for Collection objects like List, Set. It comes inside java.util package and it was introduced in java 1.2 as public interface Iterator. To generate successive elements from a Collection, we can use java iterator.

It contains three methods:

boolean hasNext(): this method returns true if this Iterator has more element to iterate.

remove(): method remove the last element return by the iterator this method only calls once per call to next().

Object next(): Returns the next element in the iteration.

To implement an iterable data structure, we need to:

-Implement Iterable interface along with its methods in the said Data Structure

-Create an Iterator class which implements Iterator interface and corresponding methods

eg. To start custom iterator

class CustomDataStructure implements Iterable<> {

// code for data structure

public Iterator<> iterator() {

return new CustomIterator<>(this);

}

}

class CustomIterator<> implements Iterator<> {

// constructor

CustomIterator<>(CustomDataStructure obj) {

// initialize cursor

}

// Checks if the next element exists

public boolean hasNext() {

}

// moves the cursor/iterator to next element

public T next() {

}

// Used to remove an element. Implement only if needed

public void remove() {

// Default throws UnsupportedOperationException.

}

}

The Iterator class can also, be implemented as an inner class of the Data Structure class since it won’t be used elsewhere.

1. How next() and hasNext() work?

To implement an Iterator, we need a cursor or pointer to keep track of which element we currently are on. Depending on the underlying data structure, we can progress from one element to another. This is done in the next() method which returns the current element and the cursor advances to next element.

Before advancing the pointer, we check whether next element exists. i.e. we can picturize the behind-the-scenes code as follows:

While(iterator.hasNext()) { //if next element exists

next(); // advance the pointer

}

2. Initializing the cursor

The cursor initialization completely depends on the data structure. For example, in a linked list we would initialize cursor to the head element. In an array list, we would initialize cursor to the 0th element.

From the point of view of implementation:

If the Iterator class is implemented as an inner class, we can simply use “this” keyword (e.g. cursor = CustomDataStructure.this.element) to access the desired element

If the Iterator class is implemented as a separate class, we can pass this object of the data structure to the iterator class constructor as demonstrated in the example below.

Below program illustrates the use of Iterable interface:

Given below is a Custom Linked List which makes use of Generics. The linked list consists of Node objects which contain a Generic data value and pointer to next node. The class provides some standard ‘get’ methods like getHead() and getTail(), and the necessary Iterator() function, which has to be implemented while implementing Iterable interface.

Then the necessary custom class ‘ListIterator’ is created, which will implement the Iterator interface, along with it the functionalities of hasNext() and next() are also to be implemented. These two functions form the core of Iterable and Iterator interface.

Eg.

import java.util.Iterator;

// Custom Linked List class using Generics

class List<T> implements Iterable<T> {

Node<T> head, tail;

// add new Element at tail of the linked list in O(1)

public void add(T data) {

Node<T> node = new Node<>(data, null);

if (head == null)

tail = head = node;

else {

tail.setNext(node);

tail = node;

}

}

// return Head

public Node<T> getHead() {

return head;

}

// return Tail

public Node<T> getTail() {

return tail;

}

// return Iterator instance

public Iterator<T> iterator() {

return new ListIterator<T>(this);

}

}

class ListIterator<T> implements Iterator<T> {

Node<T> current;

// initialize pointer to head of the list for iteration

public ListIterator(List<T> list) {

current = list.getHead();

}

// returns false if next element does not exist

public boolean hasNext() {

return current != null;

}

// return current data and update pointer

public T next() {

T data = current.getData();

current = current.getNext();

return data;

}

// implement if needed

public void remove() {

throw new UnsupportedOperationException();

}

}

// Constituent Node of Linked List

class Node<T> {

T data;

Node<T> next;

public Node(T data, Node<T> next) {

this.data = data;

this.next = next;

}

// Setter getter methods for Data and Next Pointer

public void setData(T data) {

this.data = data;

}

public void setNext(Node<T> next) {

this.next = next;

}

public T getData() {

return data;

}

public Node<T> getNext() {

return next;

}

}

// Driver class

class Main {

public static void main(String[] args) {

// Create Linked List

List<String> myList = new List<>();

// Add Elements

myList.add("abc");

myList.add("mno");

myList.add("pqr");

myList.add("xyz");

// Iterate through the list using For Each Loop

for (String string : myList)

System.out.println(string);

}

}

**SOLID Principles**

We'll be discussing the SOLID principles of Object-Oriented Design.

First, we'll start by exploring the reasons they came about and why we should consider them when designing software. Then, we'll outline each principle alongside some example code to emphasize the point.

The Reason for SOLID Principles:

The SOLID principles were first conceptualized by Robert C. Martin in his 2000 paper, Design Principles and Design Patterns. These concepts were later built upon by Michael Feathers, who introduced us to the SOLID acronym. And in the last 20 years, these 5 principles have revolutionized the world of object-oriented programming, changing the way that we write software.

So, what is SOLID and how does it help us write better code? Simply put, Martin's and Feathers' design principles encourage us to create more maintainable, understandable, and flexible software. Consequently, as our applications grow in size, we can reduce their complexity and save ourselves a lot of headaches further down the road!

The following 5 concepts make up our SOLID principles:

Single Responsibility

Open/Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

While some of these words may sound daunting, they can be easily understood with some simple code examples. In the following sections, we'll take a deep dive into what each of these principles means, along with a quick Java example to illustrate each one.

Single Responsibility:

Let's kick things off with the single responsibility principle. As we might expect, this principle states that a class should only have one responsibility. Furthermore, it should only have one reason to change.

How does this principle help us to build better software? Let's see a few of its benefits:

Testing – A class with one responsibility will have far fewer test cases

Lower coupling – Less functionality in a single class will have fewer dependencies

Organization – Smaller, well-organized classes are easier to search than monolithic ones

Take, for example, a class to represent a simple book:

public class Book {

private String name;

private String author;

private String text;

//constructor, getters and setters

}

In this code, we store the name, author, and text associated with an instance of a Book.

Let's now add a couple of methods to query the text:

public class Book {

private String name;

private String author;

private String text;

//constructor, getters and setters

// methods that directly relate to the book properties

public String replaceWordInText(String word){

return text.replaceAll(word, text);

}

public boolean isWordInText(String word){

return text.contains(word);

}

}

Now, our Book class works well, and we can store as many books as we like in our application. But, what good is storing the information if we can't output the text to our console and read it?

Let's throw caution to the wind and add a print method:

public class Book {

//...

void printTextToConsole(){

// our code for formatting and printing the text

}

}

This code does, however, violate the single responsibility principle we outlined earlier. To fix our mess, we should implement a separate class that is concerned only with printing our texts:

public class BookPrinter {

// methods for outputting text

void printTextToConsole(String text){

//our code for formatting and printing the text

}

void printTextToAnotherMedium(String text){

// code for writing to any other location..

}

}

Awesome. Not only have we developed a class that relieves the Book of its printing duties, but we can also leverage our BookPrinter class to send our text to other media.

Whether it's email, logging, or anything else, we have a separate class dedicated to this one concern.

Open for Extension, Closed for Modification:

Now, time for the ‘O' – more formally known as the open-closed principle. Simply put, classes should be open for extension, but closed for modification. In doing so, we stop ourselves from modifying existing code and causing potential new bugs in an otherwise happy application.

Of course, the one exception to the rule is when fixing bugs in existing code.

Let's explore the concept further with a quick code example. As part of a new project, imagine we've implemented a Guitar class.

It's fully fledged and even has a volume knob:

public class Guitar {

private String make;

private String model;

private int volume;

//Constructors, getters & setters

}

We launch the application, and everyone loves it. However, after a few months, we decide the Guitar is a little bit boring and could do with an awesome flame pattern to make it look a bit more ‘rock and roll'.

At this point, it might be tempting to just open up the Guitar class and add a flame pattern – but who knows what errors that might throw up in our application.

Instead, let's stick to the open-closed principle and simply extend our Guitar class:

public class SuperCoolGuitarWithFlames extends Guitar {

private String flameColor;

//constructor, getters + setters

}

By extending the Guitar class we can be sure that our existing application won't be affected.

Liskov Substitution:

Next up on our list is Liskov substitution, which is arguably the most complex of the 5 principles. Simply put, if class A is a subtype of class B, then we should be able to replace B with A without disrupting the behavior of our program.

Let's just jump straight to the code to help wrap our heads around this concept:

public interface Car {

void turnOnEngine();

void accelerate();

}

Above, we define a simple Car interface with a couple of methods that all cars should be able to fulfill – turning on the engine, and accelerating forward.

Let's implement our interface and provide some code for the methods:

public class MotorCar implements Car {

private Engine engine;

//Constructors, getters + setters

public void turnOnEngine() {

//turn on the engine!

engine.on();

}

public void accelerate() {

//move forward!

engine.powerOn(1000);

}

}

As our code describes, we have an engine that we can turn on, and we can increase the power. But wait, its 2019, and Elon Musk has been a busy man.

We are now living in the era of electric cars:

public class ElectricCar implements Car {

public void turnOnEngine() {

throw new AssertionError("I don't have an engine!");

}

public void accelerate() {

//this acceleration is crazy!

}

}

By throwing a car without an engine into the mix, we are inherently changing the behavior of our program. This is a blatant violation of Liskov substitution and is a bit harder to fix than our previous 2 principles.

One possible solution would be to rework our model into interfaces that take into account the engine-less state of our Car.

Interface Segregation

The ‘I ‘ in SOLID stands for interface segregation, and it simply means that larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.

For this example, we're going to try our hands as zookeepers. And more specifically, we'll be working in the bear enclosure.

Let's start with an interface that outlines our roles as a bear keeper:

public interface BearKeeper {

void washTheBear();

void feedTheBear();

void petTheBear();

}

As avid zookeepers, we're more than happy to wash and feed our beloved bears. However, we're all too aware of the dangers of petting them. Unfortunately, our interface is rather large, and we have no choice than to implement the code to pet the bear.

Let's fix this by splitting our large interface into 3 separate ones:

public interface BearCleaner {

void washTheBear();

}

public interface BearFeeder {

void feedTheBear();

}

public interface BearPetter {

void petTheBear();

}

Now, thanks to interface segregation, we're free to implement only the methods that matter to us:

public class BearCarer implements BearCleaner, BearFeeder {

public void washTheBear() {

//I think we missed a spot...

}

public void feedTheBear() {

//Tuna Tuesdays...

}

}

And finally, we can leave the dangerous stuff to the crazy people:

public class CrazyPerson implements BearPetter {

public void petTheBear() {

//Good luck with that!

}

}

Going further, we could even split our BookPrinter class from our example earlier to use interface segregation in the same way. By implementing a Printer interface with a single print method, we could instantiate separate ConsoleBookPrinter and OtherMediaBookPrinter classes.

Dependency Inversion:

The principle of Dependency Inversion refers to the decoupling of software modules. This way, instead of high-level modules depending on low-level modules, both will depend on abstractions.

To demonstrate this, let's go old-school and bring to life a Windows 98 computer with code:

public class Windows98Machine {}

But what good is a computer without a monitor and keyboard? Let's add one of each to our constructor so that every Windows98Computer we instantiate comes pre-packed with a Monitor and a StandardKeyboard:

public class Windows98Machine {

private final StandardKeyboard keyboard;

private final Monitor monitor;

public Windows98Machine() {

monitor = new Monitor();

keyboard = new StandardKeyboard();

}

}

This code will work, and we'll be able to use the StandardKeyboard and Monitor freely within our Windows98Computer class. Problem solved? Not quite. By declaring the StandardKeyboard and Monitor with the new keyword, we've tightly coupled these 3 classes together.

Not only does this make our Windows98Computer hard to test, but we've also lost the ability to switch out our StandardKeyboard class with a different one should the need arise. And we're stuck with our Monitor class, too.

Let's decouple our machine from the StandardKeyboard by adding a more general Keyboard interface and using this in our class:

public interface Keyboard { }

public class Windows98Machine{

private final Keyboard keyboard;

private final Monitor monitor;

public Windows98Machine(Keyboard keyboard, Monitor monitor) {

this.keyboard = keyboard;

this.monitor = monitor;

}

}

Here, we're using the dependency injection pattern here to facilitate adding the Keyboard dependency into the Windows98Machine class.

Let's also modify our StandardKeyboard class to implement the Keyboard interface so that it's suitable for injecting into the Windows98Machine class:

public class StandardKeyboard implements Keyboard { }

Now our classes are decoupled and communicate through the Keyboard abstraction. If we want, we can easily switch out the type of keyboard in our machine with a different implementation of the interface. We can follow the same principle for the Monitor class.

Excellent! We've decoupled the dependencies and are free to test our Windows98Machine with whichever testing framework we choose.

Conclusion

In this tutorial, we've taken a deep dive into the SOLID principles of object-oriented design.

We started with a quick bit of SOLID history and the reasons these principles exist.

Letter by letter, we've broken down the meaning of each principle with a quick code example that violates it. We then saw how to fix our code and make it adhere to the SOLID principles.

**Monitor**

A monitor can be considered as a building which contains a special room. The special room can be occupied by only one customer(thread) at a time. The room usually contains some data and code.

If a customer wants to occupy the special room, he has to enter the Hallway(Entry Set) to wait first. Scheduler will pick one based on some criteria(e.g. FIFO). If he is suspended for some reason, he will be sent to the wait room, and be scheduled to reenter the special room later. As it is shown in the diagram above, there are 3 rooms in this building.

In brief, a monitor is a facility which monitors the threads' access to the special room. It ensures that only one thread can access the protected data or code.

It is good to say that each object has a monitor, since each object could have its own critical section, and capable of monitoring the thread sequence.

To enable collaboration of different threads, Java provide wait() and notify() to suspend a thread and to wake up another thread that are waiting on the object respectively. In addition, there are 3 other versions:

|  |
| --- |
| wait(long timeout, int nanos)  wait(long timeout) notified by other threads or notified by timeout.  notify(all) |

Those methods can only be invoked within a synchronized statement or synchronized method. The reason is that if a method does not require mutual exclusion, there is no need to monitor or collaborate between threads, every thread can access that method freely.

**Executor Service**

[ExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html) is a framework provided by the JDK which simplifies the execution of tasks in asynchronous mode. Generally speaking, *ExecutorService* automatically provides a pool of threads and API for assigning tasks to it.

The easiest way to create ExecutorService is to use one of the factory methods of the Executors class.

For example, the following line of code will create a thread-pool with 10 threads:

|  |  |
| --- | --- |
|  | ExecutorService executor = Executors.newFixedThreadPool(10); |

The are several other factory methods to create predefined ExecutorService that meet specific use cases

The shutdown() method doesn't cause immediate destruction of the ExecutorService. It will make the ExecutorService stop accepting new tasks and shut down after all running threads finish their current work.

|  |  |
| --- | --- |
|  | executorService.shutdown(); |

The *shutdownNow()* method tries to destroy the *ExecutorService* immediately, but it doesn't guarantee that all the running threads will be stopped at the same time

**Callable**

Since the early days of Java, multithreading has been a major aspect of the language. *Runnable*is the core interface provided for representing multi-threaded tasks and *Callable* is an improved version of *Runnable* that was added in Java 1.5.

Both interfaces are designed to represent a task that can be executed by multiple threads. *Runnable* tasks can be run using the *Thread* class or *ExecutorService* whereas*Callables* can be run only using the latter.

Let's have a deeper look at the way these interfaces handle return values.

With Runnable

The Runnable interface is a functional interface and has a single run() method which doesn't accept any parameters and does not return any values.

This is suitable for situations where we are not looking for a result of the thread execution, for example, incoming events logging:

|  |  |
| --- | --- |
|  | public interface Runnable { |
|  | public void run(); |
|  | } |

Let's understand this with an example:

|  |  |
| --- | --- |
|  | public class EventLoggingTask implements Runnable{ |
|  | private Logger logger |
|  | = LoggerFactory.getLogger(EventLoggingTask.class); |
|  |  |
|  | @Override |
|  | public void run() { |
|  | logger.info("Message"); |
|  | } |
|  | } |

In this example, the thread will just read a message from the queue and log it in a log file. There's no value returned from the task; the task can be launched using ExecutorService:

|  |  |
| --- | --- |
|  | public void executeTask() { |
|  | executorService = Executors.newSingleThreadExecutor(); |
|  | Future future = executorService.submit(new EventLoggingTask()); |
|  | executorService.shutdown(); |
|  | } |

In this case, the Future object will not hold any value.

With Callable

The Callable interface is a generic interface containing a single call() method – which returns a generic value V:

|  |  |
| --- | --- |
|  | public interface Callable<V> { |
|  | V call() throws Exception; |
|  | } |

Let's have a look at calculating the factorial of a number:

|  |  |
| --- | --- |
|  | public class FactorialTask implements Callable<Integer> { |
|  | int number; |
|  |  |
|  | // standard constructors |
|  |  |
|  | public Integer call() throws InvalidParamaterException { |
|  | int fact = 1; |
|  | // ... |
|  | for(int count = number; count > 1; count--) { |
|  | fact = fact \* count; |
|  | } |
|  |  |
|  | return fact; |
|  | } |
|  | } |

The result of call() method is returned within a Future object:

|  |  |
| --- | --- |
|  | @Test |
|  | public void whenTaskSubmitted\_ThenFutureResultObtained(){ |
|  | FactorialTask task = new FactorialTask(5); |
|  | Future<Integer> future = executorService.submit(task); |
|  |  |
|  | assertEquals(120, future.get().intValue()); |

\*\*\* Remember, Future.get() is a blocking method and blocks until execution is finished, so you should always call this method with a timeout to avoid [deadlock](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html) or livelock in your application.

-Exception Handling

Let's see how suitable they are for exception handling.

With *Runnable*

Since the method signature does not have the “throws” clause specified, there is no way to propagate further checked exceptions.

With *Callable*

*Callable's call()* method contains “throws *Exception”* clause so we can easily propagate checked exceptions further:

|  |  |
| --- | --- |
|  | public class FactorialTask implements Callable<Integer> { |
|  | // ... |
|  | public Integer call() throws InvalidParamaterException { |
|  |  |
|  | if(number < 0) { |
|  | throw new InvalidParamaterException("Number should be positive"); |
|  | } |
|  | // ... |
|  | } |
|  | } |

In case of running a *Callable using* an*ExecutorService,*the exceptions are collected in the*Future*object, which can be checked by making a call to the *Future.get()*method. This will throw an *ExecutionException –*which wraps the original exception:

|  |  |
| --- | --- |
|  | @Test(expected = ExecutionException.class) |
|  | public void whenException\_ThenCallableThrowsIt() { |
|  |  |
|  | FactorialCallableTask task = new FactorialCallableTask(-5); |
|  | Future<Integer> future = executorService.submit(task); |
|  | Integer result = future.get().intValue(); |
|  | } |

In the above test, the *ExecutionException*is being thrown as we are passing an invalid number. We can call the *getCause()*method on this exception object to get the original checked exception.

If we don't make the call to the *get()*method of *Future*class – then the exception thrown by *call()*method will not be reported back, and the task will still be marked as completed:

|  |  |
| --- | --- |
|  | @Test |
|  | public void whenException\_ThenCallableDoesntThrowsItIfGetIsNotCalled(){ |
|  | FactorialCallableTask task = new FactorialCallableTask(-5); |
|  | Future<Integer> future = executorService.submit(task); |
|  |  |
|  | assertEquals(false, future.isDone()); |
|  | } |

The above test will pass successfully even though we've thrown an exception for the negative values of the parameter to *FactorialCallableTask.*

**Cyclic barrier**

*CyclicBarriers* are synchronization constructs that were introduced with Java 5 as a part of the *java.util.concurrent* package.

A CyclicBarrier is a synchronizer that allows a set of threads to wait for each other to reach a common execution point, also called a barrier.

CyclicBarriers are used in programs in which we have a fixed number of threads that must wait for each other to reach a common point before continuing execution.

The barrier is called cyclic because it can be re-used after the waiting threads are released.

**Countdown latch**

CountDownLatch is used to make sure that a task waits for other threads before it starts. To understand its application, let us consider a server where the main task can only start when all the required services have started.

Working of CountDownLatch:  
When we create an object of CountDownLatch, we specify the number of threads it should wait for, all such thread are required to do count down by calling CountDownLatch.countDown() once they are completed or ready to the job. As soon as count reaches zero, the waiting task starts running.

**Blocking Queue**

A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

BlockingQueue blockingQueue = new LinkedBlockingDeque();

The implementation of a blocking queue looks similar to a [Bounded Semaphore](http://tutorials.jenkov.com/java-concurrency/semaphores.html#bounded). Here is a simple implementation of a blocking queue:

public class BlockingQueue {

private List queue = new LinkedList();

private int limit = 10;

public BlockingQueue(int limit){

this.limit = limit;

}

public synchronized void enqueue(Object item)

throws InterruptedException {

while(this.queue.size() == this.limit) {

wait();

}

this.queue.add(item);

if(this.queue.size() == 1) {

notifyAll();

}

}

public synchronized Object dequeue()

throws InterruptedException{

while(this.queue.size() == 0){

wait();

}

if(this.queue.size() == this.limit){

notifyAll();

}

return this.queue.remove(0);

}

}

Notice how notifyAll() is only called from enqueue() and dequeue() if the queue size is equal to the size bounds (0 or limit). If the queue size is not equal to either bound when enqueue() or dequeue() is called, there can be no threads waiting to either enqueue or dequeue items.

**Multiple produce consumer**

**Readers Writers Lock**

**Re-entrant lock**

**Atomic variables**

Implement transaction

**Why wait and notify called form synchronized**

**Fork**

**Factory Design**

**ACID**

**Normalization**

**Red Black vs AVL**

**Builder Design**

**Observer Design**

**Abstract Builder Design**

**Decorator Design**

**Generics**

**Garbage Collector Functioning**

String is not a very good candidate to be used with synchronized keyword. It’s because they are stored in a string pool and we don’t want to lock a string that might be getting used by another piece of code. So I am using an Object variable. Learn more about synchronization and thread safety in java.

**Runnable vs Thread**

| **Key** | **Thread** | **Runnable** |
| --- | --- | --- |
| Basic | Thread is a class. It is used to create a thread | Runnable is a functional interface which is used to create a thread |
| Methods | It has multiple methods including start() and run() | It has only abstract method run() |
|  | Each thread creates a unique object and gets associated with it | Multiple threads share the same objects. |
| Memory | More memory required | Less memory required |
| Limitation | Multiple Inheritance is not allowed in java hence after a class extends Thread class, it can not extend any other class | If a class is implementing the runnable interface then your class can extend another class. |

class RunnableExample implements Runnable{

   public void run(){

      System.out.println("Thread is running for Runnable Implementation");

   }

   public static void main(String args[]){

      RunnableExample runnable=new RunnableExample();

      Thread t1 =new Thread(runnable);

      t1.start();

   }

}

class ThreadExample extends Thread{

   public void run(){

      System.out.println("Thread is running");

   }

   public static void main(String args[]){

      ThreadExample t1=new ThreadExample ();

      t1.start();

   }

}

**Memory management** is the process of allocating memory to new Java objects and de-allocating memory from unused objects. In Java, memory management happens automatically and memory allocation and de-allocation happen in the background. But it does not guarantee anything. So, as a Java programmer, in order to write high-performance and optimized code, it is imperative to understand how Java memory is designed. Understanding memory organization in Java allows a programmer to avoid *OutOfMemoryError* and quickly find memory leaks.

Generally, memory is divided into - stack memory and heap memory. As soon as JVM (Java Virtual Machine) starts, these memories are assigned by the system. We will discuss more on this memory in next section.

**Stack Memory**

**Stack memory** is responsible for holding references to objects which are stored in heap and also stores values of Java primitive types. It always works on LIFO (Last-In-First-Out) order. When a method is called from Java program, a new memory block is created in the stack to hold local primitive values and references to other objects. Once a method finishes execution, its memory is deallocated and assigned to next method. If stack memory is full, Java runtime throws *java.lang.StackOverFlowError*.

**Heap Memory**

The memory that actually stores the object is called **heap memory**. The objects in heap are referenced by variables in stack memory. As soon as an object is created, heap memory is created for that object with its reference in the stack. Objects stored in heap are.globally accessible and it has larger size when compared to stack memory. If heap memory is full, Java runtime throws *java.lang.OutOfMemoryError: Java Heap Space*.

The size of stack and heap memory depends upon the JVM configurations and can be changed explicitly. You can use *-Xms* and *-Xmx* JVM option to define the startup size and maximum size of heap memory. For stack memory, you can use *-Xss* option.

**Garbage Collection**

In Java, **garbage collector** automatically reclaims the memory for reuse and helps in eliminating memory leaks and other memory-related problems. When an object is no longer used, the Java garbage collector reclaims the underlying memory and reuses it for future object allocations. JVM manages the heap area for all objects and as long as the object is being referenced, it is considered alive. Once an object is no longer referenced or not reachable by an application, the garbage collector removes this object and reclaims the unused memory.

JVM uses a **mark and sweep algorithm** to determine which objects are no longer in use by the application. This is a two-step process:

1. First, the algorithm scans all object references and marks those objects which are found as alive.
2. It sweeps or reclaims all the heap memory that is not occupied by these marked objects.