WEEK1

DAY1:

Java: It is one of the most popular programming language and it is a pure object oriented programming language. By using java we can create small desktop application to large server side applications. Java was developed my James Gosling in 1995 at sun microsystems. 3 billion mobile phones run java.

\*\*Amazon, ebay, \*Google, Instagram, \*Microsoft, Netflix, Spotify, Uber are using java as the backed code.

FEATURES:

1. Object Oriented: It is a pure object oriented programming language.

-> It helps to write cleaner code and to maintain control over the functions and modules.

-> It focuses on the development of large software applications.

-> Applications of OOP

* Web applications
* Mobile applications
* Database management
* Artificial Intelligence
* Embedded systems

1. WORA: Write Once Run Anywhere

source code->Byte code

the byte code we can run on any operating system.

1. Security JVM: JVM checks the byte code every time a new code is getting executed.

* Data hiding(abstraction) is possible it provides security.
* Here any manipulations happened to your bytecode JVM thinks that it is corrupted and it won’t execute the code. JVM secure.

1. Robust: Robust means strong and the program don’t crash easily. There are 2 reasons for this

* Exception Handling: An exception is an error that occurs at run time and exceptions which we can handle. The errors which cannot be handled.
* Memory Management: The memory allocation problems are not occurred in java because the memory allocation and deallocation are done by JVM.

1. Multithreaded: A thread represents an individual process to execute a group of statements. JVM uses several threads to execute different blocks of code. Creating multiple threads is called as multithreading.

* It improves the CPU utilization.

1. High Performance: The problem with interpreter inside the JVM is that it is slow. Because of this java programs used to run slow. To overcome this problem, along with the interpreter, they have introduced JIT (Just In Time) compiler, which increase the speed of the execution. Both interpreter and JIT work together.
2. Dynamic: Java is more dynamic. It is designed to adopt to an evolving environment.

* It supports dynamic loading of classes. It means classes are loaded on demand.
* It loads the class file during runtime only. Hence, anything that happens at runtime is dynamic.

Java comes in the form of 3 categories:

1. JSE/J2SE:- java 2 standard edition /core edition

* This is the foundation of both editions in below.
* This is used for core java concepts and used for desktop applications, standalone, networking, database, GUI, multithreaded types of applications.
* Desktop Applications: These are software programs run locally on computer devices.
* Windows file explorer
* Adobe Photoshop
* Discord
* Media Player

1. JEE/J2EE:- java 2 enterprise edition (Business)

* Enterprise applications are large-scale software. It addresses an entire organization’s needs rather than an individual.
* Enterprise applications:
* Salesforce- customer relationship management platform that connects sales, marketing, commerce, and service teams.
* Microsoft office 365(word and excel).
* Accounting and Billing.
* Supply chain management.
* Cloud computing.

1. JME/J2ME:- java 2 micro edition

* It is a development and deployment platform of portable code for embedded and mobile devices (sensors, gateways, mobile phones, printers, TV set-top boxes, smart watches, Refrigerators).

# X.java (source code)🡪java compile🡪 X.class (byte code) 🡪 JVM🡪 execute this file.

JDK= (JRE+ Development tools like java compiler, debugger etc.)

JRE= (JVM + predefined library classes)

JVM is platform dependent.

Compiler is platform dependent.

WHY JAVA?

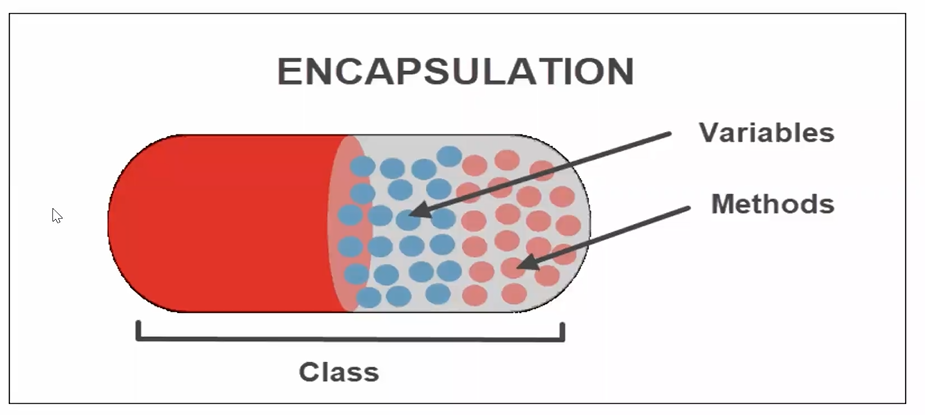
Java is like a full meal but other languages are like fast food. We can also tell the features of java also.

Object Oriented Programming:

* Encapsulation
* Abstraction
* Inheritance
* Polymorphism

1. Encapsulation:

* It is the back bone of the oop.
* A class is a combination of variables and methods.
* The variables and methods are covered by class.



1. Abstraction:

* Abstraction is a process of hiding the implementation details and showing only functionality to the user.
* By using abstraction it will provide the security.

1. Inheritance:

* Inheritance means child class can acquire the properties from parent class.

1. Polymorphism:

* Here the functionality is same but the behaviour is different.
* Example: person – Kiran

Kiran behave like son with his parents.

Kiran behave like husband with his wife.

Kiran behave like father with his son.

Kiran behave like Employee in office.

CLASS:

\*\* In one file we can create any number of classes & here there is no mandatory field that class and file have the same name.

Eg: File name: First.java

**class** Demo {

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

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

}

}

**class** Demo1

{

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

{

System.***out***.println("Hi!");

}

}

**class** Demo2

{

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

{

System.***out***.println("Hi Hello!");

}

}

Here we have file: First.java

The .java file can create 3 .class files 1) Demo.class 2) Demo1.class 3) Demo2.class

We can run 3 .class files separately.

* A special case is that any class we declared as public.

Eg: here file name: Demo.java (the file name is same as public class name).

**public class** Demo {

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

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

}

}

**class** Demo1

{

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

{

System.***out***.println("Hi!");

}

}

**class** Demo2

{

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

{

System.***out***.println("Hi Hello!");

}

}

Output: Hello

\*\*NOTE: If the class name is public we can also save the file name as class name. else we will get a compilation error\*\*

* While running the java application we can run at most only one class at a time.

\*\*NOTE: In a single .java file we can have only one public class. If any of the 2 classes having the public then it shows the compilation error.

* Here our file name: Demo.java
* It will perfectly execute the Demo class because the file name and class names are same. The remaining classes shows the compilation error.

**public class** Demo {

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

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

}

}

**public class** Demo1

{

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

{

System.***out***.println("Hi!");

}

}

**public class** Demo2

{

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

{

System.***out***.println("Hi Hello!");

}

}

Output: Hello

It executes the code and print hello but it gives the compilation error for Demo1 and Demo2.

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

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

* If any changes made to this it will not show any error but it will not run by the JVM.
* The changes are like letters, removal of flower brackets etc.
* A special case is that we can write any variable name in place of “args”.
* Don’t change anything in this line except args variable.

Compilation- done by java compiler.

Execution- done by the JVM and the JVM contains the Interpreter and JIT.

Without object creation we can execute a static block:

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

**public** **class** Demo{

**static**{

System.***out***.println("static block");

}

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

System.***out***.println("This is main method");

}

}

Output:

static block

This is main method

Without object creation we cannot execute a normal block:

Eg: **public** **class** Demo{

{

System.***out***.println("static block");

}

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

System.***out***.println("This is main method");

}

}

Output: This is main method

With object creation we can execute a normal block also:

Eg: **public** **class** Demo{

{

System.***out***.println("static block");

}

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

System.***out***.println("This is main method");

Demo d=**new** Demo();

}

}

Output: This is main method

static block

\*\* System.out.println(“hi”);

* Here “System” is a predefined class, which belongs to java.lang package , inside this class we have various members, “out” is a member it belongs to the System class only.
* The type of this “out” field is the PrintStream class. This PrintStream is another predefined class. The “println” method belongs to PrintStream class.

DAY2:

Naming Convensions:

1. All the class names, abstract class names, interface names and Enum, names should follow “PascalNamingConvention”, which means, the letter of every word should be in upper case.

Examples: String, ArrayList, InputStreamReader, StringBuffer, Student etc.

1. All the java variables, method names should follow camelNamingConvention, which means the first letter should be in lower case and the subsequent word should start with Uppercase.

Examples(variables): in, out, studentName, pageContext etc.

Examples(methods): concat(), forName(), getInputStream() etc.

1. All java constant variables must be provided in Upper case letters.

Examples: MIN\_PRIORITY, MAX\_PRIORITY, NORM\_PRIORITY

Data Types:

1. Primitive Data Types/ Primary/ Basic.
2. Non-Primitive Data Types/ reference/ Advanced.
3. User Data Types
4. Primitive Data Types:

\*\* By default every integer number is int (If in case the number is byte, short it is also treated as int).

\*\* By default every decimal is treated as double (It is mentioned like 10.45f now it will be treated as float).

There are 8 primitive data types in java

* Numeric
* Boolean
* Numeric:
* Character
* Character(char)
* char – 2 bytes (16bits)

range – ( -2^15 to 2^15-1)

default value: ‘ ‘

* Integral
* Integer (byte, short, int, long)
* byte – 1byte (8 bits)

range – (-2^7 to 2^7-1)

(-128 -127 -126 ………. -1 0 1 2 3 4………… 127)

default value- 0

* short – 2bytes (16 bits)

range – (-2^15 to 2^15-1)

default value- 0

* int – 4bytes (32bits)

range – (-2^31 to 2^31-1)

default value- 0

* long – 8bytes (64bits)

range – (-2^63 to 2^63-1)

default value- 0

* Decimal (float, decimal)
* float – 4bytes(32bits)

range – (-2^31 to 2^31-1)

default value – 0.0f

* double – 8bytes(64bits)

range – (-2^63 to 2^63-1)

default value – 0.0

* Boolean: 1 bit (true or false)

Default value: false

Example: variables and datatypes.

**public** **class** Person {

**private** **final** **double** PI=3.14; //constant- all letters are in upper case only.

**private** String MY\_BIG\_CONSTANT="Hi I am siri!"; //constant

**private** String myBigConstant=""; //normal variables follows camelCase

**private** String firstName="siri"; //String is a class

**private** **int** age; //instance variable- because it is primitive.

**private** **long** id;

**private** **char** firstLetter;

**private** **byte** myByte;

**private** Company company=**new** Company();// reference variable - because company is a class.

**private** NewsAgency na; // reference variable

}

Special case in data types:

**public** **class** Demo {

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

{

**int** i=0b11; //0b-->binary

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

**int** k=0b01 | 0b10 | 0b100;

System.***out***.println(k);

**int** s=1 |2 |3;

System.***out***.println(s);

**int** h=0x123; //0x-->hexa decimal:

//1 2 3

//1\*16^2 2\*16^1 3\*16^0

System.***out***.println(h);

**int** x=0123; //octal - It starts with 0

//1 2 3

//1\*8^2 2\*8^1 3\*8^0 = 64+16+3=83

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

}

}

Output:

3

7

3

291

BigDecimal:

String representation of double number and normal representation of double.

Example:

**import** java.math.BigDecimal;

**public** **class** DatatypeEx {

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

//BigDEcimal using double string values

BigDecimal bd1 = **new** BigDecimal("124567890.0987654321");

BigDecimal bd2 = **new** BigDecimal("987654321.123456789");

BigDecimal bg=bd1.add(bd2);

System.***out***.println(bg);

//It uses double values

BigDecimal A=BigDecimal.*valueOf*(23.56);

System.***out***.println(A);

}

}

Output:

1112222211.2222222211

23.56

Eg:

**import** java.math.BigDecimal;

**public** **class** Demo {

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

{

BigDecimal d=**new** BigDecimal("2.34");

BigDecimal d1=**new** BigDecimal("3.56");

System.***out***.println(d1.subtract(d));

}

}

Output:

1.22

Example:

**import** java.math.BigDecimal;

**import** java.math.MathContext;

**import** java.math.RoundingMode;

**public** **class** Demo {

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

{

System.***out***.println(**new** BigDecimal(".45986").add(**new** BigDecimal("5.31")));

System.***out***.println(**new** BigDecimal(".45986").subtract(**new** BigDecimal("5.31")));

System.***out***.println(**new** BigDecimal(".45986").multiply(**new** BigDecimal("5.31")));

MathContext mc=**new** MathContext(7,RoundingMode.***HALF\_UP***);

System.***out***.println(**new** BigDecimal(".45986").divide(**new** BigDecimal("5.31"),mc));

System.***out***.println(**new** BigDecimal(".45986").sqrt(mc));

System.***out***.println(**new** BigDecimal("-7").abs());

System.***out***.println(**new** BigDecimal("45986").max(**new** BigDecimal("531")));

}

}

Output:

5.76986

-4.85014

2.4418566

0.08660264

0.6781298

7

45986

BigInteger & BigDecimal:

Eg:

**import** java.math.BigDecimal;

**import** java.math.BigInteger;

**public** **class** Demo {

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

{

BigDecimal d=**new** BigDecimal("3.1452");

**byte** myByte=100;

System.***out***.println(d);

System.***out***.println(myByte);

**byte** b=d.byteValue();

System.***out***.println(b);

BigDecimal bd=**new** BigDecimal("5763");

System.***out***.println(bd.toString()+"9679");

BigInteger bt=**new** BigInteger("754865734857675843");

System.***out***.println(bt);

}

}

Output:

3.1452

100

3

57639679

754865734857675843

Number Formatting:

**import** java.math.BigDecimal;

**import** java.text.NumberFormat;

**import** java.util.Locale;

**public** **class** DatatypeEx {

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

BigDecimal d=**new** BigDecimal("87989335466452943295");

System.***out***.println(d);

NumberFormat n=NumberFormat.*getCurrencyInstance*(Locale.***UK***);

System.***out***.println(n.format(d));

}

}

Output:

87989335466452943295

£87,989,335,466,452,943,295.00

Compound Interest: using BigDecimal

Eg:

**import** java.math.BigDecimal;

**public** **class** Demo {

**public** **static** BigDecimal calculate(String principal, String rate, **int** period, String contribution) {

//Balance=p(1+r)^y + c[((1+r)^y-1)]/r]

BigDecimal a=BigDecimal.***ONE***.add(**new** BigDecimal(rate)).pow(period);//(1+r)^y

BigDecimal b=a.subtract(BigDecimal.***ONE***);//(1+r)^y-1

BigDecimal c=b.divide(**new** BigDecimal(rate));//((1+r)^y-1)]/r

BigDecimal d=c.multiply(**new** BigDecimal(contribution));//c[((1+r)^y-1)]/r]

BigDecimal e=**new** BigDecimal(principal).multiply(a);//p(1+r)^y

BigDecimal f=e.add(d);

**return** f;

}

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

{

BigDecimal ans=*calculate*("10000.00",".08",10,"1000");

System.***out***.println(ans);

}

}

Output:

36075.8124386377007104000000

Formatting Numbers:

**import** java.math.BigDecimal;

**import** java.text.NumberFormat;

**import** java.text.ParseException;

**import** java.util.Locale;

**public** **class** Demo {

**private** **static** **final** NumberFormat ***moneyFormatter***=NumberFormat.*getCurrencyInstance*(Locale.***US***);

**private** **static** **final** NumberFormat ***percentFormatter***=NumberFormat.*getPercentInstance*();

**public** **static** BigDecimal calculate(String principal, String rate, **int** period, String contribution) **throws** ParseException {

//Balance=p(1+r)^y + c[((1+r)^y-1)]/r]

String rateAsPercent=***percentFormatter***.parse(rate).toString();

BigDecimal a=BigDecimal.***ONE***.add(**new** BigDecimal(rateAsPercent)).pow(period);//(1+r)^y

BigDecimal b=a.subtract(BigDecimal.***ONE***);//(1+r)^y-1

BigDecimal c=b.divide(**new** BigDecimal(rateAsPercent));//((1+r)^y-1)]/r

BigDecimal d=c.multiply(**new** BigDecimal(***moneyFormatter***.parse(contribution).toString()));//c[((1+r)^y-1)]/r]

BigDecimal e=**new** BigDecimal(***moneyFormatter***.parse(principal).toString()).multiply(a);//p(1+r)^y

BigDecimal f=e.add(d);

**return** f;

}

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

{

BigDecimal ans=*calculate*("$25,300","8%",10,"$7,500");

System.***out***.println(***moneyFormatter***.format(ans));

}

}

Output:

$163,270.02

Customizing Numbers Formatters:

Eg:

**import** java.math.BigDecimal;

**import** java.text.DecimalFormat;

**import** java.text.NumberFormat;

**import** java.text.ParseException;

**import** java.util.Locale;

**public** **class** Demo {

**private** **static** **final** NumberFormat ***moneyFormatter***=NumberFormat.*getCurrencyInstance*(Locale.***US***);

**private** **static** **final** NumberFormat ***percentFormatter***=NumberFormat.*getPercentInstance*();

**public** **static** BigDecimal calculate(String principal, String rate, **int** period, String contribution) **throws** ParseException {

//Balance=p(1+r)^y + c[((1+r)^y-1)]/r]

String rateAsPercent=***percentFormatter***.parse(rate).toString();

BigDecimal a=BigDecimal.***ONE***.add(**new** BigDecimal(rateAsPercent)).pow(period);//(1+r)^y

BigDecimal b=a.subtract(BigDecimal.***ONE***);//(1+r)^y-1

BigDecimal c=b.divide(**new** BigDecimal(rateAsPercent));//((1+r)^y-1)]/r

BigDecimal d=c.multiply(**new** BigDecimal(***moneyFormatter***.parse(contribution).toString()));//c[((1+r)^y-1)]/r]

BigDecimal e=**new** BigDecimal(***moneyFormatter***.parse(principal).toString()).multiply(a);//p(1+r)^y

BigDecimal f=e.add(d);

**return** f;

}

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

{

DecimalFormat df=**new** DecimalFormat("$,###.##");

DecimalFormat df1=**new** DecimalFormat("$00000000,000.00");

DecimalFormat pf=**new** DecimalFormat("$(#)");

DecimalFormat df2=**new** DecimalFormat("$,###.##;$(#)");

DecimalFormat df3=**new** DecimalFormat("#,###%");

BigDecimal ans=*calculate*("$25,300","8%",10,"$7,500");

System.***out***.println(df.format(ans));

System.***out***.println(df1.format(ans));

System.***out***.println(pf.format(ans));

System.***out***.println(df2.format(ans));

System.***out***.println(df3.format(2345.08));

}

}

Output:

$163,270.02

$00,000,163,270.02

$(163270)

$163,270.02

234,508%

Formatting number:

Eg:

**import** java.math.BigDecimal;

**import** java.text.NumberFormat;

**import** java.text.ParseException;

**import** java.util.Locale;

**public** **class** Demo {

**private** **static** **final** NumberFormat ***moneyFormatter***=NumberFormat.*getCurrencyInstance*(Locale.***US***);

**private** **static** **final** NumberFormat ***percentFormatter***=NumberFormat.*getPercentInstance*();

**public** **static** BigDecimal calculate(String principal, String rate, **int** period, String contribution) **throws** ParseException {

//Balance=p(1+r)^y + c[((1+r)^y-1)]/r]

String rateAsPercent=***percentFormatter***.parse(rate).toString();

BigDecimal a=BigDecimal.***ONE***.add(**new** BigDecimal(rateAsPercent)).pow(period);//(1+r)^y

BigDecimal b=a.subtract(BigDecimal.***ONE***);//(1+r)^y-1

BigDecimal c=b.divide(**new** BigDecimal(rateAsPercent));//((1+r)^y-1)]/r

BigDecimal d=c.multiply(**new** BigDecimal(***moneyFormatter***.parse(contribution).toString()));//c[((1+r)^y-1)]/r]

BigDecimal e=**new** BigDecimal(***moneyFormatter***.parse(principal).toString()).multiply(a);//p(1+r)^y

BigDecimal f=e.add(d);

**return** f;

}

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

{

BigDecimal ans=*calculate*("$25,300","8%",10,"$7,500");

System.***out***.printf("%f%n",ans);

System.***out***.printf("$%.2f%n",ans);//2 decimal points

System.***out***.printf("$%.2f%n",ans.negate());

System.***out***.printf("$%,.2f%n",ans);

System.***out***.printf("$%,(.2f%n",ans.negate());

String myMoney=String.*format*("$%,.2f%n",ans);

System.***out***.println(myMoney);

}

}

Output:

163270.020925

$163270.02

$-163270.02

$163,270.02

$(163,270.02)

$163,270.02

Difference between Instance variable and Reference variable:

Type Casting: The procedure of converting one data type into its equivalent another data type is known as type casting.

* Implicit type casting: (upcasting/ widening)
* Explicit type casting: (down casting/ narrowing)

1. Implicit type casting: converting smaller data type to larger data type.

Example: int x=10;

long y=x;

**public** **class** Demo {

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

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

**int** x=10;

**long** y=x;

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

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

}

}

Output: 10

10

NOTE: Char and byte both are compatible to each other we don’t do down casting.

Eg1: char to byte

**public** **class** Demo

{

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

{

**byte** b1=65;

**byte** b2='A';

System.***out***.println(b1);

System.***out***.println(b2);

}

}

Output: 10 10

Eg2: byte to char

**public** **class** Demo {

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

{

**char** b1=65; // It prints ASCII value of 65

**char** b2='A';

System.***out***.println(b1);

System.***out***.println(b2);

}

}

Output: A

A

\*\* In this we cannot assign char to byte and byte to char

**char** b1='A';

**byte** b2=b1; // It is not possible to assign a character to a byte variable.

\*\* It is possible by type casting. It is not a down casting. Why because both are same.

**public** **class** Demo {

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

{

**char** b1='A';

**byte** b2=(**byte**)b1; // It is not possible to assign a character to a byte variable.

System.***out***.println(b1);

System.***out***.println(b2);

}

}

Output: A

65

1. Explicit type casting: converting bigger data type to smaller data type.

Example: int x=10;

Byte y=x;

//compile time error

Type mismatch: cannot convert from int to byte

To overcome this problem we use explicit type casting.

Example: int x=10;

byte y=(byte) x;

**public** **class** Demo {

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

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

**int** x=10;

**byte** y=(**byte**)x;

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

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

}

}

output: 10

10

\*\* In this process of converting bigger data type to smaller data type we loss some data. For example converting double (8bytes) to float (4bytes).

Here float is bigger than int so we don’t do down casting.

Example:

**public** **class** Demo {

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

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

**double** x=100.544234353464654567;

**float** y=(**float**)x;

float z=10;

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

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

System.out.println(z);

}

}

Output: 100.54423435346466 //double value

100.544235 //float value (It losses some data)

10

\*\* By default every non decimal value is treated as integer and decimal values are treated as double.

Special case: byte

**public** **class** Demo {

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

{

**byte** b=10;

**byte** c=b+1; // here 1 is integer and we are adding int to byte and the resultant is int but we are assigning it to byte. It give compilation error.

// to overcome that explicit type casting

**byte** d=(**byte**)b+1; // it will also show error. why means we are converting only b to byte not 1.

**byte** s=(**byte**)(b+1);

System.***out***.println(s); // 11

}

}

1. User defined data type: Here we can take class, interface, enum, etc as user defined data types.

Example: Employee emp;

Same as -- int x;

// emp is variable

// Employee is type

WRAPPER CLASSES:

Since java is an object-oriented programming language, there are some situations, where we need to represent the primitive data types in the form of objects.

* In java.lang package some classes are defined to represent the primitive data types in the form of objects.
* Examples:

byte: java.lang.Byte

short: java.lang.Short

int: java.lang.Integer

long: java.lang.Long

float: java.langl.Float

double: java.lang.Double

bool: java.lang.Boolean

char: java.lang.Character

* To identify the range of particular datatype these wrapper classes has provided following constant variables in all the wrapper classes except Boolean class.

MIN\_VALUE

MAX\_VALUE

Data types min max values:

Integer. MIN\_VALUE ----- Integer. MAX\_VALUE

Double. MIN\_VALUE ----- Double. MAX\_VALUE

Float. MIN\_VALUE ----- Float. MAX\_VALUE

Short. MIN\_VALUE ----- Short. MAX\_VALUE

Examples: System.***out***.println(Byte.***MIN\_VALUE***+"------>"+Byte.***MAX\_VALUE***);

Output: -128------>127

System.***out***.println(Integer.***MIN\_VALUE***+"------>"+Integer.***MAX\_VALUE***);

Output: -2147483648------>2147483647

System.***out***.println(Integer.***MIN\_VALUE***+"------>"+Integer.***MAX\_VALUE***);

Output: 4.9E-324------>1.7976931348623157E308

System.***out***.println(Short.***MIN\_VALUE***+"------>"+Short.***MAX\_VALUE***);

Output: -32768------>32767

METHOD:

What is the difference between method and function?

Here the method and function are a block of code that performs a well-defined task. the methods are present in the class only but the functions are present out of the class.

Method: A method is a block of code that performs a well-defined task. The methods are present with in the class only. Without class we cannot create methods. A method cannot exist without class.

Syntax:

returnType methodName()

{

Instructions;

}

Some methods return some value, whereas some methods does not return any value. The method which does not return any value we apply void as return type. We can pass parameters to the method.

Java belongs to a C family language: ( C, C++, java, C#, python, javascript)

COMMENTS:

Comments in java.

1. Single line comments (//)

Shortcut for this ctrl+/

1. Multi line comments (/\* \*/)

Shortcut for this ctrl+\*

JAVA LANGUAGE FUNDAMENTALS:

To prepare java applications, java has provided the following list of tokens.

1. Identifiers
2. Literals
3. Keywords/ Reserved words
4. Operators

IDENTIFIERS: The identifier is a name assigned to the programming elements like variables, methods, classes, abstract classes, interfaces etc.

Example: int age=25;

int height; // default

t value of int - 0

height=67;

Here

int: data type

age, height: variables(identifier)

=: assignment

25: value/ constant(literal)

;: terminator

Variable & Identifier:

Identifier: An identifier is only used to identify an entity uniquely in program at the time of execution.

Variable: A variable is a name given to a memory location, that is used to hold a value.

LITERALS:

Literal is a constant/value assigned to the variable.

1. Integer/ Integral Literals:

Byte, short, int, long: 10, 20, 30……

char: ‘A’, ‘B’……….

1. Floating point literals:

Float: 10.23f, 20.574f, ….

Double: 11.1234, 123.33534, ……….

1. Boolean literals:

Boolean: True/ False

1. String literals:

String: “welcome”, “Hello”, …….

KEYWORDS/ RESERVED WORDS:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| abstract | continue | for | new | switch |
| assert | default | goto | package | synchronized |
| boolean | do | if | private | this |
| break | double | implements | protected | throw |
| byte | else | import | public | throws |
| case | enum | instanceof | return | transient |
| catch | extends | int | short | try |
| char | final | interface | static | void |
| class | finally | long | strictfp | volatile |
| const | float | native | super | while |

OPERATORS: An operator is a symbol, it will perform a particular operation over the provided operands.

1. Arithmetic operators:

+, -, \*, /, %, ++, --

1. Assignment operator:

=, +=, -=, /=, %=, \*=

1. Comparison operator/ relational operator:

==, !=, <, >, <=, >=

1. Boolean logical operator:

&, |, ^

1. Bitwise logical operators:

&, |, ^, <<, >>

1. Ternary operator:

Condition? Expression1: Expression2;

1. Short-circuit operator:

&&, ||

ARITHMETIC OPERATORS:

**public** **class** Demo {

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

{

**int** a=10;

**int** b=5;

System.***out***.println(a+b); // addition

System.***out***.println(a-b); // subtraction

System.***out***.println(a\*b); // multiplication

System.***out***.println(a/b); // division

System.***out***.println(a%b); // modulo division

// increment and decrement operators

System.***out***.println(a); //10

System.***out***.println(a++); //10 //post-increment => (print, increase)

System.***out***.println(++a); //12

System.***out***.println(a--); //12

System.***out***.println(--a); //10

System.***out***.println(a); //10

}

}

Output: 15

5

50

2

0

10

10

12

12

10

10

Eg2: **public** **class** Demo {

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

{

**int** a=5;

System.***out***.println((--a+--a)\*(++a-a--)+(--a+a--)\*(++a+a++));

// (4 + 3) \* (4 - 4) + (2 + 2) \* (2 + 2)

}

}

Output:16

ASSIGNMENT OPERATORS:

**public** **class** Demo {

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

{

**int** a=10;

**int** b=5;

System.***out***.println(a+=b); // a=a+b => a=10+5=15

System.***out***.println(a-=b); // a=a-b => a=15-5=10

System.***out***.println(a\*=b); // a=a\*b => a=10\*5=50

System.***out***.println(a/=b); // a=a/b => a=50/5=10

System.***out***.println(a%=b); // a=a%b => a=10%5=0

System.***out***.println(a); // a=0

}

}

Output: 15

10

50

10

0

0

COMPARISON OPERATORS: This operator returns the Boolean values

**public** **class** Demo {

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

{

**int** a=5,b=10;**int** c=30;

System.***out***.println(a<b);

System.***out***.println(a>b);

System.***out***.println(a==b);

System.***out***.println(a!=b);

System.***out***.println(a<=b);

System.***out***.println(a>=c);

}

}

Output: true

false

false

true

true

LOGICAL OPERATORS/ SHORT-CIRCUIT OPERATORS:

Difference between logical operators and short-circuit operators:

(&) In Boolean logical operator if one condition is true/false it will go to the next statement and validate that.

(&&) In short circuit operator if one condition is false it will not go to the next condition/ statement.

(|) In Boolean logical operator if one condition is true/false it will go to the next statement and validate that.

(||) In short circuit operator if one condition is true it will not go to the next condition/ statement.

|  |  |  |
| --- | --- | --- |
| A | B | A && B |
| True | True | True |
| False | True | False |
| True | False | False |
| False | False | False |

|  |  |  |
| --- | --- | --- |
| A | B | A || B |
| True | True | True |
| False | True | True |
| True | False | True |
| False | False | False |

|  |  |  |
| --- | --- | --- |
| A | B | A xor B |
| True | True | False |
| False | True | True |
| True | False | True |
| False | False | False |

XOR 🡪 ^

Example for | and ||:

**public** **class** Demo {

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

**int** a=10;

**int** b=10;

**if**((a++ == 10) | (b++ == 10))

{

System.***out***.println(a+" "+b); //OUTPUT: 11 11

}

**int** c=10;

**int** d=10;

**if**((c++ == 10) || (d++ == 10))

{

System.***out***.println(c+" "+d); //OUTPUT: 11 10

}

}

}

Output: 11 11

11 10

Example for & and &&:

**public** **class** Demo {

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

**int** a=10;

**int** b=10;

**if**((a++ != 10) & (b++ != 10))

{ }

System.***out***.println(a+" "+b); //OUTPUT: 11 11

**int** c=10;

**int** d=10;

**if**((c++ != 10) && (d++ != 10))

{ }

System.***out***.println(c+" "+d); //OUTPUT: 11 10

}

}

Output: 11 11

11 10

BITWISE LOGICAL OPERATORS:

**public** **class** Demo {

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

{

**int** a=10;

**int** b=2;

System.***out***.println(a&b);

System.***out***.println(a|b);

System.***out***.println(a^b);

System.***out***.println(a<<b);

System.***out***.println(a>>b);

}

}

Output: 2

10

8

40

2

Explanation:

int a=10;// 1010

int b=2;// 0010

a&b--->10&2--> 0010-----> 2

a|b--->10|2--> 1010-----> 10

a^b--->10^2--> 1000-----> 8

a<<b ---> 10<<2 -----> 00001010

00101000---> 40

--> Remove 2 symbols at left side and append 2 0's at right side.

a>>b ---> 10>>2 -----> 00001010

00000010

--> Remove 2 symbols at right side and append 2 0's at left side.

Note: Removable Symbols may be o's and 1's but appendable symbols must be 0's.

TERNARY OPERATORS:

(condition)? Expression1: Expression;

JAVA STATEMENTS: Statement is the collection of expressions.

1. General Purpose Statements

Declaring variables, methods, classes etc

Creating objects, accessing variables, methods etc.

1. Conditional Statements

If, if-else, switch

1. Iterative Statements

for, while, do-while

1. Transfer Statements

Break, continue, return

1. Exception Handling Statements

Try-catch-finally, throw, throws

1. Synchronized Statements

Synchronized method, synchronized block

1. General Purpose statements:

Variable: A variable is a name given to a memory location, that is used to hold a value.

Variables are of 2 types

Global variable and local variable.

Java does not support global variables – It supports instance variables and class variable. It also supports local variables.

\*\* Instance variables and class variables are created inside the class and outside the method. In the below program we cannot print the instance variable. If we try to print (outside the method and inside the method) it will show an error.

\*\* we can only access the instance variable by using object. The life of an instance variable is depends on the life of an object.

**public** **class** Demo {

**int** x=10; // instance variable or non-static variable

**static** **int** *y*=20; // class variable or static variable

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

**int** z=40; // local variable

System.***out***.println(*y*+" "+z);

// we can also call static variable by using classname.variablename

System.***out***.println(Demo.*y*);

}

}

Output: 20 40

20

**public** **class** Demo {

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

**int** z; // local variable

System.***out***.println(z);

}

}

Output: this will give you the error. In java the local variables must be initialized before we use it.

LOCAL VARIABLE: local variables must be initialized.

1. CONDITIONAL STATEMENTS:

Syntax1:

if(condition)

{

---instructions----

}

Syntax2:

if(condition)

{

---instructions----

}

else

{

----instructions----

}

Syntax3:

if(condition)

{

---instructions----

} else if(condition)

{

---instruction----

} else if(condition)

{

---instructions----

}

....

....

else {

----instructions----

}

Example for if statement**:**

**Eg:**

**public** **class** Demo {

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

**int** z=40;// local variable

**int** y;

**if**(z==40) // if(40==40) it will not error

{

y=100;

}

System.***out***.println(y);//It will show error.

}

}

Output: In this also it will show error at printing y, why because the condition is checking with variable. The variable value may vary according to user input. Here we must initialize the local variable.

But we can check the condition with constant value it don’t show error.

To overcome that variable error in printing we use else block

Example for if-else statement:

**public** **class** Demo {

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

**int** z=40;// local variable

**int** y;

**if**(z==40)

{

y=100;

}

**else**

y=200;

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

}

}

Output: 100

Example for else-if ladder**:**

**public** **class** Demo {

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

{

**int** i=10;

**int** j;

**if**(i==10)

{

j=20;

}

**else** **if**(i==20)

{

j=30;

}

**else**

{

j=40;

}

System.***out***.println(j);

}

}

Output: 20

Switch condition:

'if' is able to provide single condition checking by default, but switch is able to provide multiple conditions checking.

Syntax:

switch(var)

{

case 1: -----instructions-----

break;

case 2: ----instructions------

break;

case n: ----instructions-----

break;

default: ----instructions-----

break;

}

Example for switch statement:

**public** **class** Demo {

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

{

**int** i=10;

**switch**(i)

{

**case** 5:

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

**break**;

**case** 10:

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

**break**;

**case** 15:

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

**break**;

**case** 20:

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

**break**;

**default**:

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

**break**;

}

}

}

Output: Ten

Special Features in switch case:

Eg: **public** **class** Demo {

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

{

//switch have return type so we add semi-colon at end of switch

String card="king";

**int** currentTotalValue=10;

**int** currentValue=**switch**(card){

**case** "king", "queen", "jack" ->10; //It returns integer->it will catch by currentValue

**case** "ace"->{

**if**(currentTotalValue<11)

{

**yield** 11;//return 11

}

**else** {

**yield** 1;

}

}

**default** ->Integer.*parseInt*(card);

};

System.***out***.println("Current card value:"+currentValue);

System.***out***.println("Total value:"+(currentTotalValue+currentValue));

}

}

Output:

Current card value:10

Total value:20

Eg://error in this

**public** **class** Demo {

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

{

**record** Person(String firstName,String lastName,**int** age) {

}

String var1="Hello world";

Integer var2=34;

String[] var3= {"siri","sha"};

Person var4=**new** Person("Saranya","R",3);

Person var5=**new** Person("arjun","M",25);

Object obj=var1;

**switch**(obj) {

**case** String msg -> System.***out***.println(msg);

**case** Integer num -> System.out.println(num);

**case** Person p && p.firstName().length()>3 -> System.out.println("looks like saranya");

**case** String[] arr -> System.out.println(arr);

**case** **default** -> System.out.println("Have no idea");

}

}

}

Rules to write switch:

1. switch is able to allow the data types like byte, short, int, char and String.

2. In switch, all cases and default are optional, we can write switch without cases and with default, we can write switch with cases and without default, we can write switch without both cases and default also

**METHOD:**

**public** **class** Demo {

**public** **static** **void** add(**int** a,**int** b) //any method daclare as static we can call that without creating object.

{

System.***out***.println(a+b);

}

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

*add*(100,200); //method calling

}

}

JAVA OOP CONCEPTS:

**Object** means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts.

Encapsulation

Polymorphism

Inheritance

Abstraction

Static loading: A block of code would be loaded into the RAM before it is executed, (after being loaded into the RAM, it may or may not execute).

--🡪 while executing, the entire application will be loaded into the RAM.

Dynamic loading: A block of code would be loaded into the RAM, only when it is required to be executed.

Why it is dynamic:

Demo.java ---🡪 compiler (java compiler) ---🡪 Demo.class ---🡪 while executing, only the required part will be loaded into the RAM, remaining part will be there inside the hard disk in the form of byte code.

\*\*NOTE: C program follows static loading, whereas java follows dynamic loading.

CLASS: A class is a blue print of an object. We can create multiple objects to a particular class. A class is a collection of data members and methods.

We can create an empty and we can generate the .class file for the empty class.

--Here we can also use class as the data type.

--class can only extend one class and implements many interfaces.

Eg: public class Demo extends Super implements Interface1,Intereface2;

--Here by default every class can extend the object class.

Example: **public** **class** Demo **extends** Object

Eg:

**package** business;

**class** Company {

NewsAgency na;

}

**package** business;

**class** NewsAgency{

Company company;

}

Explanation: here both the classes are present inside the same package that’s why they are sharable.

--both the classes are default. The default classes are sharable with in the class and outside the class and within the package.

**default** **class** Company {

NewsAgency na;

}

We cannot do like this.

Eg: Demo.java

**public** **class** Demo

{

// here we can compile the class but we cannot run the class without main method.

}

OBJECT: Object is an instance of class. An object can have state and behaviour.

State of an object: Data which is present to the object is known as state of an object.

Behaviour of an object: methods which are applicable to the object is known as behaviour of an object.

Example: - car class: can have multiple objects like – Creta, Scorpio, Safari

Car object can have their own states: Data - variables

Model

Color

Price

And behaviour: Logic – functions/ methods

Start ()

Stop ()

changeGare ()

Object creation:

* For a class reference variable 3 possible values are there:
* 1. Its own class object

Demo d=new Demo();

* 2. It’s child class object.

Demo d=new DemoChild(); // it is possible only if DemoChild class is the child class of Demo class.

* 3. Default value for any reference variable. – null

Demo d=null;

There are 2 ways to create it’s own class object:

1. Demo d=new Demo ();

Here the new keyword is allocated memory to the non-static data and that address is stored in the d variable. By using that address we are accessing all non-static data.

1. new Demo (). Fun ();

Here the memory is allocated and the address is not stored in any variable directly we are calling address. Function.

Eg: **public** **static** **void** main(String args[])

{

**int** i=230; // local primitive variable

Demo d1=**new** Demo();// reference or class variable, d1 is pointing to the demo class object.

Demo d2= 500;// we cannot assign directly a value to the class object.

}

Output: compile time error

**public** **class** Demo {

**int** x;

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

{

**int** i=230;

Demo d1=**new** Demo();

d1.x=100;

System.***out***.println(d1.x);//100

Demo d2= d1; // assigning object d1 to object d2 it is possible. the both the memory locations are also same

System.***out***.println(d2.x); //100

}

}

Output: 100

100

**public** **class** Demo {

**int** z=100; // non-static or instance variable

**void** fun() {

System.***out***.println("inside fun1");

}

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

{

**int** x=10;

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

System.***out***.println(z); // It will show the error because z is a non-static member variable we cannot access the non-static members inside static method.

}

}

Output: compile time error

To overcome this conflict we create object and access the non-static members and methods.

**public** **class** Demo {

**int** z=100; // non-static or instance variable

**void** fun() {

System.***out***.println("inside fun");

}

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

{

**int** x=10; // local variable

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

Demo obj=**new** Demo(); // class reference variable

System.***out***.println(obj.z); // By using the object the non-static members are called inside the static method.

obj.fun(); // method calling

System.***out***.println(obj); // It will display the address of the object.

}

}

Output 10

100

inside fun1

com.masai.Demo@7c30a502

3 type: object creation

\*\*\*NOTE: one object (memory space) can be referred by multiple variables simultaneously, but one variable cannot refer multiple objects simultaneously.

**public** **class** Demo {

**int** x;

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

{

Demo d1=**new** Demo();

d1.x=100;

System.***out***.println(d1.x);//100

d1=**null**;

System.***out***.println(d1.x); // null pointer exception

}

}

Output: 100

Exception in thread "main" java.lang.NullPointerException: Cannot read field "x" because "d1" is null

at Myproject/com.masai.Demo.main(Demo.java:20)

Garbage Collector: The object which is not referenced by any ref variable will be treated as garbage, and in java there is a separate thread running continuously, called garbage collector, the duty of this garbage collector is to kill that un-referenced object and free the RAM. (the above example d1 has no ref variable it is null).

--if any variable is holding the address of any object then that object is not treated as garbage.

**public** **class** Demo {

**int** x;

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

{

Demo d1=**new** Demo(); // this will be killed

d1.x=100;

System.***out***.println(d1.x);//100

d1=**new** Demo(); //this object is active now

System.***out***.println(d1.x);

}

}

Output: 100

0

\*\*how much space is available for garbage collector:

**public** **class** Demo {

**int** x;

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

{

Demo d1=**new** Demo();

Demo d2=d1;

d1=**null**;

// how many objects are available to garbage collector?

// no space is available for garbage collector

//d1 and d2 are pointing to same heap area, so d1 is null, but d2 holding that heap area.

System.***out***.println(d2.x); //0

}

}

Output: 0

Eg:

**public** **class** DatatypeEx {

**int** x=20;

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

DatatypeEx d=**new** DatatypeEx();

DatatypeEx d1=d;

System.***out***.println(d.x);

System.***out***.println(d1.x);

d=**null**;

System.***out***.println(d1.x);

System.***out***.println(d.x);

}

}

Output:

20

20

20

Exception in thread

Responsibility of “new”: in object creation

1. Reserve the memory space inside the heap area of RAM.
2. Load all non-static members of that class into this area.
3. While loading the non-static variables, if that variable is not initialized, then default initialization will be provided by the new operator.

\*\* Note: In java we cannot create nested methods. One method cannot contain another method but it calls another method.

\*\* Note: we cannot run the executable statements inside the class and outside the method. Executable statements present inside the block or method.

Example:

**public** **class** Demo {

**int** x=10; // Instance variable

// we cannot do like below

x=20; // reassigning is not possible because it is an executable statement

System.out.println(x);

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

{

**int** x=10; // local variable

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

}

}

Output: compile time error

A possibility that we can write executable files inside a block.

Example:

**public** **class** Demo {

**int** x=100; // Instance variable

{

x=20;

System.***out***.println(x); // in this there is no error but it does not print the statements inside this block when object is created then these will execute.

}

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

{

**int** x=10; // local variable

System.***out***.println(x);//10

}

}

Output: 10

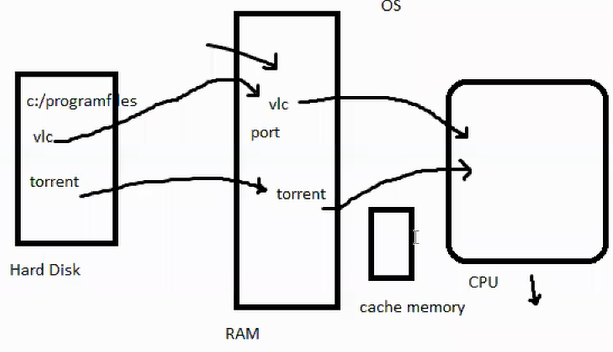
Benefits of OOP:

* Reduce complexity
* Easier maintenance
* Faster Development
* Code reusability

How our computer system works:

* RAM/ ROM (primary memory)
* Hard disk (secondary memory)
* Operating System
* CPU/ processor

This is the process for downloading an application



Cache memory store the frequently used applications. It can reduce the accessibility time.

Until anything(data) loaded into the RAM it will not available to the CPU for the execution.

**Classes and Objects:**

**Creation of objects**:

Example: creating object to class A inside main method

**class** A

{

**int** x=10;

**void** funA()

{

System.***out***.println("inside the funA");

}

}

**public** **class** Demo {

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

{

System.***out***.println("Inside main method of demo");

A obj=**new** A();

System.***out***.println(obj.x);

obj.funA();

}

}

Output: Inside main method of demo

10

inside the funA

**Accessing instance object:**

\*\*If the object is created outside the method.

**class** A

{

**int** x=10;

**void** funA()

{

System.***out***.println("inside the funA");

}

}

**public** **class** Demo {

**int** x=30;

A obj=**new** A(); // it is an instance object you want call this you can create a object to public class in main method.

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

{

System.***out***.println("Inside main method of demo");

// creating object to Demo class

Demo d=**new** Demo();

d.obj.funA();// accessing class A function

System.***out***.println("instance variable in demo class:"+d.x);

System.***out***.println("variable in class A:"+d.obj.x);

}

}

Output: Inside main method of demo

Inside the funA

Instance variable in demo class:30

Variable in class A:10

\*\*Object is created inside the method.

--there is no need of another class object for calling A class methods.

Eg: **class** A{

**void** fun()

{

System.***out***.println("inside class A");

}

}

**public** **class** Practice {

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

{

A a=**new** A();

a.fun();

}

}

Output:

inside class A

Example3:

**class** A

{

**int** x=10;

**void** funA()

{

System.***out***.println("inside the funA");

}

}

**public** **class** Demo {

**int** x; //0

A obj; // null

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

{

System.***out***.println("Inside main method of demo");

// creating object to Demo class

Demo d=**new** Demo();

d.obj=**new** A();

d.obj.funA();

System.***out***.println("instance variable in demo class:"+d.x);

System.***out***.println(d.obj);

}

}

Output:

Inside main method of demo

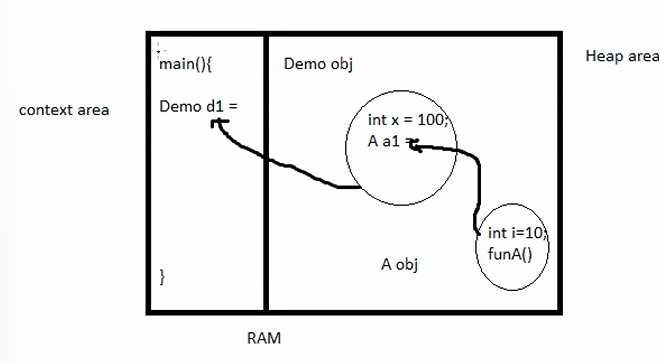
inside the funA

instance variable in demo class:0

com.masai.A@49e4cb85

\*\* Note: We can define a class as instance member of another class as well, by this we establish “Has-A” relationship between 2 objects.

--Diagram for has-a relation in above program



**public** **class** Demo

{

**int** x=100;

Demo d1=**new** Demo();

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

{

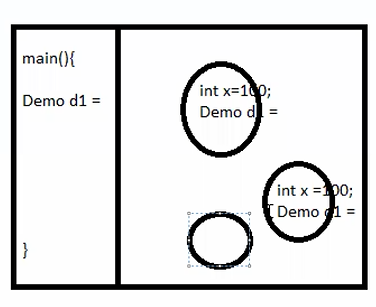
Demo d1=**new** Demo();

}

}

Output: compile time error

Stack overflow Exception (It generates infinite loop).



**public** **class** Demo

{

**int** x=100;

Demo d1; // the initial value is null

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

{

Demo d1=**new** Demo();

System.***out***.println(d1.d1);

d1.d1=**new** Demo(); // allocating memory space

System.***out***.println(d1.d1); //It will return address

System.***out***.println(d1.d1.x); // by using address we are calling non-static variable.

}

}

Output: null

com.masai.Demo@7c30a502

100

**public** **class** Demo {

**static** **void** fun()

{

System.***out***.println("Inside the fun");

}

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

{

System.***out***.println("Inside the main method");

// calling function

*fun*();

}

}

Output: Inside the main method

Inside the fun

Eg:

**class** A

{

**void** funA()

{

System.***out***.println("Inside class A function A");

}

}

**public** **class** Demo {

**static** **void** fun(A a)

{

System.***out***.println("Inside the fun");

a.funA(); // inside this function we are calling funA() of class A by using object.

}

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

{

System.***out***.println("Inside the main method");

// calling function

A a=**new** A();

*fun*(a);// we are passing class A object to the fun().

}

}

Output: Inside the main method

Inside the fun

Inside class A function A

Eg: null value object

**class** A

{

**void** funA()

{

System.***out***.println("Inside class A function A");

}

}

**public** **class** Demo {

**static** **void** fun(A a)

{

**if**(a!=**null**)

{

System.***out***.println("Inside the fun");

a.funA(); // inside this function we are calling funA() of class A by using object.

}

**else**

{

System.***out***.println("Don't pass null values");

}

}

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

{

System.***out***.println("Inside the main method");

//creating object

A a=**null**;

*fun*(a);// we are passing class A object to the fun().

}

}

Output: Inside the main method

Don’t pass null values

Method with return type integer and returning an object:

**class** A

{

**void** funA()

{

System.***out***.println("Inside calss A function A");

}

}

**public** **class** Demo {

**static** **int** fun()

{

**int** x=100;

**return** x;

}

**static** A funX()// A is a class

{

A a=**new** A();

**return** a;

}

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

{

System.***out***.println("Inside the main method");

**int** res=*fun*();

System.***out***.println("method fun() return value:"+res);

A obj=*funX*();

obj.funA();

}

}

Output: Inside the main method

method fun() return value:100

Inside class A function A

Accessing static members: To access the static members from the main method, we have 3 options:

1. We can access directly without any ref variable (with in the same class)
2. By using class name. (dot) // it is most recommended way we can call like this in another methods also.
3. By creating an object of that class and through object reference.

Eg:

**public** **class** Demo

{

**static** **int** *x*=100;

**static** **void** fun()

{

System.***out***.println("inside fun of demo");

}

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

{

// 1st way

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

*fun*();

// 2nd way

System.***out***.println(Demo.*x*);

Demo.*fun*();

// 3rd way

Demo d=**new** Demo();

System.***out***.println(d.*x*);

d.*fun*();

}

}

Output:

100

inside fun of demo

100

inside fun of demo

100

inside fun of demo

**Static object**:

**class** A

{

**void** fun()

{

System.***out***.println("inside the fun in class A");

}

}

**public** **class** Demo

{

**static** **int** *x*=100;

**static** A *a1*=**new** A(); //creating object to class A

**int** y=20;

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

{

System.***out***.println(Demo.*x*);

System.***out***.println(Demo.*a1*);// a1 is a static object.

Demo.*a1*.fun();

}

}

Output: 100

com.masai.A@49e4cb85

inside the fun in class A

The above example is same as – System.out.println();

-- fun() belongs to the A class, similarly println() method belongs to PrintStream class.

-- PrintStream class is statically defined inside the System class with ‘out’ variable.

-- example of System class

class System

{

static PrintStream out= new PrintStream();

}

Eg:

**public** **class** Demo

{

**static** **int** *x*=100;

**static** Demo *d1*=**new** Demo();;

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

{

Demo d1=**new** Demo();

d1=**null**; // without this also it prints same

System.***out***.println(d1.*x*); //100

System.***out***.println(d1.*d1*.*x*); //100

}

}

Output:

100

100

**Static Block**:

**public** **class** Demo{

**public** **static** **int**[] *nums*=*number*();

**static** {

*nums*=**new** **int**[4];

*nums*[0]=1;

*nums*[1]=2;

*nums*[2]=3;

*nums*[3]=4;

**for**(**int** i:*nums*)

{

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

}

}

**static** **int**[] number() {

**int**[] nums=**new** **int**[4];

nums[0]=1;

nums[1]=2;

nums[2]=3;

**return** nums;

}

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

{

}

}

Output:

static:1

static:2

static:3

static:4

METHOD:

Java has two types of methods

1. Normal or concrete method: method with body
2. Abstract method: method without body

The method is present only inside of the class.

* We can call static methods directly like fun(); Or we can call by (classname.methodname).
* Non- static methods are called by using objects only.

Example:

**public** **class** Demo

{

**int** x=200;

// method declaration

**public** **void** fun(**int** x)

{

System.***out***.println("inside the fun method "+x);

}

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

{

Demo d=**new** Demo();

//method calling

d.fun(12);

}

}

Output: inside the fun method 12

Passing function parameter as class object:

**class** A

{

**void** funA()

{

System.***out***.println("inside the funA in class A");

}

}

**public** **class** Demo

{

**int** x=200;

// method declaration

**public** **void** fun(A a1)

{

System.***out***.println("inside the fun method "+a1);

}

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

{

Demo d=**new** Demo();

//method calling

d.fun(**null**);// here we are passing object address as “null”

}

}

Output:

inside the fun method null

* passing object address as parameter.

**class** A

{

**void** funA()

{

System.***out***.println("inside the funA in class A");

}

}

**public** **class** Demo

{

**int** x=200;

// method declaration

**public** **void** fun(A a1)

{

System.***out***.println("inside the fun method "+a1);

}

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

{

Demo d=**new** Demo();

A obj=**new** A();

//method calling

d.fun(obj);

d.fun(**new** A()); //creating object and directly passing address to the function.

}

}

Output: inside the fun method com.masai.A@1d251891

inside the fun method [com.masai.A@2133c8f8](mailto:com.masai.A@2133c8f8)

d.fun(obj); //after the method execution A obj will not be eligible for garbage collection.

d.fun(**new** A()); // after the method execution A obj will be eligible for garbage collection.

Method with return type:

**public** **class** Demo

{

**int** fun()

{

System.***out***.println("inside fun of demo");

**byte** b=10;

**return** b;

}

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

{

Demo d=**new** Demo();

**int** i=d.fun();

**byte** b=(**byte**)d.fun();

**long** l=d.fun();

System.***out***.println(i+" "+b+" "+l);

}

}

Output:

inside fun of demo

inside fun of demo

inside fun of demo

10 10 10

Object of a class be the return type of a method:

**class** A

{

**void** funA()

{

System.***out***.println("inside the funA of A");

}

}

**public** **class** Demo

{

A fun()

{

System.***out***.println("inside fun of demo");

A a1=**new** A();

**return** a1;

//return null;

//return new A();

}

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

{

Demo d=**new** Demo();

A obj=d.fun();

obj.funA();// by using address of obj calling class A method.

}

}

Output:

inside fun of demo

inside the funA of A

passing arguments and return type:

**class** A

{

**void** funA()

{

System.***out***.println("inside the funA of A");

}

}

**public** **class** Demo

{

A fun(String username, String password)

{

**if**(username.equals("admin") && password.equals("123"))

{

A a1=**new** A();

**return** a1;

}

**else**

**return** **null**;

}

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

{

Demo d=**new** Demo();

//Object obj=d.fun("admin", "123");

A a1=d.fun("admin", "123");

a1.funA();

}

}

Output: inside the funA of A

**NOTE**:

--In java there is a class called Object class, it is a predefined class in java library which is a super class (parent class) of any classes in java world.

--so if a method returning an object, then we can hold that object either in it’s own class reference variable.

-- to the object class variable we can assign/ store any class object.

Object obj=d.fun("admin", "123");

A a1=d.fun("admin", "123");

POLYMORPHISM:

Defining more than one method with the same name inside the class.

Two types of polymorphisms:

1. Static polymorphism (compile time polymorphism): more than one method with the same name but with different parameters (which method will be executed will be decided at compile time), we achieve static polymorphism by using method overloading.
2. Dynamic polymorphism (runtime polymorphism): more than one method with same name and same parameter, dynamic polymorphism we achieve through inheritance (by using method overriding), which method will be executed will be decided at runtime.

Example: overloading concept

**public** **class** Demo

{

**void** fun()

{

System.***out***.println("inside fun of demo");

}

**void** fun(**int** x)

{

System.***out***.println("inside fun(int x) of demo");

}

**void** fun(**float** f)

{

System.***out***.println("inside fun(float f) of demo");

}

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

{

Demo d=**new** Demo();

d.fun(10);

d.fun();

d.fun(10.5f);

}

}

Output:

inside fun(int x) of demo

inside fun of demo

inside fun(float f) of demo

Eg2:

**public** **class** Demo

{

**void** fun(**int** x,**float** y)

{

System.***out***.println("inside fun(int x, float y) of demo");

}

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

{

Demo d=**new** Demo();

d.fun(10,10);

}

}

Output: inside fun(int x, float y) of demo

Disadvantage with overloading concept:

The main drawback of the static polymorphism is, compiler may goes to the ambiguous state

Example:

**public** **class** Demo

{

**void** fun(**int** x,**float** y)

{

System.***out***.println("inside fun(int x, float y) of demo");

}

**void** fun(**float** x,**int** y)

{

System.***out***.println(" fun(int x, float y) of demo");

}

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

{

Demo d=**new** Demo();

d.fun(10,10);

}

}

Output: It will give the compilation error

Eg2:

**class** A

{

**void** funA()

{

System.***out***.println("inside the funA of A");

}

}

**class** B

{

**void** funB()

{

System.***out***.println("inside the funB of B");

}

}

**public** **class** Demo

{

**void** fun(A a1)

{

System.***out***.println("inside fun(A a1) of demo");

}

**void** fun(B b1)

{

System.***out***.println("inside fun(B b1) of demo");

}

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

{

Demo d=**new** Demo();

//d.fun(null);// ambiguity

d.fun(**new** A());

d.fun(**new** B());

}

}

Output: inside fun(A a1) of demo

inside fun(B b1) of demo

\*\*Calling class A main method inside Demo class main method:

A.java

**public** **class** A {

**int** i=10;

**void** funA()

{

System.***out***.println("inside funA of A");

}

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

{

System.***out***.println("inside main of A");

}

}

Demo.java

**public** **class** Demo

{

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

{

A.*main*(**null**); -- we can pass args in place of null

//calling A class main method.

}

}

Output: inside main of A

**CONSTRUCTOR:**

It is a non-static method which will be executed automatically at the time of creating an object.

NOTE: when we compile a .java file of a class, Java compiler verifies, is there any constructor in our .java file or not. If we place any constructor manually inside a .java file, java compiler will place the same constructor inside the .class file, but if we don’t place any constructor explicitly inside our .java file then java compiler will place a default constructor inside the .class file.

class Demo()

{

}

NOTE: we can have a .java file for a class without a constructor, but we can’t have .class file for a class without a constructor.

--default constructor given by the java compiler will always public and zero argument and it is empty body.

Eg: Default constructor

Public demo()

{

// super();

}

--strikly speaking, it is not a empty body, there is one hidden statement is there inside the default constructor as a first statement.

--until the last statement of the constructor is not executed, object is not created completely. So far an object creation, constructor execution is mandatory.

**Difference between normal method and constructor**:

|  |  |
| --- | --- |
| Method | Constructor |
| 1.Method name can be any name. | 1.constructor name must be the class name. |
| 2.A method must have a return type at least void. | 2.constructor does not have return type. |
| 3.Method can be static. | 3.Static keyword is not applicable with constructor. |
| 4.on a single object we can call a method multiple time. | 4.on a single object a constructor will be called only once. |
| 5.A method can be abstract and can be final also. | 5.abstarct and final keywords is not applicable with constructor. |

**Similarities: method & constructor**

1.Both are the code block, we can write multiple executable statements.

2.As we can overload a method, we can overload a constructor also. And all the static polymorphism rules are applicable with the constructor overloading.

Constructor overloading:

**public** **class** Demo {

Demo()

{

System.***out***.println("Constructor Demo().....");

}

Demo(**int** i)

{

System.***out***.println("constructor Demo(int i)..");

}

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

{

Demo d1=**new** Demo();

Demo d2=**new** Demo(10);

}

}

Output: Constructor Demo().....

constructor Demo(int i)..10

Eg2: **public** **class** Demo {

Demo(**int** i)

{

System.***out***.println("constructor Demo(int i).."+i);

}

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

{

Demo d1=**new** Demo(); //it won't take the object as dafault constructor because we already we defined a parameterized constructor in class.

Demo d2=**new** Demo(10); // this object will work.

}

}

Output: compile time error

Constructor overloading:

**public** **class** Demo {

Demo()

{

System.***out***.println("Constructor Demo().....");

}

Demo(**int** i)

{

System.***out***.println("constructor Demo(int i)..."+i);

}

Demo(String s)

{

System.***out***.println("constructor Demo(String )..."+s);

}

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

{

Demo d1=**new** Demo();

Demo d2=**new** Demo(100);

Demo d3=**new** Demo("SIRI");

}

}

Output: Constructor Demo().....

constructor Demo(int i)...100

constructor Demo(String )...SIRI

\*\*In constructor we can also pass the class object as parameter

**public** **class** Demo {

Demo(A a1)

{

System.***out***.println("constructor Demo(A a1)..."+a1);

a1.funA();

}

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

{

A a1=**new** A();

Demo d1=**new** Demo(a1);

Demo d2=**new** Demo(**new** A());

}

}

Output:

constructor Demo(A a1)...com.mic.A@1d251891

inside the fuuA of A

constructor Demo(A a1)...com.mic.A@2133c8f8

inside the fuuA of A

**this Keyword**:

---It will represent the current class object.

--- There are three uses of ‘this’ keyword

1. To represent the current class object.
2. To differentiate the instance variable and the local variable.
3. To call a constructor of a class from the another constructor of the same class.

Type 2:

**public** **class** Demo {

**int** x=100;//non-static variable

// non-static variables are sharable

**void** fun()

{

**int** x=483;

System.***out***.println("inside dun of Demo");

System.***out***.println(x); // local variable

System.***out***.println(**this**.x); // instance variable

System.***out***.println(**this**);

// current object on which fun1 is called.

}

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

{

Demo d1=**new** Demo();

System.***out***.println(d1);

d1.fun();

}

}

Output: com.mic.Demo@27f674d

inside dun of Demo

Local variable:483

Instance variable:100

com.mic.Demo@27f674d

\*\*\*\*\*NOTE: ‘this’ keyword we cannot use inside the static area. this keyword refer to the object.

If we use it give the compile time error. Because object is not created to the static members.

--- If you want to call constructor, then that call must be from another constructor of the same class (by using ‘this’ keyword) or from the constructor of child class (by using ‘super’ keyword).

--- that call of the constructor must be the first statement inside a constructor.

**public** **class** Demo {

Demo()

{

System.***out***.println("inside Demo()...");

}

Demo(**int** i)

{

System.***out***.println("inside Demo(int i)..."+i);

}

Demo(String s)

{

System.***out***.println("inside Demo(String s)..."+s);

}

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

{

Demo d1=**new** Demo("hello");// here it only load the Demo(string s) it will not load remaining constructors.

}

}

Output: inside Demo(String s)… hello

\*\* to overcome above problem

**public** **class** Demo {

Demo()

{

**this**(10); // here it calls the constructor who take integer as parameter.

System.***out***.println("inside Demo()...");

}

Demo(**int** i)

{

**this**("siri"); // constructor calling is the first statement in the body.

System.***out***.println("inside Demo(int i)..."+i);

}

Demo(String s)

{

System.***out***.println("inside Demo(String s)..."+s);

}

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

{

Demo d1=**new** Demo();// by using single object we can call all constructors by using 'this' keyword.

}

}

Output:

inside Demo(String s)...siri

inside Demo(int i)...10

inside Demo()...

\*\* there is no order for calling constructors.

Example:

**public** **class** Demo {

Demo()

{

System.***out***.println("inside Demo()...");

}

Demo(**int** i)

{

**this**("siri");

System.***out***.println("inside Demo(int i)..."+i);

}

Demo(String s)

{

**this**();

System.***out***.println("inside Demo(String s)..."+s);

}

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

{

Demo d1=**new** Demo(10);

}

}

Output: inside Demo()...

inside Demo(String s)...siri

inside Demo(int i)...10

PURPOSE OF CONSTRUCTORS:

1. If we want to execute some statements at the time of our object creation, then we can keep those statements inside the constructor.
2. To initialize the instance variable (initialize an object).

Example:

**class** Student {

**int** roll;

String name;

**int** marks;

//zero argument constructor

Student()

{

}

//parameterized constructor

Student(**int** roll,String name,**int** marks)

{

**this**.roll=roll;// we are assigning local variable to instance variable.

**this**.name=name;

**this**.marks=marks;

}

**public** **void** showDetails()

{

System.***out***.println("Roll is:"+roll);

System.***out***.println("Name is:"+name);

System.***out***.println("Marks is:"+marks);

}

}

**public** **class** Demo {

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

{

Student s=**new** Student(1,"siri",750);

//without using constructor we can write

Student s1=**new** Student();

s1.roll=101;

s1.name="DHAANVI";

s1.marks=800;

s.showDetails();

s1.showDetails();

}

}

Output: Roll is:1

Name is: siri

Marks is:750

Roll is:101

Name is: DHAANVI

Marks is:800

Pure Encapsulation:

---mark our class variable as private and expose them outside the class through the public getter and setter method.

---don’t show our data directly.

--getter & setter: Getters and setters are used to protect your data, particularly when creating classes. For each instance variable, a getter method returns its value while a setter method sets or updates its value. Given this, getters and setters are also known as accessors and mutators, respectively.

Eg: **private** **int** roll;

**private** String name;

**private** **int** marks;

private variables are accessed only inside the class. We cannot access variables outside the class.

--To overcome that we use BEAN class.

JAVA BEAN CLASS: Pure encapsulated  
1. This class should be public.

2.variables/fields should be private.

3. for each variable/field there should be corresponding public getter and setter method.

4. This must have zero argument constructor/default constructor.

5. This class may have parameterized constructor.

Example:

Student.java

**public** **class** Student {

**private** **int** roll;

**private** String name;

**private** **int** marks;

**public** Student() {

}

**public** Student(**int** roll, String name, **int** marks) {

**this**.roll = roll;

**this**.name = name;

**this**.marks = marks;

}

**public** **int** getRoll() {

**return** roll;

}

**public** **void** setRoll(**int** roll) {

**this**.roll = roll;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getMarks() {

**return** marks;

}

**public** **void** setMarks(**int** marks) {

**this**.marks = marks;

}

}

Demo.java

**public** **class** Demo {

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

{

Student s1=**new** Student(10,"krishna",200);

Student s=**new** Student();

s.setRoll(19);

s.setName("radha");

s.setMarks(500);

System.***out***.println("Roll is:"+s.getRoll());

System.***out***.println("Name is:"+s.getName());

System.***out***.println("Marks is:"+s.getMarks());

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

System.***out***.println("Roll is:"+s1.getRoll());

System.***out***.println("Name is:"+s1.getName());

System.***out***.println("Marks is:"+s1.getMarks());

}

}

Output:

Roll is:19

Name is:radha

Marks is:500

=================

Roll is:10

Name is:krishna

Marks is:200

Taking input from user:

3 ways to take input from user

1. Scanner class
2. Using command line argument
3. Using BufferedReader class

With the help of CLA, and BR class we can take input only in the form of String.

--- using Scanner class we can take the input in almost all the primitive datatype also.

---Scanner class belongs to java.util package;

--- in java common classes (String, Object, System, etc….) they belong to java.lang package.

Creating scanner class object:

----Scanner sc=new Scanner(System.in); --System.in represent the address of the keyboard.

nextInt();

nextLong();

nextFloat();

nextBoolean();

next(); // read the string

-------- > These methods will read the one token only

nextLine(); // it will read entire line

\*\*difference between next () and nextLine ():

Explanation: next ()

Enter roll:

10 “/n”

Enter name:

Sirisha Challagiri “/n” // here challagiri is consumed as next token as marks value.

Enter marks: // It will show the error input mismatch exception. Because it take the input as challagiri

500 “/n”

Explanation: nextLine ():

Enter roll:

10 “/n”

Enter name: //here sc.nextLine() will consume the above “/n”. It will not accept string .

Kiran Kumar “/n”

Enter marks:

600 “/n”

---So to solve the above problem we have following solutions:

1. Never use nextLine() always read the token by using next();

// with this we cannot read full space separated name.

1. Always use nextLine(); // even to read the primitives also.

Example:

**import** java.util.\*;

**public** **class** Demo {

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

{

Scanner sc=**new** Scanner(System.***in***);

System.***out***.println("Enter roll:");

**int** roll=Integer.*parseInt*(sc.nextLine());

System.***out***.println("Enter name:");

String name=sc.nextLine();

System.***out***.println("Enter marks:");

**int** marks=Integer.*parseInt*(sc.nextLine());

System.***out***.println("The roll is:"+roll);

System.***out***.println("The name is:"+name);

System.***out***.println("The marks is:"+marks);

sc.close();

}

}

Output: Enter roll:

12

Enter name:

challagiri Sirisha

Enter marks:

680

The roll is:12

The name is:challagiri Sirisha

The marks is:680

--How to take input as matrix format:

10 20 30

40 50 60

Example:

**import** java.util.\*;

**public** **class** Demo {

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

{

Scanner sc=**new** Scanner(System.***in***);

System.***out***.println("Enter a matrix:");

**int** n1=sc.nextInt();

**int** n2=sc.nextInt();

**int** n3=sc.nextInt();

**int** n4=sc.nextInt();

**int** n5=sc.nextInt();

**int** n6=sc.nextInt();

System.***out***.println(n1+" "+n2+" "+n3);

System.***out***.println(n4+" "+n5+" "+n6);

sc.close();

}

}

Output:

Enter a matrix:

10 20 30

40 50 60

MATRIX:

10 20 30

40 50 60

--there are multiple overloaded println() method is defined inside the PrintStream class.

1. println(){

// printing line break

}

2.println(primitives){

//print the primitives

}

3.println(String s){

// it will print the content

}

4.println(Object obj){

//it will print the address

}

5.println(char[] chr)

Example:

**import** java.util.\*;

**public** **class** Demo {

Demo(String s)

{

System.***out***.println(s);

}

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

{

String s1=**new** String("welcome");

Demo d1=**new** Demo("hello");

System.***out***.println(s1); // welcome

System.***out***.println(d1);// address

//System.out.println(null);// ambiguity compile time error.

//for null--The method println(char[]) is ambiguous for the type PrintStream

}

}

Output: hello

welcome

com.mic.Demo@7c30a502

**STRING:**

In java String is an object that represent a sequence of characters. Strings are immutable.

---once a string object is created, we cannot modify that object, if we want to modify it by calling its method, those method will return a string object. Instead of modifying that existing string object.

String – java.lang

Eg: String str= ”Welcome to java”;

**public** **class** Demo

{

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

{

String s1="Hello";

String s2="Hello";

String s3=**new** String("Hello"); // object

System.***out***.println(s1);

System.***out***.println(s3);

System.***out***.println(s1==s2);

System.***out***.println(s1==s3);

String s4=**new** String("Welcome");

}

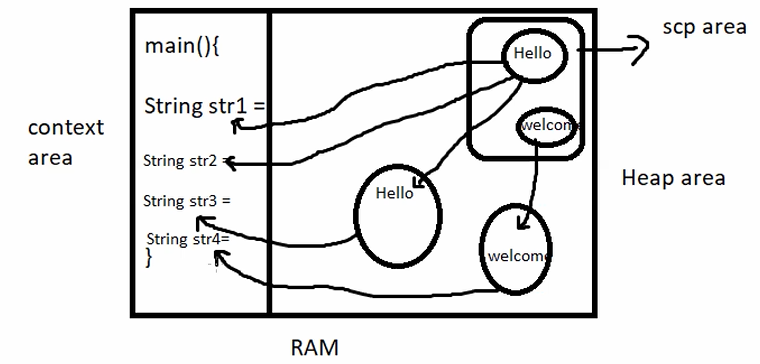
}

Output: Hello

Hello

true

false



SCP area – String constant pool area

String operations:

* concat(): it will not perform operation on the same string it will return the string

Example1:

**public** **class** Demo

{

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

{

String s1="Hello";

// s1.concat("world"); // it will not modify the s1 it will return string.

String s2=s1.concat(" world");

System.***out***.println(s2);

}

}

Output: Hello world

Eg2:

**public** **class** Demo {

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

{

String s1="Welcome";

s1=s1.concat(" "+s1);//it not done the modification on the s1. we are re-assigning string to s1.

System.***out***.println(s1);

}

}

Output: Welcome Welcome

Eg3:

**public** **class** Demo {

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

{

String str=" Apple color red";

String s1="green";

String s2="pink";

String s3=s1.concat(str).concat(" "+s2);

System.***out***.println(s3);

System.***out***.format("%s %s\n",s1,s2);

String sr=String.*format*("%s %s\n", s1,s2);

System.***out***.println(sr);

}

}

Output: green Apple color red pink

green pink

green pink

* toUpperCase(): it converts string into upper case.

Example:

**public** **class** Demo

{

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

{

String s1="Hello";

String s2= s1.toUpperCase();

System.***out***.println(s2);

}

}

Output: HELLO

* equalsIgnoreCase(): It checks two strings without case sensitivity .

**public** **class** Demo {

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

{

String s1="Welcome";

String s2=s1.toUpperCase();

System.***out***.println(s1.equalsIgnoreCase(s2));

}

}

Output: true

* trim(): it ignore the starting and ending spaces.

Eg: **public** **class** Demo {

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

{

String s1=" Welcome ";

String s2=s1.trim();

System.***out***.println(s1==s2);

System.***out***.println(s1.equals(s2));

System.***out***.println(s1);

System.***out***.println(s2);

}

}

Output:

false

false

Welcome

Welcome

* substring(start index, end index):

Eg:

**public** **class** Demo {

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

{

String str="apple";

String first=str.substring(0,1).toUpperCase();

String second=str.substring(1);

StringBuilder sb=**new** StringBuilder();

System.***out***.println(sb.append(first).append(second));

}

}

Output: Apple

Eg: **public** **class** Demo {

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

{

String str="Apple";

String str1="green";

//index

System.***out***.println(str.indexOf("p"));

System.***out***.println(str1.indexOf("green"));

System.***out***.println("The string is not present:"+str.indexOf("car"));

//lastIndexOf

System.***out***.println(str.lastIndexOf('p'));

//indexOf(char,starting)

System.***out***.println(str.indexOf('p',3));

System.***out***.println(str1.indexOf('r',1));

}

}

Output:

1

0

The string is not present:-1

2

-1

* replace():

Eg: **public** **class** Demo {

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

{

String str="life is a journey enjoy the ride";

System.***out***.println(str.replace("ride", "process and except it"));

System.***out***.println(str.replace('j','S'));

}

}

Output: life is a journey enjoy the process and except it

life is a Sourney enSoy the ride

* isEmpty() & isBlank():

**Eg:**

**public** **class** Demo {

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

{

String str=" ";

System.***out***.println(str.isEmpty());

System.***out***.println(str.isBlank());

}

}

Output: false

True

* strip(): removing white space

Eg:

**public** **class** Demo {

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

{

String str=" java ";

String string="""

i am siri

<html>

<div> </div>

</html>

""";

System.***out***.println(str.stripLeading()); //remove the front space

System.***out***.println(str.stripTrailing());//back side

System.***out***.println(string.stripIndent());

}

}

Output:

java

java

i am siri

<html>

<div> </div>

</html>

Creating our own split method:

**public** **class** Demo {

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

{

String str=" java ";

System.***out***.println(*split*(str)); //it prints by removing spaces

}

**public** **static** String split(String s)

{

**return** s.replace(" ", "");

}

}

Output: java

* charAt():

**public** **class** Demo {

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

{

String str="Dhaanvika";

System.***out***.println(str.charAt(3));

}

}

Output: a

* compareTo() & compareToIgnoreCase():

--comparing string for alphabetical order.

Eg: **public** **class** Demo {

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

{

String str="Apple";

String str1="apple";

System.***out***.println(str.compareTo(str1));

System.***out***.println(str.compareToIgnoreCase(str1));

}

}

Output: -32

0

Creating our own compareToIgnoreCase():

Eg: **public** **class** Demo {

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

{

String str="Apple";

String str1="apple";

System.***out***.println(*compareToIgnoreCase*(str,str1));

}

**public** **static** **int** compareToIgnoreCase(String s1,String s2)

{

String lowers1=s1.toLowerCase();

String lowers2=s2.toLowerCase(); //we can convert it to upper also

**return** lowers1.compareTo(lowers2);

}

}

Output:0

* contains():

Eg: **public** **class** Demo {

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

{

String str="Apple color red";

System.***out***.println(str.contains("red"));

}

}

Output: true

* length():

**public** **class** Demo {

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

{

String str="Apple";

String str1="green";

System.***out***.println(str.length());

**char**[] ch=str.toCharArray();

System.***out***.println(ch[2]);

StringBuilder br=**new** StringBuilder(str.length()+str1.length()+1);

br.append(str).append(" ").append(str1);

System.***out***.println(br);

}

}

Output: 5

p

Apple green

* split():

Eg: **public** **class** Demo {

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

{

String text="""

siri,ch,20,kcl

naga,R,21,hyd

saranya,ch,3,vij

dhaanvi,M,5,hyd

""";

String[] people=text.split("\n");

System.***out***.println("number of people:"+people.length);

System.***out***.println(people[0]);

String[] siri=people[0].split(",");

System.***out***.println(siri.length);

System.***out***.println(siri[0]);

}

}

Output:

number of people:4

siri,ch,20,kcl

4

Siri

* startsWith() & endsWith():

Eg:

**public** **class** Demo {

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

{

String filename=" myfile.txt ".strip();

String file=" file.doc";

System.***out***.println(filename.endsWith("txt"));

System.***out***.println(filename.startsWith("my"));

System.***out***.println(file.strip().startsWith("file"));

}

}

Output:

true

true

true

* contentEquals(): compare the StringBuffer and StringBuilder.

--compare Normal string and StringBuilder strings.

Eg:

**public** **class** Demo {

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

{

String first="Apple";

StringBuilder br=**new** StringBuilder("Apple");

System.***out***.println(first.equals(br));

System.***out***.println(first.contentEquals(br));

}

}

Output: false

true

\*\*NOTE: everything that will be appended at the right side is evaluated as string only. It is evaluated from left to right.

Eg: **public** **static** **void** main(String args[])

{

System.***out***.println("Heloo"+10+20);

}

Output: Hello1020

Eg2: **public** **static** **void** main(String args[])

{

System.***out***.println(10+20+"Heloo");

}

Output: 30Hello

**Substring & indexOf**:

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

Example:

**public** **class** Demo {

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

{

String phone="(234) 333-5551";

String areaCode=*parseAreaCode*(phone);

String exchange=*parseExchange*(phone);

String lineNumber=*parseLineNumber*(phone);

System.***out***.println(areaCode);

System.***out***.println(exchange);

System.***out***.println(lineNumber);

}

**private** **static** String parseLineNumber(String phone) {

**int** spacelast=phone.indexOf("-");

String lineNumber=phone.substring(spacelast+1);

**return** lineNumber;

}

**private** **static** String parseExchange(String phone) {

**int** space=phone.indexOf(" ");

**int** hyphen=phone.indexOf("-");

String exchange=phone.substring(space+1,hyphen);

**return** exchange;

}

**private** **static** String parseAreaCode(String phone) {

**int** openParens=phone.indexOf("(");

**int** closeParens=phone.indexOf(")");

String areaCode=phone.substring(openParens+1,closeParens);

**return** areaCode;

}

}

Output:

234

333

5551

\*\*Difference between the (==) and equals:

==: it means the content and address are also same then only it will return true.

equals: it only checks the content if content is same then it will return true.

Example:

**import** java.util.\*;

**public** **class** Demo {

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

{

String s1="Welcome";

String s2="Welcome";

String s3=**new** String("Welcome");

String s4=**new** String("Hello");

System.***out***.println(s1==s2);

System.***out***.println(s1.equals(s2));

System.***out***.println(s1==s3);

System.***out***.println(s1.equals(s3));

}

}

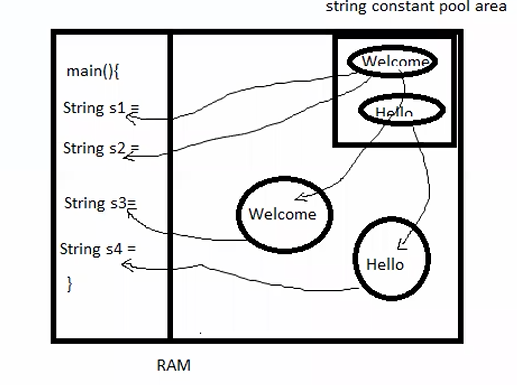
Output:

true

true

false

true



Note: Garbage collector does not have any effect on SCP area.

---SCP area will be sharable in entire application, it is one per JVM.

Eg: String s1="Welcome";

s1=**null**;

--s1 will not give to the garbage collector. It not effect the SCP area.

\*Practical implementation:

**public** **class** Demo {

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

{

String s1="Welcome";

System.***out***.println(System.*identityHashCode*(s1));

s1=**null**;

String s2="Welcome";

System.***out***.println(System.*identityHashCode*(s2));

}

}

Output: 41903949

41903949

Both the addresses are same that means after s1=null the welcome(SCP area) is not erased by garbage collector.

--In case of the both the string variables are from 2 different classes then also it point to the same SCP area.

Eg:

**class** K

{

String s="Welcome";

}

**public** **class** Demo {

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

{

String s1="Welcome";

K a1=**new** K();

System.***out***.println(s1==(a1.s));

}

}

Output: true

\*Here we want to perform modifications on the same object. We use 2 concepts.

---since string object is immutable. we cannot modify an existing object, each modification will return a new Object, but if we want to get immutability, then we should use

---StringBuffer and StringBuilder (it is faster than SB)

\*\*Difference between StringBuffer and StringBuilder:

--- most of the methods of StringBuffer is synchronized i.e thread safe, and give slow performance compare to StringBuilder.

---whereas most of the methods of StringBuilder is non-synchronized, i.e not thread-safe it will give fast performance.

Example: Reversing String

**public** **class** Demo {

**public** **static** String reverseString(String s)

{

StringBuilder sb=**new** StringBuilder("");

**char** chr[]=s.toCharArray();

**for**(**int** i=chr.length-1;i>=0;i--)

sb.append(chr[i]);

**return** sb.toString();

}

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

{

System.***out***.println(*reverseString*("Welcome"));

}

}

Output: emocleW

Example:

**public** **class** Demo {

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

{

String str="Apple color red";

String s1="green";

String s2="pink";

//capacity

String string=**new** StringBuilder(str.length()+s1.length()+1)

.append(str).append(" ")

.append(s1).toString();

System.***out***.println(string);

}

}

Output:

Apple color red green

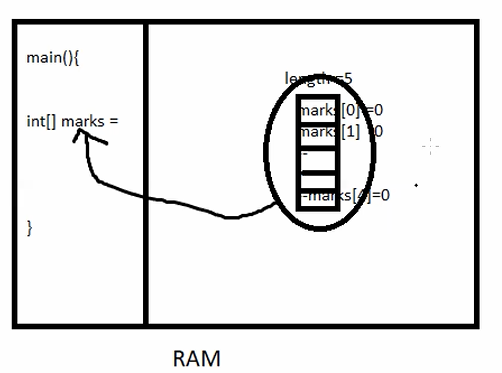
ARRAY:

Array is a collection of homogeneous data elements.

Declaration of array:

Int [] marks=new int [5];

Memory allocation:



Example:

**public** **class** Demo {

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

{

**int**[] arr=**new** **int**[5];

System.***out***.println(arr); //It prints the address of array

System.***out***.println(arr[2]);

System.***out***.println(arr[4]);

System.***out***.println("length of array:"+arr.length);

System.***out***.println(arr[6]);

}

}

Output: [I@27f674d

0

0

length of array:5

Eg: **public** **class** Demo {

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

{

**int**[] arr=**new** **int**[5];

arr[0]=10;

arr[1]=20;

arr[4]=40;

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

{

System.***out***.println(arr[i]);

}

// for(int i:arr)

// {

// System.out.println(i);

// }

}

}

Output: 10

20

0

0

40

Difference:

**int**[] arr;

arr=**new** **int**[5];//valid

**int**[] arr3;

arr3=arr;//valid

**int**[] arr1= {10,20,30,40,50};//valid

**int**[] arr2;

arr2= {10,20,30,40};//Invalid

array object creation:

A[] arr = new A[2]; // arr size 2 it creates 3 variables.

//A a1, a2, a3

1. arr –reference variable of array type object
2. arr[0] --- ref variable A type with default value null.
3. Arr[1] --- ref variable A type with default value=null.

Example:

**class** A

{

**int** x=100;

**void** funA()

{

System.***out***.println("inside funA of A");

}

}

**public** **class** Demo {

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

{

A[] arr=**new** A[3];

arr[0]=**new** A();

arr[1]=**new** A();

arr[2]=**new** A();

**for**(A a:arr) {

a.funA();

}

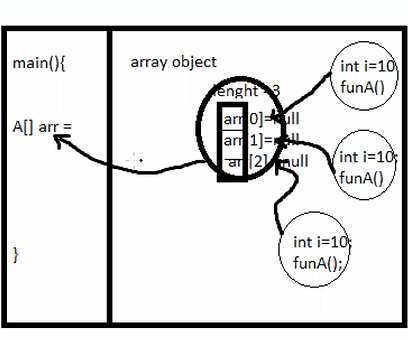
}

}

Output: inside the fuuA of A

inside the fuuA of A

inside the fuuA of A



The simplified form of array object:

A[] arr1= {**new** A(),**new** A(),**new** A(),**new** A()};

**for**(A a:arr1)

{

a.funA();

}

Output: inside the fuuA of A

inside the fuuA of A

inside the fuuA of A

inside the fuuA of A

String Array:

String[] cities= {"vij","hyd","uk"};

**for**(String city:cities)

{

System.***out***.println(city.toUpperCase());

}

Output: VIJ

HYD

UK

Eg:

**public** **class** Demo {

**void** fun(**int**[] arr)

{

**if**(arr != **null**)

{

System.***out***.println("inside fun of Demo");

**for**(**int** i:arr)

{

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

}

}

**else**

System.***out***.println("null is not allowed");

}

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

{

Demo d=**new** Demo();

//int[] ar=new int[3]; // without values it assign default value.

**int**[] ar= {10,20,30,40};

d.fun(ar);

//d.fun(null);

}

}

Output: inside fun of Demo

10

20

30

40

---counting total marks using array:

**public** **class** Demo {

**void** fun(**int**[] arr)

{

**if**(arr !=**null**)

{

System.***out***.println("inside fun of demo");

**int** total=0;

**for**(**int** i:arr)

{

total+=i;

}

System.***out***.println("Total marks:"+total);

}

**else**

System.***out***.println("null is not allowed");

}

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

{

**int**[] arr= {60,80,78,90,59};

Demo d=**new** Demo();

d.fun(arr);

}

}

Output: inside fun of demo

Total marks:367

Student Details:

**import** java.util.\*;

**public** **class** Demo {

**void** fun(Student[] stu)

{

**if**(stu!=**null**)

{

System.***out***.println("inside fun of Demo");

**for**(**int** i=0;i<stu.length;i++)

{

System.***out***.println(stu[i].getRoll());

System.***out***.println(stu[i].getName());

System.***out***.println(stu[i].getMarks());

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

}

}

**else**

System.***out***.println("null is not alowed ");

}

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

{

Demo d=**new** Demo();

Scanner sc=**new** Scanner(System.***in***);

System.***out***.println("Enter number of students:");

**int** n=sc.nextInt();

Student[] stu=**new** Student[n];

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

{

System.***out***.println("Enter roll:");

**int** roll=sc.nextInt();

System.***out***.println("Enter name:");

String name=sc.next();

System.***out***.println("Enter marks:");

**int** marks=sc.nextInt();

//stu[i]=new Student(roll,name,marks);

stu[i]=**new** Student();

stu[i].setRoll(roll);

stu[i].setName(name);

stu[i].setMarks(marks);

}

d.fun(stu);

}

}

Output: Enter number of students:

2

Enter roll:1

101

Enter name:1

SIRI

Enter marks:1

670

Enter roll:2

102

Enter name:2

MADHU

Enter marks:2

960

inside fun of Demo

101

SIRI

670

================

102

MADHU

960

================

Returning array:

---Array as return type

**public** **class** Demo {

Student[] getDetails(String user,String password)

{

**if**(user.equals("admin") && password.equals("123"))

{

// student array of size 3

Student[] stu=**new** Student[3];

//initializing values

stu[0]=**new** Student(101,"name1",678);

stu[1]=**new** Student(102,"name2",778);

stu[2]=**new** Student(103,"name3",878);

**return** stu;

}

**else**

**return** **null**;

}

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

{

Demo d=**new** Demo();

Student[] stu=d.getDetails("admin", "123");

**if**(stu!=**null**)

{

**for**(**int** i=0;i<stu.length;i++) {

System.***out***.println("Roll is:"+stu[i].getRoll());

System.***out***.println("Name is:"+stu[i].getName());

System.***out***.println("Marks is:"+stu[i].getMarks());

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

}

}

**else**

System.***out***.println("invalid return");

}

}

Output:

Roll is:101

Name is:name1

Marks is:678

=================

Roll is:102

Name is:name2

Marks is:778

=================

Roll is:103

Name is:name3

Marks is:878

=================

Two dimensional array:

---multi-dimensional array is treated as array of array.

Eg: **public** **class** Demo {

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

{

**int**[][] arr=**new** **int**[3][2];

System.***out***.println(arr); //[[I@

System.***out***.println(arr.length);//3

System.***out***.println(arr[0]);//[I@

System.***out***.println(arr[0].length);

System.***out***.println(arr[0][1]);//0

}

}

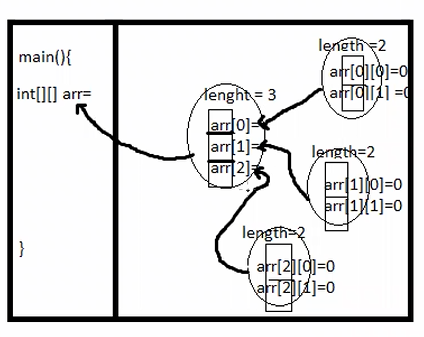
Output: [[I@27f674d

3

[I@49e4cb85

2

0



Example for 2D array:

**public** **class** Demo {

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

{

**int**[][] arr=**new** **int**[4][4];

**for**(**int** i=0;i<arr.length;i++)

{

**for**(**int** j=0;j<arr[i].length;j++)

{

arr[i][j]=j;

}

}

**for**(**int**[] ar:arr)

{

**for**(**int** i:ar)

{

System.***out***.print(i+" ");

}

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

}

}

}

Output:

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

Passing array as parameter:

**public** **class** Demo {

**static** **void** fun(String[] arr)

{

System.***out***.println("inside the fun of Demo");

**for**(String s:arr)

{

System.***out***.println(s.toUpperCase());

}

}

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

{

String[] names= {"siri","naga","dhaanvi"};

//String[] names=new String[0];

//String[] names=null;

//empty string array

//String[] names={};

*fun*(names);

}

}

Output:

inside the fun of Demo

SIRI

NAGA

DHAANVI

Command line arguments:

----JVM will call the main method initially by supporting empty String array object.

----at the time of running our application through the CLA what ever value we will supply, that value will be loaded into that string array object.

----and that argument is called as CLA, which is available to our application in the form of string array.

Eg:

**public** **class** Demo {

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

{

System.***out***.println(args);//

System.***out***.println(args.length);//0

System.***out***.println(args[0]);//error

}

}

Output: [Ljava.lang.String;@27f674d

2

**INHERITENCE**:

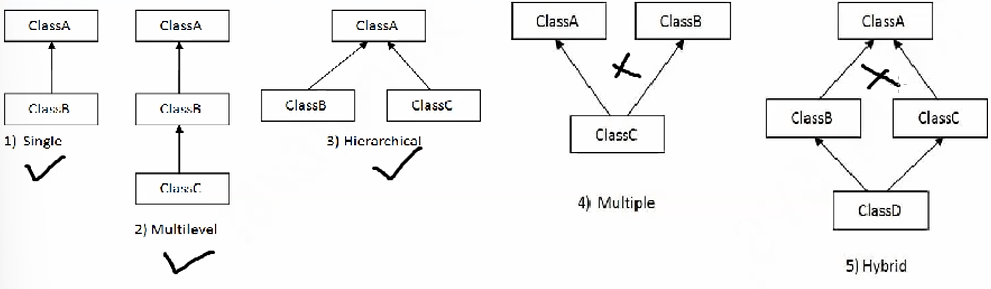
---Getting the properties of one object of a class to another object of another class.

--derived class Acquire the properties from the base class.

\* keyword – extends

--- static members will not participate in inheritance. Even though the child class object we can access the static members of the parent class. These members will be accessed from the context area. Not from the object.

TYPES OF INHERITENCE:



A – parent class or super class or base class

|

B – child class or sub or derived class

Is-A relationship --- >Dog is a - Animal

Advantages:

1. Code reusability
2. Runtime polymorphism

\*code reusability:

Animal

Sleep ()

Walk ()

Eat ()

MakeNoise ()

Dog Cat Tiger Lion

Sleep ()//1000 statements

Walk ()// 2000 statements

Eat ()//3000 …

MakeNoise ()

//we don’t write so many times same methods. All methods are put it in a single parent class.

Example of single inheritance:

**class** A

{

**int** i=100;

**void** funA()

{

System.***out***.println("inside funA of A");

}

}

**public** **class** Demo **extends** A {

**int** x=230;

**void** funX()

{

System.***out***.println("inside the funX of Demo");

}

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

{

Demo d=**new** Demo();

System.***out***.println(d.x);

d.funX();

System.***out***.println(d.i);

d.funA();

}

}

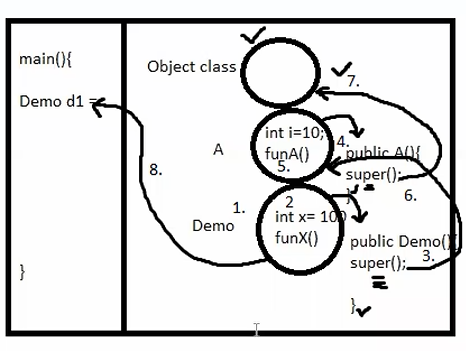
Output: 230

inside the funX of Demo

100

inside funA of A

Implementation:



With respect to diagram:

1. Object of the super class is created first completely before object of the child class created completely.
2. Super class object will be created from the constructor of the sub class. (by using super();)

Demo(){

Super(); //create the parent class object by executing parent class

}

1. Super class object will be created in association with the sub class object.
2. Object class is the super

NOTE: meaning of super() is to create the parent class object by executing parent class zero argument constructor.

---if inside the parent class we have only parameterized constructor then inside the child class we need to call that parent class constructor explicitly by supplying appropriate argument ex: super(10).

Otherwise our inheritance will fail.

A.java

A(**int** m)

{

System.***out***.println("inside A(int m) constructor");

}

A()

{

this(10);

System.***out***.println("inside A constructor");

}

Demo.java

Demo()

{

Super();

System.***out***.println("inside Demo constructor");

}

Output: inside A(int m) constructor

inside A constructor

inside Demo constructor

\*\*\*\*\*\*\* this () and super () cannot be called in single loop body – because the 2 keywords are the first statements in the body. This is only for constructor.

\*\* to call a method using super keyword we can call any where in the loop or body.

---we cannot use this, super inside the static area. ( eg: main method)

---super keyword will represent the immediate parent class object.

---this keyword will represent the current class object.

There are 2 uses of the super keyword:

1. To represent the immediate parent class object.
2. To call the parent class constructor.

1.super().

2.super(), super(10).

METHOD OVERRIDING: Dynamic polymorphism (run time)

--- methods with the same name.

**class** A

{

**int** i=100;

**void** funA()

{

System.***out***.println("inside funA of A");

}

}

**public** **class** Demo **extends** A {

**int** x=230;

**void** funX()

{

System.***out***.println("inside the funX of Demo");

}

@Override // this annotation make sure that we have override correct method.

**void** funA()

{

System.***out***.println("inside funA of Demo");

System.***out***.println("This is second statement inside funA of Demo");

}

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

{

Demo d=**new** Demo();

//System.out.println(d.x);

d.funX();

//System.out.println(d.i);

d.funA();

}

}

Output:

inside the funX of Demo

inside funA of Demo

This is second statement inside funA of Demo

ACCESS MODIFIERS:

Public > default > protected > private

Public:

--this modifier will work on in and outside of the package and class.

Default:

--It will work with in the package and in and outside of the class.

--The default access modifier of any java class member is “default” –No.

--The default access modifier is the absence of a modifier.

\*\* from java 8 onwards, the default keyword is also used to specify that a method in an interface provides the default implementation of an optional method.

Eg: we cannot do this -- XX

**default** **class** Company {

NewsAgency na;

}

Protected: protected data will work only with in the class and package.

Private: It work only with in the class.

\*--private methods are called inside of the public method of its own class.

Eg:

**class** classA{

**public** **void** fun1()

{

System.***out***.println("Inside the public method");

fun2();

}

**private** **void** fun2()

{

System.***out***.println("Inside the private method");

}

**protected** **void** fun3()

{

System.***out***.println("Inside the protected method");

}

**void** fun4()

{

System.***out***.println("Inside the default method");

}

}

**public** **class** Demo **extends** Object{

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

{

classA a=**new** classA();

a.fun1();

//a.fun2(); // here we cannot call the private method- because it works only inside that class only.

// if we want to call the private method. we call that method inside the public method.

a.fun3();

a.fun4(); //default method calling.

}

}

Output:

Inside the public method

Inside the private method

Inside the protected method

Inside the default method

Method overriding rules:

1. While overriding a method inside the child class we cannot reduce its accessibility. But we can increase.
2. Related with the exception handling.

overriding static methods: This concept is called as method hiding.

---if we define same static method inside the child class which is already defined statically inside the parent class, then this concept is known as method hiding, it is not a method overriding, it will hide the parent class static method inside the context area.

Eg:

**class** A

{

**int** i=100;

**static** **void** funA()

{

System.***out***.println("inside funA of A");

}

}

**public** **class** Demo **extends** A {

**int** x=230;

**static** **void** funA()

{

System.***out***.println("inside funA of Demo");

System.***out***.println("This is second statement inside funA of Demo");

}

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

{

Demo d=**new** Demo();

//System.out.println(d.i);

d.*funA*();

//Demo.funA(); //we can also call static methods like this.

//A.funA(); // error- it won’t override the class A method and don’t print the Demo class method.

}

}

Output: inside funA of Demo

This is second statement inside funA of Demo

Super class reference to child class object – up casting is only possible with this.

---according to this rule, we can assign any child class object to its parent class reference variable.

--- NOTE: if a super class reference pointing to the child class object, then with that object we can refer only members of the parent class and, if that method is overridden inside the child class then child class method will gets the priority.

But with the reference we cannot call the child class specific functionality. Will get a compiler error.

A a=new A();

A a=null;

A a=new child(); //demo();

**class** Animal

{

int i=10;

**void** eat()

{

System.***out***.println("I can eat");

}

}

**public** **class** Demo **extends** Animal {

int x=20;

@Override

**void** eat()

{

System.***out***.println("I can eat - Demo");

}

**void** display()

{

System.***out***.println("this is display");

}

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

{

//super class reference to child class object.

Animal a=**new** Demo();//run time polymorphism

a.eat(); //with this 'a' object we can only call the parent class and override methods of child class.

//a.display(); // we cannot call this it is a child class method.

//to write this we use object down casting it is possible only when super class ref points to the child class object.

Demo d=(Demo)a;

d.display();

}

}

Output: I can eat - Demo

this is display

--- in order to call the child class specific functionality from the parent class object, we need to downcast the super class reference to the appropriate child class object.

Above example: Demo d=(Demo)a;

Eg2:

**class** A

{

**int** i=100;

**void** funA()

{

System.***out***.println("inside funA of A");

}

**void** funB()

{

System.***out***.println("inside funB of A");

}

}

**public** **class** Demo **extends** A {

**int** x=230;

@Override

**void** funA()

{

System.***out***.println("inside funA of Demo");

System.***out***.println("This is second statement inside funA of Demo");

}

**void** funX()

{

System.***out***.println("inside funX of Demo");

}

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

{

A a=**new** Demo();

a.funA();

a.funB();

// we cannot access the child class member which is not a member of parent class.

//we use down casting

Demo d=(Demo)a;

d.funX();

}

}

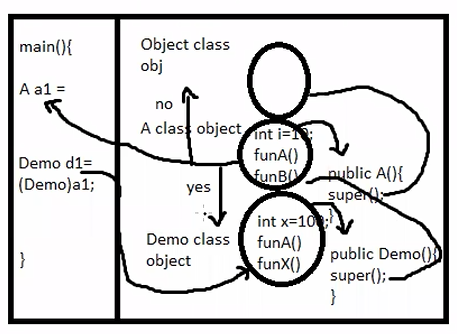
Output: inside funA of Demo

This is second statement inside funA of Demo

inside funB of A

inside funX of Demo

Explanation:



The full example of inheritance concept:

**class** LgOldTv

{

**public** **void** start()

{

System.***out***.println("start TV inside LgOldTv");

}

**public** **void** increaseVolume()

{

System.***out***.println("increaseVolume inside LgOldTv class");

}

}

**class** LgSmartTv **extends** LgOldTv

{

**public** **void** playGame()

{

System.***out***.println("playGame inside the LgSmartTv");

}

@Override

**public** **void** increaseVolume()

{

System.***out***.println("inside increaseVolume of LgSmartTv");

}

**public** **void** changeChannel()

{

System.***out***.println("changeChannel inside LgSmartTv");

}

}

**public** **class** Demo{

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

{

//creating object to LgOldTv

LgOldTv lot=**new** LgOldTv();

//with this we only call parent class methods only

lot.start();

lot.increaseVolume();

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

//creating object to LgSmartTv

LgSmartTv lst=**new** LgSmartTv();

//with this we can call parent & child class methods

lot.start();

lst.changeChannel();

lst.increaseVolume();

lst.playGame();

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

//creating object to super class reference to child class object

LgOldTv st=**new** LgSmartTv();

//with this object i can directly call the LgOldTv methods and Override methods in child class.

st.start();

st.increaseVolume();//override method of child class

//if we want to call child class methods. we use object down casting.

LgSmartTv lg=(LgSmartTv)st;

lg.changeChannel();

lg.playGame();

}

}

Output:

start TV inside LgOldTv

increaseVolume inside LgOldTv class

================================

start TV inside LgOldTv

changeChannel inside LgSmartTv

inside increaseVolume of LgSmartTv

playGame inside the LgSmartTv

=================================

start TV inside LgOldTv

inside increaseVolume of LgSmartTv

changeChannel inside LgSmartTv

playGame inside the LgSmartTv

Example1: object passing and object returning

Object passing as argument

**class** A

{

}

**public** **class** Demo{

**public** **void** fun(Object obj)

{

System.***out***.println("inside fun of demo");

}

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

{

Demo d=**new** Demo();

d.fun(**new** A());

d.fun(**null**);

}

}

Output: inside fun of demo

inside fun of demo

Example2: return type as object

**class** A

{

**void** funA()

{

System.***out***.println("inside funA of A");

}

}

**class** B{

**void** funB()

{

System.***out***.println("inside funB of A");

}

}

**public** **class** Demo{

**public** Object fun(**int** x)

{

System.***out***.println("inside fun of demo");

**if**(x>10)

**return** **new** B();

**else**

**return** **new** A();

}

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

{

Demo d=**new** Demo();

//A a=d.fun(); //compile time error

Object obj=d.fun(23);

B b=(B)obj; // we can do down casting for A and B also

b.funB();

}

}

Output:

inside fun of demo

inside funB of A

Example3:

**class** A

{

**void** funA()

{

System.***out***.println("inside funA of A");

}

}

**class** B{

**void** funB()

{

System.***out***.println("inside funB of A");

}

}

**public** **class** Demo{

**public** Object fun(**int** x)

{

System.***out***.println("inside fun of demo");

**if**(x>10)

**return** **new** B();

**else**

**return** **new** A();

}

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

{

Demo d=**new** Demo();

//A a=d.fun(); //compile time error

Object obj=d.fun(5);

//we can down cast in a single line also

// A a=(A)d.fun(6);

**if**(obj **instanceof** B) {

//The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

B b=(B)obj;

b.funB();

}

**else** {

A b=(A)obj;

b.funA();

}

}

}

Output:

inside fun of demo

inside funA of A

toString():

---toString() method belongs to Object class.

---java.lang.Object

Note: Object class method are also called as universal method, we can call these methods on any class objects.

Full form: public String toString()

---the above functionality is written inside the toString(0 method of the object class.

Default working of toString():

---if we call this method on any object then it will convert that object address into the string and return that string.

Eg:

**public** **class** Demo{

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

{

Demo d=**new** Demo();

String s=d.toString();

System.***out***.println(s); // string

System.***out***.println(d);

}

}

Output:

com.mic.Demo@27f674d

com.mic.Demo@27f674d

user created toString() method:

Eg1:

**public** **class** Demo {

**public** String toString()

{

**return** "siri";

}

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

{

Demo d=**new** Demo();

System.***out***.println(d);

}

}

Output: siri

Eg2:In below example we can print our own string instead of address of object and we can also override the string.

**class** Kab

{

**public** String toString()

{

**return** "hi";

}

}

**public** **class** Demo **extends** Kab{

@Override

**public** String toString()

{

**return** "siri";

}

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

{

Kab f=**new** Demo();

//Demo d=new Demo();

System.***out***.println(f);

}

}

Output:

siri

Eg2:

**public** **class** Demo{

// by default this method extends the object class.

//that's why it is override

@Override

**public** String toString() {

**return** "welcome to my world";

}

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

{

Demo d=**new** Demo();

String s=d.toString();

System.***out***.println(s); // string

System.***out***.println(d);

}

}

Output:

welcome to my world

welcome to my world

---toString() is called by the concept of dynamic polymorphism.

---which method will be called decided at runtime it is also known as late binding.

Early binding vs late binding:

Binding: connecting the method body with method call is known as binding.

--if it is decided at compile time then it is known as early binding (method overloading or static polymorphism).

-- if it is decided at run time then it is known as late binding (method overriding or dynamic polymorphism).

\*\*\*\*\*\*In any class we want to print user given data instead of object address of a variable.

Example:

**class** A

{

**int** roll;

String name;

A(**int** r,String n)

{

roll=r;

name=n;

}

**public** String toString()

{

//return "object of classA";

//return name; //o/p: siri

//return roll+""; //roll is a integer but we return string so we added "" o/p:10

**return** "["+roll+","+name+"]"; //o/p: [10,siri]

}

}

**public** **class** Demo{

@Override

**public** String toString() {

**return** "welcome to my world";

}

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

{

A a=**new** A(10,"siri");

System.***out***.println(a);

System.***out***.println(a);//It will print address. instead of printing address we can print user defined data also.

}

}

Output:

[10, siri]

[10, siri]

FINAL KEYWORD:

---If we define a variable as final it is constant which we cannot modify or reassign.

Example:

**public** **class** Demo{

**final** **int** x=10;

**final** **int** y;

Demo(**int** y)

{

**this**.y=y;

}

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

{

Demo d=**new** Demo(20);

}

}

---if we define a method as final then we cannot override that method.

**public** **final** **void** fun()

{

}

---if we define a class as final then we cannot inherit that class.

**public** **final** **class** Demo{

}

---String class is a final class.

**Access Modifiers**:

---if specifies the accessibility of a class or its members outside that class or package.

We have 4 types

1. Public: accessibility is global (other package or other classes also)
2. Default: it is similar to public as long as inside the same package, but outside the package it works as private.
3. Protected: it is similar to default, but outside the package we can access it by using inheritance.
4. Private: the accessibility is restricted to that class only.

NOTE:

1.A normal class/ outer-class can only be either default or public but that class members (variables, methods, constructors, inner classes) can be private, public, default, protected.

2.We cannot apply access modifiers to the local variables.

---default constructor given by java compiler is always public.

---if we make constructor of a class a default then we cannot create its object or extends this class outside that package.

---if we make constructor of a class a private

**Varargs in java**:

--varargs: Variable Arguments (Varargs) in Java is a method that takes a variable number of arguments.

--we don’t that how many parameters taken by our method in that case we use varargs.

Eg:

**public** **class** Demo{

**static** **void** fun(**int**... i)

{

System.***out***.println("inside fun of Demo");

**for**(**int** a:i) {

System.***out***.println(a);

}

}

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

{

*fun*(10,20,30,40,50);

}

}

Output:

inside fun of Demo

10

20

30

40

50

Eg2:

**public** **class** Demo{

**void** fun(**int** s,**int**... k) {

System.***out***.println("siri:"+s);

**for**(**int** i:k)

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

}

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

{

Demo d=**new** Demo();

d.fun(10,30,40);

}

}

Output:

siri:10

30

40

---we cannot pass vararg argument as first argument in a list of arguments passing to a method.

Eg: **void** fun(**int**... k,**int** s);

Rules to create varargs:

1.If we want to declare any other parameters along with the var-args then this var arg parameter must be last parameter.

2.Inside a single method we cannot have 2 var-args as a parameter.

3.If multiple overloaded methods are there then var-args gets the least priority.

Example:

**public** **class** Varargs {

**public** **static** **int** add(**int**... a)

{

System.***out***.println("inside var-arg method");

**int** result=0;

**for**(**int** i:a)

{

result+=i;

}

**return** result;

}

**public** **static** **int** add(**int** x,**int** y)

{

System.***out***.println("inside single argument add method");

**return** x;

}

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

System.***out***.println(*add*());

System.***out***.println(*add*(10,20));

**int**[] arr= {10,20,30};

System.***out***.println(*add*(arr));

**int**[] arr1=**new** **int**[]{1,2,3};

System.***out***.println(*add*(arr1));

System.***out***.println(*add*(10));

System.***out***.println(*add*(10,20));

}

}

Output: inside var-arg method

0

inside single argument add method

10

inside var-arg method

60

inside var-arg method

6

inside var-arg method

10

inside single argument add method

10

4.Along with var-args we cannot define another overloaded method which takes array of the same parameter. It will become duplicate method definition.

Example:

**public** **static** **int** add(**int**... a)

{

}

**public** **static** **int** add(**int** z,**int**... b)

{

}

//It is not possible. Error

**public** **class** Varargs {

**public** **static** **void** fun(**int**... i)

{

}

**public** **static** **void** fun(**int** j,**int**... k)

{

}

**public** **static** **void** fun(**int** a,**int** b,**int**... c)

{

}

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

*fun*(10,20,30);//ambiguous for type var-args(error)

}

}

**Enum in java**:

---with the help of enum, we can create our own data type (enumerated type) which will going to contains some fixed set of constants.

Declaring enum: either inside a class or as external file.

Java compiler will generate the .class file for an enum also.

---we can declare the main method also inside an enum and we can run a enum file as a class also.

--every enum is internally implemented by using class concept.

--every enum constant represents an object of type enum.

 enum constants are public, static and final

Internal enum:

**public** **class** Demo{

//inside class

**enum** Color{

***green***,***yellow***,***red***,***pink***;

}

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

{

Color c=Color.***green***;

}

}

Values() and ordinal() method:

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

--Every enum implicitly contains values() method that returns all the values presents inside an enum.

--values() method will return all the values in the form of array. This value() method is a static method which we can call on any enum.

--with in the enum every constant placed based on the order, and we can find the ordinal values of enum constant by using ordinal() method.

--this ordinal value are zero bases index value.

--ordinal() method is non-static method.

Eg: Internal structure of enum class

Public final class Month extends Enum {

Public static final JAN=new Month();

Public static final FEB=new Month();

Public static final MAR=new Month();

Public static final APR=new Month();

Public static final MAY=new Month();

}

Eg:

**public** **enum** Month {

***JAN***,***FEB***,***MAR***,***APR***,***MAY***;

}

Class

**public** **class** EnumEx {

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

{

Month m1=Month.***APR***;

System.***out***.println(m1);

Month[] month=Month.*values*();

**for**(Month m:month)

{

System.***out***.println(m+"====="+m.ordinal());

}

}

}

Output:

JAN=====0

FEB=====1

MAR=====2

APR=====3

MAY=====4

Constructor in enum:

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

--we can place constructor also inside an enum, that constructor can only be private whether we mention it or not.

--a constructor of an enum will be executed separately for every enum constant at the time of enum class loaded into the memory.

Example: enum ( parameterized and zero argument constructor)

**public** **enum** Month {

***JAN***,***FEB***(28),***MAR***(31),***APR***,***MAY***(31);

**private** Month() {

System.***out***.println("inside constructor");

}

**private** Month(**int** num)

{

System.***out***.println("inside constructor of Month(int num)");

}

}

Class:

**public** **class** EnumEx {

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

{

Month m1=Month.***APR***; //Month is a enum class loaded into the memory.

}

}

Output:

inside constructor

inside constructor of Month(int num)

inside constructor of Month(int num)

inside constructor

inside constructor of Month(int num)

--enum ordinal, enum fields, enum ValueOf

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

Card.java

**package** com.example.cards;

**public** **class** Card {

**private** Suit suit;

**private** Rank rank;

**public** Card(Suit suit, Rank rank) {

**this**.suit = suit;

**this**.rank = rank;

}

**public** Card(String suit, String rank) {

**this**.suit=Suit.valueOf(suit.toUpperCase());

**this**.rank=Rank.valueOf(rank.toUpperCase());

}

**public** Suit getSuit() {

**return** suit;

}

**public** **void** setSuit(Suit suit) {

**this**.suit = suit;

}

**public** Rank getRank() {

**return** rank;

}

**public** **void** setRank(Rank rank) {

**this**.rank = rank;

}

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

Card c1=**new** Card(Suit.DIAMONDS,Rank.FIVE);

Card c2=**new** Card(Suit.HEARTS,Rank.EIGHT);

Card c3=**new** Card(Suit.CLUBS,Rank.FOUR);

Card c4=**new** Card("spades","queen");

System.***out***.println(c1);

System.***out***.println(c2);

System.***out***.println(c3);

System.***out***.println(c4);

}

@Override

**public** String toString() {

**return** **this**.rank.toString()+" "+**this**.suit.toString();

}

**public** **int** getValue() {

**return** **this**.rank.getValue();

//ordinal(): Returns the ordinal of this enumeration constant (its position in its enum declaration, where the initial constant is assigned an ordinal of zero).

}

}

Output:

FIVE ♦

EIGHT ♥

FOUR ♣

QUEEN ♠

CardTest.java

**package** com.example.cards;

**import** org.junit.jupiter.api.Test;

**import** **static** org.junit.jupiter.api.Assertions.assertEquals;

**public** **class** CardTest {

@Test

**void** canGetValueOfTwoCard() {

Card card1=**new** Card(Suit.DIAMONDS,Rank.TWO);

assertEquals(2,card1.getValue());

}

@Test

**void** canGetValueOfThreeCard() {

Card card1=**new** Card(Suit.DIAMONDS,Rank.THREE);

assertEquals(3,card1.getValue());

}

@Test

**void** canGetValueOfAceCard() {

Card card1=**new** Card(Suit.DIAMONDS,Rank.ACE);

assertEquals(1,card1.getValue());

}

@Test

**void** canGetValueOfJackCard() {

Card card1=**new** Card(Suit.DIAMONDS,Rank.JACK);

assertEquals(10,card1.getValue());

}

@Test

**void** canGetValueOfQueenCard() {

Card card1=**new** Card(Suit.DIAMONDS,Rank.JACK);

assertEquals(10,card1.getValue());

}

@Test

**void** canGetValueOfKingCard() {

Card card1=**new** Card(Suit.DIAMONDS,Rank.KING);

assertEquals(10,card1.getValue());

}

}

Enum—Suit.java

**package** com.example.cards;

**public** **enum** Suit {

***DIAMONDS***('\u2666'),//Unicode symbols

***HEARTS***('\u2665'),

***CLUBS***('\u2663'),

***SPADES***('\u2660');

**private** **char** symbol;

Suit(**char** symbol) {

**this**.symbol = symbol;

}

@Override

**public** String toString() {

**return** Character.*toString*(**this**.symbol);

}

}

SuitTest.java

**package** com.example.cards;

**import** org.junit.jupiter.api.Test;

**import** **static** org.junit.jupiter.api.Assertions.\*;

**class** SuitTest {

@Test

**void** heartPrintsHeart() {

assertEquals("\u2665",Suit.HEARTS.toString());

}

}

Enum—Rank.java

**package** com.example.cards;

**public** **enum** Rank {

***ACE***(1),

***TWO***(2),

***THREE***(3),

***FOUR***(4),

***FIVE***(5),

***SIX***(6),

***SEVEN***(7),

***EIGHT***(8),

***NINE***(9),

***TEN***(10),

***JACK***(10),

***QUEEN***(10),

***KING***(10);

**private** **int** value;

Rank(**int** value) {

**this**.value=value;

}

//Rank Jack=new Rank("JACK");

//Rank QUEEN=new Rank("QUEEN");

//Rank KING=new Rank("KING");

//assertEquals(10,Rank.KING.getValue());

**public** **int** getValue() {

**return** value;

// return switch (this) {

// case JACK, KING, QUEEN -> 10;

// default -> this.ordinal() + 1;

// };

//ordinal(): Returns the ordinal of this enumeration constant (its position in its enum declaration, where the initial constant is assigned an ordinal of zero).

}

}

RankTest.java

**package** com.example.cards;

**import** org.junit.jupiter.api.Test;

**import** **static** org.junit.jupiter.api.Assertions.\*;

**class** RankTest {

@Test

**void** getValueOfKing() {

//Rank Jack=new Rank("JACK");

//Rank QUEEN=new Rank("QUEEN");

//Rank KING=new Rank("KING");

assertEquals(10,Rank.KING.getValue());

}

}

PACKAGE:

There are 3 uses of the packages in java:

1.With the package, we can bind the related concept logically.

com.masai.db:

100: --with intract with the DB

com.masai.service:

200: --in which we define business logic to process the data

com.masai.presentation:

200: --in which we define the presentation logic.

Packages:

Java.lang: it contains all the common classes (eg: String, system, Object etc).

Java.io: contains all the classes and interface which are required to perform i/o operations.

Java.sql:

Java.util:

Java.net:

2.with the help of package we can avoid the naming conflict.

String

Java.lang.String --- fully qualified name

com.masai.String --- user created package

3.to provide some level of security, by using access modifiers.

--Note: in real time we should not define a class, interface without a package.

--in java every package is a folder, where as every folder is not a package.

--when we keep any classes inside any package, then we can compile that class by following 2 ways:

1. manually: here first we need to compile the class and then we need to create folder with the name of the package and keep all the .class files inside that folder. After that we can run our application by using fully qualified command.

> java p1.Demo

2. Automatically: here we need to compile our class by using following command:

> javac -d . Demo.java: this command will create a folder with the name of packages automatically.

**ABSTRACT CLASS**:

---abstract class

---abstract method

--by using an abstract class we achieve partial abstraction in java.

Note: an abstract class does not have meaning, unless it is extended by the child class, and we are allowed to create only child class object.

---In abstract class we can have abstract methods and normal methods also.

--what ever we done with the normal class we can also do with abstract class also.

Difference b/w normal class & abstract class:

1. For an abstract class we cannot create its object directly, (new keyword is not applicable)
2. Inside an abstract class, we may have an abstract method also. (it is not mandatory)

Note: we can have an empty abstract class also

1. Final keyword is not applicable with the abstract class (final and abstract both are enimies)

Abstract Method:

---method with body is also known as implemented method or concreate method.

---method without body is known as abstract method or unimplemented method. These types of methods should have abstract keyword.

Eg: public abstract void fun();

---illegal combinations

Final – abtrsct

Abstract – static

Abstract - private

Note: inside a normal class/ concreate class, we cannot have an abstract method.

---if we want to place the abstract method then we make class as abstract.

**public** **abstract** **class** Demo{

//without body

**public** **abstract** **void** fun();

//with body

**void** funDemo()

{

System.***out***.println("inside funDemo of Demo");

}

}

--if we are extending any abstract class then the extended class or child is also abstract. Or otherwise we override all the methods of abstract class inside the child class.

**abstract** **class** A

{

}

**public** **abstract** **class** Demo **extends** A{

//without body

**public** **abstract** **void** fun();

//with body

**void** funDemo()

{

System.***out***.println("inside funDemo of Demo");

}

}

\*\*\*\*\*\*\*\*\*\*\*\*

The only way to create object to abstract class is super class reference & child class object.

Parent p=new Child();

A.java

--------

**public** **abstract** **class** A {

**public** A() {

System.***out***.println("inside the constructor of A");

}

**public** **abstract** **void** funAbs();

**public** **abstract** **void** funAbs1();

**void** funA()

{

System.***out***.println("inside funA of A");

}

}

AChild.java

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

**public** **class** AChild **extends** A {

AChild()

{

System.***out***.println("inside constructor of Achild");

}

@Override

**public** **void** funAbs() {

System.***out***.println("inside abstract funAbs of AChild");

}

@Override

**public** **void** funAbs1() {

System.***out***.println("inside abstract funAbs1 of AChild");

}

}

Demo.java

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

**public** **class** Demo{

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

{

A a=**new** AChild();

a.funA();

a.funAbs();

a.funAbs1();

}

}

Output:

inside the constructor of A

inside constructor of Achild

inside funA of A

inside abstract funAbs of AChild

inside abstract funAbs1 of AChild

purpose of abstract class:

--abstract class: nobody cannot create object to the abstract class.

--abstract method: to hide the implementation details of methods. Abstract methods cannot be static.

Note: A static method belongs to class not to object instance thus it cannot be overridden or implemented in a child class.

Abstract animal

Eat()

Sleep()

Walk()

makeNoise()

child classes:

Dog Cat Tiger Lion

--these all have differently makeNoise(). Then we put that makeNoise() as abstract and use in all child classes according to there sounds.

**INTERFACE**:

--in java we have 3 types of valid structures

1. normal or concreate class (fully implemented structure)

---A method with body

2. abstract class (partial implemented structure)

3. interface (fully unimplemented structure) & 100% abstraction

---java compiler generates .class file for all the above valid structure.

---with the help of an interface we achieve 100% abstraction in java.

\*\*An interface contains only abstract methods and constant variables (public static final).

---from java-8 onwards an interface can contain some special kinds of method, which will have body also.

---we create an interface by using “interface” keyword.

Example:

Interface Intr

{

int x=10; // it will public static final

void funX(); // it will become public and abstract

void funY();

}

---as we extends a class inside another class, we implements an interface inside another class.

---if a class is implementing an interface, then that class must override all the abstract methods present inside that interface otherwise we need to mark that implementation class as an abstract class.

Eg:

**interface** Intr

{

**int** ***x***=10;

**void** funX();

**void** funY();

**default** **void** funZ()

{

System.***out***.println("inside default funZ of Intr");

}

**static** **void** funI()

{

System.***out***.println("inside static funI of Intr");

}

}

**public** **class** Demo **implements** Intr{

@Override

**public** **void** funX() {

System.***out***.println("inside funX of Demo");

}

@Override

**public** **void** funY() {

System.***out***.println("inside funY of Demo");

}

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

{

Intr it=**new** Demo();

it.funX();

it.funY();

Intr.*funI*(); // static method calling

}

}

Output:

inside funX of Demo

inside funY of Demo

inside static funI of Intr

---from jdk 1.8 onwards we can place method with body also inside an interface (default method or static method).

\*\*\*Note: with the help of an interface we achieve loose coupling in java.

---inside an interface if we place any method without body, that method will be public and abstract whether we mention it or not.

Eg:

**public** **interface** X {

**public** **abstract** **void** fun();

**void** fun1();

}

class A{

}

A.java ------> compile ------> A.class (here we have default constructor)

X.java ------> compile ------> X.class

--constructor concept is not applicable with an interface.

--as a class is extended by another class, an interface needs to be implemented by another class.

--if a class implements an interface, then that class has to override all the abstract method defined inside that interface otherwise we need

X.java

--------

**public** **interface** X {

**void** fun();

**void** fun1();

}

XImple.java

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

**public** **class** XImple **implements** X {

@Override

**public** **void** fun() {

System.***out***.println("inside fun of XImple");

}

@Override

**public** **void** fun1() {

System.***out***.println("inside fun1 of XImple");

}

//specific method

**public** **void** fun2()

{

System.***out***.println("inside fun2 of XImple");

}

}

Note: we cannot create object of an interface. But we can define a reference variable for an interface.

X a=new X(); // we cannot create object to interface—compiler error

X a=? //2 possible value

1. X a=new XImple();
2. X a=null;

Note: we can define variable of any 3 valid structure like (concrete class, abstract class or an interface) but the object should be created only for the concrete class.

Demo.java

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

**public** **class** Demo{

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

{

X x=**new** XImple();

x.fun();

x.fun1();

XImple xi=(XImple)x; //down casting

xi.fun2();

xi.fun1();

//implementation class

XImple xx=**new** XImple();

xx.fun();

xx.fun1();

xx.fun2();

}

}

Output:

inside fun of XImple

inside fun1 of XImple

inside fun2 of XImple

inside fun1 of XImple

inside fun of XImple

inside fun1 of XImple

inside fun2 of XImple

---inside an interface, in addition to an abstract method, we can have variables also.

---if we define any variable inside an interface, it will be by default “public static final” whether we mention it or not.

---the variable must be initialized at time of declaration.

---variable defined inside an interface can be accessed by the implemented class object also.

--we cannot modify the variable values, because the variables are default final.

Example:

**public** **interface** X {

**int** ***i***=20;

**void** fun();

**void** fun1();

}

**public** **class** Demo{

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

{

X x=**new** XImple();

x.fun();

x.fun1();

XImple xi=(XImple)x; //down casting

//implementation class

XImple xx=**new** XImple();

System.***out***.println(X.***i***);

System.***out***.println(xx.***i***);

}

}

Output:

20

20

20

Interface as a method parameter:

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

\*\*\*\*one interface can be implemented in multiple classes.

---if a method is defined to take an interface, then we can call that method by supplying any of its implemented class object or null.

Eg:

**public** **interface** X {

**int** ***i***=20;

**void** fun();

**void** fun1();

}

**public** **class** XImple **implements** X {

@Override

**public** **void** fun() {

System.***out***.println("inside fun of XImple");

}

@Override

**public** **void** fun1() {

System.***out***.println("inside fun1 of XImple");

}

//specific method

**public** **void** fun2()

{

System.***out***.println("inside fun2 of XImple");

}

}

**public** **class** AnotherImple **implements** X {

@Override

**public** **void** fun() {

System.***out***.println("inside fun of AnotherImple");

}

@Override

**public** **void** fun1() {

System.***out***.println("inside fun1 of AnotherImple");

}

}

**public** **class** Demo{

**public** **void** funDemo(X x)

{

**if**(x!=**null**)

{

System.***out***.println("inside funDemo of Demo");

x.fun();

x.fun1();

}

**else**

System.***out***.println("null not allowed");

}

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

{

Demo d=**new** Demo();

//XImple x=new XImple();

//d.funDemo(x);

//d.funDemo(null);

d.funDemo(**new** XImple());

d.funDemo(**new** AnotherImple());

}

}

Output:

inside funDemo of Demo

inside fun of XImple

inside fun1 of XImple

inside funDemo of Demo

inside fun of AnotherImple

inside fun1 of AnotherImple

Interface as a method return type:

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

Eg:

**public** **interface** X {

**int** ***i***=20;

**void** fun();

**void** fun1();

}

**public** **class** XImple **implements** X {

@Override

**public** **void** fun() {

System.***out***.println("inside fun of XImple");

}

@Override

**public** **void** fun1() {

System.***out***.println("inside fun1 of XImple");

}

//specific method

**public** **void** fun2()

{

System.***out***.println("inside fun2 of XImple");

}

}

**public** **class** AnotherImple **implements** X {

@Override

**public** **void** fun() {

System.***out***.println("inside fun of AnotherImple");

}

@Override

**public** **void** fun1() {

System.***out***.println("inside fun1 of AnotherImple");

}

}

**public** **class** Demo{

**public** X funDemo()

{

System.***out***.println("inside funDemo of Demo");

**return** **new** XImple();

//return new AnotherImple();

//return null;

}

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

{

Demo d=**new** Demo();

// X x=d.funDemo();

// if(x!=null)

// {

// x.fun();

// x.fun1();

// }

// else

// System.out.println("null not allowed");

Object obj=d.funDemo();

//first level down casting

X x=(X)obj;

x.fun();

x.fun1();

//second level down casting

XImple xx=(XImple)x;

xx.fun();

xx.fun1();

xx.fun2();

//directly down casting obj to XImple object

XImple x1=(XImple)obj;

x1.fun();

x1.fun1();

x1.fun2();

}

}

Output:

inside funDemo of Demo

inside fun of XImple

inside fun1 of XImple

inside fun of XImple

inside fun1 of XImple

inside fun2 of XImple

inside fun of XImple

inside fun1 of XImple

inside fun2 of XImple

---class is a blue print of object, where as interface is also like blueprint of class.

Example: using interface

Hotel.java

**public** **interface** Hotel {

**void** chickenBiryani();

**void** masalaDosa();

}

TajHotel.java

**public** **class** TajHotel **implements** Hotel {

@Override

**public** **void** chickenBiryani() {

System.***out***.println("inside the chickenBiryani of TajHotel");

}

@Override

**public** **void** masalaDosa() {

System.***out***.println("inside the masalaDosa of TajHotel");

}

**public** **void** paneerMasalaDosa()

{

System.***out***.println("inside the paneerMasalaDosa of TajHotel");

}

}

RoadSideHotel.java

**public** **class** RoadSideHotel **implements** Hotel {

@Override

**public** **void** chickenBiryani() {

System.***out***.println("inside the chickenBiryani of RoadSideHotel");

}

@Override

**public** **void** masalaDosa() {

System.***out***.println("inside the masalaDosa of RoadSideHotel");

}

}

Demo.java

**public** **class** Demo{

//return type interface

**public** Hotel provideFood(**int** amount)

{

Hotel hotel=**null**;

**if**(amount>500)

{

hotel=**new** TajHotel();

}

**else** **if**(amount>200 && amount<=500)

{

hotel=**new** RoadSideHotel();

}

**return** hotel;

}

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

{

Demo d=**new** Demo();

Hotel h=d.provideFood(600);

**if**(h!=**null**) {

h.chickenBiryani();

h.masalaDosa();

**if**(h **instanceof** TajHotel)

{

TajHotel taj=(TajHotel)h;

taj.paneerMasalaDosa();

}

}

**else**

{

System.***out***.println("null not allowed");

}

}

}

Output:

inside the chickenBiryani of TajHotel

inside the masalaDosa of TajHotel

inside the paneerMasalaDosa of TajHotel

\*\*\*\*\*\*\*\*

---interface cannot extends another class and cannot implements any interface.

---but one interface can extend more than one interface simultaneously.

\*\*\*\*\*\*\*\*

Extends Interface:

Eg:

**interface** Intr1

{

**void** funA();

**void** funB();

}

**interface** Intr2

{

**void** funC();

**void** funD();

}

**interface** Intr3 **extends** Intr1,Intr2

{

**void** funX();

**void** funY();

}

**public** **class** Demo **implements** Intr3{

@Override

**public** **void** funA() {

System.***out***.println("inside funA of Imple class");

}

@Override

**public** **void** funB() {

System.***out***.println("inside funB of Imple class");

}

@Override

**public** **void** funC() {

System.***out***.println("inside funC of Imple class");

}

@Override

**public** **void** funD() {

System.***out***.println("inside funD of Imple class");

}

@Override

**public** **void** funX() {

System.***out***.println("inside funX of Imple class");

}

@Override

**public** **void** funY() {

System.***out***.println("inside funY of Imple class");

}

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

{

Demo d=**new** Demo();

d.funA();

d.funB();

d.funC();

d.funD();

d.funX();

d.funY();

//object to intr1

Intr1 i1=**new** Demo();

//we can call only A B functions

i1.funA();

i1.funB();

//object to intr2

Intr2 i2=**new** Demo();

i2.funC();

i2.funD();

//object to intr3

Intr3 i3=**new** Demo();

i3.funA();

i3.funB();

i3.funC();

i3.funD();

i3.funX();

i3.funY();

}

}

\*\*java 1.8 onwards some new feature introduced in interface:

1.Default method

2.Static method

--both methods should have body.

1. default method:

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

--we can define a default method with the body inside an interface.

--this default method need not override inside the implementation classes.

--If we want, we can override this default method inside any implementation classes.

--these(interface) default methods are by default inherited inside the implementation classes.

--we can call these(interface) default method from any implementation class object.

Eg:

Interface

**public** **interface** Hotel {

**void** chickenBiryani();

**void** masalaDosa();

//default method

**default** **void** iceCream(){

System.***out***.println("inside the iceCream of Hotel Interface");

}

}

TajHotel.java

**public** **class** TajHotel **implements** Hotel {

@Override

**public** **void** chickenBiryani() {

System.***out***.println("inside the chickenBiryani of TajHotel");

}

@Override

**public** **void** masalaDosa() {

System.***out***.println("inside the masalaDosa of TajHotel");

}

//specified method

**public** **void** paneerMasalaDosa()

{

System.***out***.println("inside the paneerMasalaDosa of TajHotel");

}

@Override //override the default interface method

**public** **void** iceCream() {

//Hotel.super.iceCream();

System.***out***.println("inside the iceCream from TajHotel");

}

}

RoadSideHotel.java

**public** **class** RoadSideHotel **implements** Hotel {

@Override

**public** **void** chickenBiryani() {

System.***out***.println("inside the chickenBiryani of RoadSideHotel");

}

@Override

**public** **void** masalaDosa() {

System.***out***.println("inside the masalaDosa of RoadSideHotel");

}

}

Demo.java

**public** **class** Demo{

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

{

Hotel h=**new** TajHotel();

h.chickenBiryani();

h.iceCream();

Hotel hr=**new** RoadSideHotel();

hr.chickenBiryani();

hr.iceCream();

}

}

Output:

inside the chickenBiryani of TajHotel

inside the iceCream from TajHotel

inside the chickenBiryani of RoadSideHotel

inside the iceCream of Hotel Interface

2.Static method:

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

--we can define a static method also inside an interface from java 1.8.

--this static method must have body.

--static method of an interface will not be inherited inside the implementation class object.

--so we cannot call this static method of an interface by using implementation class object.

NOTE:

--we can call the static method of an interface only by using interface name.

We cannot call static method of an interface even by using interface variable also.

--\*\* we can define same static method as static or non-static method inside the implementation class also. Which is already defined statically inside the interface. (this concept is called as method hiding).

**public** **interface** Hotel {

**void** chickenBiryani();

**void** masalaDosa();

//default method

**default** **void** iceCream(){

System.***out***.println("inside the iceCream of Hotel Interface");

}

**static** **void** drinkingWater()

{

System.***out***.println("inside the drinkingWater from Hotel");

}

}

Demo.java

**public** **class** Demo{

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

{

Hotel h=**new** TajHotel();

h.chickenBiryani();

h.iceCream();

Hotel hr=**new** RoadSideHotel();

hr.chickenBiryani();

hr.iceCream();

//static method calling

Hotel.*drinkingWater*();

}

}

NOTE: one class can extends another class and implements multiple interfaces.

Eg:

class A extends B implements X,Y

NOTE: we can have an empty interface also.

Eg:

Public interface X {

}

---these type of empty interface is also known as tag or marker interface.

---some of the predefined marker interfaces in java are:

Java.io.Serializable

Java.lang.Cloanable

--these kind of marker interfaces used to specify certain type of special behaviour of our object.

class Student implements Serializable {

}

Student s1=new Student(10,”ram”,768); // moving out this student object from the RAM to outside the RAM is known as serialization and reverse is known as deserialization.

MARKER INTERFACE:

--if we want to group multiple objects to a special type then also we can use Marker Interface.

Eg:

Special.java

**public** **interface** Special {

}

A.java

**public** **class** A **implements** Special{

**public** A() {

System.***out***.println("inside the constructor of A");

}

**void** funA()

{

System.***out***.println("inside funA of A");

}

}

Demo.java

**public** **class** Demo{

**public** **void** fun(Special special)

{

System.***out***.println("inside the fun of Demo");

}

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

{

Demo d=**new** Demo();

d.fun(**new** A());

}

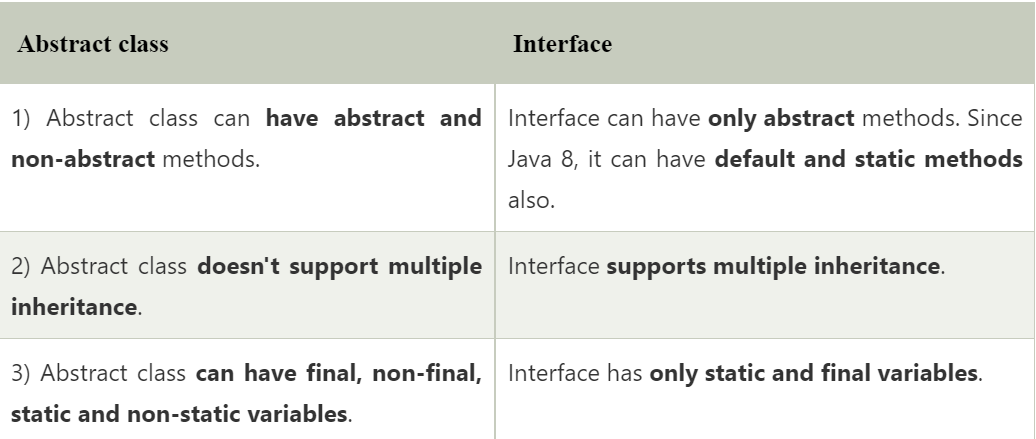
}

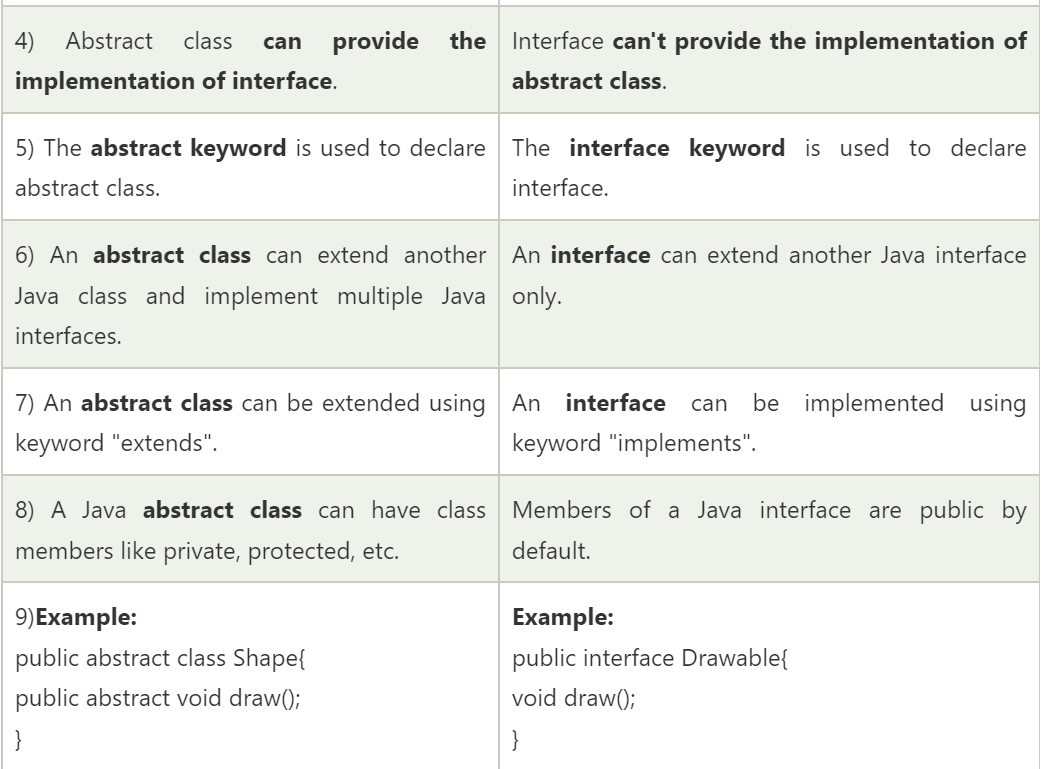
Output:

inside the constructor of A

inside the fun of Demo

Difference between Abstract class and Interface:





**Regular Expression**:

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

--It is an expression which represents a group of strings according to a particular pattern.

---We can write a regular expression to represent all valid mail ids.

---we can write a regular expression to represent all valid Indian mobile numbers.

Application areas where we can use RE:

-Form validation

-Pattern matching application (search) ctrl+f

-Design translater, like compiler, assembler.

etc, .

To represent and use Regular Expressions in java applications, JAVA has provided some classes and interfaces belongs to “java.util.regex” package.

--we mainly uses following 2 classes from the java.util.regex package to provide facility of RE:

1. java.util.regex.Pattern: A pattern object represents “compiled version of regular expression”.

2. java.util.regex.Matcher: Matcher object we can use to match the given pattern in the target string.

To create RE in the form of pattern object we have to use the following method from java.util.regex.Pattern class.

Public static Pattern compile(String regex);

Example:

Pattern p=Pattern.compile(“ab”);

We can create a Matcher object by using the matcher() method of the pattern class.

Public Matcher macher(String target);

Example:

Macher m=p.matcher(“abbbabbaba”);

To get details about the matches identified in the Macher object we have to use the following methods of the Macher object.

Public Boolean find(): it attempts to find the next match and return true if it is available otherwise returns false.

Public int start(): returns the starting index of the matched subsequence.

Public int end(): returns the ending index of the matched subsequence.

Public String group(): returns the matched subsequence.

To prepare RE, we have to use the following elements.

--Character classes

--Quantifiers

**Matches**:

Example:

**public** **class** Demo {

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

{

System.***out***.println("cat".matches("cat"));

System.***out***.println("Cat".toLowerCase().matches("cat"));

System.***out***.println("Cat".matches("[cC]at"));

System.***out***.println("Bat".matches("[cC]at"));

System.***out***.println("Bat".matches("[cCBb]at"));

System.***out***.println("gun".matches("[a-g]un"));

System.***out***.println("@siri".matches("[^c]siri"));

System.***out***.println("long".matches("[^a-z]ong"));

System.***out***.println("Bike".matches("\\wike"));// \\w means it accept one character

System.***out***.println("\_ike".matches("\\wike"));

System.***out***.println("\_\_\_".matches("\\w\\w\\w")); // w-word

//numbers

System.***out***.println("123".matches("\\d\\d\\d"));//d-number

//phone number matching

System.***out***.println("321-333-7652".matches("\\d\\d\\d-\\d\\d\\d-\\d\\d\\d\\d"));

System.***out***.println("321-333-7652".matches("\\d{3}-\\d{3}-\\d{4}"));

System.***out***.println("321.333.7652".matches("\\d{3}[-.,]\\d{3}[-.,]\\d{4}"));

System.***out***.println("321,333.7652".matches("\\d{3}[-.,]\\d{3}[-.,]\\d{4}"));

System.***out***.println("321 333 7652".matches("\\d{3}[-.,\\s]\\d{3}[-.,\\s]\\d{4}")); // s-space

System.***out***.println("321 333 7652".matches("\\d{3}[-.,\\s]+\\d{3}[-.,\\s]+\\d{4}"));// + - multiple or 1 or more

System.***out***.println("3213337652".matches("\\d{3}[-.,\\s]+\\d{3}[-.,\\s]+\\d{4}"));

System.***out***.println("3213337652".matches("\\d{3}[-.,\\s]\*\\d{3}[-.,\\s]\*\\d{4}"));//\*- zero or more

System.***out***.println("321333 7652".matches("\\d{3}[-.,\\s]?\\d{3}[-.,\\s]?\\d{4}"));//?- zero or one

System.***out***.println("321 333 652".matches("\\d{3}[-.,\\s]\*\\d{3}[-.,\\s]\*\\d{3,4}"));//{3,4}- 3 digits or 4 digits

System.***out***.println("321 333 7652798".matches("\\d{3}[-.,\\s]\*\\d{3}[-.,\\s]\*\\d{3,}"));// {3,}-3 or more digits

System.***out***.println("321 333 4321".matches("(\\d{3}[,-.\\s]?){2}\\d{4}"));

}

}

Output: true

true

true

false

true

true

true

false

true

true

true

true

true

true

true

true

true

true

false

true

true

true

true

true

Example:

**public** **class** Example{

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

String str="Flinstone, Fred, 1/1/1900, Programmer, {locpd=2000,yoe=10,iq=140}";

String regex="(?<lastName>\\w+),\\s\*(?<firstName>\\w+),\\s\*(?<dob>\\d{1,2}/\\d{1,2}/\\d{4}),\\s\*(?<role>\\w+),\\s\*\\{(\\w+=\\d{4},\\w+=\\d{2},\\w+=\\d{3})\\}";

System.***out***.println(str.matches(regex));

}

}

Output:true

**Matcher and pattern**:

Eg:

**import** java.util.regex.Matcher;

**import** java.util.regex.Pattern;

**public** **class** Demo {

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

{

String regex="((\\d{1,2})[,-.\\s]?)?((\\d{3})[,-.\\s]?)((\\d{3})[,-.\\s]?)(\\d{4})";

String phoneNum="91.231.345,5677";

System.***out***.println(phoneNum.matches(regex));

Pattern ptr=Pattern.*compile*(regex);

Matcher mat=ptr.matcher(phoneNum);

**if**(mat.matches())

{

System.***out***.println("country code:"+mat.group(2));

System.***out***.println("area code:"+mat.group(4));

System.***out***.println("exchange:"+mat.group(6));

System.***out***.println("line number:"+mat.group(7));

System.***out***.println(mat.group(0));

}

}

}

Output:

true

country code:91

area code:231

exchange:345

line number:5677

91.231.345,5677

Explanation:

((\\d{1,2})[,-.\\s]?) ---it is 1

(\\d{1,2}) – it is 2

To overcome above:

**import** java.util.regex.Matcher;

**import** java.util.regex.Pattern;

**public** **class** Demo {

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

{

String regex="(?:(\\d{1,2})[,-.\\s]?)?(?:(\\d{3})[,-.\\s]?)(?:(\\d{3})[,-.\\s]?)(\\d{4})";

String phoneNum="91.231.345,5677";

System.***out***.println(phoneNum.matches(regex));

Pattern ptr=Pattern.*compile*(regex);

Matcher mat=ptr.matcher(phoneNum);

**if**(mat.matches())

{

System.***out***.println("country code:"+mat.group(1));

System.***out***.println("area code:"+mat.group(2));

System.***out***.println("exchange:"+mat.group(3));

System.***out***.println("line number:"+mat.group(4));

System.***out***.println(mat.group(0));

}

}

}

Output:

true

country code:91

area code:231

exchange:345

line number:5677

91.231.345,5677

Explanation:

(?:(\\d{1,2})[,-.\\s]?) –

?: - non capture .

--it ignore the big group and it consider 1st group as (\\d{1,2})

?<>

-----

Within angular braces we can give a name to the group and we can call that using group name.

Eg: **import** java.util.regex.Matcher;

**import** java.util.regex.Pattern;

**public** **class** Demo {

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

{

String regex="(?:(?<countryCode>\\d{1,2})[,-.\\s]?)?(?:(?<areaCode>\\d{3})[,-.\\s]?)(?:(?<exchange>\\d{3})[,-.\\s]?)(?<lineNumber>\\d{4})";

String phoneNum="91.231.345,5677";

System.***out***.println(phoneNum.matches(regex));

Pattern ptr=Pattern.*compile*(regex);

Matcher mat=ptr.matcher(phoneNum);

**if**(mat.matches())

{

System.***out***.println("country code:"+mat.group("countryCode"));

System.***out***.println("area code:"+mat.group("areaCode"));

System.***out***.println("exchange:"+mat.group("exchange"));

System.***out***.println("line number:"+mat.group("lineNumber"));

System.***out***.println(mat.group(0));

}

}

}

Output:

true

country code:91

area code:231

exchange:345

line number:5677

91.231.345,5677

**#COMMENTS**

Example: Eg for brackets (234)

(?:\\(?(?<areaCode>\\d{3})\\)?[,-.\\s]?)

**import** java.util.regex.Matcher;

**import** java.util.regex.Pattern;

**public** **class** Demo {

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

{

String regex="""

#this is my regex to parse the parts of a phone number

(?:(?<countryCode>\\d{1,2})[,-.\\s]?)? #country code

(?:\\(?(?<areaCode>\\d{3})\\)?[,-.\\s]?) #area code

(?:(?<exchange>\\d{3})[,-.\\s]?) #exchange

(?<lineNumber>\\d{4}) #line number

""";

String phoneNum="1.(231).345,5677";

System.***out***.println(phoneNum.matches(regex));

Pattern ptr=Pattern.*compile*(regex,Pattern.***COMMENTS***);

Matcher mat=ptr.matcher(phoneNum);

**if**(mat.matches())

{

System.***out***.println("country code:"+mat.group("countryCode"));

System.***out***.println("area code:"+mat.group("areaCode"));

System.***out***.println("exchange:"+mat.group("exchange"));

System.***out***.println("line number:"+mat.group("lineNumber"));

System.***out***.println(mat.group(0));

}

}

}

Output:

false

country code:1

area code:231

exchange:345

line number:5677

1.(231).345,5677

**DOT(.):**

Eg:

**public** **class** Demo {

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

{

System.***out***.println("good".matches("....")); //4 dots - 4 characters

System.***out***.println("good".matches(".+"));

System.***out***.println("good".matches("^....$"));// ^->start of string $->end of string.

System.***out***.println("good".matches("^..$..")); //$->is placed b/w string is not end of string

System.***out***.println("good girl".matches("....\\s\\b....")); //b->boundary

System.***out***.println("---".matches("\\W\\W\\W"));//W->capital W->means non-word number

System.***out***.println("123".matches("\\W\\W\\W"));//W don't allow numbers

System.***out***.println("123".matches("\\D\\D\\D"));//D->it don't allow numbers.it will allow non-numbers.

System.***out***.println("hot".matches("\\D\\D\\D"));//string is non-number

System.***out***.println("".matches("\\s"));

System.***out***.println(".".matches("\\s"));

}

}

Output:

true

true

true

false

true

true

false

false

true

false

**Parsing a real text document:**

**import** java.util.regex.Matcher;

**import** java.util.regex.Pattern;

**public** **class** Demo {

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

{

String text="""

Student Number: 1234567890 Grade: 12

Birthdate: 01/02/2001 Gender: F

State ID: 2343454565

Cumulative GPA (weight) 3.82

Cumulative GPA (Unweighted) 3.46

""";

String regex="""

Student\\sNumber:\\s(?<studentNum>\\d{10}).\*#grab student number

Grade:\\s+(?<grade>\\d{1,2}).\* #grab grade

Birthdate:\\s+(?<month>\\d{2})/(?<day>\\d{2})/(?<year>\\d{4}).\* #grab data of birth

Gender:\\s+(?<gender>\\w+)\\b.\* #grab gender

State\\sID:\\s+(?<stateID>\\d+)\\b.\* #grab state

weight\\)\\s+(?<weight>[\\d\\.]+)\\b.\* #grab weight

weighted\\)\\s+(?<weighted>[\\d\\.]+)\\b.\* #grab weighted

""";

Pattern ptr=Pattern.*compile*(regex,Pattern.***DOTALL*** | Pattern.***COMMENTS***);

//DOTALL--> .\*

Matcher mat=ptr.matcher(text);

**if**(mat.matches())

{

System.***out***.println(mat.group("studentNum"));

System.***out***.println(mat.group("grade"));

System.***out***.println(mat.group("month"));

System.***out***.println(mat.group("day"));

System.***out***.println(mat.group("year"));

System.***out***.println(mat.group("gender"));

System.***out***.println(mat.group("stateID"));

System.***out***.println(mat.group("weight"));

System.***out***.println(mat.group("weighted"));

}

}

}

Output:

1234567890

12

01

02

2001

F

2343454565

3.82

3.46

Greedy Operator: (.\*?)

State\\sID:\\s+(?<stateID>\\d+)\\b.\*? #grab state

Cumulative.\*?(?<weight>[\\d\\.]+)\\b.\* #grab weight

Eg:

**import** java.util.regex.Matcher;

**import** java.util.regex.Pattern;

**public** **class** Demo {

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

{

String text="""

challagiri,sirisha,06/07/2002

R,naga,30/05/2001

ch,kiran,1/8/2000

""";

String regex="(?<lastName>\\w+),(?<firstName>\\w+),(?<dob>\\d{1,2}/\\d{1,2}/\\d{4})";

Pattern ptr=Pattern.*compile*(regex);

Matcher mat=ptr.matcher(text);

mat.find();

System.***out***.println(mat.group("lastName"));

System.***out***.println(mat.group("firstName"));

System.***out***.println(mat.group("dob"));

System.***out***.println(mat.start("lastName"));

System.***out***.println(mat.end("lastName"));

System.***out***.println(mat.start());

mat.find(25);

System.***out***.println(mat.group("lastName"));

System.***out***.println(mat.group("firstName"));

System.***out***.println(mat.group("dob"));

System.***out***.println(mat.start("lastName"));

System.***out***.println(mat.end("lastName"));

System.***out***.println(mat.start());

}

}

Output:

challagiri

sirisha

06/07/2002

0

10

0

R

naga

30/05/2001

30

31

30

Regex using loops:

Eg:

**import** java.util.regex.Matcher;

**import** java.util.regex.Pattern;

**public** **class** Demo {

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

{

String text="""

challagiri,sirisha,06/07/2002

R,naga,30/05/2001

ch,kiran,1/8/2000

""";

String regex="(?<lastName>\\w+),(?<firstName>\\w+),(?<dob>\\d{1,2}/\\d{1,2}/\\d{4})";

Pattern ptr=Pattern.*compile*(regex);

Matcher mat=ptr.matcher(text);

**while**(mat.find()) {

System.***out***.printf("%s %s %s%n",mat.group("lastName"),mat.group("firstName"),mat.group("dob"));

}

}

}

Output:

challagiri sirisha 06/07/2002

R naga 30/05/2001

ch kiran 1/8/2000

Example:

**import** java.util.regex.Matcher;

**import** java.util.regex.Pattern;

**public** **class** Example {

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

String str="Flinstone, Fred, 1/1/1900, Programmer, {locpd=2000,yoe=10,iq=140}";

String regex="(?<lastName>\\w+),\\s\*(?<firstName>\\w+),\\s\*(?<dob>\\d{1,2}/\\d{1,2}/\\d{4}),\\s\*(?<role>\\w+),\\s\*\\{\\w+\\=(?<locpd>\\w+)\\,\\w+\\=(?<yoe>\\w+)\\,\\w+\\=(?<iq>\\w+)\\}";

System.***out***.println(str.matches(regex));

String str1="locpd=2000,yoe=10,iq=140";

String progRegex="\\w+\\=(?<locpd>\\w+)\\,\\w+\\=(?<yoe>\\w+)\\,\\w+\\=(?<iq>\\w+)";

System.***out***.println(str1.matches(progRegex));

Pattern ptr=Pattern.*compile*(regex);

Matcher mat=ptr.matcher(str);

**while**(mat.find()){

System.***out***.printf("%s %s %s %s %s %s %s%n",mat.group("lastName"),mat.group("firstName"),mat.group("dob"),mat.group("role"),mat.group("locpd"),mat.group("yoe"),mat.group("iq"));

}

**if**(mat.matches()){

System.***out***.println(mat.group("lastName"));

}

}

}

Output:

true

true

Flinstone Fred 1/1/1900 Programmer 2000 10 140

Flinstone

Example:

**import** java.util.regex.Matcher;

**import** java.util.regex.Pattern;

**public** **class** Demo{

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

{

**int** count=0;

Pattern p=Pattern.*compile*("ab");

Matcher m=p.matcher("abbbabbaba");

**while**(m.find())

{

count++;

System.***out***.println(m.start()+"-----"+m.end()+"-----"+m.group());

}

System.***out***.println("The number of occurences:"+count);

}

}

Output:

0-----2-----ab

4-----6-----ab

7-----9-----ab

The number of occurences:3

1.Character class:

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

Character classes are used to specify alphabets and digits in RE

[abc]-------------------------------Either ‘a’ or ‘b’ or ‘c’

[^abc] ------------------------------Except ‘a’ and ‘b’ and ‘c’

[a-z] ---------------------------------any lowercase alphabet symbol

[A-Z] --------------------------------any uppercase alphabet symbol

[a-zA-Z] ----------------------------any alphabet symbol

[0-9] ---------------------------------any digit from 0 to 9

[a-zA-Z0-9] ------------------------any alphanumeric character

[^a-zA-Z0-9] -----------------------any special character

[a-z&&[^bc]] -----------------------a through z, except for b and c

[a-z&&[^m-p]] ---------------------a through z, and not m through p

Regex Metacharacter:

The RE metacharacters work as short codes.

\s----------space character

\d---------any digit from 0 to 9 [0-9]

\w--------any word character [a-zA-Z0-9]

. ----------any character including special characters.

\S---------any character except for space character

\D--------any character except for digit

\W-------any character except for word character (special character)

2.\*\*Regex Quantifiers:

The quantifiers specify the number of occurrences of a character.

re\* -- 0 or any number of occurrences of the preceding expression.

re+ -- 1 or more (at least) number of occurrences of the preceding expression.

re? – 0 or 1 (at most 1) number of occurrences of the preceding expression.

re{n} -- exactly n number of occurrences of the preceding expression.

re{n,} – n or more occurrences of the preceding expression.

re{n,m} – at least n and at most m.

a|b – either a or b

Eg:

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class Demo{

public static void main(String args[])

{

int count=0;

Pattern p=Pattern.compile("[abc]");

Matcher m=p.matcher("abbbabbaba");

while(m.find())

{

count++;

System.out.println(m.start()+"-----"+m.end()+"-----"+m.group());

}

System.out.println("The number of occurences:"+count);

}

}

Output:

0-----1-----a

1-----2-----b

2-----3-----b

3-----4-----b

4-----5-----a

5-----6-----b

6-----7-----b

7-----8-----a

8-----9-----b

9-----10-----a

The number of occurences:10

EXCEPTION HANDLING:

There are 2 types of errors:

1.syntax error: improper environment and syntactical error. It results in compilation error.

2.Logical error: sometimes JVM would be unable to execute statements in the bytecode due to some logical error occur in our program.

--whenever a logical error occurs in our application, JVM will terminate the program abnormally, and control comes out from the program unconditionally without executing remaining statements.

--for each logical error, there are some predefined classes in java, whose objects are created by the JVM.

Eg:

Java.lang.ArthimeticException

Java.lang.ArrayIndexOutOfBoundsException

Java.lang.NullPointerException

Java.lang.classCastException

Java.lang.NumberFormatException

Java.io.IOException

Java.sql.SQLException

Etc…

--whenever JVM encounters a logical error, it creates an object of the corresponding predefined exception class, and put that in our application, if we don’t handle that object explicitly, this object will reach back to the JVM, then JVM will receive that object and terminates the program abnormally.

--to avoid the abnormal termination, we need to avoid that exception class object should not reach back to the JVM, and we do so by handling that exception class object.

--we handle the exception class object by just assigning that object to the corresponding exception class variable.

--we have 2 keywords try and catch to implements exception handling:

1.try: try keyword will recognize the exception class object created by the JVM and it gets hold on that object and transfer that object to the catch block.

2.catch: catch block, we should be able to assign those objects to the appropriate class reference variable so that they will not reach back to the JVM and therefore program will not be terminated abnormally.

\*\*Important points:

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

1. The main objective of exception handling is normal/ graceful termination of the program.
2. We can know in which part of the program the logical error has occurred.
3. We are not allowing the JVM to terminate the entire application, only the try block in which the logical error has occurred would be terminated.
4. If no exception class object is created inside the try block, control will never enter to the catch block.
5. Once the control enters in the catch block, it executes the statements in the catch block and moves further. It will not come to the try block again, because of which the catch block has executed.
6. Every try block should be associated with at least one catch block.

Note: in a catch block, we not only handle the exception class object to the corresponding variable, in addition to that catch block provides an alternate way to define the logic whenever any exception occurred inside the try block.

Example:

**public** **class** Demo {

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

{

**try** {

**int** x=100;

**int** y=0;

**int** z=x/y;

System.***out***.println("The result is:"+z);

}

**catch**(ArithmeticException ae){

System.***out***.println("inside catch block");

System.***out***.println(ae.getMessage());

}

System.***out***.println("end of main");

}

}

Output:

inside catch block

/ by zero

end of main

Advantages and Disadvantages of using multiple catch blocks for a single try block:

Advantages:

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

--we have more granular way to handle each type of execution, we can write alternate logic for each type of exception.

Disadvantages:

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

1.Developer should have through knowledge about which statement may throw which type of exception, which may not be possible always.

2.Writing too many catch block is also feasible.

Object

|

Throwable

|

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

| |

Exception (checked exception) Error

|

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

| | | |

Run time Exception ClassNotFoundException SQLException IOException etc..

|

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

| | |

ArthimeticException NullPointerException ArrayIndexOutOfBoundException

\*-- the classes which are direct child of the exception class (except RunTimeException class) is known as checked exception class (including exception class also).

\*-- the classes which are the direct child of RunTimeException including the RunTimeException class is known as unchecked exception.

--Exception class is the super class/ Base class of all other exception classes.

--above all the classes belongs to java.lang package.

--as we know that super class reference can point to the class object, so we can take only one catch block that can handle any type of exception class object.

Rule for try with multiple catch block:

--sibling (NullPointerException, ArthimeticException, etc.) can be in any order, but parent (Exception) must be at last.

logical errors are also of two types:

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

1. Simple logical error: these logical errors are permitted to handle, all the classes which represent the simple logical error comes under the exception category, ie. They are the child of the exception class.
2. Serious logical error: Not permitted to handle, all the classes which represents serious logical error comes under the error class category, i.e they are the child of error class.

Note: catch block is designed in such a way that it will take only object of Throwable class and its child class. It cannot take object class.

--In catch block technically we can take the Throwable but it is bad practice, because it will handle error class object also, (which we are not permitted to handle).

--In java we have all together 5 keywords in the concept of exception handling:

1. try

2. catch

3. throw

4. throws

5. finally

Exception classes are classified in 2 categories:

1. Checked Exception: checked by the java compiler whether we have handled that exception class object or not at compile time.
2. Unchecked Exception: not checked by the compiler whether we handled exception class object or not.

NOTE: whether exception is a checked or unchecked exception, exceptions always occur at runtime, it never occurs at compile time, compile time only occurs compilation error.

.java -------------- java compiler ------------ .class ---------------- jvm ------------------------- execute that class

Responsibility of java compiler:

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

1. Convert .java (source code) to the .class (byte code).
2. It will scan our.java file and check any kind of syntax error, and generates a compile time error.
3. If inside the class we don’t have any constructor then java compiler provides a default constructor to the .class.
4. For each and every statement which may have any logical error, compiler will check whether we are handling the corresponding exception class or not.

Checked Exception:

Applying the exception handling concept is mandatory at compile time only, if we don’t handle them, then compiler will generate a compile time error.

\*\*the exception class object for which compiler force you to handle at compile time only is known as checked exception.

Eg:

ClassNotFoundException

IOException

SQLException

FileNotFoundException

Unchecked Exception:

The classes which are the direct child of RunTimeException including the RunTimeException class is known as unchecked exception.

Eg:

ArthimeticException

NullPointerException

ArrayIndexOutOfBoundsException etc.

Throw Keyword:

--with ‘throw’ keyword we can generate the exception class object in our program explicitly.

--Generally JVM will create the exception class object, whenever a logical error occurs in our application, and put/ throw that exception class object inside our application (at the same statement where logical error occurs).

--but sometimes, it is required that programmer need to create the exception class object explicitly and throw that exception class object inside the application manually, whenever a logical error occurs due to the business logic violation.

Note: this business logic violation error related logical error, is not a logical error to the point of view of JVM.

Eg:

**public** **class** Dhaanvi {

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

**int** a=10,b=0,c=0;

**try** {

**if**(b==0) {

ArithmeticException ae=**new** ArithmeticException("Zero division exception");

**throw** ae;

}

**else** {

System.***out***.println(a/b);

}

}

**catch**(ArithmeticException e) {

System.***out***.println(e.getMessage());

}

}

Eg:

**public** **class** TestThrow1 {

    //function to check if person is eligible to vote or not

**public** **static** **void** validate(**int** age) {

**if**(age<18) {

            //throw Arithmetic exception if not eligible to vote

**throw** **new** ArithmeticException("Person is not eligible to vote");

        }

**else** {

            System.out.println("Person is eligible to vote!!");

        }

    }

    //main method

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

        //calling the function

        validate(13);

        System.out.println("rest of the code...");

  }

Eg:

**public** **class** ExceptionEx {

**public** **int** getPension (**int** age,**int** salary)

{

**int** pension=0;

**if**(age>40 && age<100)

{

pension=(age\*salary)/100;

}

**else** {

ArithmeticException ae=**new** ArithmeticException("Invalid age:"+age);

**throw** ae;

}

**return** pension;

}

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

ExceptionEx ee=**new** ExceptionEx();

**try** {

**int** result=ee.getPension(50, 20000);

System.***out***.println(result);

**int** result1=ee.getPension(10, 30000);

System.***out***.println(result1);

}

**catch**(ArithmeticException ae){

System.***out***.println(ae.getMessage());

}

}

}

Output: 10000

Invalid age:10

Throws Exception:

--this keyword is used along with method signature to announce the caller that the method may throw/ generate an exception class object at calling place.

Eg:

**public** **int** getPension(**int** age,**int** salary) **throws** ArithmeticException

{

**int** pension=0;

**if**(age>40 && age<100)

{

pension=(age\*salary)/100;

}

**else** {

ArithmeticException ae=**new** ArithmeticException ("Invalid age:"+age);

**throw** ae;

}

**return** pension;

}

Difference between throw & throws:

|  |  |
| --- | --- |
| throw | throws |
| Used to throw an exception for a method | Used to indicate what exception type may be thrown by a method |
| Cannot throw multiple exceptions | Can declare multiple exceptions |
| **Syntax:**  ->throw is followed by an object (new *type*)  ->used inside the method | **Syntax:**  ->throws is followed by a class  ->and used with the method signature |

Rule 1:

--If we throwing any unchecked exception class object from a method then announcing that exception with method signature is optional. Otherwise (if we throw any checked class object) then it is mandatory.

Rule 2:

If a method throws any checked exception then caller have 2 choices at compile time:

1. Caller need to wrap the method call inside try-catch block (handling the exception) it is mandatory.
2. Caller can delegate the exception handling duty to its own caller by using “throws” keyword (skipping the exception handling duty).

\*\*NOTE:

We only handle the exception using try-catch block, using throws keyword we just skip the exception handling duty and delegate it to the caller.

--we can apply the throws keyword with the main method also, but in real time application it is never recommended to use throws keyword along with main method. Because doing so, we are ready to accept abnormal termination of our application.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*If the exception is checked we must handle the exception. For Checked method we must use try-catch block or throws at method extension.

\*If the exception is unchecked it is no mandatory to handle the exception. For Unchecked exception method need not use any extra blocks.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Checked Example: one way

**public** **class** ExceptionEx {

**public** **void** fun() **throws** ClassNotFoundException,FileNotFoundException {

System.***out***.println("inside fun...");

}

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

ExceptionEx ee=**new** ExceptionEx();

**try** {

ee.fun();

} **catch** (ClassNotFoundException | FileNotFoundException e) {

e.printStackTrace();

}

}

}

Another way:

**public** **class** ExceptionEx {

**public** **void** fun() **throws** ClassNotFoundException,FileNotFoundException {

System.***out***.println("inside fun...");

}

**public** **static** **void** main(String[] args) **throws** ClassNotFoundException,FileNotFoundException{

ExceptionEx ee=**new** ExceptionEx();

ee.fun();

}

}

Output: for both

Inside fun ….

NOTE: when we use checked or unchecked exceptions

Whenever we want to make exception handling duty mandatory to the caller then throw the checked exception.

User-defined Exception:

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

**class** IllegalException **extends** Exception{

**public** IllegalException()

{

}

}

**public** **class** ExceptionEx {

**public** **void** fun() **throws** IllegalException {

System.***out***.println("inside fun...");

}

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

ExceptionEx ee=**new** ExceptionEx();

ee.fun();

}

}

Output:

Inside fun…

Nested try catch block:

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

--we can use try-catch inside a try block or a catch block and even in finally block also.

Eg:

**public** **class** NestedTryCatch

{

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

**try** {

System.***out***.println("inside try 1");

System.***out***.println(100/10);

**try** {

System.***out***.println("inside try 2");

//some another class - month

Month a=**null**;

a.getTowers();

} **catch** (ArithmeticException ae) {

System.***out***.println("Inside catch of try2 -- arithmetic exception");

}

System.***out***.println("end of try 1");

}

**catch**(ArithmeticException ae) {

System.***out***.println("Inside catch of try 1 -- Arithmetic exception");

}

**catch**(Exception e){

System.***out***.println("Inside catch of try 1 -- Exception");

}

}

}

Output:

inside try 1

10

inside try 2

Inside catch of try 1 -- Exception

NOTE: If inner try is unable to handle the exception then that exception is propagated to the outer try block, by suspending the remaining statement of inner try block.

--we can also write try catch block in nested catch block also.

Throws in constructor:

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

--since constructor is also a block of code, and there might be a chance of logical error.

--so constructor can also throw an exception.

Eg:

**class** IllegalException **extends** Exception{

**public** IllegalException()

{

}

}

**public** **class** ExceptionEx {

//constructor

**public** ExceptionEx() **throws** IllegalException{

System.***out***.println("Inside default constructor");

}

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

**try** {

ExceptionEx ee=**new** ExceptionEx();

} **catch** (IllegalException e) {

e.printStackTrace();

}

}

}

Output:

Inside default constructor

--so if any constructor throws a checked exception then we need to create object of that class inside the try-catch block or we can skip the exception handling duty by using throws keyword.

Throws with interface:

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

--if a parent class constructor throws any checked exception then in order to extends that class we need to resolve the parent class constructor by calling the parent class constructor from the child class constructor.

Eg:

A.java

**class** A{

**int** i=10;

**public** A() **throws** IllegalException{

System.***out***.println("inside the constructor of class A");

}

**public** A(**int** i) **throws** IllegalException{

System.***out***.println("inside the constructor(int i) of class A");

}

**void** fun() {

System.***out***.println("inside fun of A");

}

}

Demo.java

**public** **class** ExceptionEx **extends** A{

//constructor

**public** ExceptionEx() **throws** IllegalException{

**super**(10);

System.***out***.println("Inside default constructor");

}

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

**try** {

ExceptionEx ee=**new** ExceptionEx();

} **catch** (IllegalException e) {

e.printStackTrace();

}

}

}

Output:

inside the constructor (int i) of class A

Inside default constructor

Method overriding rules with the exception handling:

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

--if a parent class method throws any checked exception, then while overriding that method inside the child class we have following options:

1. child class overridden method may not throw any kind of exception.

2. child class overridden method may throw the same exception.

3. may throw any kind of unchecked exception.

--But child class overridden method cannot throw super type (parent) of specified exception.

Eg:

**class** A{

**int** i=10;

//extending class is bigger

**void** fun() **throws** Exception{

System.***out***.println("inside fun of A");

}

}

**public** **class** ExceptionEx **extends** A{

//child of parent exception

@Override

**void** fun() **throws** ArithmeticException{

System.***out***.println("inside fun of ExceptionEx");

}

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

ExceptionEx ee=**new** ExceptionEx();

ee.fun();

}

}

Output:

inside fun of ExceptionEx

finally block:

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

--whether try block is executing or catch block is executing finally must execute.

--Irrespective of try and catch if we want to execute some code then we can use finally block.

--a finally block used with try and catch block only.

--we can use a finally block without a catch block also, i.e try with finally (but not recommended to use).

Eg:

**public** **class** ExceptionEx{

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

**try** {

//Connecting to the DB server

//after getting the DB connection, we need to perform DB operation.

System.***out***.println(100/0);

//after performing the DB operation we should close the connection

}**catch** (Exception e) {

System.***out***.println("inside catch");

}

**finally** {

//close the connection

System.***out***.println("inside finally");

}

//close the connection - outside the block

//by doing this we violate 2 coding principles.

//1. coding modularity 2. coding integrity

//coding modularity: related code should be in related unit.

//coding integrity: one logic should not mix up with other logic.

//any code is changed that may effect on the other code also. so it is better to place the code in block.

}

}

Output:

inside catch

inside finally

try with resource features:

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

**try** ("create connection") {

//Connecting to the DB server

//after getting the DB connection, we need to perform DB operation.

System.***out***.println(100/0);

//after performing the DB operation we should close the connection

}**catch** (Exception e) {

System.***out***.println("inside catch");

}

Exception rethrowing:

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

Converting one exception to another exception object.

Eg:

**public** **class** ExceptionEx{

**void** fun(**int** n1,**int** n2) **throws** IllegalException{

**try** {

System.***out***.println(n1/n2);

}

**catch** (ArithmeticException e) {

System.***out***.println("inside catch");

IllegalException ie=**new** IllegalException("Number2 should not be 0");

**throw** ie;

}

}

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

ExceptionEx ee=**new** ExceptionEx();

**try** {

ee.fun(100, 0);

} **catch** (IllegalException e) {

System.***out***.println("inside catch of main method");

e.printStackTrace();

}

}

}

Output:

inside catch

inside catch of main method

com.mic.IllegalException

at Practice/com.mic.ExceptionEx.fun(ExceptionEx.java:27)

at Practice/com.mic.ExceptionEx.main(ExceptionEx.java:34)

**Functional Programming:**

--this concept is introduced in java 1.8 onwards.

--In this type of paradigm, a function is treated as a value. (we can assign the entire function in a variable, or we ca pass a function to another function parameter or we can return a function from another function as well.)

Int i=10;

--the main advantage of the Functional programming is less coding, polymorphism and easy to understand.

--to achieve the functional programming in java we need a functional interface.

Functional interface:

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

--an interface which has only one abstract method is called FI (Functional Interface).

--A FI may have ‘N’ number of static and default methods.

--A FI may contain some data members(variables) also.

Example:

@FunctionalInterface

**public** **interface** Intr { //by default it is public and abstract

**void** fun();

}

}

@FunctionalInterface annotation make sure we have a valid FI.

Eg2:

@FunctionalInterface

**public** **interface** Intr {

**int** ***i***=20;

//by default it is public and abstract

**void** fun();

//it belongs to object class

**public** **abstract** String toString();

**public** **static** **void** fun2() {

}

**public** **default** **void** fun3() {

}

}

Some of the predefined functional interface in java:

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

Java.lang.Comparable : public int compareTo(Object obj);

Java.util.Comparator : public int compare(Object obj1, Object obj2);

Java.lang.Iterable : public Iterator iterator();

Java.lang.Runnable : public void run();

NOTE: with the help of FI we achieve FP in java using Lambda expression.

Eg:

Intr.java

@FunctionalInterface

**public** **interface** Intr {

**void** fun(**int** i);

}

Demo.java

**public** **class** Demo **implements** Intr{

@Override

**public** **void** fun(**int** i) {

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

}

}

Eg1:

**public** **class** Demo **implements** Intr{

@Override

**public** **void** fun(**int** i) {

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

}

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

Demo d=**new** Demo();

Intr it=**new** Demo();

//using anonymous inner class

Intr i1=**new** Intr() {

@Override

**public** **void** fun(**int** i) {

System.***out***.println("welcome using anonymous inner class:"+i);

}

};

i1.fun(12);

it.fun(0);

d.fun(56);

//using lambda expression(only for functional interface)

Intr i2= i -> System.***out***.println("Lambda expression:"+i);

i2.fun(222);

}

}

Output:

welcome using anonymous inner class:12

Integer:0

Integer:56

Lambda expression:222

Lambda Expression:

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

--It is an expression using which we can provide the implementation of a functional Interface.

Lambda Expression comprises 3 things:

1.parameter (data type is optional): If only one parameter is then () small bracket is also optional, small () bracket is mandatory when we have zero or more than one parameter.

🡪 method have zero arguments then — ()

2.lambda operation: ->

3.method body: If we write only one statement inside the implementation body then {} is optional.

Note: LE does not consider the method name of FI.

Eg:

@FunctionalInterface

**public** **interface** Intr {

**void** fun(**int** i,String name);

}

Demo.java

**public** **class** Demo **implements** Intr{

@Override

**public** **void** fun(**int** i,String name) {

// **TODO** Auto-generated method stub

}

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

Intr i=(num,n)-> System.***out***.println("number:"+num+"-----"+"age: "+n);

i.fun(500,"siri");

}

}

Output:

number:500-----age: siri

Functional Interface method with return type:

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

--If inside the method body only one statement is there then {} is optional, and return keyword is not allowed.

--return keyword is allowed only with {} brackets.

Eg:

Intr.java

@FunctionalInterface

**public** **interface** Intr {

String fun(**int** i,String name);

}

Demo.java

**public** **class** Demo **implements** Intr{

@Override

**public** String fun(**int** i,String name) {

// **TODO** Auto-generated method stub

**return** "hi";

}

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

Intr i=(num,n)-> "This is the number:"+num;

//one way of returning

String s=i.fun(10, "siri");

System.***out***.println(s);

//another way

System.***out***.println(i.fun(500, "sara"));

}

}

Output:

This is the number:10

This is the number:500

Eg2: return type Student

--Student bean class is created with variables roll, name, marks.

Intr.java

@FunctionalInterface

**public** **interface** Intr {

Student createObject(**int** roll,String name,**int** marks);

}

Demo.java

**public** **class** Demo **implements** Intr{

@Override

**public** Student createObject(**int** roll, String name, **int** marks) {

// **TODO** Auto-generated method stub

**return** **null**;

}

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

//format1

Intr i=(r,n,m)-> **new** Student(r,n,m);

Student s=i.createObject(101, "siri",550);

System.***out***.println(s);

//format2

Intr i1=(r,n,m)->{

Student s1=**new** Student(r,n,m);

**return** s1;

};

Student s2=i1.createObject(102, "Madhav", 600);

System.***out***.println(s2);

}

}

Output:

Student [roll=101, name=siri, marks=550]

Student [roll=102, name=Madhav, marks=600]

Eg3: sorting student marks

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Comparator;

**import** java.util.List;

**public** **class** Demo **implements** Intr{

@Override

**public** Student createObject(**int** roll, String name, **int** marks) {

// **TODO** Auto-generated method stub

**return** **null**;

}

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

List<Stu> student=**new** ArrayList<>();

student.add(**new** Stu(10,"n1",550));

student.add(**new** Stu(20,"n2",450));

student.add(**new** Stu(30,"n3",510));

student.add(**new** Stu(40,"n4",520));

student.add(**new** Stu(50,"n5",400));

//using external class implementation

Collections.*sort*(student,**new** StudentMarksComp());

//using anonymous class

Collections.*sort*(student, **new** Comparator<Stu>() {

**public** **int** compare(Stu s1, Stu s2) {

**return** s1.getMarks()>s2.getMarks()? +1:-1;

}

});

//using lambda expression

Collections.*sort*(student,(s1,s2)-> s1.getMarks()>s2.getMarks()? +1:-1);

}

}

Eg4: passing functional interface as method parameter

Intr.java

@FunctionalInterface

**public** **interface** Intr {

**void** fun();

}

ImpleIntr.java

**public** **class** ImpleIntr **implements** Intr {

@Override

**public** **void** fun() {

// **TODO** Auto-generated method stub

}

}

Demo.java

**public** **class** Demo {

**public** **void** fun(Intr i1) {

System.***out***.println("inside fun of Demo");

System.***out***.println(i1);//address

i1.fun();

}

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

Demo d=**new** Demo();

//one way

Intr i1= () -> System.***out***.println("hi hello");

d.fun(i1);

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

//lambda expression

d.fun(() -> System.***out***.println("hi hello"));

}

}

Output:

inside fun of Demo

com.mic.Demo$$Lambda$23/0x00000008000c0400@43a25848

hi hello

===================

inside fun of Demo

com.mic.Demo$$Lambda$24/0x00000008000c0618@5594a1b5

hi hello

NOTE: we cannot override the object class methods example toString(), equals(), hashCode() etc.

Method Reference:

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

--It is the simplified form (short-cut) of Lambda expression.

--It is represented by using :: double colon symbol.

--Instead of creating a LE with all the details, with the help of method reference we can refer an existing class method to the functional interface variable.

Note:

we can take a reference of a static method using ClassName::methodName;

we can take a reference of a non-static method using ClassObject::methodName;

we can take a reference of a constructor also using ClassName::new;

Eg1: static method

@FunctionalInterface

**public** **interface** Intr {

**void** fun();

}

Demo.java

**public** **class** Demo {

**public** **static** **void** funX() {

System.***out***.println("inside funX of Demo");

System.***out***.println("other statements of Demo");

}

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

//our own message

Intr i1=() -> System.***out***.println("inside LE");

i1.fun();

//assigning a method block of statements

Intr i2=Demo::*funX*;

i2.fun();

}

}

Output:

inside LE

inside funX of Demo

other statements of Demo

Eg2: static, non-static, constructor methods

@FunctionalInterface

**public** **interface** Intr {

**void** fun(String name);

}

Demo.java

**public** **class** Demo {

**public** Demo(String name)

{

System.***out***.println("inside constructor of Demo:"+name);

}

**public** Demo() {

}

**public** **static** **void** funX(String name) {

System.***out***.println("inside funX of Demo:"+name);

}

**public** **void** funY(String name) {

System.***out***.println("inside funY of Demo:"+name);

}

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

//It refer the static method

Intr i1=Demo::*funX*;

//refer the non - static method

//Demo d=new Demo();

Intr i2=**new** Demo()::funY;

//refer the constructor

Intr i3=Demo::**new**;

//our own message

Intr i4= n-> System.***out***.println("Using LE:"+n);

i1.fun("Name1");

i2.fun("Name2");

i3.fun("Name3");

i4.fun("Name4");

}

}

Output:

inside funX of Demo: Name1

inside funY of Demo: Name2

inside constructor of Demo: Name3

Using LE: Name4

Eg:

@FunctionalInterface

**public** **interface** Intr {

**int** getTheNumber(String name);

}

Demo.java

**public** **class** Demo {

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

//anonymous

Intr i1=(num)-> {

**int** result=Integer.*valueOf*(num);

**return** result;

};

**int** res=i1.getTheNumber("20");

System.***out***.println(res);

//lambda expression

Intr i2=(n)-> Integer.*parseInt*(n);

System.***out***.println(i2.getTheNumber("100"));

Intr i3=Integer::*parseInt*;

System.***out***.println(i3.getTheNumber("200"));

}

}

Output:

20

100

200

Eg:

@FunctionalInterface

**public** **interface** Intr {

**void** printSomething(String message);

}

Demo.java

**public** **class** Demo {

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

Intr i1= m -> System.***out***.println(m);

i1.printSomething("welcome");

Intr i2= System.***out***::println;

i2.printSomething("Hello");

}

}

Output:

welcome

Hello

Eg: list

Intr.java

**import** java.util.List;

@FunctionalInterface

**public** **interface** Intr {

**void** printSomething(List<String> cities);

}

ImpleIntr.java

**import** java.util.List;

**public** **class** ImpleIntr **implements** Intr {

@Override

**public** **void** printSomething(List<String> cities) {

**for**(String c:cities)

{

System.***out***.println(c.toUpperCase());

}

}

}

Demo.java

**import** java.util.Arrays;

**public** **class** Demo {

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

//external class

Intr i1=**new** ImpleIntr();

i1.printSomething(Arrays.*asList*("a","x","g"));

//lambda expression

Intr i2= cities -> {

**int** count=0;

**for**(String s:cities) {

System.***out***.println((++count)+"==="+s);

}

};

i2.printSomething(Arrays.*asList*("a","c","y"));

}

}

Output:

A

X

G

1===a

2===c

3===y

Some of the functional interface introduced in java 8 to perform Functional Programming:

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

--these interfaces belongs to java.util.function package

1. Predicate(I)

2. Consumer(I)

3. Supplier(I)

4. Function(I)

Predicate(I):

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

This interface has only one abstract method

Interface Predicate<T> {

Public Boolean test (T t);

}

--this test () method checks whether supplied object is satisfying a condition or not.

Example: test a student object whether his / her marks is greater than 500 or not.

MyPredicate.java

**import** java.util.function.Predicate;

**class** MyPredicate **implements** Predicate<Student> {

@Override

**public** **boolean** test(Student s) {

// if(s.getMarks()>500)

// return true;

// else

// return false;

**return** s.getMarks()>500;

}

}

Demo.java

**import** java.util.function.Predicate;

**public** **class** Demo {

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

Predicate<Student> p=**new** MyPredicate();

System.***out***.println(p.test(**new** Student(10,"name1",450)));

//Lambda Expression

Predicate<Student> p1= s -> s.getMarks() >500;

System.***out***.println((p1.test(**new** Student(20,"name2",600))));

}

}

Output:

false

true

--from java 1.8 onwards inside the collection interface one new method is added called

Public Boolean removeIf (Predicate filter);

(Predicate filter – lambda expression)

--based on the condition of Predicate, this method will remove/ filter the elements from the collection classes.

Example:

MyPredicate.java

**import** java.util.function.Predicate;

**class** MyPredicate **implements** Predicate<Student> {

@Override

**public** **boolean** test(Student s) {

// if(s.getMarks()>500)

// return true;

// else

// return false;

**return** s.getMarks()>500;

}

Demo.java

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Demo {

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

List<Student> students=**new** ArrayList<>();

students.add(**new** Student(10,"n1",750));

students.add(**new** Student(20,"n2",850));

students.add(**new** Student(30,"n3",480));

students.add(**new** Student(40,"n4",450));

students.add(**new** Student(50,"n5",550));

// MyPredicate m=new MyPredicate();

// students.removeIf(m);

//external class

students.removeIf(**new** MyPredicate());

**for**(Student s:students) {

System.***out***.println(s);

}

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

//Lambda Expression

students.removeIf(s-> s.getMarks()>500);

**for**(Student s:students) {

System.***out***.println(s);

}

}

}

Output:

Student [roll=30, name=n3, marks=480]

Student [roll=40, name=n4, marks=450]

===================

Student [roll=30, name=n3, marks=480]

Student [roll=40, name=n4, marks=450]

Consumer(I):

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

Consumer<T>

Public void accept (T t);

--this method only accept the object of generic type and does not return anything.

Example:

MyConsumer.java

**import** java.util.function.Consumer;

**public** **class** MyConsumer **implements** Consumer<Stu> {

@Override

**public** **void** accept(Stu s) {

System.***out***.println("Roll: "+s.getRoll());

System.***out***.println("Name: "+s.getName());

System.***out***.println("Marks: "+s.getMarks());

}

}

Demo.java

**import** java.util.function.Consumer;

**public** **class** Demo {

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

//external class

Consumer<Stu> c=**new** MyConsumer();

c.accept(**new** Stu(10,"Siri",500));

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

//Lambda Expression: without using external class(MyConsumer)

Consumer<Stu> c1= s -> {

System.***out***.println("Roll: "+s.getRoll());

System.***out***.println("Name: "+s.getName());

System.***out***.println("Marks: "+s.getMarks());

};

c1.accept(**new** Stu(20,"Naga",550));

}

}

Output:

Roll: 10

Name: Siri

Marks: 500

=============

Roll: 20

Name: Naga

Marks: 550

forEach method:

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

Public void forEach (Consumer action); //action for each element of a collection

--this method is a default method belongs to Iterable interface.

--as we know that every collection is iterable (refer the collection hierarchy diagram).

--so we can call this forEach method on any collection object.

Eg:

**import** java.util.Arrays;

**import** java.util.List;

**public** **class** Demo {

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

List<String> list=Arrays.*asList*("armaan","arjun","abhi");

//normal for loop

//enhanced for loop

//Iterator

//ListInterator

list.forEach(n -> System.***out***.println(n.toUpperCase()));

}

}

Output:

ARMAAN

ARJUN

ABHI

Eg2:

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Demo {

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

List<Stu> students=**new** ArrayList<>();

students.add(**new** Stu(10,"n1",550));

students.add(**new** Stu(20,"n2",450));

students.add(**new** Stu(30,"n3",500));

students.add(**new** Stu(40,"n4",480));

students.forEach(n -> {

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

//write that object to the file (serialize the object)

});

}

}

Output:

Stu [roll=10, name=n1, marks=550]

Stu [roll=20, name=n2, marks=450]

Stu [roll=30, name=n3, marks=500]

Stu [roll=40, name=n4, marks=480]

Supplier(I):  
--------------

--java.util.function.Supplier;

public T get ();

Eg:

MySupplier.java

**import** java.util.function.Supplier;

**public** **class** MySupplier **implements** Supplier<String> {

@Override

**public** String get() {

**return** "inside the MySupplier external class ";

}

}

Demo.java

**import** java.util.function.Supplier;

**public** **class** Demo {

**public** **int** getANumber() {

**return** 10000;

}

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

//external class

Supplier<String> s=**new** MySupplier();

String str=s.get();

System.***out***.println(str);

//lambda expression

Supplier<String> s1= () -> "inside the lambda expression";

System.***out***.println(s1.get());

//LE: using Student

Supplier<Stu> s2= () -> **new** Stu(10,"n1",800);

System.***out***.println(s2.get());

//assigning another method: using method reference

Supplier<Integer> s3= **new** Demo()::getANumber; //non-static method

**int** i=s3.get();

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

}

}

Output:

inside the MySupplier external class

inside the lambda expression

Stu [roll=10, name=n1, marks=800]

10000

Function(I):

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

--java.util.function.Function<T,R>

Public R apply (T t);

Example: Getting a student object and returning

**class** MyFunction **implements** Function<Stu, String>{

@Override

**public** String apply(Stu s) {

// if(s.getMarks()>500)

// return "pass";

// else

// return "fail";

**return** s.getMarks()>500? "pass":"fail";

}

}

Demo.java

**import** java.util.function.Function;

**public** **class** Demo {

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

//external class

Function<Stu,String> f=**new** MyFunction();

Stu s1=**new** Stu(10,"name1",550);

System.***out***.println(f.apply(s1));

//lambda expression

Function<Stu,String> f1= s -> s.getMarks()>500?"pass":"fail";

System.***out***.println(f1.apply(**new** Stu(20,"name2",450)));

}

}

Output:

pass

fail

**STREAM API:**-- this api belongs to “java.util.stream” package.

--this api is different from IO stream, this IO – stream api belongs to java.io package and java.nio package here we represent flow of data between peripherals (input output devices) in the form of bytes or characters.

This java.io stream represents flow of data in bytes or characters

--this java.util.stream package contains some library classes and interfaces by using which we can perform functional style of programming on a group of objects (Collection of data) in the form of objects.

This java.util stream represents flow of data in the form of objects.

\*\*this api has one main interface:

Java.util.stream.Stream(I)

NOTE: object of this stream interface represents flow/ sequence of objects from a source like collection objects.

Feature of stream:

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

1.stream does not store the elements in a sequence.

Eg: write does not hold/ store the electricity

2. It represent only flow of objects, not the primitives.

3. operations (filtering/ mapping etc) performed on the stream object does not modify its source.

Eg: filtering a stream obtained from a source (collection) produces a new stream with the filtered elements rather than removing the elements from the source collection.

3.With the help of stream object we can perform various operations on the collection data in functional style, like filtering some elements, printing some elements, transforming some elements etc.

--Collection interface provides 2 methods to get a Stream object.

1. Stream<T> stream ();

2. Stream<T> parallelStream (); //this stream object is used on multithreading application.

Methods of the Stream(I) interface:

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

There are 2 types of method in the Stream interface:

1. Intermediate methods
2. Terminal methods

1.Intermediate methods: these methods return a new Stream object, instead of a final output.

--these methods never give the final output.

Some of the commonly used intermediate methods are:

Map (), filter () method

2.terminal methods: stream objects return the final output only when terminal method is called on the stream object.

These methods consume that stream object. And after that we cannot re-use that stream object again.

Note: If we try to use a consumed stream object once again, then we will get an exception.

forEach (Consumer action) //similar to the Iterable interface forEach method

collect ()

min ()

max ()

count ()

get ()

anyMatch ()

allMatch ()

Example:

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.stream.Stream;

**public** **class** Demo {

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

List<String> list=Arrays.*asList*("one","two","three");

Stream<String> s=list.stream();

s.forEach(n -> System.***out***.println(n)); //terminal method

//run time exception : we cannot use the stream once it used

s.forEach(i -> System.***out***.println(i));

}

}

Output:

one

two

three

Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed

Filter () method:

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

--It is one the intermediate method.

--this method takes a predicate object as an argument, and filter the stream based on the predicate condition, and returns the filtered elements in another stream object.

Eg:

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Stream;

**public** **class** Demo {

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

List<Stu> students=**new** ArrayList<>();

students.add(**new** Stu(10,"N1",750));

students.add(**new** Stu(12,"N2",850));

students.add(**new** Stu(13,"N3",450));

students.add(**new** Stu(14,"N4",550));

//from the above list get another list of students whose marks is less than 500.

Stream<Stu> str1=students.stream();

Stream<Stu> str2=str1.filter(s -> s.getMarks()<500);

str2.forEach(s -> System.***out***.println(s));

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

//short form

students.stream().filter(s -> s.getMarks()<500).forEach(s -> System.***out***.println(s));

}

}

Output:

Stu [roll=13, name=N3, marks=450]

======================

Stu [roll=13, name=N3, marks=450]

Eg2: list of marks. Marks are <500

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Collectors;

**public** **class** Demo {

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

List<Stu> students=**new** ArrayList<>();

students.add(**new** Stu(10,"N1",750));

students.add(**new** Stu(12,"N2",850));

students.add(**new** Stu(13,"N3",450));

students.add(**new** Stu(14,"N4",410));

//short form

List<Stu> filteredList=students.stream()

.filter(s -> s.getMarks()<500)

.collect(Collectors.*toList*());

System.***out***.println(filteredList);

}

}

Output:

[Stu [roll=13, name=N3, marks=450], Stu [roll=14, name=N4, marks=410]]

Map () method:

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

--It is also intermediate method.

--this method is used to transform the object.

--this method takes java.util.function.Function(I) object as an argument and map/ transfers the element to a new element and returns the mapped elements in another stream.

Example:

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Collectors;

**import** java.util.stream.Stream;

**public** **class** Demo {

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

List<Stu> students=**new** ArrayList<>();

students.add(**new** Stu(10,"N1",750));

students.add(**new** Stu(12,"N2",850));

students.add(**new** Stu(13,"N3",450));

students.add(**new** Stu(14,"N4",410));

Stream<Stu> str1=students.stream();

Stream<Stu> str2=str1.map(s -> {

Stu s1=**new** Stu(s.getRoll(),s.getName(),s.getMarks()+50);

**return** s1;

});

List<Stu> modifiedList=str2.collect(Collectors.*toList*());

System.***out***.println(modifiedList);

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

//short form

List<Stu> shortForm=students.stream()

.map(s -> **new** Stu(s.getRoll(), s.getName(), s.getMarks()+50))

.collect(Collectors.*toList*());

System.***out***.println(shortForm);

}

}

Output:

[Stu [roll=10, name=N1, marks=800], Stu [roll=12, name=N2, marks=900], Stu [roll=13, name=N3, marks=500], Stu [roll=14, name=N4, marks=460]]

========================

[Stu [roll=10, name=N1, marks=800], Stu [roll=12, name=N2, marks=900], Stu [roll=13, name=N3, marks=500], Stu [roll=14, name=N4, marks=460]]

Min and max method:

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

These methods are the terminal methods which will takes a Comparator object, using which we can decide min and max elements.

--this min () and max () method will return the minimum and maximum object in the form of java.util.Optional class object.

--this optional class introduced in java 1.8 and it is basically used to avoid the NullPointerException.

--to get the elements from this optional class, we need to call get () method.

Eg: min

Demo.java

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.Optional;

**import** java.util.stream.Stream;

**public** **class** Demo {

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

List<Stu> students=**new** ArrayList<>();

students.add(**new** Stu(10,"N1",750));

students.add(**new** Stu(12,"N2",850));

students.add(**new** Stu(13,"N3",450));

students.add(**new** Stu(14,"N4",410));

Stream<Stu> str1=students.stream();

Optional<Stu> opt=str1.min((s1,s2) -> s1.getMarks()>s2.getMarks()?1:-1);

Stu s=opt.get();

System.***out***.println(s);

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

Stu minStudent=students.stream()

.min((s1,s2) -> s1.getMarks()>s2.getMarks()?1:-1)

.get();

System.***out***.println(minStudent);

}

}

Eg2: count ()

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Demo {

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

List<Stu> students=**new** ArrayList<>();

students.add(**new** Stu(10,"N1",750));

students.add(**new** Stu(12,"N2",850));

students.add(**new** Stu(13,"N3",450));

students.add(**new** Stu(14,"N4",410));

**long** res=students.stream().filter(s -> s.getMarks()<500).count();

System.***out***.println(res);

}

}

Output:

2

allMatch () anyMatch (), nonmatch ():

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

Eg:

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Demo {

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

List<Stu> students=**new** ArrayList<>();

students.add(**new** Stu(10,"N1",750));

students.add(**new** Stu(12,"N2",850));

students.add(**new** Stu(13,"N3",450));

students.add(**new** Stu(14,"N4",410));

//anyMatch()

**boolean** res=students.stream().anyMatch(s -> s.getMarks() < 1000);

System.***out***.println(res);

//allMatch()

**boolean** res1=students.stream().allMatch(s -> s.getMarks() < 1000);

System.***out***.println(res1);

//noneMatch()

**boolean** res2=students.stream().noneMatch(s -> s.getMarks() < 1000);

System.***out***.println(res2);

}

}

Output:

true

true

false

**COLLECTIONS FRAMEWORK**:

Collection: if we want to represent a group of objects in a single unit as a single object.

--collection of objects

FRAMEWORK:

--the main objective of a framework is easy developer work.

--it is semi implemented architecture.

--A FW comprises some abstract design with built-in behaviour (functionality), in order to use it, we need to insert our functionality in various places of FW.

--A software FW is a universal, reusable SW (software) platform to develop software applications, products or solutions.

Limitations of array:

1. Size is fixed, we cannot increase or decrease it dynamically.
2. It supports homogeneous type of data/ elements.

--this limitation we can overcome by taking object class array.

3. array concept is not implemented based on ready made method support. For each and every requirement we need to write our own logic. Even for printing the elements from an array we need to write for loop.

--in each array object we have only one non-static variable i.e ‘length’.

--to overcome the above limitations we need to use collection FW.

Features of Collection FW:

--collections are growable and shrinkable in nature.

--collections can hold both homogeneous and heterogeneous type of elements.

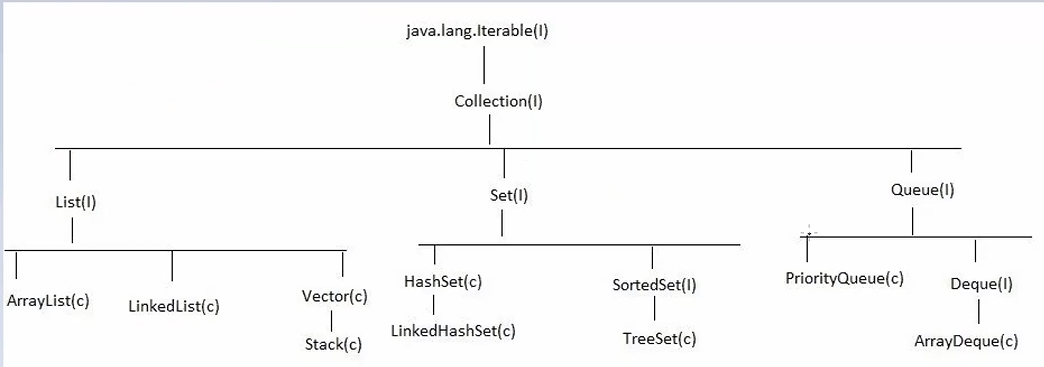
--every collection classes are implemented based on some standard data-structure, hence readymade method support is available for most of the requirements.

\*\*\*\*All (99%) the collection FW related classes and interfaces belongs to java.util package.

--collection FR having 2 sections:

1. normal collection (here we manage object uniformally/ individually/ singular manner).

2. Map (here we manage objects in key-value pair).



List: when we need to arrange the elements in a sequence (index-based manner) and duplicate elements are allowed.

Set: when we need uniqueness (duplicates are not allowed).

Queue: when we need to arrange the elements prior to processing (FIFO order is default, but we can manipulate).

Collection Interface:

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

--it is the foundation upon which the collection FR is built.

--it declares some of the core methods that all collection classes will have.

--these methods we can call on any collection FR related classes objects.

Some of the methods of collection interface:

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

--int size (); // how many elements are there in that collection object.

--boolean isEmty();

--boolean contains (object obj); //for searching any element.

--Iterator iterator ();

--Object[] toArray();

--T[] toArray(T[]); //T means type integer, float etc. (convert only limited size element to array).

--boolean add(E); // it is a most commonly used method, to add any element inside any type of collection.

--boolean remove(Object obj);

--boolean containsAll(Collection col);

--boolean addAll(Collection col); // copy all the element from one collection to our collection object.

--boolean removeAll(Collection col);

--boolean retainAll(Collection col); //except supplied collection elements all other will be removed.

--void clear(); //remove all elements from the collection.

//these following 2 methods belongs to object class

--boolean equals(Object obj); //compare 2 collection object.

--int hashCode(); //return the hashCode of collection obj.

//these methods introduced in java 1.8 to perform functional programming.

--boolean removeIf(java.util.function.Predicate<?super E>);

--java.util.Spliterator<E> spliterator();

-- java.util.Stream<E> stream();

-- java.util.stream.Stream<E> parallelStream();

LIST:

🡪it is the child interface of collection interface and declares the behaviour of a collection to preserve the sequence of an element.

🡪sequence will be maintained.

🡪duplicates are allowed.

--in addition to the collection interface methods, list interface defines some of its own methods also.

--public Object get(int index);

--public Object set(int index, object newobj); //it will return the override object.

--public Object remove(int index);

--public int indexOf(Object obj);

ArrayList:

--it is the implementation of the list interface.

--it is basically a dynamic array (it dynamically increase and decrease in size).

--ArrayList class is the best choice if our frequent operation is retrieval based on index.

--duplicates are allowed.

--null insertion is possible (multiple time also).

Eg:

ArrayList al=new ArrayList();

System.out.println(al); // []

Note:

All the collection classes has overridden the toString() method internally. So they will print the elements inside that collection in [] square bracket.

--all the collection classes are like a container or bag which holds multiple objects.

--int the above example/ statement we have created an empty AL object with the default initial capacity 10.

--once AL reaches to its max capacity then a new AL object will be created in the memory automatically with the new capacity using following formula:

\* newCapacity = (currentCapacity \* 3/2) +1;

ArrayList al=new ArrrayList(1000); //Al created with the initial capacity 1000.

Auto-Boxing && Auto-Unboxing:

Boxing: converting primitives into the object (box) it is known as boxing and reverse is called unboxing.

--8 primitive datatypes:

--for each primitive datatypes we have corresponding wrapper classes are there.

1. int – java.lang.Integer

2. byte – java.lang.Byte

3. char – java.lang.Character

4. Boolean – java.lang.Boolean

5. short—java.lang.Short

6. long – java.lang.Long

7. float – java.lang.Float

8. double – java.lang.Double

--before java 1.5 in order to add the primitives in the collection we need to wrap that primitives to their corresponding wrapper class object.

Example:

int i=10;

Integer i1=Integer.valueOf(i); //boxing

int x=i1.intValue();

converting array to array list List.of() and Arrays.asList()

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

--any change made to the array it reflects on Arrays.asList() but it does not reflect on List.of()

list.add(34);

System.***out***.println(list);//It gives exception.

//we cannot perform operations on List.of().

Eg:

Integer[] array = **new** Integer[]{1, 2, 3, 4};

List<Integer> list = Arrays.*asList*(array);

Example:

--------

**public** **class** Example{

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

Integer[] array = **new** Integer[]{1, 2, 3, 4};

List<Integer> lst=Arrays.*asList*(array);//any change made to the array it reflects on arrays.asList() but it does not reflect on List.of()

List<Integer> list = List.*of*(array);

array[2]=100;

System.***out***.println(lst);

System.***out***.println(list);

}

}

Output:

[1, 2, 100, 4]

[1, 2, 3, 4]

Example:

**public** **class** Example{

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

List<Integer> lst=List.*of*(1,2,3);

System.***out***.println(lst);

}

}

Output: [1,2,3]

Example:

**public** **class** Example{

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

List<Integer> lst=Arrays.*asList*(1,2,3);

System.***out***.println(lst);

}

}

Output: [1,2,3]

Auto-boxing and un-boxing:

List:

**import** java.util.ArrayList;

**public** **class** Demo{

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

{

ArrayList<Integer> al=**new** ArrayList<>();

System.***out***.println("empty list:"+al);

//al.add(null); // we can add null value to the integer object but iteration is not possible and it give null pointer exception.

//al.add(null);

al.add(1); //auto-boxing: here al is a object we are adding it to primitive type int values.

al.add(2);

System.***out***.println("after adding elements:"+al);

System.***out***.println("for loop for iterating");

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

{

System.***out***.println(al.get(i));

}

System.***out***.println("for each loop for iterating");

**for**(**int** s:al) //Auto-unboxing: al is a object of integer type we are converting it into primitive type int.

{

System.***out***.println(s);

}

}

}

Output:

empty list: []

after adding elements: [1, 2]

for loop for iterating

1

2

for each loop for iterating

1

2

Iterator using while loop

Example:

**import** java.util.ArrayList;

**public** **class** Demo{

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

{

ArrayList al=**new** ArrayList();

al.add("India");

al.add("Delhi");

al.add("switzerland");

al.add("Landon");

al.add(**new** A()); //class is alive

al.add(**new** Student(101,"sita",567));

al.add(**null**);

al.add(**null**);

al.add(120); //integer

al.add(**true**); // boolean

al.add(10.55); // double

System.***out***.println(al);

System.***out***.println(al.size());

Object obj=al.get(2);

String s=(String)obj; //down casting

System.***out***.println(s);

System.***out***.println(obj);

//Object obj=al.get(2);

**int** i=(**int**)al.get(8);

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

Object obj1=al.get(5);

System.***out***.println(obj1);

}

}

Output:

inside the constructor of A

[India, Delhi, switzerland, Landon, com.mic.A@3ac3fd8b, com.mic.Student@5594a1b5, null, null, 120, true, 10.55]

11

switzerland

switzerland

120

com.mic.Student@5594a1b5

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

--in the above application our ArrayList object is not a type safe collection.

--if our collection is not type safe collection then we can add any type of object at any position inside our collection.

--here while getting the elements from the type unsafe collection every time we need to downcast the element, which is not feasiable. There might be a chance of ClassCastException.

--so in real time, our collection should be type safe collection.

--type safe collection means making our collection homogeneous.

Benefits of type safe collection:

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

1. If we try to add any other type of element then compiler will stop at compile time.
2. We will get rid of down casting problem.

--the above type safe collection concept is called Generics concept.

ArrayList<Object> al=**new** ArrayList<>();

--taking the type as Object is similar to creating type-unsafe collection.

Example:

**import** java.util.ArrayList;

**import** java.util.Iterator;

**public** **class** Demo{

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

{

ArrayList<String> al=**new** ArrayList<>();

System.***out***.println("empty list:"+al);

al.add(**null**);

al.add(**null**);

al.add("A");

al.add("B");

System.***out***.println("after adding elements:"+al);

System.***out***.println("for loop for iterating");

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

{

System.***out***.println(al.get(i));

}

System.***out***.println("for each loop for iterating");

**for**(String s:al)

{

System.***out***.println(s);

}

System.***out***.println("Iterator using while loop");

Iterator<String> itr=al.iterator();

**while**(itr.hasNext())

{

String str=itr.next();

System.***out***.println(str);

}

}

}

Output:

empty list: []

after adding elements: [null, null, A, B]

for loop for iterating

null

null

A

B

for each loop for iterating

null

null

A

B

Iterator using while loop

null

null

A

B

\*\*//ArrayList class pseudo code before generics

Class ArrayList implements List {

Public Boolean add (Object obj) {

//it will add the object obj to the AL

}

Public int get (int index) {

//it will return obj to the specified index

}

//remaining methods

}

\*\*// ArrayList class pseudo code after generics

Class ArrayList<T> implements List {

Public Boolean add (T t) {

//it will add the object obj to the AL

}

Public T get (int index) {

//it will return obj to the specified index

}

//remaining methods

}

EXAMPLE:

**import** java.util.ArrayList;

**import** java.util.Scanner;

**public** **class** Demo{

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

{

ArrayList<Student> students=**new** ArrayList<>();

Scanner sc=**new** Scanner(System.***in***);

**int** count=1;

**while**(**true**)

{

System.***out***.println("Enter details of student:"+(count++));

System.***out***.println("enter roll:");

**int** roll=sc.nextInt();

System.***out***.println("Enter name:");

String name=sc.next();

System.***out***.println("Enter marks:");

**int** marks=sc.nextInt();

Student student=**new** Student(roll,name,marks);

students.add(student);

System.***out***.println("student object added successfully");

System.***out***.println("want more (y/n)?");

String choice=sc.next();

**if**(choice.equalsIgnoreCase("n"))

**break**;

}

//enhanced for loop (for-each)

**for**(Student stu:students)

{

System.***out***.println("roll is:"+stu.getRoll());

System.***out***.println("name is:"+stu.getName());

System.***out***.println("marks is:"+stu.getMarks());

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

}

//normal for loop

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

{

Student stu=students.get(i);

System.***out***.println("Roll is:"+stu.getRoll());

System.***out***.println("name is:"+stu.getName());

System.***out***.println("marks is:"+stu.getMarks());

}

}

}

Output:

Enter details of student:1

enter roll:

101

Enter name:

siri

Enter marks:

768

student object added successfully

want more (y/n)?

n

roll is:101

name is:siri

marks is:768

===========================

Note: from the list object we can get elements one by one by using following approaches also:

1. By using Iterator object
2. By using ListIterator object
3. By using forEach() method
4. By using Stream api

Where as from Set and Queue:

--we can not use normal for loop we can only use:

1. enhanced for loop

2. by using Iterator

3. by using forEach() method

4. by using Stream api

**LINKED LIST**:

--It is also one of the implementation of list interface.

--LinkedList class from java 1.5 onwards also implements deque interface.

--LinkedList class is the best choice if our frequent operation is insertion or deletion from the middle.

--LinkedList class also follows the properties of list and deque (preserve the sequence and index concept)

\*\*with the linked list if we delete or insert elements then too much shifting operation is not required.

--In Java Linked List is implemented using Doubly Linked List data

**Example:**

**import** java.util.ArrayList;

**import** java.util.LinkedList;

**import** java.util.Scanner;

**public** **class** Demo{

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

{

LinkedList<Integer> ll=**new** LinkedList<>();

ll.add(10);

ll.add(20);

ll.add(30);

ll.add(40);

ll.add(40);

ll.add(**null**);

ll.add(**null**);

System.***out***.println(ll);

// for(int x:ll)//here we will get null pointer exception

// {

// System.out.println(x);

// }

**for**(Integer y:ll)//no error

{

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

}

}

}

Output:  
[10, 20, 30, 40, 40, null, null]

10

20

30

40

40

null

null

**VECTOR**:

--It is also similar to ArrayList class, with following difference.

1. AL introduced in java 1.2 where as vector class is a legacy collection class introduced in java 1.0 version.

2. where AL reaches to the maximum capacity the new AL object will be created internally with new capacity.

newCapacity = (currentCapacity \* 3/2) +1

where as when vector class reaches to its maximum capacity then a new vector object is created in the memory by double capacity.

3.Most of the methods of AL class is non-synchronized (not thread safe) where as most of the methods of vector class is synchronized, i.e, thread-safe.

--Al will give better and fast performance compare to vector class.

Eg:

totalTicket=10;

public synchronized void bookTicket(int numberOfTicket)

{

If(numberOfTicket <= totalTicket)

{

totalTicket-=numberOfTicket;

}

}

Public synchronized void viewAvailability(){

}

--until we have specific requirement we should use synchronized keyword.

--we have an option to make our AL object methods as synchronized.

Java.util.Collection(I): root interface of Collection FW

Java.util.Collection (c): utility class

Collections.synchronizedList(al); //it will convert the AL to the synchronized list (thread safe object).

Vector ---bleck & white TV

AL ----colour TV ---reduce the colour

Example of vector:

**import** java.util.Vector;

**public** **class** Demo{

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

{

Vector<Integer> ll=**new** Vector<>();

ll.add(10);

ll.add(20);

ll.add(30);

ll.add(40);

ll.add(40);

ll.add(**null**);

ll.add(**null**);

System.***out***.println(ll);

// for(int x:ll)//here we will get null pointer exception

// {

// System.out.println(x);

// }

**for**(Integer y:ll)//no error

{

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

}

}

}

Output:

[10, 20, 30, 40, 40, null, null]

10

20

30

40

40

null

null

**STACK**:

--All the operation of the stack we can perform with the help of Linked List also.

Example:

**import** java.util.Stack;

**public** **class** Demo{

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

{

Stack<Integer> st=**new** Stack<>();

st.add(10);

st.add(20);

st.add(30);

st.add(40);

st.add(40);

st.add(**null**);

st.add(**null**);

System.***out***.println(st);

System.***out***.println(st.pop());

System.***out***.println(st);

}

}

Output:

[10, 20, 30, 40, 40, null, null]

null

[10, 20, 30, 40, 40, null]

\*\*the different possibilities to write a collection

ArrayList<String> al=new ArrayList<>(): //too specific

List<String> al=new ArrayList<>(); // recommended approach

Collection<String> al=new ArrayList<>(); //

Object al=new ArrayList<>():

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

Example:

Student.java

**public** **class** Student {

**private** **int** roll;

**private** String name;

**private** **int** marks;

**public** Student() {

}

**public** Student(**int** roll, String name, **int** marks) {

**this**.roll = roll;

**this**.name = name;

**this**.marks = marks;

}

**public** **int** getRoll() {

**return** roll;

}

**public** **void** setRoll(**int** roll) {

**this**.roll = roll;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getMarks() {

**return** marks;

}

**public** **void** setMarks(**int** marks) {

**this**.marks = marks;

}

}

Demo.java

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Demo{

**public** ArrayList<Student> getStudent()

{

ArrayList<Student> stu=**new** ArrayList<>();

stu.add(**new** Student(10,"n1",850));

stu.add(**new** Student(12,"n2",750));

stu.add(**new** Student(14,"n3",550));

**return** stu;

}

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

{

Demo d=**new** Demo();

List<Student> stu=d.getStudent();

**for**(Student student:stu)

{

System.***out***.println(student);

//System.out.println("roll is:"+student.getRoll());

//System.out.println("name is:"+student.getName());

//System.out.println("marks is:"+student.getMarks());

}

}

}

Output:

com.mic.Student@1d251891

com.mic.Student@2133c8f8

com.mic.Student@43a25848

**Iterator and ListIterator: (I)**

--Iterator interface belongs to java.util package.

--Using this iterator we can interate each elements from any collection one by one in a standard manner.

--Iterator is also known as universal cursor. Because it is applicable with any type of collection classes.

--using Iterator, while iterating elements from any collection, we can remove the elements also.

Example:

**import** java.util.Arrays;

**import** java.util.Iterator;

**import** java.util.List;

**public** **class** IteratorEx {

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

List<String> list=Arrays.*asList*("a","b","c");

Iterator<String> itr=list.iterator();

**while**(itr.hasNext())

{

String s=itr.next();

System.***out***.println(s);

}

}

}

Output:

a

b

c

**ListIterator:**

--It is the child interface of the Iterator interface.

--It is the most powerful cursor in collection framework with only one limitation, with ListIterator we can work with only list category. We can work with set, queue.

--with this ListIterator we can navigate both side (forward and backword).

--It is a bi-directional cursor.

Example:

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.ListIterator;

**public** **class** ListInteratorEx {

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

{

List<String> list=Arrays.*asList*("a","b","c","d");

ListIterator<String> litr=list.listIterator();

**while**(litr.hasNext())

{

System.***out***.println(litr.next());

}

System.***out***.println(list);

}

}

Output:

a

b

c

d

[a, b, c, d]

**HashSet**:

🡪it is the first implementation of the set interface.

🡪elements are added based on their hash code.

🡪there is no sequence.

🡪duplicate will not be allowed.

🡪\*\*HashSet class is the best choice, if our frequent operation is searching.

🡪searching a particular element based on hash code will have time complexity O (1).

🡪here an empty HS object is created with the initial capacity 16 and the default load factor (fill ratio) is 0.75.

🡪here fill ratio means after completion of 75% the new HS object will be created in the memory.

Eg:

HashSet<Integer> hs=new HashSet<>(1000,0.8f);

--here initial capacity is 1000 and once reaches to the 80% then a new HS object is created in the memory.

Example:

**import** java.util.HashSet;

**import** java.util.Iterator;

**public** **class** Demo{

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

{

HashSet<String> al=**new** HashSet<>();

System.***out***.println("empty set:"+al);

al.add(**null**);

al.add(**null**);

al.add("A");

al.add("B");

System.***out***.println("after adding elements:"+al);

System.***out***.println("for each loop for iterating");

**for**(String s:al)

{

System.***out***.println(s);

}

System.***out***.println("Iterator using while loop");

Iterator<String> itr=al.iterator();

**while**(itr.hasNext())

{

String str=itr.next();

System.***out***.println(str);

}

}

}

Output:

empty set:[]

after adding elements:[null, A, B]

for each loop for iterating

null

A

B

Iterator using while loop

null

A

B

Note: to access the elements one by one from the HS class we cannot use normal for loop, but we can use enhanced for loop.

Eg:

for (Integer ir:hs)

{

System.out.println(ir);

}

Note: HashSet is very much related with HashMap, it internally uses the HashMap to store the element.

Object Equality:

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

--equals() method belongs to object class.

Public Boolean equals (Object obj);

--this method is implemented inside the object class as follows:

Public Boolean equals (Object obj)

{

if(obj==this)

return true;

else

return false;

}

Or

return obj==this;

--in order to make our class objects logically equal we need to override the above equals() method from the object class to our student class.

Eg:

Demo.java

**public** **class** Demo{

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

{

Student s1=**new** Student(10,"n1",750);

Student s2=**new** Student(10,"n1",750);

System.***out***.println(s1.hashCode());

System.***out***.println(s2.hashCode());

}

}

Output:

488970385

557041912

Student.java

@Override

**public** **boolean** equals(Object obj) {

Student s1=**this**;

Student s2=(Student)obj;

// if(s1.getRoll() ==s2.getRoll() && s1.getName().equals(s2.getName())&& s1.getMarks()==s2.getMarks())

// return true;

// else

// return false;

**return** (s1.getRoll() ==s2.getRoll() && s1.getName().equals(s2.getName())&& s1.getMarks()==s2.getMarks());

}

\*\*\*\*\*\*\*\*\*\*\*\*after writing this

@Override

**public** **int** hashCode() {

**return** roll;

}

The output for Demo after hashCode() method

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

10

10

Note: this equals() method has a best friend called hashCode() method, it is also defined inside the object class.

Public int hashCode();

---equals() and hashCode method is like a contract, if we override the equals() method to make our objects logically equal then we have to override the hashCode() method also.

--if we call equals() method to compare 2 objects and if it returns true then those objects hashCode value should also be same.

Eg:

**public** **class** Demo{

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

{

A a1=**new** A();

A a2=**new** A();

System.***out***.println(a1.hashCode());

System.***out***.println(a2.hashCode());

}

}

Output:

inside the constructor of A

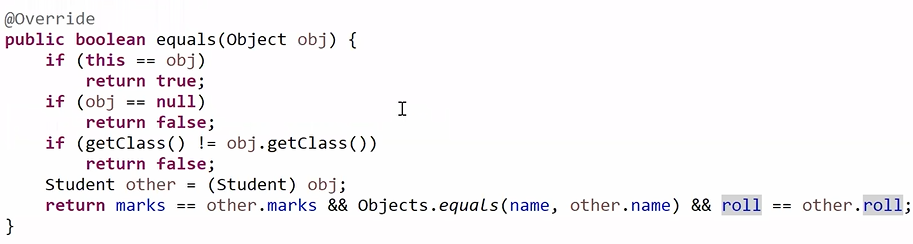
inside the constructor of A

557041912

1134712904

Override equals() method:

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



Override hashCode() method:

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

@Override

**public** **int** hashCode() {

**return** Objects.hash(marks,name,roll);

}

--if for 2 objects will returns true then their hashCode value Should return the same value for boyh objects.

Basically 3 cases where we need to override equals and hashCode method inside our class.

1. Whenever we try to make our objects logically equal.
2. Whenever we try to add our objects inside the HashSet or LinkedHashSet class.
3. Whenever we try to add our objects as a key inside the HashMap class.

**LinkedHashSet**:

--it is the child class of HashSet class, it also does not allow duplicate, but it maintains the insertion order.

Note: In collection FR all the collection classes are mutually inclusive. i.e. we can convert any collection classes to any other collection class very easily.

Example: converting array list to hash set

**import** java.util.ArrayList;

**import** java.util.LinkedHashSet;

**public** **class** Demo{

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

{

ArrayList<Integer> al=**new** ArrayList<>();

al.add(10);

al.add(20);

al.add(30);

al.add(10);

al.add(20);

System.***out***.println(al);

LinkedHashSet<Integer> hs=**new** LinkedHashSet<>();

**for**(**int** il:al)

{

hs.add(il);

}

System.***out***.println(hs);

//converting hash set to array list

al=**new** ArrayList<>();

}

}

Output:

[10, 20, 30, 10, 20]

[10, 20, 30]

By using Easy way:

**import** java.util.ArrayList;

**import** java.util.LinkedHashSet;

**public** **class** Demo{

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

{

ArrayList<Integer> al=**new** ArrayList<>();

al.add(10);

al.add(20);

al.add(30);

al.add(10);

al.add(20);

System.***out***.println(al);

LinkedHashSet<Integer> hs=**new** LinkedHashSet<>(al);

System.***out***.println(hs);

al=**new** ArrayList<>(hs);

System.***out***.println(al);

}

}

Output:

[10, 20, 30, 10, 20]

[10, 20, 30]

[10, 20, 30]

Program: remove duplicate elements from the string

**import** java.util.LinkedHashSet;

**public** **class** Demo{

**public** String removeDuplicateFromString(String original)

{

**char**[] ch=original.toCharArray();

LinkedHashSet<Character> lhs=**new** LinkedHashSet<>();

**for**(**char** chr:ch)

{

lhs.add(chr);

}

StringBuilder br=**new** StringBuilder("");

**for**(**char** ch1:lhs)

br.append(ch1);

**return** br.toString();

}

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

{

Demo d=**new** Demo();

String str=d.removeDuplicateFromString("sirisha");

System.***out***.println(str);

}

}

Output:

sirha

Remove duplicates from list:

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

**import** java.util.ArrayList;

**import** java.util.LinkedHashSet;

**import** java.util.List;

**public** **class** Demo{

**public** List<String> removeDuplicateFromString(List<String> cities)

{

LinkedHashSet<String> lhs=**new** LinkedHashSet<>(cities);

ArrayList<String> lst=**new** ArrayList<>(lhs);

**return** lst;

//return new ArrayList<>(lhs);

}

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

{

Demo d=**new** Demo();

List<String> cities=**new** ArrayList<>();

cities.add("delhi");

cities.add("delhi");

cities.add("mumbai");

cities.add("chenni");

List<String> ls=d.removeDuplicateFromString(cities);

System.***out***.println(ls);

}

}

Output:

[delhi, mumbai, chenni]

**TreeSet**:

--It has the nature of Collection, set, SortedSet.

--In java TreeSet class implemented by using balanced tree data structure.

--we use the Treeset class, when we want to arrange our object in sorted (natural sorting order).

--duplicates are not allowed.

--Insertion order is not preserved.

--Even a single null also not allowed, if we try to adda a null value then at runtime we get a NullPointerException.

Example1:

**import** java.util.TreeSet;

**public** **class** Demo{

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

{

Demo d=**new** Demo();

TreeSet<Integer> ts=**new** TreeSet<>();

ts.add(2);

ts.add(10);

ts.add(15);

ts.add(5);

ts.add(50);

System.***out***.println(ts);

}

}

Output:

[2, 5, 10, 15, 50]

Example2:

**import** java.util.TreeSet;

**public** **class** Demo{

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

{

Demo d=**new** Demo();

TreeSet<String> cities=**new** TreeSet<>();

cities.add("delhi");

cities.add("delhi");

cities.add("mumbai");

cities.add("chenni");

cities.add("chandigarh");

System.***out***.println(cities);

}

}

Output:

[chandigarh, chenni, delhi, mumbai]

Note:

If we try to add any element inside the TreeSet object, then those elements should be comparable.

i.e That object class should implements “java.lang.Comparable” interface. Otherwise we will get a ClassCastException at run time.

NOTE: all the wrapper classes and String class implements comparable interface.

**Comparable interface:**

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

--this interface belongs to java.lang package.

--by using this interface, we will specify the sorting rules/ technique inside the class.

--this interface is having only one abstract method:

Public int compareTo(Object obj);

--if we try to add any class object inside the TreeSet, then that class must comparable interface and override this compareTo method, and inside this method we need to specify the sorting technique of that class object.

Example:

Public int compareTo(Object o)

{

//in this method, we need to specify the sorting rules.

//this method internally called by the add() method of TreeSet object, when we try to add any element.

//s1.compareTo (s2);

//if s1 is bigger than s2 it returns +1

//if s2 is bigger than s1 it returns -1

//if both s1 and s2 are equal then it returns 0;

}

Example:

Student.java

**public** **class** Student **implements** Comparable {

**private** **int** roll;

**private** String name;

**private** **int** marks;

**public** Student() {

}

**public** Student(**int** roll, String name, **int** marks) {

**this**.roll = roll;

**this**.name = name;

**this**.marks = marks;

}

**public** **int** getRoll() {

**return** roll;

}

**public** **void** setRoll(**int** roll) {

**this**.roll = roll;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getMarks() {

**return** marks;

}

**public** **void** setMarks(**int** marks) {

**this**.marks = marks;

}

@Override

**public** String toString()

{

**return** "Student [roll:"+roll+",name:"+name+", marks:"+marks+"]";

}

@Override

**public** **int** compareTo(Object o)

{

Student s1=**this**;

Student s2=(Student)o;

**if**(s1.getMarks()>s2.getMarks())

**return** +1;

**else** **if**(s1.getMarks()<s2.getMarks())

**return** -1;

**else**

**return** 0;

}

}

Demo.java

**import** java.util.TreeSet;

**public** **class** Demo{

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

{

TreeSet<Student> ts=**new** TreeSet<>();

ts.add(**new** Student(10,"n1",678));

ts.add(**new** Student(12,"n3",878));

ts.add(**new** Student(14,"n2",978));

ts.add(**new** Student(16,"n4",778));

System.***out***.println(ts);

}

}

Output:

[Student [roll:10, name: n1, marks:678], Student [roll:16, name: n4, marks:778], Student [roll:12, name: n3, marks:878], Student [roll:14, name: n2, marks:978]]

--the above logic is based on the marks.

NOTE: in order to check the object uniqueness

HashSet and LinkedHashSet class uses equals () and hashCode () method.

Whereas Treeset class uses compareTo () method. If compareTo () returns 0.

Above example using Generics:

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

**public** **class** Student **implements** Comparable<Student> {

@Override

**public** **int** compareTo(Student s) {

**if**(**this**.getMarks()>s.getMarks())

**return** +1;

**else**

**return** -1;

}

}

Example: sort the student based on name

**public** **class** Student **implements** Comparable<Student> {

@Override

**public** **int** compareTo (Student s) {

**return** **this**.getName().compareTo(s.getName());

}

}

Reverse order:

@Override

**public** **int** compareTo (Student s) {

**return** s.getName().compareTo(**this**.getName());

}

If marks are same then sort them according to their name:

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

**public** **class** Student **implements** Comparable<Student> {

@Override

**public** **int** compareTo(Student s) {

**if**(**this**.getMarks()>s.getMarks())

**return** +1;

**else** **if**(**this**.getMarks()<s.getMarks())

**return** -1;

**else**

**return** **this**.getName().compareTo(s.getName());

}

}

**Java.util.Comparator(I)**:

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

--by using interface, we can define the sorting technique for a class object from outside of that class.

NOTE: if we want to define a sorting rule of a class object inside the same class then we should use java.lang.Comparable interface.

--whereas if we want to define the sorting logic outside of that class then we need to use java.util.Comparator interface.

--with the help of Comparable we can define only one sorting logic, whereas with the help of Comparator we can define multiple sorting logic.

--this Comparator interface has only one abstract method.

public int compare(Object obj1, Object obj2);

--to use the Comparator:

Step1: here we need not pollute the Student class (java bean class) by implementing Comparable interface.

Step2: create a separate class by any name and implements the Comparator interface and define the sorting logic by overriding the compare(---) method.

Example:

StudentMarksComp.java

**package** com.mic;

**import** java.util.Comparator;

**public** **class** StudentMarksComp **implements** Comparator {

@Override

**public** **int** compare(Object o1, Object o2) {

Student s1=(Student)o1;

Student s2=(Student)o2;

**if**(s1.getMarks() > s2.getMarks())

**return** +1;

**else** **if**(s1.getMarks() < s2.getMarks())

**return** -1;

**else**

**return** 0;

}

}

Comparator using Generics:

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

**import** java.util.Comparator;

**public** **class** StudentMarksComp **implements** Comparator<Student> {

@Override

**public** **int** compare(Student s1,Student s2) {

**if**(s1.getMarks() > s2.getMarks())

**return** +1;

**else** **if**(s1.getMarks() < s2.getMarks())

**return** -1;

**else**

**return** 0;

}

}

Step 3:

--create the above StudentMarksComp class object and pass that object to the constructor of the TreeSet class.

Example:

Demo.java

**import** java.util.TreeSet;

**public** **class** Demo{

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

{

StudentMarksComp mcom=**new** StudentMarksComp();

TreeSet<Student> ts=**new** TreeSet<>(mcom);

ts.add(**new** Student(10,"n1",678));

ts.add(**new** Student(12,"n3",878));

ts.add(**new** Student(14,"n2",978));

ts.add(**new** Student(16,"n4",778));

System.***out***.println(ts);

}

}

**import** java.util.Comparator;

**public** **class** StudentMarksComp **implements** Comparator<Student> {

@Override

**public** **int** compare(Student s1,Student s2) {

**return** s1.getName().compareTo(s2.getName());

}

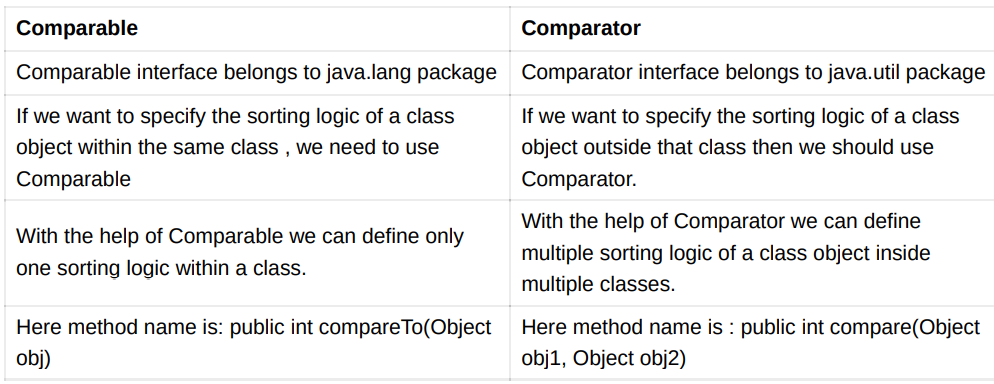
}

Output:

[Student [roll:10,name:n1, marks:678], Student [roll:16,name:n4, marks:778], Student [roll:12,name:n3, marks:878], Student [roll:14,name:n2, marks:978]]

Difference between comparable and comparator:

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



Utility or tools classes in collection FW:

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

1. Java.util.Array class: It is mostly used for performing utility operation on the normal arrays.

For example: sorting, searching, reversing, converting to list, printing the arrays elements etc.

1. java.util.Collection class: it is mostly used to perform utility operations for the collection FR related classes.

java.util.Arrays:

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

--inside this class we have various static methods are there by using which we can perform various operations on the normal array object.

Eg:

printing the elements from array:

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

**import** java.util.Arrays;

**public** **class** Demo{

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

{

**int**[] arr= {12,23,34,45,56,67};

// String result=Arrays.toString(arr);

// System.out.println(result);

System.***out***.println(Arrays.*toString*(arr));

}

}

Output:

[12, 23, 34, 45, 56, 67]

Sorting an array:

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

**import** java.util.Arrays;

**public** **class** Demo{

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

{

**int**[] arr= {12,23,34,45,56,67,32,15};

System.***out***.println(Arrays.*toString*(arr));

Arrays.*sort*(arr);

System.***out***.println(Arrays.*toString*(arr));

}

}

Output:

[12, 23, 34, 45, 56, 67, 32, 15]

[12, 15, 23, 32, 34, 45, 56, 67]

Searching an element in array:

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

--first of all our array should be sorted, otherwise we may not get correct result.

--if the value is found then we get index value otherwise we get the negative value.

Example:

**import** java.util.Arrays;

**public** **class** Demo{

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

{

**int**[] arr= {12,23,34,45,56,67,32,15};

Arrays.*sort*(arr);

System.***out***.println(Arrays.*toString*(arr));

**int** index=Arrays.*binarySearch*(arr,34);

System.***out***.println("index:"+index);

}

}

Output:

[12, 15, 23, 32, 34, 45, 56, 67]

index:4

converting list in easiest manner: Arrays.asList();

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

**import** java.util.Arrays;

**import** java.util.List;

**public** **class** Demo{

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

{

//int[] arr= {12,23,34,45,56,67,32,15};

List<String> lst=Arrays.*asList*("delhi","mumbai","chennai","kolkata");

List<Integer> lst1=Arrays.*asList*(10,20,30,40);

List<Student> stu=Arrays.*asList*(

**new** Student(10,"n1",780),

**new** Student(12,"n2",880),

**new** Student(14,"n3",980)

);

}

}

Collections class:

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

--this class also defines some of the static methods to perform some utility operations on the Collection FW related classes.

Collections.sort:

**import** java.util.Arrays;

**import** java.util.Collections;

**import** java.util.List;

**public** **class** Demo{

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

{

List<String> lst=Arrays.*asList*("delhi","mumbai","chennai","kolkata");

Collections.*sort*(lst);

System.***out***.println(lst);

}

}

Output:

[chennai, delhi, kolkata, mumbai]

\*\*Example:

Student.java:

**public** **class** Student **implements** Comparable<Student> {

**private** **int** roll;

**private** String name;

**private** **int** marks;

**public** Student() {

}

**public** Student(**int** roll, String name, **int** marks) {

**this**.roll = roll;

**this**.name = name;

**this**.marks = marks;

}

**public** **int** getRoll() {

**return** roll;

}

**public** **void** setRoll(**int** roll) {

**this**.roll = roll;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getMarks() {

**return** marks;

}

**public** **void** setMarks(**int** marks) {

**this**.marks = marks;

}

@Override

**public** String toString()

{

**return** "Student [roll:"+roll+",name:"+name+", marks:"+marks+"]";

}

@Override

**public** **int** compareTo(Student s) {

**if**(**this**.getMarks()>s.getMarks())

**return** +1;

**else** **if**(**this**.getMarks()<s.getMarks())

**return** -1;

**else**

**return** **this**.getName().compareTo(s.getName());

}

}

StudentMarksComp.java:

**import** java.util.Comparator;

**public** **class** StudentMarksComp **implements** Comparator<Student> {

@Override

**public** **int** compare(Student s1,Student s2) {

**return** s1.getName().compareTo(s2.getName());

}

}

Demo.java

**import** java.util.Arrays;

**import** java.util.Collections;

**import** java.util.List;

**public** **class** Demo{

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

{

List<Student> stu=Arrays.*asList*(

**new** Student(10,"n1",780),

**new** Student(12,"n2",880),

**new** Student(14,"n3",980)

);

Collections.*sort*(stu,**new** StudentMarksComp());

System.***out***.println(stu);

}

}

Output:

[Student [roll:10,name:n1, marks:780], Student [roll:12,name:n2, marks:880], Student [roll:14,name:n3, marks:980]]

Getting number of occurrences in a list:

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

**import** java.util.Arrays;

**import** java.util.Collections;

**import** java.util.List;

**public** **class** Demo{

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

{

List<Integer> lst=Arrays.*asList*(10,20,30,40,10);

**int** result=Collections.*frequency*(lst,10);

System.***out***.println(result);

}

}

Output: 2

Converting list to the synchronized list:

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

**import** java.util.Arrays;

**import** java.util.Collections;

**import** java.util.List;

**public** **class** Demo{

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

{

List<Integer> lst=Arrays.*asList*(10,20,30,40,10);

List<Integer> l=Collections.*synchronizedList*(lst);

System.***out***.println(l);

}

}

Output:

[10, 20, 30, 40, 10]

**Queue:**

--if we want to represent a group of object prior to processing (before processing we want to arrange the object) then we should use the queue.

Eg:

Sending mail or SMS to multiple people.

--Queue follows FIFO (first in first out) order but based on our requirement we can implement our own order also by using Priority Queue class.

**PriorityQueue:**

--Here null insertion is not possible even a single null also.

--inside the PriorityQueue if we want any elements then that element should be Comparable, (or we can use Comparator) otherwise we get ClassCastException.

Note:

Inside the Queue the elements will not be as sorted as inside the TreeSet, but by using its peek() and poll() method, remove() method it will work according to the Comparable or Comparator(Sorting order).

**NOTE:**

By using priority queue we can prioritise the order. By using Linked List the results follow the FIFO order.

Example1:

**import** java.util.PriorityQueue;

**import** java.util.Queue;

**public** **class** PriorityQueueEx {

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

{

Queue<Integer> q=**new** PriorityQueue<>();

q.add(10);

q.add(20);

q.add(12);

q.add(30);

q.add(22);

System.***out***.println(q);//without sorting

Integer i;

**while**((i=q.poll())!=**null**)

{

//after using method all are sorted

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

}

}

}

Output:

[10, 20, 12, 30, 22]

//sorting method done internally

10

12

20

22

30

Example2:

**import** java.util.Comparator;

**import** java.util.PriorityQueue;

**import** java.util.Queue;

**class** Student1{

**private** **int** roll;

**private** String name;

**private** **int** marks;

**public** Student1() {

}

**public** Student1(**int** roll, String name, **int** marks) {

**this**.roll = roll;

**this**.name = name;

**this**.marks = marks;

}

**public** **int** getRoll() {

**return** roll;

}

**public** **void** setRoll(**int** roll) {

**this**.roll = roll;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getMarks() {

**return** marks;

}

**public** **void** setMarks(**int** marks) {

**this**.marks = marks;

}

}

**class** StudentMarksComp **implements** Comparator<Student1>{

@Override

**public** **int** compare(Student1 s1, Student1 s2) {

**return** s1.getMarks()>s2.getMarks()? +1:-1;

}

}

**public** **class** PriorityQueueAdv {

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

Queue<Student1> q=**new** PriorityQueue<>(**new** StudentMarksComp());

q.add(**new** Student1(10,"N1",500));

q.add(**new** Student1(20,"N2",450));

q.add(**new** Student1(30,"N3",520));

q.add(**new** Student1(40,"N4",400));

q.add(**new** Student1(50,"N5",300));

System.***out***.println(q);

Student1 s;

**while**((s=q.poll())!=**null**)

{

System.***out***.println(s);

}

}

}

Output:

[Student [roll=50, name=N5, marks=300], Student [roll=40, name=N4, marks=400], Student [roll=30, name=N3, marks=520], Student [roll=10, name=N1, marks=500], Student [roll=20, name=N2, marks=450]]

Student [roll=50, name=N5, marks=300]

Student [roll=40, name=N4, marks=400]

Student [roll=20, name=N2, marks=450]

Student [roll=10, name=N1, marks=500]

Student [roll=30, name=N3, marks=520]

Example3: Queue using linked list class- default FIFO order

**import** java.util.LinkedList;

**import** java.util.Queue;

**public** **class** QueueLinkedList {

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

{

Queue<Stu> q=**new** LinkedList<>();

q.add(**new** Stu(10,"N1",500));

q.add(**new** Stu(20,"N2",450));

q.add(**new** Stu(30,"N3",520));

q.add(**new** Stu(40,"N4",400));

q.add(**new** Stu(50,"N5",300));

System.***out***.println(q);

//here we don't use any any external method for iterating.

**for**(Stu s:q)

{

System.***out***.println(s);

}

}

}

Output:

[Stu [roll=10, name=N1, marks=500], Stu [roll=20, name=N2, marks=450], Stu [roll=30, name=N3, marks=520], Stu [roll=40, name=N4, marks=400], Stu [roll=50, name=N5, marks=300]]

Stu [roll=10, name=N1, marks=500]

Stu [roll=20, name=N2, marks=450]

Stu [roll=30, name=N3, marks=520]

Stu [roll=40, name=N4, marks=400]

Stu [roll=50, name=N5, marks=300]

**MAP:**

Using map also we can group multiple objects in the form of key-value pairs. Hence each key-value pair is known as an element.

A map is useful if we want to search, update or delete elements on the basis of a key.

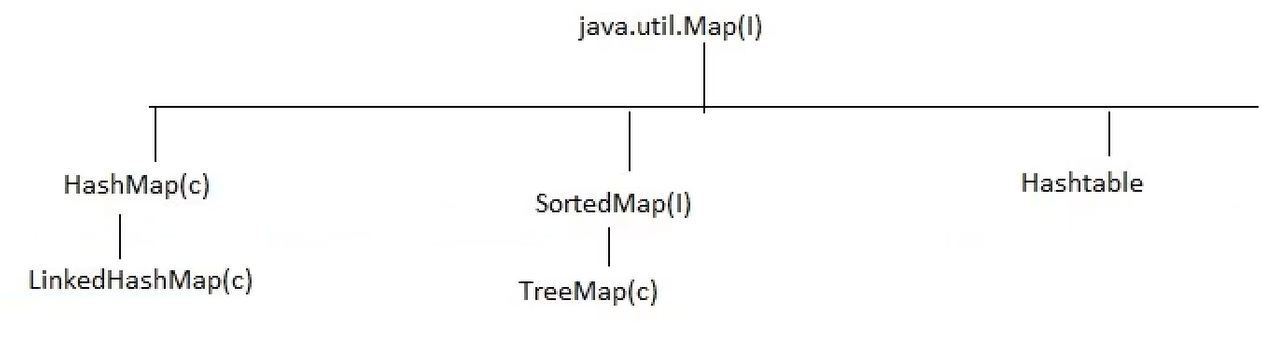
Examples:

-A map of error codes and their descriptions.

-A map of zip codes and cities.

-A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.

-A map of classes and students. Each class (key) is associated with a list of students (value).



Some of the Methods Available in Map:

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

->Public V put (K kay, V value): if entry is inserted it returns null value.

Eg: map.put (10, “delhi”); //null

Map.put (10, “pune”); // delhi, it returns overwritten value.

->public Set<K> keySet (); It will return all the key in the form of Set.

->public Collection<V> values (); It will return all the values in the form of collection objects.

->public Set<Map.Entry<K, V>> entrySet (); It will return all the kay-value pair(entry) in the form of Set.

Package java.util;

Interface Map {

Public V put (K k, V v);

Interface Entry

{

Object getKey ();

Object getValue ();

Object setValue (Object value);

}

}

--Interface can have inner class and inner interface.

--Entry as an inner interface so,

Map.Entry

Example:

**import** java.util.Collection;

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Set;

**public** **class** MapEx {

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

{

HashMap<Integer,String> map=**new** HashMap<>();

map.put(1, "one");

map.put(2, "two");

map.put(3, "three");

map.put(4, "four");

map.put(**null**, "five");

//printing map

System.***out***.println(map);

//printing all keys

Set<Integer> set=map.keySet();

System.***out***.println(set);

System.***out***.println(map.keySet());

//printing all values

Collection<String> collection=map.values();

System.***out***.println(collection);

System.***out***.println(map.values());

//printing all entries

Set<Map.Entry<Integer, String>> set1=map.entrySet();

System.***out***.println(set1);

**for**(Map.Entry<Integer, String> hm:set1)

{

System.***out***.println(hm.getKey()+"===="+hm.getValue());

}

System.***out***.println(map.size());

System.***out***.println(map.get(1));

}

}

Output:

{null=five, 1=one, 2=two, 3=three, 4=four}

[null, 1, 2, 3, 4]

[null, 1, 2, 3, 4]

[five, one, two, three, four]

[five, one, two, three, four]

[null=five, 1=one, 2=two, 3=three, 4=four]

null====five

1====one

2====two

3====three

4====four

5

one

Example2:

**import** java.util.HashMap;

**import** java.util.Map;

**class** Student{

**private** **int** roll;

**private** String name;

**private** **int** marks;

**public** Student()

{

}

**public** Student(**int** roll, String name, **int** marks) {

**this**.roll = roll;

**this**.name = name;

**this**.marks = marks;

}

**public** **int** getRoll() {

**return** roll;

}

**public** **void** setRoll(**int** roll) {

**this**.roll = roll;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getMarks() {

**return** marks;

}

**public** **void** setMarks(**int** marks) {

**this**.marks = marks;

}

@Override

**public** String toString() {

**return** "Student [roll=" + roll + ", name=" + name + ", marks=" + marks + "]";

}

}

**public** **class** MapSetOfValues {

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

{

HashMap<String,Student> hm=**new** HashMap<>();

hm.put("Hyd", **new** Student(101,"siri",980));

hm.put("chennai",**new** Student(102,"mouni",982));

hm.put("mumbai", **new** Student(103,"jaya",985));

System.***out***.println(hm);

//Set<Map.Entry<String,Student>> set=hm.entrySet();

**for**(Map.Entry<String, Student> m:hm.entrySet())

{

System.***out***.println("City name is:"+m.getKey());

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

Student s=m.getValue();

System.***out***.println("Roll:"+s.getRoll());

System.***out***.println("Name:"+s.getName());

System.***out***.println("Marks:"+s.getMarks());

}

}

}

Output:

{mumbai=Student [roll=103, name=jaya, marks=985], Hyd=Student [roll=101, name=siri, marks=980], chennai=Student [roll=102, name=mouni, marks=982]}

City name is:mumbai

==========

Roll:103

Name:jaya

Marks:985

City name is:Hyd

==========

Roll:101

Name:siri

Marks:980

City name is:chennai

==========

Roll:102

Name:mouni

Marks:982

NOTE:

--If inside a HashMap or LinkedHashMap we try to add any key as a user-defined object, then it is always recommended that, we should override equals () and hashCode () method inside that class, which is used as a key.

--Where as inside the TreeMap, the key object should implement Comparable interface or we need to use Comparator.

**Generics:**

--the main objective of Generics is to achieve type safety.

--It makes the code stable by detecting the bugs at compile time.

--Generics concept mostly used with collection framework related classes, but we can apply the generics concept without collection f/w also.

Note: Generics concept is not applicable with primitive data types, it supports only the objects.

--we can create our own generic classes or generic methods, and that class or method can work with any kind of data.

Advantages:

1.type checking at compile time.

2.get rid of down casting problem.

3with this our code will become bug (most of them detected at compile time) free and more stable.

Example:

MyGen.java

//Generic can work with any type of data

**public** **class** MyGen<T> {

T data;

**public** MyGen(T data)

{

**this**.data=data;

}

**public** T getData() {

**return** data;

}

}

Demo.java

**public** **class** Demo {

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

MyGen<Integer> mg=**new** MyGen<>(10);

System.***out***.println(mg.getData());

MyGen<String> mg1=**new** MyGen<>("Hello");

System.***out***.println(mg1.getData());

MyGen<Stu> mg2=**new** MyGen<>(**new** Stu(101,"siri",500));

System.***out***.println(mg2.getData());

}

}

Output:

10

Hello

Stu [roll=101, name=siri, marks=500]

Constraints in generics:

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

--In generics classes, we can apply the constraints or bound the type parameter for a particular range by using “extends” keyword.

Eg1:

Class Demo<T>{

}

--It is unbound type, without any constraint, we can pass any type of parameter.

Demo<Integer> d1=new Demo<>();

Demo<String> d2=new Demo<>();

Demo<Student> d3=new Demo<>();

Eg2: bounded type

Class Demo<T extends X>{ //either X or any child class of X

}

--here if X is a class type then we can pass either X types elements or sub class elements of X type as parameter type.

Eg3:

Class Demo<T extends Number>{ //here number class belongs to java.lang package

}

Number class will act as a parent class, for all the wrapper classes which represent the numeric data types.

//valid

Demo<Number> d1=new Demo<>();

Demo<Integer> d2=new Demo<>();

Demo<Double> d3=new Demo<>();

//Invalid

Demo<String> d1=new Demo<>();

If X is an interface then we are able to pass either X type elements or X implementation type elements as parameter.

Eg:

Interface Java{

}

Class CoreJava implements Java{

}

Class AdvJava implemets Java{

}

//In generics

Class Course<T extends Java>{ //Java is Interface

}

Course<Java> c1=new Course<>();

Course<CoreJava> c2=new Course<>();

Course<AdvJava> c3=new Course<>();

NOTE:

\*Bounded parameters constraints at class level are not allowing “implements” keyword and “super” keyword or any type of wildcards(?).

\*In generics classes, we can use more than one type as bounded parameter by using & operator.

Eg:

Class Demo<T extends Number & Serializable>{

}

--here Demo class is able to allow the elements which must be either same as Number or sub class and must implements Serializable interface.

Generic Method:

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

--in the MyGen class example we place get() method inside our generics class as a generic method.

--but we can declare a generic method inside a non-generic class also.

Eg:

MyGen.java

**public** **class** MyGen<T> {

T data;

**public** MyGen(T data)

{

**this**.data=data;

}

**public** T getData() {

**return** data;

}

}

Demo.java

**public** **class** Demo {

**public** **static** <T> **void** fun(T t)

{

System.***out***.println(t);

}

**public** **static** <T> **void** fun1(T[] tr) {

**for**(T t:tr)

{

System.***out***.println(t);

}

}

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

*fun*(100);

*fun*("Hello");

*fun*(**new** Stu(10,"na",400));

//int[] nums= {10,20,30,40}; //In generic primitive are not supported

Integer[] nums= {10,20,30,40};

String[] str= {"a","b","c"};

*fun1*(nums);

*fun1*(str);

}

}

Output:

100

Hello

Stu [roll=10, name=na, marks=400]

10

20

30

40

a

b

c

\*Generic method with more than one generic type parameters and return generic type:

**public** **class** GenericMethod {

**public** **static** <T> **void** fun(T t) {

System.***out***.println(t);

}

//more than one generic parameters

**public** **static** <K,V> **void** fun1(K k,V v) {

System.***out***.println(k+"==="+v);

}

//generic with return type

**public** **static** <K,V,L> V fun2(K k,V v,L l) {

**return** v;

}

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

*fun*(1);

*fun*("hi");

*fun*(**new** Stu(10,"siri",900));

*fun1*(1,43);

*fun1*("hello","world");

*fun1*("hi",18);

*fun1*("Hyd",**new** Stu(20,"mahi",910));

System.***out***.println(*fun2*(1,2,3));

System.***out***.println(*fun2*("hi",**new** Stu(200,"maha",890),"mumbai"));

Object o=*fun2*(10,250,345);

System.***out***.println(o);

}

}

Output:

1

hi

Stu [roll=10, name=siri, marks=900]

1===43

hello===world

hi===18

Hyd===Stu [roll=20, name=mahi, marks=910]

2

Stu [roll=200, name=maha, marks=890]

250

Wildcards in generics:

--It is mostly used with collection type parameters inside a method.

--we cannot use wildcard with class level.

1.upper bound ->? extends type

2.lower bound ->? Super type

3.unbounded wildcard ->?

Unbounded wildcard:

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

--here we are using list.

--we can pass any data type values to the list.

Eg: without wildcard

Public <T> void fun(List<T> list){

}

Eg: using wildcard

--the above example is similar to not applying generics.

Public void fun(List<?> list){

}

**import** java.util.Arrays;

**import** java.util.List;

**public** **class** UnboundedWildcard {

**public** **static** **void** fun(List<?> list)

{

System.***out***.println("inside fun..");

**for**(Object i:list)

{

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

}

}

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

List<Integer> iList=Arrays.*asList*(1,2,3,4,5);

List<String> sList=Arrays.*asList*("a","b","c","d");

*fun*(iList);

*fun*(sList);

}

}

Output:

inside fun..

1

2

3

4

5

inside fun..

a

b

c

d

Eg:

Public static void fun(List list){

}

--the above code we write without the wildcard also.

Upper Bound Wildcard:

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

Public static void fun(List<? Extends Number> list) {

}  
**import** java.util.Arrays;

**import** java.util.List;

**public** **class** UnboundedWildcard {

**public** **static** **void** fun(List<? **extends** Number> list)

{

System.***out***.println("inside fun..");

//upper bound

//Number will take - integer, byte, float, double, long.

**for**(Number i:list)

{

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

}

}

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

List<Integer> iList=Arrays.*asList*(1,2,3,4,5);

List<Float> fList=Arrays.*asList*(2.1f,3.4f,6.8f);

*fun*(iList);

*fun*(fList);

}

}

Output:

inside fun..

1

2

3

4

5

inside fun..

2.1

3.4

6.8

Eg2:

class Person {

void show() {

}

}

Class Student extends Person {

}

Class Demo{

Static void fun (List<? Extends Person> list) {

{

For(Person i:list)

{

}

}

Public static void main (String [] args) {

List<Student> sList=Arrays.asList(

new Student(1,”n1”,100),

…………..,

);

}

Lower Bound Wildcard:

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

Public static void fun (List<? Super Integer> list)

{

//parent of integer are NUMBER also OBJECT also.

}

Eg:

**import** java.util.Arrays;

**import** java.util.List;

**public** **class** UnboundedWildcard {

**public** **static** **void** fun(List<? **super** Integer> list)

{

System.***out***.println("inside fun..");

//lower bound

//Integer take parent class as- NUMBER or OBJECT.

//only the Object is the type

**for**(Object i:list)

{

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

}

}

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

List<Integer> iList=Arrays.*asList*(1,2,3,4,5);

List<Number> nList=Arrays.*asList*(2,3,6,8,10);

List<Object> oList=Arrays.*asList*(10,20,30,40,50);

*fun*(iList);

*fun*(nList);

*fun*(oList);

}

}

Output:

inside fun..

1

2

3

4

5

inside fun..

2

3

6

8

10

inside fun..

10

20

30

40

50

Eg2:

Public static void fun (List<? Super Student> list) {

For(Object o:list) {

System.out.println(o);

}

}

--here we can call the above method by passing List of Student or List of Person or List of Object.

**Date and Time api:**

Date and time are from

Java.util package

Java.text package

--The classes are:

Java.util.Date

Java.util.Calander

Java.text.SimpleDateFormat

\*From java 1.8 on words a new date and time api introduced belongs to java.time package.

->important classes of the java.time package

1. java.time.LocalDate:

2. java.time.LocalTime:

3. java.time.LocalDateTime:

4. java.time.Duration:

5. java.time.Duration:

6. java.time.format.DateTimeFormatter:

Example1:

**import** java.time.LocalDate;

**import** java.time.LocalDateTime;

**import** java.time.LocalTime;

**import** java.time.ZonedDateTime;

**public** **class** DateTimeEx {

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

LocalDate ld=LocalDate.*now*();

System.***out***.println(ld);//yyyy-mm-dd

LocalTime lt=LocalTime.*now*();

System.***out***.println(lt);

LocalDateTime ldt=LocalDateTime.*now*();

System.***out***.println(ldt);

ZonedDateTime zdt=ZonedDateTime.*now*();

System.***out***.println(zdt);

}

}

Output:

2022-11-25

16:17:19.582904600

2022-11-25T16:17:19.582904600

2022-11-25T16:17:19.583906100+05:30[Asia/Calcutta]

Eg2:

**import** java.time.LocalDate;

**public** **class** DateTimeEx {

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

System.***out***.println(args.length);

LocalDate date=LocalDate.*now*();

LocalDate yesterday=date.minusDays(1);

LocalDate tomorrow=yesterday.plusDays(2);

System.***out***.println("Taday date:"+date);

System.***out***.println("Yesterday date:"+yesterday);

System.***out***.println("Tomorrow date:"+tomorrow);

}

}

Output:

3

Taday date:2022-11-25

Yesterday date:2022-11-24

Tomorrow date:2022-11-26

Eg3:

**import** java.time.LocalDate;

**import** java.time.LocalDateTime;

**public** **class** DateTimeEx {

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

LocalDate date=LocalDate.*of*(2017, 1, 13);

LocalDateTime dateTime=date.atTime(1,50,9);

System.***out***.println(dateTime);

}

}

Output:

2017-01-13T01:50:09

Eg4: DateTimeFormatter

Patterns:

G - Era(AD BC)

y - year( yy(18) or yyyy(2018))

M - Month(M(9) or MM(09) or MMM(Sep))(MMMMM--September)

d - day(d(23) or dd(23) or ddd(023))

E - day in a weak(E (sun))(EEEE--Sunday)

a - am pm

h - hour in am or pm (1-12)

hh - hour in am or pm (01-12)

H - hour of day in 24 hour form (0-23)

HH - hour of day in 24 hour form (00-23)

m - minute (4)

mm - minute (04)

s - second (4)

ss - second(04)

program: User defined date time format

**import** java.time.LocalDate;

**import** java.time.format.DateTimeFormatter;

**public** **class** DateTimeEx {

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

LocalDate date=LocalDate.*now*();

//first format

DateTimeFormatter dtf=DateTimeFormatter.*ofPattern*("dd/MM/yyyy");

String str=date.format(dtf);

System.***out***.println(str);

//second format

String str1=date.format(DateTimeFormatter.*ofPattern*("dd-MM-yyyy"));

System.***out***.println(str1);

}

}

Output:

25/11/2022

25-11-2022

Eg5:

Converting string date to LocalDate

**import** java.time.LocalDateTime;

**import** java.time.format.DateTimeFormatter;

**public** **class** DateTimeEx {

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

String dob="23/06/2000==11:50:40";

//first format

LocalDateTime ld=LocalDateTime.*parse*(dob,DateTimeFormatter.*ofPattern*("dd/MM/yyyy==HH:mm:ss"));

System.***out***.println(ld);

//second format

DateTimeFormatter dtf=DateTimeFormatter.*ofPattern*("dd/MM/yyyy==HH:mm:ss");

LocalDateTime lD=LocalDateTime.*parse*(dob,dtf);

System.***out***.println(lD);

}

}

Output:

2000-06-23T11:50:40

2000-06-23T11:50:40

Eg6: leap year

**import** java.time.LocalDate;

**public** **class** DateTimeEx {

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

LocalDate ld=LocalDate.*of*(1999,2,23);

LocalDate ld1=LocalDate.*now*();

System.***out***.println(ld.isLeapYear());

System.***out***.println(ld1.isLeapYear());

}

}

Output:

false

false

Eg7:

**import** java.time.LocalDate;

**import** java.time.format.DateTimeFormatter;

**import** java.util.Scanner;

**public** **class** DateTimeEx {

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

Scanner sc=**new** Scanner(System.***in***);

System.***out***.println("Enter date of birth in dd/MM/yyyy pattern");

String str=sc.next();

DateTimeFormatter dtf=DateTimeFormatter.*ofPattern*("dd/MM/yyyy");

LocalDate ld=LocalDate.*parse*(str,dtf);

String result=ld.format(DateTimeFormatter.*ofPattern*("EEEE"));

System.***out***.println("Your Birthday is:"+result); }

}

Output:

Enter date of birth in dd/MM/yyyy pattern

23/06/2023

Your Birthday is:Friday

Eg8: period

**import** java.time.LocalDate;

**import** java.time.Period;

**public** **class** DateTimeEx {

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

LocalDate ld=LocalDate.*now*();

LocalDate dob=LocalDate.*of*(1997, 06, 23);

Period p=Period.*between*(ld, dob);

System.***out***.println(p);

}

}

Output:

P-25Y-5M-2D

Eg9: ChronoUnit

**import** java.time.LocalDateTime;

**import** java.time.temporal.ChronoUnit;

**public** **class** DateTimeEx {

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

LocalDateTime oldDate=LocalDateTime.*of*(2002,07,16,5,30,55);

LocalDateTime newDate=LocalDateTime.*of*(2022,07,16,5,30,55);

System.***out***.println(oldDate);

System.***out***.println(newDate);

System.***out***.println(ChronoUnit.***MONTHS***.between(oldDate, newDate));

}

}

Output:

2002-07-16T05:30:55

2022-07-16T05:30:55

240

Format Specifiers:

OBJECT ORIENTATION EXAMPLE: UDEMY

Nested Classes in java:

Eg:

**public** **class** Demo {

**int** x;

**class** Cal

{

**void** display()

{

System.***out***.println("inside cal class");

}

}

//static nested class

**static** **class** Calc

{

**void** display()

{

System.***out***.println("inside calc class");

}

}

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

{

**class** Pet

{

**void** display1()

{

System.***out***.println("inside pet class");

}

}

Pet obj1=**new** Pet();

obj1.display1();

Demo d=**new** Demo();

System.***out***.println(d.x);

Demo.Cal obj2=d.**new** Cal();

//OuterClassName.InnerClassName object= OuterClassObject.new InnerClassName();

obj2.display();

//object to nested static class

Calc c=**new** Calc();

c.display();

}

}

Output:

inside pet class

0

inside cal class

inside calc class

**Testing Code**:

First Test: Eg

Caluclator.java

**package** com.mic.caluclator;

**public** **class** Caluclator {

**public** **int** add(**int** num1, **int** num2) {

**return** num1+num2;

}

}

CaluclatorTest.java

**package** com.mic.caluclator;

**import** org.junit.jupiter.api.BeforeEach;

**import** org.junit.jupiter.api.Test;**import** **static** org.junit.jupiter.api.Assertions.assertEquals;

**public** **class** CaluclatorTest {

//2)private final Caluclator cal = new Caluclator();

//3

**private** Caluclator cal;

@BeforeEach

**void** setUp() {

cal = **new** Caluclator();

}

@Test

**public** **void** canAddZeroPlusZero(){

//1)Caluclator cal = new Caluclator();

**int** sum= cal.add(0,0);

assertEquals (0,sum,"was expecting sum of 0");

}

@Test

**public** **void** canAddOnePlusOne(){

//1)Caluclator cal = new Caluclator();

**int** sum=cal.add(1,1);

assertEquals (2,sum);

}

}

Testing Edge Cases:

**package** com.mic.caluclator;

**import** org.junit.jupiter.api.BeforeEach;

**import** org.junit.jupiter.api.Disabled;

**import** org.junit.jupiter.api.Test;

**import** **static** org.junit.jupiter.api.Assertions.assertEquals;

**public** **class** CaluclatorTest {

//2)private final Caluclator cal = new Caluclator();

//3

**private** Caluclator cal;

@BeforeEach

**void** setUp() {

cal = **new** Caluclator();

}

@Test

**public** **void** canAddZeroPlusZero(){

//1)Caluclator cal = new Caluclator();

**int** sum= cal.add(0,0);

assertEquals (0,sum,"was expecting sum of 0");

}

@Test

**public** **void** canAddOnePlusOne(){

//1)Caluclator cal = new Caluclator();

**int** sum=cal.add(1,1);

assertEquals (2,sum);

}

}

@Test()

@Disabled//it disable the error occurred test.

//@Ignore - junit4

**public** **void** canMaxIntPlusOne(){

**int** sum=cal.add(Integer.***MAX\_VALUE***,1);

System.***out***.println(sum);//max integer value +1 ->it rotate and give negative value

assertEquals (Integer.***MAX\_VALUE***+1L,sum);

//we are expecting a long value(+ve number) but the sum is integer value(-ve number)

}

}

Testing Annuity Calculation:

Caluclator.java:

**package** com.mic.caluclator;

**import** java.math.BigDecimal;

**import** java.text.NumberFormat;

**import** java.util.Locale;

**public** **class** Caluclator{

**public** String calcAnnuity(String R, **int** t, String r, **int** n) {

BigDecimal a=**new** BigDecimal(r).divide(**new** BigDecimal(n));

BigDecimal b=BigDecimal.***ONE***.add(a);

BigDecimal c=b.pow(n\*t);

BigDecimal d=c.subtract(BigDecimal.***ONE***);

BigDecimal e=d.divide(a);

BigDecimal f=**new** BigDecimal(R).multiply(e);

NumberFormat numberFormat=NumberFormat.*getCurrencyInstance*(Locale.***US***);

**return** numberFormat.format(f);

}

}

CaluclatorTest.java:

**package** com.mic.caluclator;

**import** org.junit.jupiter.api.Test;

**import** **static** org.junit.jupiter.api.Assertions.assertEquals;

**public** **class** CaluclatorTest {

**private** **final** Caluclator cal = **new** Caluclator();

@Test

**void** annuityExample(){

String ans = cal.calcAnnuity("22000",7,".06",1);

assertEquals("$184,664.43",ans);

}

}

Implementing guessing game with TDD (test driven development):

GuessingGame.java:

**package** com.mic.game;

**import** java.util.Random;

**public** **class** GuessingGame {

**private** **final** **int** randomNum=**new** Random().nextInt(10)+1;

**private** **int** counter=0;

**public** String guess(**int** guessedNumber) {

counter++;

**if**(counter==4 && guessedNumber!=getRandomNumber())

{

**return** "You didn't get it and you've had four tries.Game Over";

}

String tryText=counter==1?"try":"tries";

String winingMsg=String.*format*("You got it in %d %s",counter,tryText);

**return** guessedNumber==getRandomNumber()?winingMsg:"You didn't get it";

}

**public** **int** getRandomNumber() {

// Random r=new Random();

// return 1+r.nextInt(10);

**return** randomNum;

}

}

GuessingGameTest.java

**package** com.mic.game;

**import** org.junit.jupiter.api.BeforeEach;

**import** org.junit.jupiter.api.RepeatedTest;

**import** org.junit.jupiter.api.Test;

**import** **static** org.junit.jupiter.api.Assertions.assertEquals;

**public** **class** GuessingGameTest {

**private** GuessingGame game;

@BeforeEach

**void** setUp() {

game = **new** GuessingGame();

}

@Test

**public** **void** testSimpleWinSituation(){

**int** randomNumber = game.getRandomNumber();

String msg=game.guess(randomNumber);

System.***out***.println(msg);

assertEquals("You got it in 1 try",msg);

}

@Test

**public** **void** testOneWrongGuessSituation(){

**int** randomNumber = game.getRandomNumber();

String msg=game.guess(-5);

assertEquals("You didn't get it",msg);

}

@Test

//@RepeatedTest(number)-we can repeat the test number of times we want

**public** **void** testRandomNumberGeneration(){

//1 2 3 4 5 6 7 8 9 10

//1 1 1 1 0 1 1 0 1 1 =>10

**int**[] randomNumCount=**new** **int**[11];

**for**(**int** counter=0;counter<50;counter++)

{

GuessingGame game=**new** GuessingGame();

**int** randomNumber = game.getRandomNumber();

randomNumCount[randomNumber]=1;

}

**int** sum=0;

**for**(**int** counter=0;counter<11;counter++)

{

sum+=randomNumCount[counter];

}

System.***out***.println(sum);

//String msg=game.guess(-5);

assertEquals(10,sum);

}

@Test

**public** **void** testFourWrongGuesses(){

game.guess(-3);

game.guess(-3);

game.guess(-3);

String msg=game.guess(-3);

assertEquals("You didn't get it and you've had four tries.Game Over",msg);

}

@Test

**public** **void** testThreeWrongGuessesAndOneCorrect(){

game.guess(-3);

game.guess(-3);

game.guess(-3);

**int** correctAnswer=game.getRandomNumber();

String msg=game.guess(correctAnswer);

assertEquals("You got it in 4 tries",msg);

}

@Test

**public** **void** testTwoWrongGuessesAndOneCorrect(){

game.guess(-3);

game.guess(-3);

**int** correctAnswer=game.getRandomNumber();

String msg=game.guess(correctAnswer);

assertEquals("You got it in 3 tries",msg);

}

}