Java FullStack Development Program



=========================================================

Full stack development = Front-end development +

Back-end development +

Database

Front-end technologies : HTML, CSS, Javascript, Bootstrap, Angular, React JS

Back-end technologies : Java, Python, .Net, PHP,..

Database : Oracle, MySQL, PostgreSQL, MongoDB,..

* Front-end technologies are used to develop web pages.
* web pages are the pages shown to the user on his browser.
* A user interacts with the application through web pages.
* on web pages you can see the elements like links or buttons to interact with the application.

Example:

* If you open amazon.in from your browser, then you will get a landing page.
* when you click on a product, then you will get another page with complete details of that product.
* This landing page, this product details page are web pages.
* Amazon sells millions of products in online to the customers.
* The details of all the products are stored in a database.
* Back-end application of amazon will interact/talk to the database, gets the details of the products and send that details to the front-end application.

what to learn in Java back-end development?

Core JAVA

Advanced JAVA

Spring Framework

Spring Boot

Microservices

Java Realtime project

CORE JAVA

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

Language? : is a medium to convey anything.

Natural languages: English, Hindi, Telugu, Tamil,…

Formal languages: are used for specific purpose.

For ex: accounts is a formal language which talks abount profit and loss.

For ex: statistics is a formal language which talks about data analysis.

* Programming languages are also comes under formal languages.
* Programming languages are used to interact with the systems/computers/machines.
* Java, Python, .Net, Kotlin, C, C++,.. are called high-level programming languages.
* The program statements looks like English sentences. that’s why these are called high-level programming languages.
* Computers can only understand Binary language(0’s and 1’s).
* The programs developed in high-level language are translated into binary language.
* The translators like compilers and interpreters will convert the high-level language statements into Binary language.

How to learn a programming language?

Paragraph Program

sentences statements

(Grammer) (syntax)

words Tokens

Tokens:

Tokens are 5 types in any programming language.

1. keywords
2. identifiers
3. operators
4. literals
5. strings

keywords: These are the words which have pre-defined meaning in that programming language.

ex: if, for, while, break, class, ……

identifiers: These are the words which are defined by the programmer in a programming language.

ex: Employee, Customer, BankAccount, etc…….

operators: These will perform a well-defined task.

Literals: These are values in a program.

numberOfMinutesPerHour = 60;

username = “ashokit”;

strings: These are group of characters enclosed within double quotes.

statements:

Statements are 5 types in any programming language.

1. input statement
2. output statement
3. memory statement
4. arithmetic and logic statement
5. control statement

* The first name of Java was OAK.
* OAK was developed in 1991 and renamed to Java in 1995.
* James Gosling is said to be the father of Java.
* Java is the most popular and widely used programming language for Enterprise applications.
* Enterprises are nothing but large business organizations.
* Java was developed by Sun Micro Systems company. But in 2010, Oracle Corporation has acquired/purchased Sun Micro Systems. So, today Java is from Oracle Corporation.

Java Modules/Editions:

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

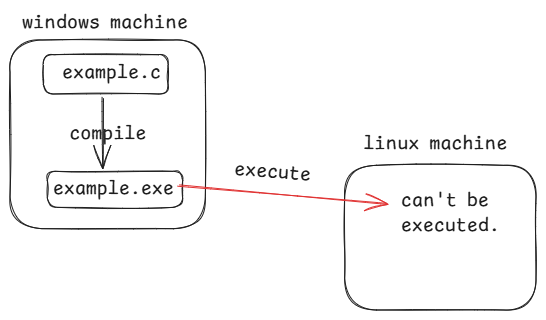
1. Java SE (Standard Edition)
2. Java EE (Enterpise Edition)
3. Java ME (Micro Edition)

* Previous name of Java SE was J2SE.
* Java SE is also called Core JAVA.
* Java SE module is designed to develop business logics in Java.
* Java SE module provides libraries to build desktop applications.
* A desktop application is nothing but, it is an application/software, we will download, install and then we use it.
* For example, MS-Office, Skype, Zoom, Media Player, etc.. are desktop applications.
* Java SE is the base module to learn other two modules.
* Java EE modules is to build large-scale, multi-tiered enterprise applications.
* For example, developing online banking systems, e-commerce websites, CRM(Customer Relationship Management) applications.
* CRM applications means, they have sales module, Service module and Support module.
* Examples of CRM applications like Samsung, LG, Sony, etc..
* Examples of e-commerce websites are, Amzaon, Flipkart, eBay, meesho, etc..
* Examples of online banking systems are like SBI online, HDFC, Kotak Bank, etc…
* Java ME module is used to build small-scale device applicatoins, like embedded systems applications, mobile phones applications, smart card applications, IoT(Internet of Things) devices.
* A Java Back-end developer has to learn Java SE and Java EE modules.

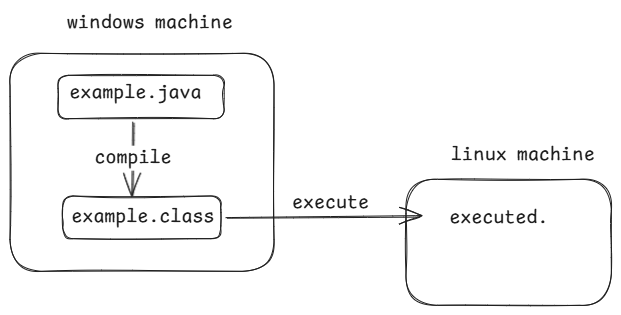
Java features:

1. Platform-independent:

* platform-dependent means, if we compile a program in one operating system then we have to execute the program in the same operating system only. But compiling in one OS and executing in another OS is not possible.
* example, C/C++ language.



* platform-independent means, we can compile a program in one operating system and we can exeute in another operating system also.
* Java is a platform-independent programming language. When you compile the Java code, it is converted to the Byte code and this Byte code can run on any operating system.



* example.java --- source code
* example.class --- byte code
* Because of byte code, Java has a capability called “Write Once, Run Anywhere(WORA)”.

1. Portable:

* Portable means, easy to carry.
* Java is portable, because it is independent of hardware and operating system. The same program can run on any device without modifications.

1. Robust:

* If a programming language provides automatic memory management and runtime error handling then it is called Robust programming language.
* A Java program is executed in a Java Virtual Machine(JVM). In JVM, there is Garbage Collector(GC) and it will automatically de-allocates the memory for unused objects.
* Java has Exception handling mechanism to handle the runtime errors. So, Java is called Robust programming language.

1. multithreading:

* multithreading means, executing more than one task parallelly in a single application.
* For example, IRCTC is a single application, and it can execute thea tasks like booking seats and cancelling seats parallelly. So it is a multithreaded application.
* Java is a multithreaded programming language. So, in a java application we can execute multiple tasks simultaneously.

1. secure:

* When you compile the java program, byte code will be generated and it will be stored in a .class file.
* If a hacker has entered into .class file and makes any changes to the byte code then JVM will not allow that .class file to enter and execute in the JVM.
* That’s why we call Java as a secured programming language.

1. Object Oriented:

* Java is an object oriented programming language.
* Java provides a way to implement the principles like encapsulation, abstraction, inheritance and polymorphism.
* With Object oriented programming approach, we can develop software application for the real-time requirements.

Java versions:

JDK – Java Development Kit

JRE – Java Runtime Environment

JVM – Java Virtual Machine

JDK1.0 -- Jan, 1996

. .

. .

J2SE5.0 -- Sep, 2004

. .

. .

Java SE8 -- Mar, 2014 (imp version)

. .

. .

Java SE11 -- Sep, 2018 (imp version)

. .

. .

Java SE17 -- Sep, 2021 (imp version)

. .

. .

Java SE21 -- Sep, 2023 (imp version)

Java SE22 -- Mar, 2024 (Current version)

Installing Java 17 version:

1. visit <https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html>
2. download windows x64 installer
3. jdk-17.0.11\_windows-x64\_bin.exe file is downloaded.
4. Double click on the downloaded file, follow next buttons, then finally close button.
5. Now the Java software is installed at

C:\Program Files\Java\jdk-17 folder.

Java Path setting:

1. In windows search, type environment
2. click on Edit system environment variables
3. click on Environment variables button
4. goto system variables, choose path variable,

then click on Edit button.

1. click on New button.
2. now copy the location of bin folder under jdk-17

(C:\Program Files\Java\jdk-17\bin) and paste it

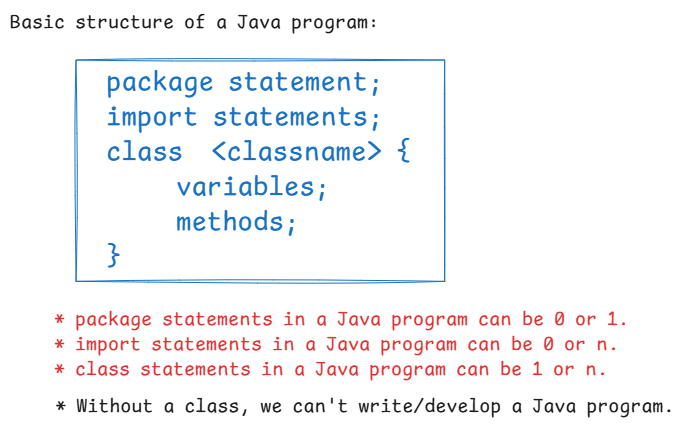
in the empty text box.

1. now move this value to the top with move up button.
2. click on ok to close the opened message boxes.

verify the path setting:

1. open a command prompt
2. run the following command

* java -version



where can we write a Java program:

* we can write a Java program in 2 places.

1. In a Text editor

( notepad/notepad++/sublime/edit plus,..)

1. In an IDE(Integrated Development Environment)

(Eclipse/IntelliJ/VSCode/STS,...)

First Java Program:

1. open notepad
2. write the program code

class HelloWorld {

public static void main(String[] args) {

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

}

}

1. save the program ( D:\69-JFSD\HelloWorld.java)

(note: change the save as type to All Files, before entering the filename)

1. Now goto D:\ drive then open 69-JFSD folder
2. type cmd in the search bar, a command prompt is opened.
3. compile the program.

D:\69-JFSD> javac HelloWorld.java

1. run the program.

D:\69-JFSD> java HelloWorld

output: Hello World

print() & println():

* print() method will display the output and places the cursor on the same line.
* println() method will display the output and moves the cursor to the next line.

for ex:

class Demo

{

public static void main(String[] args)

{

System.out.print("Welcome");

System.out.print("Java FullStack");

}

}

* javac Demo.java
* java Demo

output:

WelcomeJava FullStack

for example:

class Demo{

public static void main(String[] args)

{

System.out.println("Welcome");

System.out.print("Java FullStack");

}

}

* javac Demo.java
* java Demo

output:

Welcome

Java FullStack

Q) If I want to execute a Java program for 5 times, how many times I have to compile the program?

A) one time

Q) If I make some changes to the program code, do I need to recompile the program again?

A) Yes. If you execute the program without re-compiling then you will get the previous output.

Q) what is a keyword?

A) It is a word which has a pre-defined meaning in the programming language.

identifiers:

* identifiers are user-defined words in a program.
* class names, interface names, variable names, method names and package names are all called identifiers in the program.
* Identifier rules.

1. Identifiers can have letters(upper case or lower case) , digits, underscore and a dollar. Other special characters are not allowed.
2. Identifier must not start with a digit.
3. Identifier can’t be a keyword.
4. Identifier should not contain a whitespace.
5. Identifier is case sensitive.

* examples:

MyVar --- valid

my\_Var --- valid

my var --- invalid

7season --- invalid

season7 --- valid

season@7 --- invalid

seasion$7 --- valid

season-7 --- invalid

\_\_temp --- valid

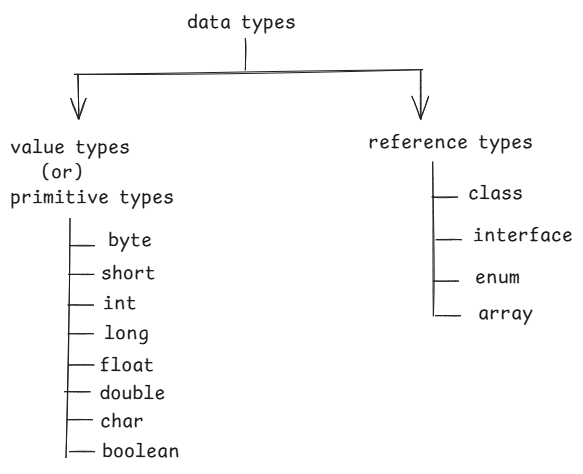
\_temp\_ --- valid

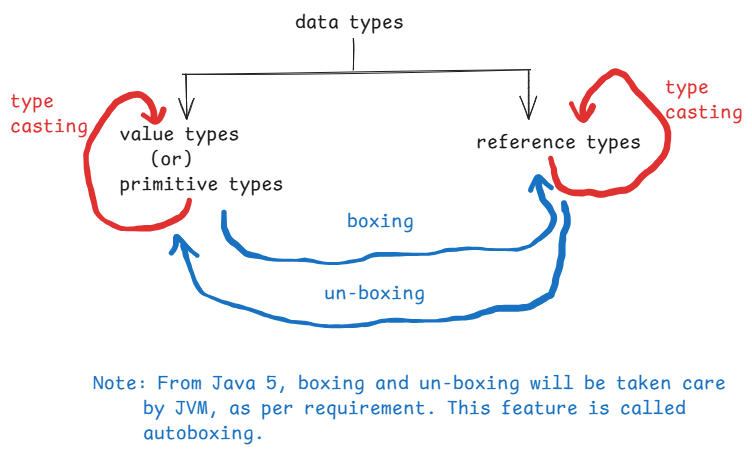
$\_\_temp --- valid

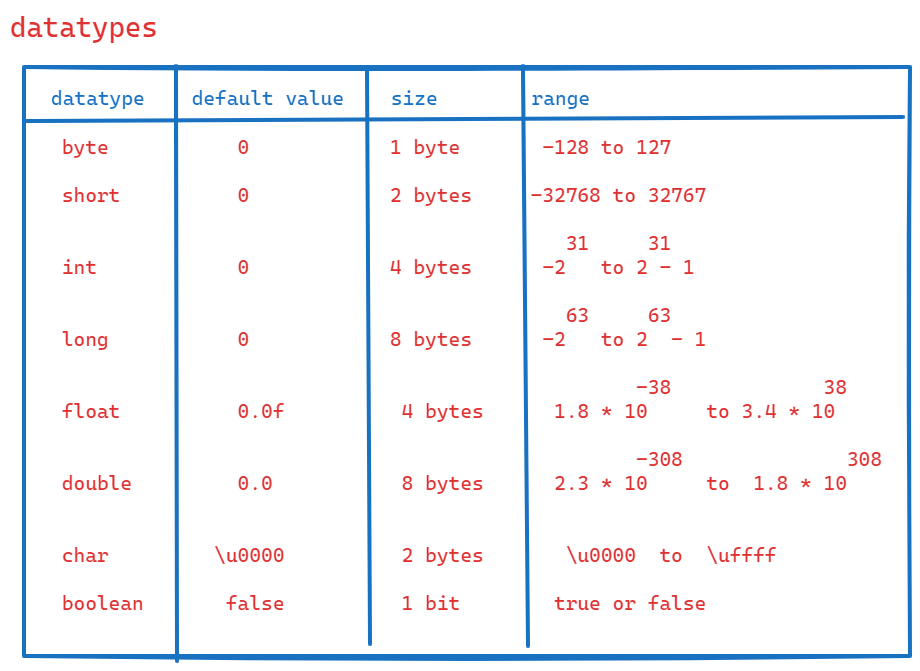
Customer$Service --- valid

\_9test --- valid

datatypes:







Note: In Java, a char variable can store a unicode character.

Unicode is a large character set, which includes almost all the languages character set.

To know the unicodes table, visit

<https://symbl.cc/en/unicode-table/>

Memory statement:

* memory statement is used to create a variable.
* a variable is a name given to some memory location.

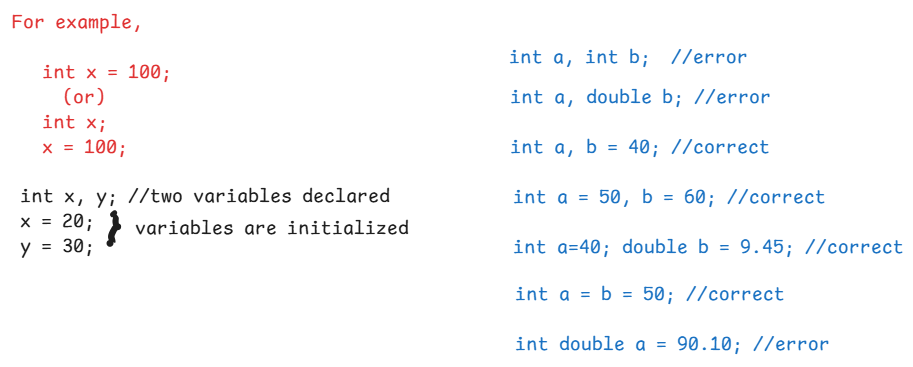
syntax:

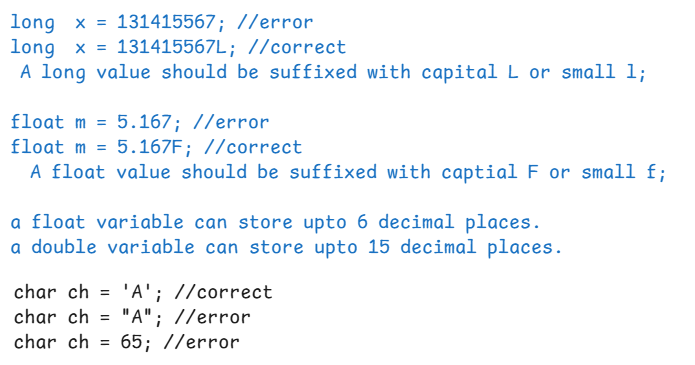
datatype variablename = value;

(or)

datatype variablename; //variable declaration

variable = value; //variable initialization





* The below code is to find addition of two integers.

class Demo2

{

public static void main(String[] args)

{

int a, b;

a = 10;

b = 20;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("SUM = " + (a + b));

}

}

* javac Demo2.java
* java Demo2

output:

a = 10

b = 20

SUM = 30

* write the below program in notepad, then compile and run the program from cmd.

class Demo2

{

public static void main(String[] args)

{

int a = 20; int b = 90;

System.out.println("a = " + a);

System.out.println("b = " + b);

float x = 37.6532143F;

//a float can store upto 6 decimal places

//so, last digit 3 will not be displayed.

System.out.println("x = " + x);

}

}

Installing Eclipse IDE:

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

1. visit <https://www.eclipse.org/downloads/>
2. click on download button
3. eclipse-inst-jre-win64.exe file is downloaded.
4. run the .exe file
5. click on Eclipse IDE for Enterprise Java and Web development.
6. click on Install button.

First Java program on Eclipse:

1. create a workspace folder.

ex: Goto D: drive, create a new folder

Eclise-Workspace-730AM

1. start the eclipse, using icon on desktop.
2. In the Eclipse IDE Launcher windows, click on browse button, choose the workspace folder, then click on launch button.
3. click on File 🡪 New 🡪 Project… 🡪 Java project 🡪 Project name: DemoApplication 🡪 choose execution jre: JavaSE 17 🡪 next 🡪 finish.
4. expand project folder 🡪 expand src folder 🡪 delete module-info.java file.
5. right click on src folder 🡪 New 🡪 class 🡪 enter the name: Demo 🡪 select public static void main(String[] args) check box 🡪 finish.
6. write the logic in main method.

for ex:

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

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

**int** a = 10, b = 20;

**float** x = 2.19f, y = 3.89f;

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

System.***out***.println("x + y = " + (x + y));

}

}

1. Right click on Demo.java 🡪 Run As 🡪 Java application
2. output will be shown on console window.

a + b = 30

x + y = 6.08

Type casting:

* Type casting means, conversion of a value from one datatype to another data type.
* Type casting is of 2 types.

1. implicit type casting/up-casting/widening
2. explicit type casting/down-casting/narrowing

* implicit type casting means, conversion of a value from smaller data type to larger data type.
* explicit type casting means, conversion of a value from larger data type to smaller data type.
* Implicit type casting will be done automatically.
* In explicit type casting, we need to specify the target type in paranthesis, before the variable name.

ex1: int x = 10;

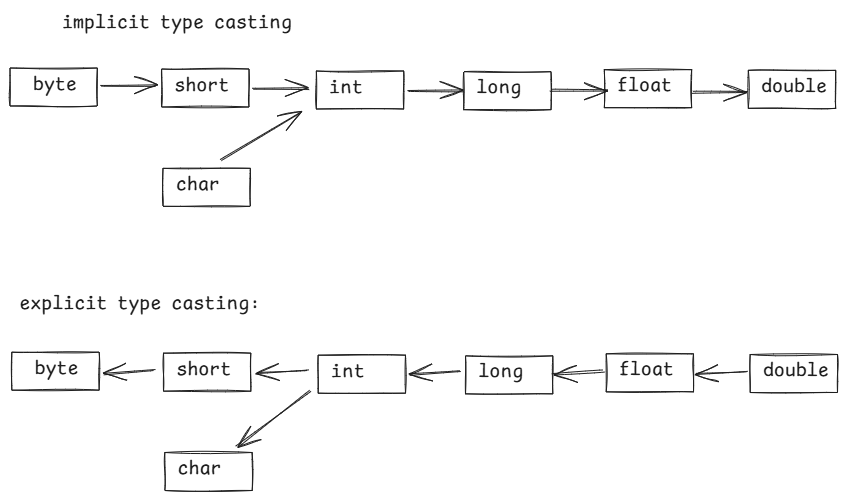
long y = x; //implicit

ex2:

long a = 587;

int b = a; //error

int b = (int)a; //explicit



Note: boolean can’t be converted to any other type and any other type can’t be converted to the boolean type.

**public** **class** TypeCastingDemo {

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

//type casting byte to float

**byte** b = 125;

**float** x = b; //implicit

System.***out***.println("Type casting byte to float");

System.***out***.println("byte : " + b);

System.***out***.println("float : " + x);

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

//type casting double to int

**double** d = 5.79;

**int** y = (**int**) d; //explicit

System.***out***.println("Type casting double to int");

System.***out***.println("double : " + d);

System.***out***.println("int : " + y);

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

//type casting char to int

**char** ch = 'a';

**int** z = ch; //implicit

System.***out***.println("Type casting char to int");

System.***out***.println("char : " + ch);

System.***out***.println("int : " + z);

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

//type casting int to char

**int** m = 65;

**char** ch2 = (**char**)m; //explicit

System.***out***.println("Type casting int to char");

System.***out***.println("int : " + m);

System.***out***.println("char : " + ch2);

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

}

}

output:

Type casting byte to float

byte : 125

float : 125.0

================================

Type casting double to int

double : 5.79

int : 5

===============================

Type casting char to int

char : a

int : 97

=============================

Type casting int to char

int : 65

char : A

Accepting the input from the user:

* To accept the input from a user, Java language has provided Scanner class in java.util package.
* In our program, first we have to create Scanner class object like below.

Scanner scan = new Scanner(System.in);

* We have to call the below methods to read the different types of input values.

1. nextByte()
2. nextShort()
3. nextInt()
4. nextLong()
5. nextFloat()
6. nextDouble()
7. nextBoolean()
8. nextLine()

Note: we don’t have nextChar() method.

/\*

\* write a program to find the addition

\* of two numbers by accepting the input

\* from the user.

\*/

// ctrl + shift + o is a shortcut to get imports

**import** java.util.Scanner;

**public** **class** MainClass {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter first number");

**int** x = scan.nextInt();

System.***out***.println("Enter second number");

**int** y = scan.nextInt();

**int** z = x + y;

System.***out***.println("Sum = " + z);

scan.close(); //optional

}

}

/\*

\* write a program to convert a fahrenheit

\* to celcius value

\*

\* formula: c = (f-32) \* 5 / 9

\*/

**import** java.util.Scanner;

**public** **class** FahrenheitToCelcius {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter fahrenheit value");

**double** f = scan.nextDouble();

**double** c = ( f - 32 ) \* 5 / 9;

System.***out***.println("celcius value = " + c);

scan.close();

}

}

/\*

\* write a program to calculate emi

\* To calculate emi, we need 3 inputs

\* 1. principle amount

\* 2. rate of interest per annum

\* 3. tenure in months n n

\* formula: emi = p \* r \* (1 + r) / (1 + r) - 1

\* here, r is the rate of interest per month

\* rate of interest per month can be calculated as,

\* r = pa / 12 / 100;

\*/

**import** java.util.Scanner;

**public** **class** EmiClass {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter principle amount");

**double** p = scan.nextDouble();

System.***out***.println("Enter rate of interest per annum");

**double** pa = scan.nextDouble();

System.***out***.println("Enter tenure in months");

**int** n = scan.nextInt();

//converting roi from pa to per month

**double** r = pa / 12 / 100;

**double** x = Math.*pow*(1+r, n);

**double** emi = p \* r \* x / (x - 1);

System.***out***.println("EMI = " + emi);

scan.close();

}

}

/\*

\* write a program to convert the given minutes

\* into days : hours : minutes format.

\* for example,

\* minutes = 1462

\* output:

\* 1 day : 0 hours : 22 minutes

\*/

**import** java.util.Scanner;

**public** **class** MinutesToDays {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter minutes ");

**int** minutes = scan.nextInt();

**int** d = minutes / 60 / 24;

**int** h = minutes / 60 % 24;

**int** m = minutes % 60;

System.***out***.println(d + " : days, " + h + " : hours, " + m + " : minutes") ;

}

}

Operators:

* an operator is a symbol, which performs a well-defined task.
* operator works with operands.
* we have unary operator, binary operator and a ternary operator.
* unary operator works with single operand
* binary operator works with two operands.
* ternary operator works with 3 parts.

unary operators:

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

++ increment operator

-- decrement operator

operand++ -- post increment

increment the operand after use

++operand -- pre increment

increment the operand before use

operand-- -- post decrement

decrement the operand after use

--operand -- pre decrement

decrement the operand before use

ex1:

int x = 10;

int y = x++;

S.o.p(x); // 11

S.o.p(y); // 10

ex2:

int x = 10;

int y = ++x;

S.o.p(x); //11

S.o.p(y); // 11

ex3:

int x = 10;

int y = 20;

int z = x++ + y--;

S.o.p(z); //30

ex4:

int x = 10;

int y = 20;

int z = ++x – y--;

S.o.p(x);

S.o.p(y);

S.o.p(z);

ex5:

int x = 10;

int y = ++x + x++;

S.o.p(x); //12

S.o.p(y); //22

Binary operators:

1. arithmetic operators
2. relational operators
3. logical operators
4. assignment operators
5. bitwise operators

arithemetic operators:

+ addition

* subtraction

\* multiplication

/ division(quotient)

% modulus(remainder)

ex1:int x = 10;

S.o.p( x / 2 ); // 5

S.o.p( x % 2 ); // 0

S.o.p( ++x % 2 ); // 1

S.o.p( x-- % 2 ); // 1

S.o.p(x); // 10

relational operators: returns either true or false

< less than

> greater than

<= less than or equals

>= greater than or equals

== equals

!= not equals

* relational operators are used to construct the prepositions/conditions.

ex:

int x = 10;

S.o.p( x++ > 10 ); // false

S.o.p( ++x > 11 ); // true

S.o.p( x == 12 ); // true

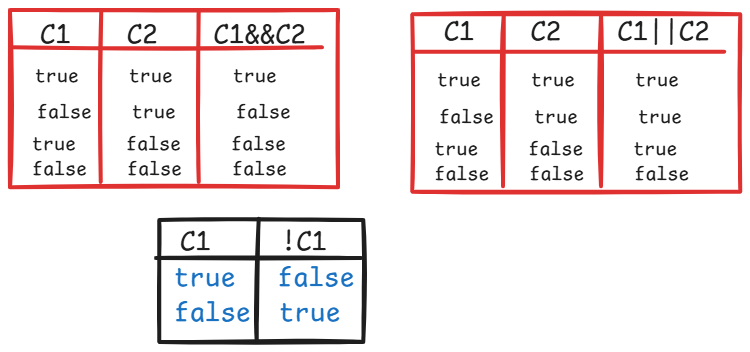
logical operators:

* These are used to combine two conditions into a single condition.

&& and

|| or

! not



ex1:

int x = 10, y = 20;

S.o.p( x <= 10 && y <= 24); // true

S.o.p( x <= 10 || y > 30 ); // true

S.o.p( !(x > 10) ); // true

ex2:

int x = 10;

S.o.p( x++ > 11 && ++x < 10); // false

S.o.p( x ); // 11

ex3:

int x = 11;

S.o.p( x++ > 10 || ++x < 11); //true

S.o.p(x); //12

ex4:

int x = 10;

S.o.p( ++x > 10 && x-- < 10 ); // false

S.o.p(x); // 10

Note: && operator verifies the second condition, only if the first condition is true.

The || operator verifies the second condition, only if the first condition is false.

assignment operators:

= assignment

+= add and assign

-= subtract and assign

\*= multiply and assign

/= divide and assign

%= modulus and assign

for ex, += is a compound assignment operator. It adds the right-hand operand to the left-hand operand and assigns the result to the left-hand operand.

ex: int x=10, y=20;

x += y;

S.o.p(x); // 30

S.o.p(y); // 20

ex:

int x = 5;

x++ %= 2;

S.o.p(x); // 1

ex:

char ch = ‘A’;

ch++;

S.o.p(ch); //B

ex:

boolean flag = false;

S.o.p( flag=true && !flag == flase); // true

Bitwise operators:

* Bitwise operators works on bits of the operands.
* Bitwise operators are mostly used in creating embedded system applications, device driver applications, cryptography applications, etc..

& bitwise AND

| bitwise OR

^ bitwise XOR

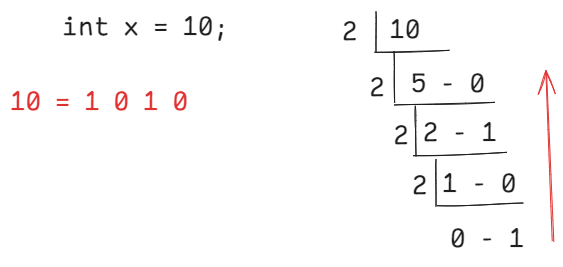
<< left shift

>> right shift

To work with bitwise operators, we need to convert a value to the equvalent binary number.

To convert a value to the binary system, divide the value with 2 until we get the quotient as zero.

ex:



ex:

int x = 16;

16 = 1 0 0 0 0

Bitwise AND(&): It returns 1, if the corresponding bits of the operands is 1, otherwise returns 0.

ex: int x = 10;

int y = 15;

10 = 1 0 1 0

15 = 1 1 1 1

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

10 & 15=1 0 1 0

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

S.o.p( x & y ); // 10

ex:

int x = 10;

int y = 20;

10 = 0 1 0 1 0

20 = 1 0 1 0 0

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

10 & 20= 0 0 0 0 0

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

S.o.p( x & y ); // 0

Bitwise OR( | ) : It returns 0, if the corresponding bits of the operands is 0, otherwise returns 1.

ex:

int x = 10;

int y = 15;

10 = 1 0 1 0

15 = 1 1 1 1

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

10 | 15= 1 1 1 1

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

S.o.p( x | y ); // 15

ex:

int x = 12;

int y = 17;

12 = 0 1 1 0 0

17 = 1 0 0 0 1

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

12 | 17 = 1 1 1 0 1

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

S.o.p( x | y ); //29

Bitwise XOR( ^ ) : It returns a bit 1, if the corresponding bits of the operands is opposite. Otherwise, returns a bit 0.

ex: int x = 10;

int y = 20;

10 = 0 1 0 1 0

20 = 1 0 1 0 0

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

10 ^ 20 = 1 1 1 1 0

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

S.o.p( x ^ y ); //30

ex: int x = 13;

int y = 19;

13 = 0 1 1 0 1

19 = 1 0 0 1 1

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

13 ^ 19 = 1 1 1 1 0

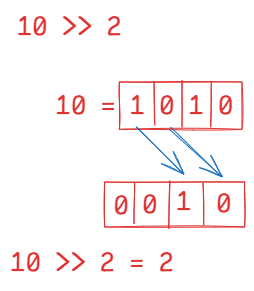
----------------------

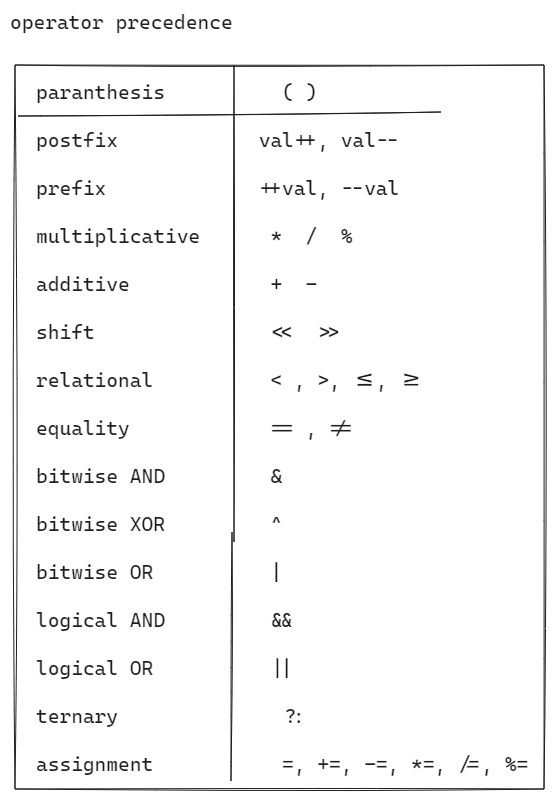
S.o.p( x ^ y ); // 30

left shift(<<) : It will shift the bits of an operand to the left by the given number of times and fills the empty spaces at right side with zero.



right shift(>>) : It will shift the bits of an operand to the right by the given number of times and fills the empty spaces at left side with zero.





ex:

int x = 3 + 9 % 2 \* 7 / (5 – 2);

S.o.p(x); // 5

ex:

int x = 7 + 7 / 7 / 7 \* 2 – (3 % 1);

S.o.p(x); // 7

ex:

boolean flag = (10 & 15) == 10;

S.o.p(flag); // true

ex:

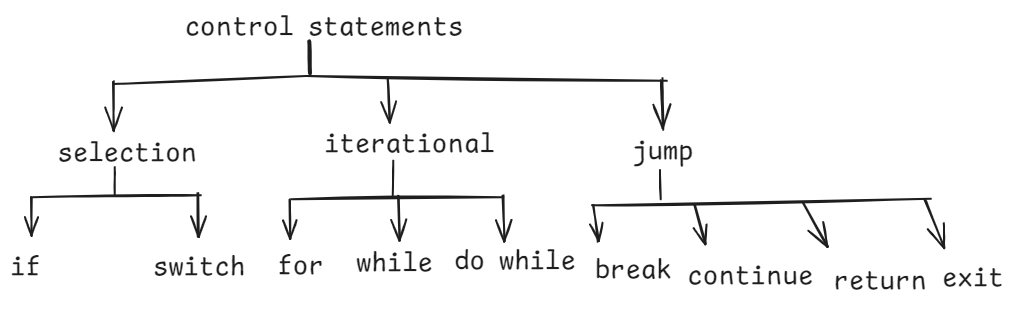
int x = 10 & 15 << 2;

S.o.p(x); // 8

control statements

* control statements will manage the flow of control in a program.
* control statements are 3 types.

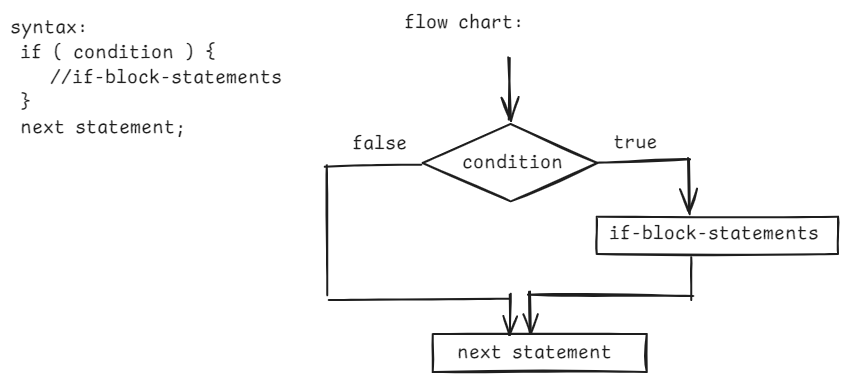
1. selection control statements
2. iterational control statements
3. jump control statements



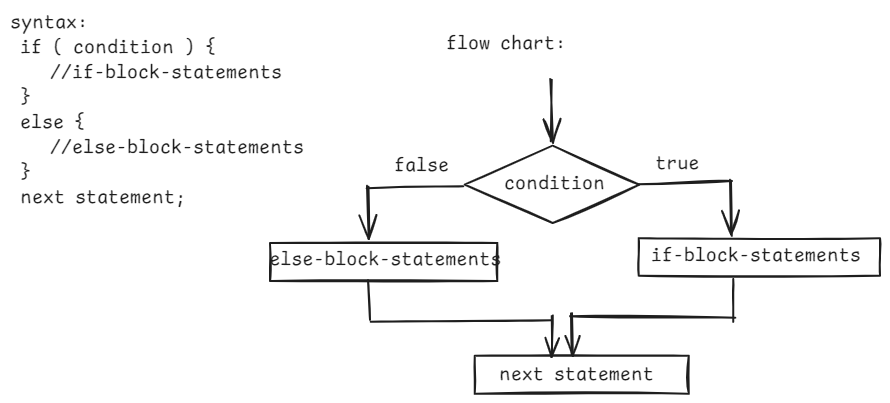
if statement:

1. simple if
2. if else
3. if else ladder
4. nested if

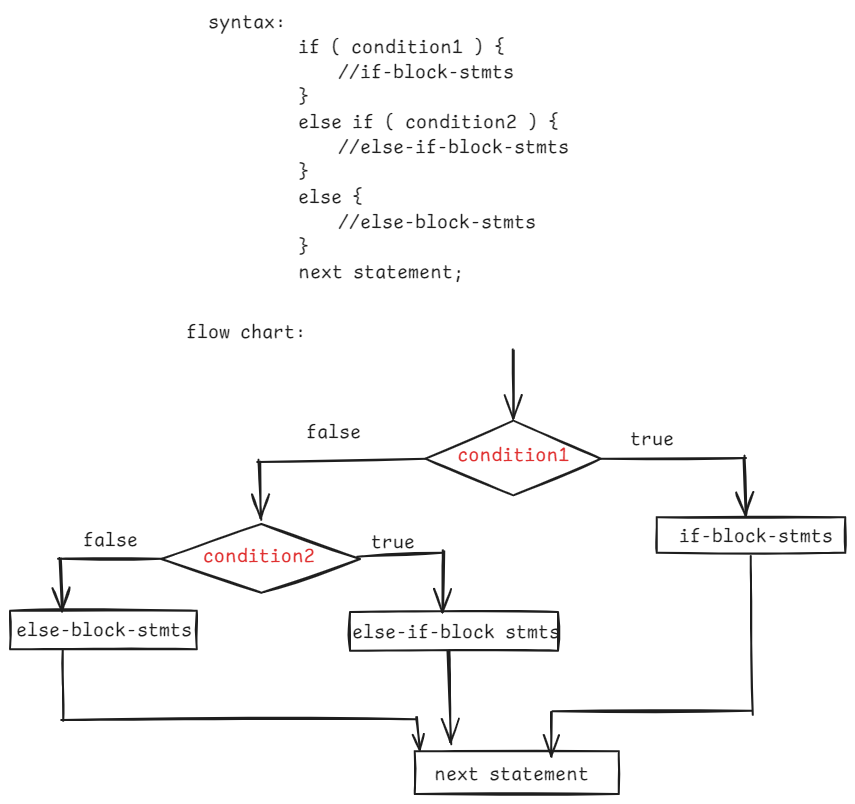
simple if:

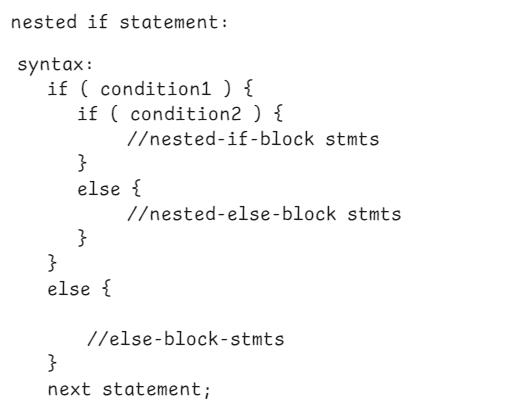


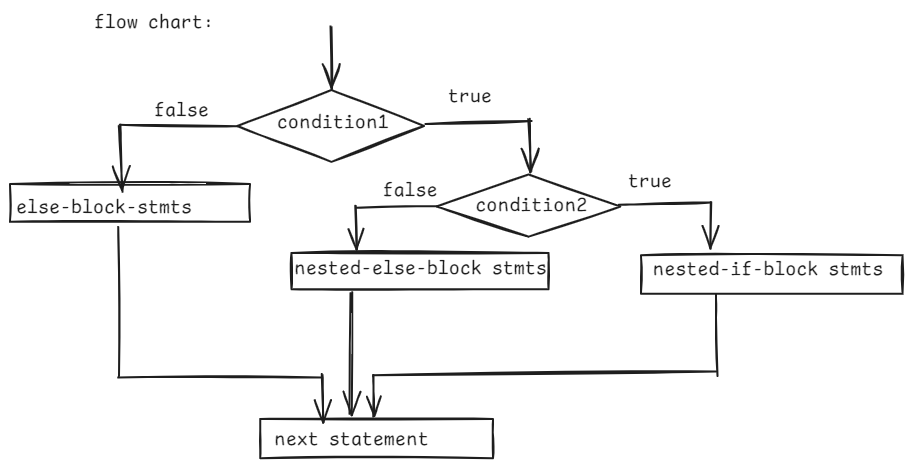
if else :



if else ladder:







/\*

\* write a program to check whether

\* a given number is even or odd

\*/

**import** java.util.Scanner;

**public** **class** Main {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter a number");

**int** n = scan.nextInt();

**if** ( n % 2 == 0 )

System.***out***.println( n + " : is even number");

**else**

System.***out***.println( n + " : is odd number");

scan.close();

}

}

/\*

\* write a program for the below requirement.

\* Take two numbers as input

\* if they are same then display the sum

\* if they are different and if one of the number

\* is 9 then display product of the two numbers

\* otherwise, display the output as -1

\*/

**import** java.util.Scanner;

**public** **class** MainClass {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter first number");

**int** a = scan.nextInt();

System.***out***.println("Enter second number");

**int** b = scan.nextInt();

**if** ( a == b ) {

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

}

**else** **if**( a == 9 || b == 9 ) {

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

}

**else** {

System.***out***.println("-1");

}

}

}

/\*

\* write a program to find the biggest

\* of three numbers using nested if conditions.

\*/

**import** java.util.Scanner;

**public** **class** MainClass {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter first number");

**int** a = scan.nextInt();

System.***out***.println("enter second number");

**int** b = scan.nextInt();

System.***out***.println("enter third number");

**int** c = scan.nextInt();

**if** ( a > b ) {

**if** ( a > c ) {

System.***out***.println( a + " : is biggest number");

}

**else** {

System.***out***.println(c + " : is biggest number");

}

}

**else** **if** ( b > c ) {

System.***out***.println( b + " : is biggest number");

}

**else** {

System.***out***.println( c + " : is biggest number");

}

}

}

/\*

\* write a program to implement the following.

\* Take the distance in kms as input and

\* calculate the delivery fee as below.

\* For first 3 kms delivery fee is 0

\* For next 3 kms delivery fee is 5 rupees per km

\* For the remaining kms delivery fee is 10 rupees per km

\*/

**import** java.util.Scanner;

**public** **class** MainClass {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter the distance in kilometers");

**int** distance = scan.nextInt();

**double** deliveryFee = 0.0;

**if**( distance <= 3 ) {

deliveryFee = 0.0;

}

**else** **if** ( distance <= 6 ) {

deliveryFee = (distance - 3) \* 5;

}

**else** {

deliveryFee = (distance - (distance -3)) \* 5 + (distance - 6) \* 10;

}

System.***out***.println("Delivery fee : " + deliveryFee);

}

}

switch statement:

* If you want to compare a single variable/a single expression with multiple values and the condition is equality then we can use switch statement.
* With statement, we can make the code more readable.

ex1:

i=1 one

i=9 nine

Here, we can want to compare a single variable with multiple values.

if(i==1)

S.o.p(“one”);

else if(I == 2)

S.o.p(“two”);

.

.

else if( I ==9)

S.o.p(“nine”);

\* In the above example, we can use switch statement also.

ex2:

if(a == b)

S.o.p(“same”);

else if(a > b)

S.o.p(“a is big”);

else

S.o.p(“b is big”);

\* here, we have greater than condition, so we can’t use switch statement for the above example.

ex3:

int x=4, y=1;

x+y is 5 then display “hello”

x+y is 8 then display “hi”

otherwise display “bye”

if(x+y == 5)

sop(“hello”);

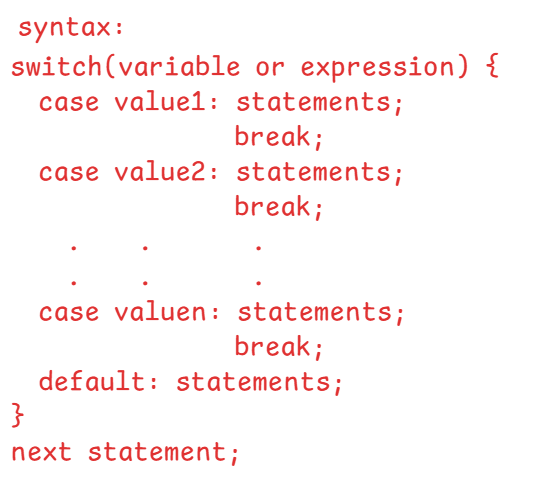
else if(x+y == 8);

s.o.p(“hi”);

else

sop(“bye”);

* here, a single expression is compared with multiple values and the conditions is equality. So, we can also use, switch statement.



* In switch statement, the variable can be integer or character or string type.
* If you provide expression as a parameter, then the result of that expression also must be an integer or character or string type.
* case values also, must be either integer or character or string type.
* switch statement compares the variable/expression parameter value with a case value.
* if a case value matches, then the corresponding statements of the case are executed.
* If break statement is not added, then the following cases statements are also executed until the break statement occurs.
* writing a default case is optional. If no case value matches the variable/expression value then default case is executed.

ex1:

**int** x = 3;

**switch**(x) {

**case** 1: System.***out***.println("ONE");

**break**;

**case** 2: System.***out***.println("TWO");

**break**;

**case** 3: System.***out***.println("THREE");

**break**;

**case** 4: System.***out***.println("FOUR");

**break**;

**default**: System.***out***.println("None");

}

output: THREE

ex2:

**int** x = 3;

**switch**(x) {

**case** 1: System.***out***.println("ONE");

**break**;

**case** 2: System.***out***.println("TWO");

**break**;

**case** 3: System.***out***.println("THREE");

**case** 4: System.***out***.println("FOUR");

**break**;

**default**: System.***out***.println("None");

}

output: THREE

FOUR

ex3:

**int** x = 7;

**switch**(x) {

**default**: System.***out***.println("None");

**case** 1: System.***out***.println("ONE");

**case** 2: System.***out***.println("TWO");

**case** 3: System.***out***.println("THREE");

**case** 4: System.***out***.println("FOUR");

}

output: None

ONE

TWO

THREE

FOUR

Note: If you are default case either at above all the cases or in the middle of the cases then we should add break statement in the default case.

ex4:

**char** ch = 'a';

**switch**(ch) {

**case** 97: System.***out***.println("ONE");

**break**;

**case** 98: System.***out***.println("TWO");

**break**;

**case** 99: System.***out***.println("THREE");

**break**;

**case** 100: System.***out***.println("FOUR");

**break**;

}

output: ONE

ex5:

**char** ch = 'd';

**switch**(ch) {

**case** 'a': System.***out***.println("ONE");

**break**;

**case** 'b': System.***out***.println("TWO");

**break**;

**case** 'c': System.***out***.println("THREE");

**break**;

**case** 'd': System.***out***.println("FOUR");

**break**;

}

output: FOUR

ex6:

**char** ch = 'e';

**switch**(ch) {

**case** 'a':

**case** 'e':

**case** 'i':

**case** 'o':

**case** 'u': System.***out***.println("vowel");

**break**;

**default**: System.***out***.println("consonent");

}

output: vowel

ex7:

String str ="ashok";

**switch**( str+" it" ) {

**case** "ashokit": System.***out***.println("ashokit, ameerpet, hyd");

**break**;

**case** "ashok it": System.***out***.println("ashok it, ameerpet, hyd");

**break**;

**default**: System.***out***.println("none");

}

output: ashok it, ameerpet, hyd

ternary operator(?:):

result = condition ? value1 : value2;

* If the condition is true, value1 will be assigned to the result, otherwise value2 will be assigned to the result.

ex1:

int a = 10, b = 5;

int c = a > b ? a : b;

S.o.p( c ); // 10

ex2:

int a = 5, b = 10, c = 8;

int max = (a>b && a>c)?a:(b>c)?b:c;

S.o.p(max); // 10

ex3:

**int** year = 2022;

String str = ( year % 4 == 0 && (year % 100 != 0 || year % 400 == 0)) ? "leap year" : "not a leap year";

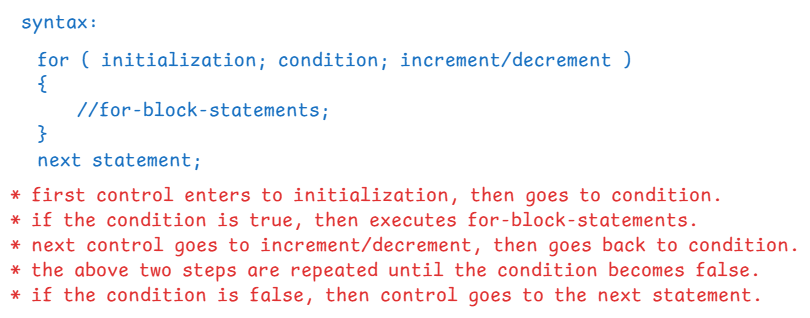
System.***out***.println(str);

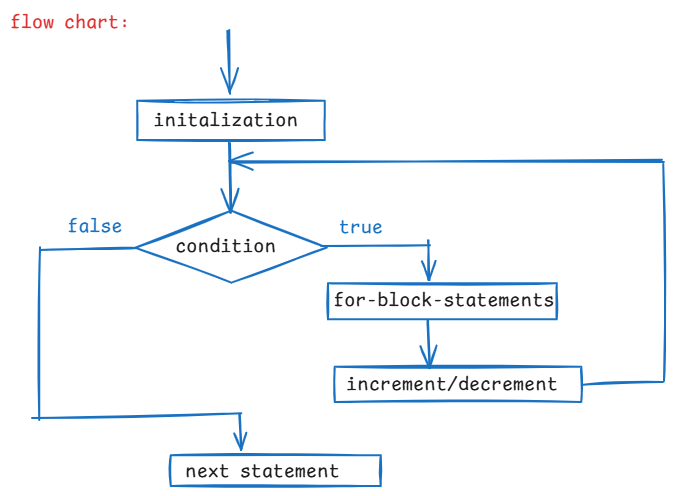
output: not a leap year

Iterational control statements

1. for loop :

* If you want to execute a group of statements repeatedly for a known number of times then use for loop.





ex1:

for ( int i=1; i <= 5; i++ ) {

System.out.println(“i = “+ i);

}

output:

i = 1

i = 2

i = 3

i = 4

i = 5

ex2:

for(int i=1; i<=5; i+=2) {

System.out.println(“i = “ + i);

}

output:

i = 1

i = 3

i = 5

ex3:

int i=1;

for( ; i <= 3; i++)

{

S.o.p(“i = “+ i);

}

S.o.p(“i = “ + i);

output:

i=1

i=2

i=3

i=4

ex4:

for(int i=1; i<=3; i++) {

S.o.p(“i=”+i);

}

S.o.p(“i=”+i);

output:

compile-time error, because the variable i is created in for statement, so it is visible in the for statement only and not visible at outside the for statement.

ex5:

for(int i=1,int j=5; i<=j; i++,j--) {

S.o.p(i+j);

}

output:

compile-time error. Because, in the initialization of the for statement, we have declared the datatype again for the variable j.

ex6:

for(int i=1, j=5; i <= j; i++, j--) {

S.o.p(i+j);

}

output:

6

6

6

ex7:

**for**( **int** i=1; **double** j=5; i<=j; i++, j-- ) {

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

}

output:

compile-time error. Because in a for statement we can have two semicolons only.

ex8:

int i=1;

for(;;) {

if( i \* 2 <= 6) {

S.o.p(i);

}

if(i > 3) {

break;

}

i++;

}

output:

1

2

3

ex9:

int i=1;

for(; i<=5; i++);

{

S.o.p(i);

}

output: 6

ex10:

for(int i=5; i > 1; i--) {

S.o.p( “ i = “ + i);

}

output:

i = 5

i = 4

i = 3

i = 2

ex11:

int n=3;

for(int i=1; 2\*i <=n; i++) {

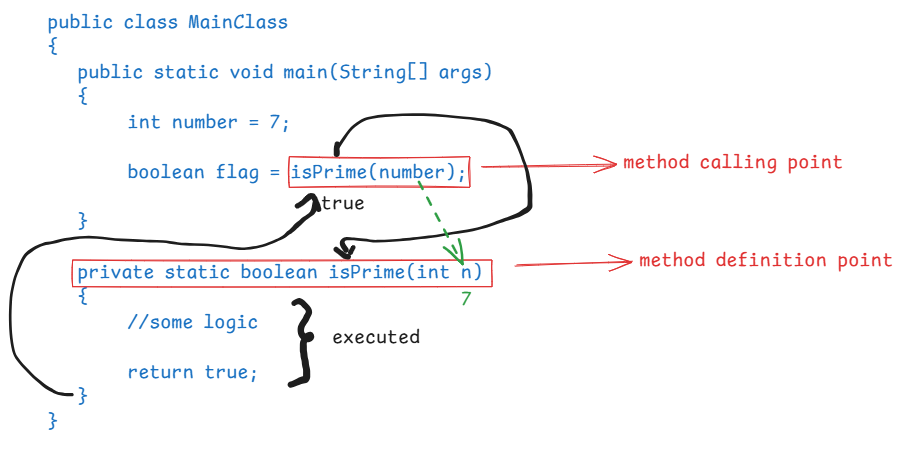
S.o.p(i);

}

output:

1

=======================================



/\*

\* write a program to find the sum of

\* n natural numbers.

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter a number");

**int** number = scan.nextInt();

*findSum*(number);

}

**private** **static** **void** findSum(**int** number) {

**int** sum = 0;

**for**( **int** i = 1; i <= number; i++ )

{

sum = sum + i;

}

System.***out***.println("sum = " + sum);

}

}

/\*

\* write a program to check whether a given

\* number is a prime number or not.

\*

\* prime number: If a number has only two factors, i.e., one and itself

\* then it is a prime number.

\* ex: number = 7

\* 7 : is a prime number

\* number = 9

\* 9 : is not a prime number

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter a number");

**int** number = scan.nextInt();

**boolean** flag = *isPrime*(number);

**if** ( flag == **true** )

System.***out***.println(number + " : is prime number");

**else**

System.***out***.println(number + " : is not a prime number");

}

**private** **static** **boolean** isPrime(**int** n) {

**boolean** flag = **true**;

**for** ( **int** i = 2; i <= Math.*sqrt*(n); i++) {

**if** ( n % i == 0 ) {

flag = **false**;

**break**;

}

}

**return** flag;

}

}

===========================================

/\*

\* write a program to check whether a given

\* number is a perfect number or not.

\*

\* perfect number: If sum of the factors of a number excluding

\* the number is equal to the same number, then

\* it is a perfect number.

\* ex: number = 6

\* 1 + 2 + 3 = 6

\* 6 : is a perfect number

\* ex: number = 24

\* 1 + 2 + 3 + 4 + 6 + 8 + 12 = 36

\* 24 : is not a perfect number

\* ex:

\* number = 28

\* 1 + 2 + 4 + 7 + 14 = 28

\* 28 : is a perfect number

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter the number");

**int** number = scan.nextInt();

**boolean** flag = *isPerfect*(number);

**if** ( flag == **true** )

System.***out***.println(number + " : is a perfect number");

**else**

System.***out***.println(number + " : is not a perfect number");

}

**private** **static** **boolean** isPerfect(**int** number) {

**int** sum = 0;

**for** ( **int** i = 1; i <= number / 2; i++ ) {

**if** ( number % i == 0 ) {

sum += i;

}

}

**if**(sum == number)

**return** **true**;

**else**

**return** **false**;

}

}

/\*

\* write a program to find the factorial

\* of a given number.

\* ex: number = 5

\* output: 120

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter a number");

**int** number = scan.nextInt();

**long** f = *factorial*(number);

System.***out***.println("Factorial of : " + number + " , is : " + f);

}

**private** **static** **long** factorial(**int** n) {

**long** fact = 1;

**for** ( **int** i = n; i >= 1; i-- ) {

fact \*= i;

}

**return** fact;

}

}

/\*

\* write a program to print the multiplication

\* table of a given number.

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter a number");

**int** number = scan.nextInt();

*printMultiplicationTable*(number);

}

**private** **static** **void** printMultiplicationTable(**int** n) {

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

System.***out***.println( n + " \* " + i + " = " + (n \* i) );

}

}

}

/\*

\* write a program to print the fibonacci series

\* of n terms.

\*

\* The first two terms of the fibonacci series are 0 and 1

\* The next term is a sum of the previous two terms.

\*

\* For example, if n = 5

\* output: 0 1 1 2 3

\* For example, if n = 10

\* output: 0 1 1 2 3 5 8 13 21 34

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter number of terms");

**int** terms = scan.nextInt();

*fibonacci*(terms);

}

**private** **static** **void** fibonacci(**int** terms) {

**int** firstTerm = 0;

**int** secondTerm = 1;

System.***out***.print(firstTerm + " " + secondTerm + " ");

**for**(**int** i = 3; i <= terms; i++) {

**int** nextTerm = firstTerm + secondTerm;

System.***out***.print(nextTerm + " ");

firstTerm = secondTerm;

secondTerm = nextTerm;

}

}

}

Nested for loop:

* writing a for loop within another for loop is called nested for loop.
* For each iteration of outer loop, the inner loop will be completely executed.

for ex:

for(int i=1; i<=3; i++) //outer loop

{

for(int j=1; j<=3; j++) //inner loop

{

S.o.println(j);

}

}

output: 1

2

3

1

2

3

1

2

3

ex2:

for(int i=1; i<=3; i++)

{

for(int j=1; j<=5; j++)

{

if( i + j == 4)

break;

S.o.p(j);

}

}

output:1

2

1

ex3:

for(int i=5; i > 1; i--)

{

for(int j=1; j<=i; j++)

{

if( 2 \* j + i > 10)

break;

S.o.p(j);

}

}

output:1

2

1

2

3

1

2

3

1

2

ex4: for(int i=1; i<=3; i++) {

for(int j=1; j<=i; j++);

S.o.p(i+j);

}

output: compile-time error

The error is not because of semi colon

for the inner loop.

The error is the variable j is visible

to the inner loop only. But the

S.o.p(i+j) statement is in outer loop,

so, j is not visible then compile-time

error.

ex5:

for(int i=1; i<=3; i++) {

int k = 0;

for(int j=1; j<=3; j++)

k += j;

S.o.p(k);

}

output:

6

6

6

Note: The variables created in outer loop are visible to the inner loop. But the variables created in the inner loop are not visible to the outer loop.

/\*

\* write a program to print all the prime numbers

\* within a given range

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter first number");

**int** x = scan.nextInt();

System.***out***.println("Enter second number");

**int** y = scan.nextInt();

**for**(**int** i = x; i <= y; i++ ) {

**boolean** flag = *isPrime*(i);

**if**(flag == **true**) {

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

}

}

}

**private** **static** **boolean** isPrime(**int** n) {

**boolean** flag = **true**;

**for**(**int** i = 2; i <= Math.*sqrt*(n); i++) {

**if** ( n % i == 0) {

flag = **false**;

**break**;

}

}

**return** flag;

}

}

/\*

\* write a program to print all perfect numbers

\* within a given range.

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter first number");

**int** x = scan.nextInt();

System.***out***.println("Enter second number");

**int** y = scan.nextInt();

**for**(**int** i = x; i <= y; i++ ) {

**boolean** flag = *isPerfect*(i);

**if**(flag == **true**) {

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

}

}

}

**private** **static** **boolean** isPerfect(**int** n) {

**int** sum = 0;

**for** ( **int** i=1; i <= n/2; i++ ) {

**if** ( n % i == 0 ) {

sum += i;

}

}

**if**( sum == n)

**return** **true**;

**else**

**return** **false**;

}

}

/\*

\* write a program to print the right angle

\* triangle star pattern for a given rows/lines.

\* For example, rows = 5

\* output:

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

\* \* \* \* \* \*

\* Note:

\* 1. we always use nested for loops to print the pattern

\* 2. we repeat the outer loop for rows/lines.

\* 3. we repeat the inner loop for columns, by somehow finding

\* the relationship between rows and columns.

\* 4. we always print star(\*) in the inner loop.

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter number of rows");

**int** rows = scan.nextInt();

*printPattern*(rows);

}

**private** **static** **void** printPattern(**int** rows) {

//outer loop for rows/lines

**for**( **int** i = 1; i <= rows; i++ ) {

//inner loop for columns

**for** ( **int** j = 1; j <= i; j++ ) {

System.***out***.print("\*" + " ");

}

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

}

}

}

/\*

\* write a program to print inverted right angle triangle

\* pattern for the given number of rows.

\* For example,

\* if rows = 5

\* output :

\* \* \* \* \* \*

\* \* \* \* \*

\* \* \* \*

\* \* \*

\* \*

\*

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter number of rows");

**int** rows = scan.nextInt();

*printPattern*(rows);

}

**private** **static** **void** printPattern(**int** rows) {

**for** ( **int** i = 1; i <= rows; i++ ) {

**for** ( **int** j = 1; j <= rows - i + 1; j++ ) {

System.***out***.print("\*" + " ");

}

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

}

}

}

/\*

\* write a program to print left angle

\* triangle pattern for the given rows.

\* For example,

\* if rows = 5,

\* output:

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

\* \* \* \* \* \*

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter number of rows");

**int** rows = scan.nextInt();

*printPattern*(rows);

}

**private** **static** **void** printPattern(**int** rows) {

//outer for loop : rows

**for** ( **int** i = 1; i <= rows; i++ ) {

//inner for loop1 : spaces

**for** ( **int** j = 1; j <= 2 \* (rows - i); j++ ) {

System.***out***.print(" ");

}

//inner for loop2: stars

**for** ( **int** k = 1; k <= i; k++ ) {

System.***out***.print("\*" + " ");

}

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

}

}

}

/\*

\* write a program to print the inverted left angle triangle

\* pattern for the given rows.

\* For example,

\* if rows = 5,

\* output:

\* \* \* \* \* \*

\* \* \* \* \*

\* \* \* \*

\* \* \*

\* \*

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter number of rows");

**int** rows = scan.nextInt();

*printPattern*(rows);

}

**private** **static** **void** printPattern(**int** rows) {

//outer for loop : rows

**for** ( **int** i = 1; i <= rows; i++ ) {

//inner for loop1 : spaces

**for** ( **int** j = 1; j <= 2 \* (i - 1); j++ ) {

System.***out***.print(" ");

}

//inner for loop2: stars

**for** ( **int** k = 1; k <= rows - i + 1; k++ ) {

System.***out***.print("\*" + " ");

}

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

}

}

}

/\*

\* write a program to print the pyramid pattern

\* for the given rows.

\* For example,

\* if rows = 5,

\* output:

\* \*

\* \* \* \*

\* \* \* \* \* \*

\* \* \* \* \* \* \* \*

\* \* \* \* \* \* \* \* \* \*

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter number of rows");

**int** rows = scan.nextInt();

*printPattern*(rows);

}

**private** **static** **void** printPattern(**int** rows) {

//outer for loop : rows

**for** ( **int** i = 1; i <= rows; i++ ) {

//inner for loop1 : spaces

**for** ( **int** j = 1; j <= 2 \* (rows - i); j++ ) {

System.***out***.print(" ");

}

//inner for loop2: stars

**for** ( **int** k = 1; k <= 2 \* i - 1; k++ ) {

System.***out***.print("\*" + " ");

}

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

}

}

}

/\*

\* write a program to display inverted pyramid pattern

\* for the given rows.

\* For example,

\* if rows = 5,

\* output:

\* \* \* \* \* \* \* \* \* \*

\* \* \* \* \* \* \* \*

\* \* \* \* \* \*

\* \* \* \*

\* \*

\*/

**import** java.util.Scanner;

**public** **class** Main {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter number of rows");

**int** rows = scan.nextInt();

*printPattern*(rows);

}

**private** **static** **void** printPattern(**int** rows) {

//outer for loop : rows

**for** ( **int** i = 1; i <= rows; i++ ) {

//inner for loop1 : spaces

**for** ( **int** j = 1; j <= 2 \* (i - 1); j++ ) {

System.***out***.print(" ");

}

//inner for loop2: stars

**for** ( **int** k = 1; k <= 2 \* ( rows - i) + 1; k++ ) {

System.***out***.print("\*" + " ");

}

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

}

}

}

/\*

\* write a program to print the below Hollow

\* square pattern for the given rows.

\* For example,

\* if rows = 5,

\* output:

\* \* \* \* \* \*

\* \* \*

\* \* \*

\* \* \*

\* \* \* \* \* \*

\*/

**import** java.util.Scanner;

**public** **class** MainClass {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter number of rows");

**int** rows = scan.nextInt();

*printPattern*(rows);

}

**private** **static** **void** printPattern(**int** rows) {

**for** ( **int** i = 1; i <= rows; i++ ) {

**for** ( **int** j = 1; j <= rows; j++ ) {

**if** ( i == 1 || j == 1 || i == rows || j == rows ) {

System.***out***.print("\*" + " ");

}

**else** {

System.***out***.print(" "); //2 white spaces

}

}

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

}

}

}

/\*

\* write a program to print the below diamond

\* pattern for the given rows.

\* For example,

\* if rows = 5,

\* output:

\* \*

\* \* \* \*

\* \* \* \* \* \*

\* \* \* \* \* \* \* \*

\* \* \* \* \* \* \* \* \* \*

\* \* \* \* \* \* \* \*

\* \* \* \* \* \*

\* \* \* \*

\* \*

\*

\*/

**import** java.util.Scanner;

**public** **class** MainClass {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter number of rows");

**int** rows = scan.nextInt();

*printPattern*(rows);

}

**private** **static** **void** printPattern(**int** rows) {

// outer for loop : rows

**for** (**int** i = 1; i <= rows; i++) {

// inner for loop1 : spaces

**for** (**int** j = 1; j <= 2 \* (rows - i); j++) {

System.***out***.print(" ");

}

// inner for loop2: stars

**for** (**int** k = 1; k <= 2 \* i - 1; k++) {

System.***out***.print("\*" + " ");

}

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

}

// outer for loop : rows

**for** (**int** i = 2; i <= rows; i++) {

// inner for loop1 : spaces

**for** (**int** j = 1; j <= 2 \* (i - 1); j++) {

System.***out***.print(" ");

}

// inner for loop2: stars

**for** (**int** k = 1; k <= 2 \* (rows - i) + 1; k++) {

System.***out***.print("\*" + " ");

}

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

}

}

}

/\*

\* write a program to print the below pattern

\* For example

\* if rows = 5,

\* output:

\* 1

\* 1 2

\* 1 2 3

\* 1 2 3 4

\* 1 2 3 4 5

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter number of rows");

**int** rows = scan.nextInt();

*printPattern*(rows);

}

**private** **static** **void** printPattern(**int** rows) {

//outer loop for rows/lines

**for**( **int** i = 1; i <= rows; i++ ) {

//inner loop for columns

**for** ( **int** j = 1; j <= i; j++ ) {

System.***out***.print(j + " ");

}

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

}

}

}

/\*

\* write a program to print the pascal triangle

\* for the given rows.

\* For example,

\* rows = 5,

\* output:

\* 1

\* 1 1

\* 1 2 1

\* 1 3 3 1

\* 1 4 6 4 1

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter rows");

**int** rows = scan.nextInt();

*printPascalTriangle*(rows);

}

**private** **static** **void** printPascalTriangle(**int** rows) {

//outer loop : rows

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

//inner loop1 : spaces

**for**(**int** j=1; j <= rows - i - 1; j++) {

System.***out***.print(" ");

}

//inner loop2: terms

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

System.***out***.print(*factorial*(i)/ (*factorial*(j) \* *factorial*(i-j)) + " ");

}

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

}

}

**private** **static** **int** factorial(**int** n) {

**int** fact = 1;

**for**(**int** i=1; i<=n; i++)

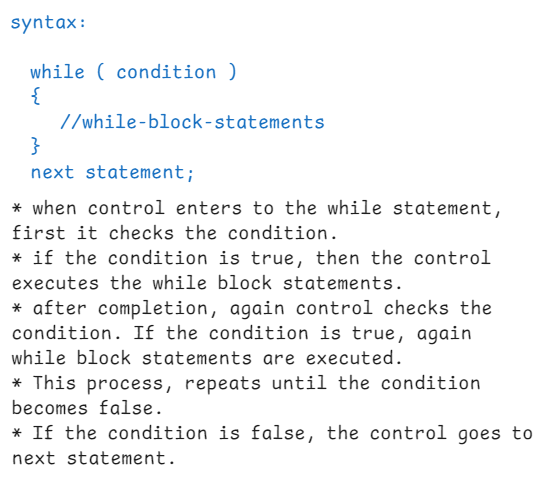
fact = fact \* i;

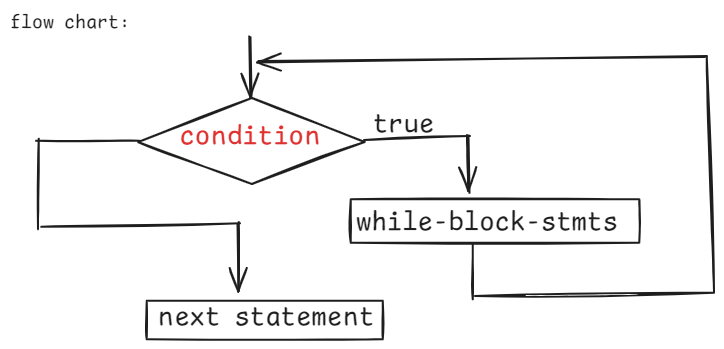
**return** fact;

}

}

while loop





Q) what is the difference between for loop and while loop?

A) 1. If you know how many times to repeat the loop, then use for loop and if you don’t know how many times to repeat then choose while loop.

2. In for statement, we can write initialization, condition, increment/decrement in a single line. But in while statement, we can write only the condition.

/\*

\* write a program to find the sum

\* of digits of a given number

\* for example,

\* if number = 719

\* output : 7 + 1 + 9 = 17

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter a number");

**int** number = scan.nextInt();

*digitsSum*(number);

}

**private** **static** **void** digitsSum(**int** number) {

**int** sum = 0;

**int** temp = number;

**while** ( number > 0 ) {

**int** rem = number % 10;

sum = sum + rem;

number = number / 10;

}

System.***out***.println("number : " + temp);

System.***out***.println("sum of digits : " + sum);

}

}

/\*

\* write a program to check whether a given

\* number is strong number or not.

\* strong number : if sum of the factorials of each digit of

\* a number is equal to the same number then

\* it is a strong number.

\* For example:

\* number = 145

\* output: 1! + 4! + 5! = 1 + 24 + 120 = 145

\* 145: is a strong number

\*/

**import** java.util.Scanner;

**public** **class** MainClass {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter a number");

**int** number = scan.nextInt();

**boolean** flag = *checkStrong*(number);

**if** ( flag == **true** )

System.***out***.println(number + " : is a strong number");

**else**

System.***out***.println(number + " : is not a strong number");

}

**private** **static** **boolean** checkStrong(**int** number) {

**int** sum = 0;

**int** temp = number;

**while** ( number > 0 ) {

**int** rem = number % 10;

**int** fact = 1;

**for**(**int** i = 1; i <= rem; i++) {

fact = fact \* i;

}

sum = sum + fact;

number = number / 10;

}

**if** ( sum == temp )

**return** **true**;

**else**

**return** **false**;

}

}

/\*

\* write a program to check whether a given

\* number is an armstrong number or not.

\* armstrong number : if sum of the n th power of each digit

\* of a number is equal to the same number

\* then it is an armstrong number.

\* Here n is the length of the number.

\* For example:

\* number = 153

\* 3 3 3

\* output : 1 + 5 + 3

\* : 1 + 125 + 27

\* : 153

\* 153 : is an armstrong number

\* For example:

\* number = 1634

\* 4 4 4 4

\* output: 1 + 6 + 3 + 4

\* : 1 + 1296 + 81 + 256

\* : 1634

\* 1634 : is an armstrong number

\*

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter a number");

**int** number = scan.nextInt();

**boolean** flag = *checkArmStrong*(number);

**if** ( flag == **true** )

System.***out***.println(number + " : is an armstrong number");

**else**

System.***out***.println(number + " : is not an armstrong number");

}

**private** **static** **boolean** checkArmStrong(**int** number) {

**int** sum = 0;

**int** temp = number;

//convert the number to string

String str = String.*valueOf*(number);

//find the length of the string

**int** length = str.length();

**while**( number > 0 ) {

**int** rem = number % 10;

sum = sum + (**int**) Math.*pow*(rem, length);

number = number / 10;

}

**if** ( sum == temp )

**return** **true**;

**else**

**return** **false**;

}

}

/\*

\* write a program to convert a binary number

\* to a decimal number

\* For example:

\* if number = 1010

\* output: 10

\* if number = 101010

\* output: 42

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter a binary number");

**int** number = scan.nextInt();

*binaryToDecimal*(number);

}

**private** **static** **void** binaryToDecimal(**int** number) {

**int** decimal = 0;

**int** x = 0;

**int** temp = number;

**while**(number > 0) {

**int** rem = number % 10;

decimal = decimal + rem \* (**int**)Math.*pow*(2, x);

number = number / 10;

x++;

}

System.***out***.println("Binary number : " + temp);

System.***out***.println("Decimal number : " + decimal);

}

}

/\*

\* write a program to check whether a given

\* number is a spy number or not.

\*

\* spy number : If sum of the digits of a number and

\* product of the digits of a number are

\* same, then it is a spy number.

\* for example: number = 123

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter a number");

**int** number = scan.nextInt();

**boolean** flag = *isSpy*(number);

**if** ( flag == **true** )

System.***out***.println(number + " : is a spy number");

**else**

System.***out***.println(number + " : is not a spy number");

}

**private** **static** **boolean** isSpy(**int** number) {

**int** sum = 0;

**int** product = 1;

**while**( number > 0 ) {

**int** rem = number % 10;

sum = sum + rem;

product = product \* rem;

number = number / 10;

}

**if**(sum == product)

**return** **true**;

**else**

**return** **false**;

}

}

/\*

\* write a program to check whether a given

\* number is palindrome or not.

\* palindrome : if reverse of a number is equals to the same number

\* then it is palindrome.

\* ex: 121, 1221, 343

\*

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter a number");

**int** number = scan.nextInt();

**boolean** flag = *isPalindrome*(number);

**if**( flag == **true** )

System.***out***.println(number + " : is a palindrome");

**else**

System.***out***.println(number + " : is not a palindrome");

}

**private** **static** **boolean** isPalindrome(**int** number) {

**int** reverse = 0;

**int** temp = number;

**while**( number > 0 ) {

**int** rem = number % 10;

reverse = reverse \* 10 + rem;

number = number / 10;

}

**if** ( reverse == temp )

**return** **true**;

**else**

**return** **false**;

}

}

/\*

\* write a program to check whether a given number is

\* automorphic number or not.

\* automorphc number : If the square of a number ends with the same digits

\* of a number then it is automorphic number.

\* ex:

\* 5 = 5 \* 5 = 25

\* 25 = 25 \* 25 = 625

\*/

**import** java.util.Scanner;

**public** **class** Main {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter a number");

**int** number = scan.nextInt();

**boolean** flag = *isAutomorphic*(number);

**if**( flag == **true**)

System.***out***.println(number + " : is automorphic number");

**else**

System.***out***.println(number + " : is not automorphic number");

}

**private** **static** **boolean** isAutomorphic(**int** number) {

**int** temp = number;

**int** count = 0;

//This while loop is for counting

//the number of digits in the given

//number

**while**( number > 0 ) {

number = number / 10;

count++;

}

**int** squareNum = temp \* temp;

**int** result = squareNum % (**int**)Math.*pow*(10, count);

**if** ( result == temp )

**return** **true**;

**else**

**return** **false**;

}

}

/\*

\* write a program to print the given number

\* in words.

\* For example,

\* number = 1326

\* output: ONE THREE TWO SIX

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter a number");

**int** number = scan.nextInt();

*printInWords*(number);

}

**private** **static** **void** printInWords(**int** number) {

String str ="";

**int** temp = number;

**while**(number > 0) {

**int** rem = number % 10;

**switch**(rem) {

**case** 0: str = "Zero" + str;

**break**;

**case** 1: str = "One" + str;

**break**;

**case** 2: str = "Two" + str;

**break**;

**case** 3: str = "Three" + str;

**break**;

**case** 4: str = "Four" + str;

**break**;

**case** 5: str = "Five" + str;

**break**;

**case** 6: str = "Six" + str;

**break**;

**case** 7: str = "Seven" + str;

**break**;

**case** 8: str = "Eight" + str;

**break**;

**case** 9: str = "Nine" + str;

**break**;

}

number = number / 10;

}

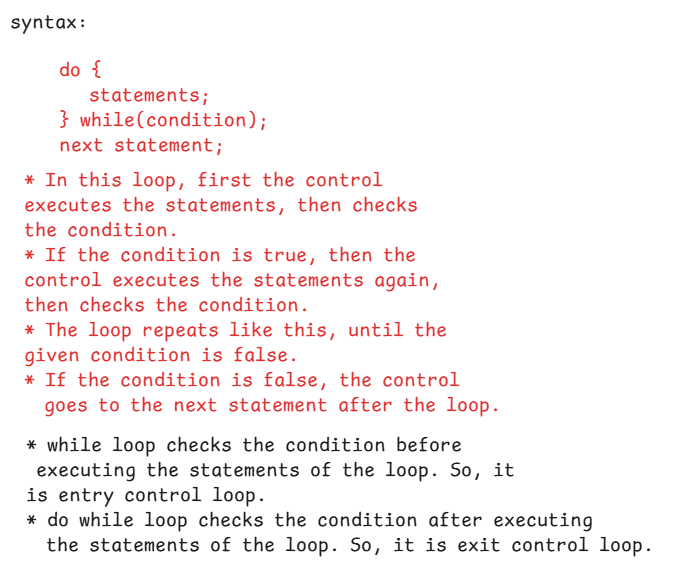
System.***out***.println("Number : " + temp);

System.***out***.println("In Words : " + str);

}

}

do while loop:



**import** java.util.Scanner;

**public** **class** MainClass {

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

Scanner scan = **new** Scanner(System.***in***);

**int** option = 0;

**do** {

System.***out***.println("===Menu===");

System.***out***.println("1. add");

System.***out***.println("2. multiply");

System.***out***.println("3. divide");

System.***out***.println("4. exit");

System.***out***.println("Enter your option ");

option = scan.nextInt();

**switch** ( option ) {

**case** 1: System.***out***.println("enter first number");

**int** a = scan.nextInt();

System.***out***.println("enter second number");

**int** b = scan.nextInt();

**int** c = a+b;

System.***out***.println("Adition : " + c);

**break**;

**case** 2: System.***out***.println("enter first number");

**int** x = scan.nextInt();

System.***out***.println("enter second number");

**int** y = scan.nextInt();

**int** z = x \* y;

System.***out***.println("Multiplication : " + z);

**break**;

**case** 3: System.***out***.println("enter first number");

**int** m = scan.nextInt();

System.***out***.println("enter second number");

**int** n = scan.nextInt();

**int** r = m / n;

System.***out***.println("Division : " + r);

**break**;

} //end switch

} **while**( option != 4);

}

}

Jumping control statements:

* break
* continue
* return
* exit

. break statement can be used to break the control from a switch statement or from a loop statement.

ex1:

**int** x = 10;

**for** ( **int** i=1; i <= 10; i+=2) {

**if**( 2 \* i == x )

**break**;

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

}

output: 1

3

ex2:

**for** ( **int** i=10; i > 0; i--) {

**for**( **int** j = 1; j <= i; j++ ) {

**if**( i + j < 7)

**break**;

System.***out***.print(j +" ");

}

}

output: 1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 1 2 3 4 5 6

ex3:

**for**(**int** i=1; i<=5; i++) {

**for**(**int** j=1; j<=i; j++) {

**if**(2 \* (i+j) > 8 )

**break**;

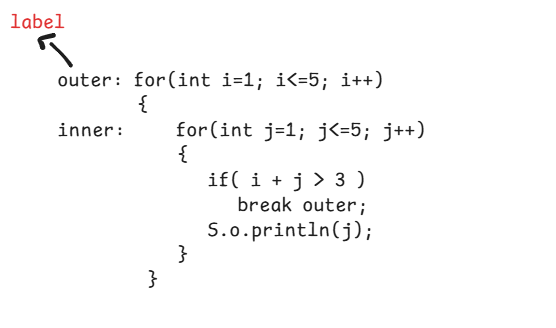
System.***out***.print(j + " ");

}

}

output: 1 1 2 1

ex4:



output: 1

2

continue :

The continue statement is used to skip the current iteration of a loop and move the control to the next iteration.

Typically we use continue statement, to skip the execution of remaining code in a loop if a specific condition is true and to move the control to the next iteration.

ex:

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

if ( i % 2 == 0 )

continue;

S.o.println(i);

}

output:

1

3

5

7

9

ex:

outer: for(int i=1; i<=5; i++) {

inner: for(int j=1; j<=5; j++) {

if( i + j > 5)

continue outer;

S.o.print(j + “ “);

}

}

output: 1 2 3 4 1 2 3 1 2 1

Arrays

* To store a value we need a variable.
* To store mulitple values, we need multiple variables.
* If we create mulitple variables in a program, the complexity will increase and readability will be decreased.
* If multiple variables are created then the memory is allocated at different addresses in the JVM, so fetching or storing the value into a variable is a time taken process.
* So, the solution to store mulitple values in a single variable at sequential addressess is an array.
* Each value stored in array variables is an element and each element has an index.
* The index of the first element is 0.

creating an array:

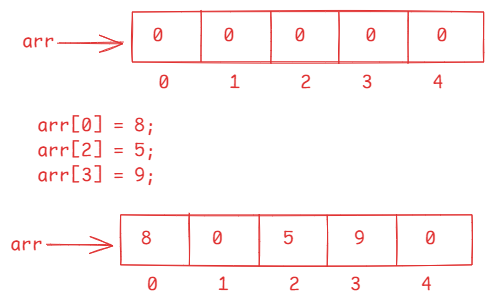
datatype[] arrayname = new datatype[size];

(or)

datatype arrayname[] = new datatype[size];

ex:

int[] arr = new int[5];



* Once an array is created, its size is fixed. We can’t increase or decrese the size after creation.
* We can use length attribute to find the size of an array.

for ex:

int[] arr = new int[10];

System.out.println( arr.length ); // 10

arr[10] = 97; // at runtime, ArrayIndexOutOfBoundsException will be thrown.

* you can create a static array like below.

for ex:

int[] arr = new int[] { 3, 0, 2, 7, 5 };

(or)

int[] arr = { 3, 0, 2, 7, 5 };

/\*

\* write a program to find the sum of the array

\* elements

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter the size of the array");

**int** n = scan.nextInt();

//create an array

**int** arr[] = **new** **int**[n];

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

System.***out***.println("Enter the element at index : " + i);

arr[i] = scan.nextInt();

}

*findSumOfElements*(arr);

}

**private** **static** **void** findSumOfElements(**int**[] arr) {

**int** sum = 0;

**for**(**int** i = 0; i < arr.length; i++ ) {

sum += arr[i];

}

System.***out***.println("Sum of the elements : " + sum);

}

}

/\*

\* write a program to find the sum of even

\* and odd elements of an array separately.

\*/

**import** java.util.Scanner;

**public** **class** MainClass {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter the size of the array");

**int** n = scan.nextInt();

//create an array

**int** arr[] = **new** **int**[n];

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

System.***out***.println("Enter the element at index : " + i);

arr[i] = scan.nextInt();

}

*findEvenOddSumOfElements*(arr);

}

**private** **static** **void** findEvenOddSumOfElements(**int**[] arr) {

**int** evenSum = 0, oddSum = 0;

**for** (**int** i= 0; i < arr.length; i++ ) {

**if** ( arr[i] % 2 == 0 )

evenSum += arr[i];

**else**

oddSum += arr[i];

}

System.***out***.println("even sum = " + evenSum);

System.***out***.println("odd sum = " + oddSum);

}

}

/\*

\* write a program to search for an element in

\* the given array using linear search

\* linear search: compare each element of an array

\* with searching element. if matched

\* then element is found.

\* if no element of an array matches with

\* searching element, then element is not found.

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter the size of the array");

**int** n = scan.nextInt();

//create an array

**int** arr[] = **new** **int**[n];

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

System.***out***.println("Enter the element at index : " + i);

arr[i] = scan.nextInt();

}

System.***out***.println("enter searching element");

**int** searchingElement = scan.nextInt();

*linearSearch*(arr, searchingElement);

}

**private** **static** **void** linearSearch(**int**[] arr, **int** searchingElement) {

**boolean** found = **false**;

**for**( **int** i = 0; i < arr.length; i++ ) {

**if**( arr[i] == searchingElement ) {

System.***out***.println("Element found at index : "+ i);

found = **true**;

**break**;

}

}

**if**(found == **false**) {

System.***out***.println("Element not found");

}

}

}

/\*

\* write a program to search for an element in

\* the given array using binary search.

\* binary search: For binary search, the array elements must be

\* sorted in ascending order.

\* Find the midpoint of the array and compare the

\* midpoint element with searching element.

\* if searching element > mid element, then do the

\* search at right side of the array. It means from

\* mid+1 to n-1.

\* if searching element < mid element, then do the

\* search at left side of the array. It means from

\* 0 to mid-1.

\* Apply this binary search recursively, until the

\* element is found at the mid point.

\*/

**import** java.util.Arrays;

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter the size of the array");

**int** n = scan.nextInt();

**int**[] arr = **new** **int**[n];

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

System.***out***.println("Enter the value at index : " + i);

arr[i] = scan.nextInt();

}

System.***out***.println("Enter searching element");

**int** searchingElement = scan.nextInt();

*binarySearch*(arr, searchingElement);

}

**private** **static** **void** binarySearch(**int**[] arr, **int** searchingElement) {

//sorting the array(ascending order)

Arrays.*sort*(arr);

**int** low = 0, high = arr.length - 1;

**boolean** flag = **false**;

**while**( low <= high ) {

**int** mid = (low + high)/2;

**if**( searchingElement == arr[mid]) {

System.***out***.println("Element found at index : " + mid);

flag = **true**;

**break**;

}

**else** **if** ( searchingElement > arr[mid]) {

low = mid+1;

}

**else** {

high = mid-1;

}

}

**if**( flag == **false**)

System.***out***.println("Element not found");

}

}

/\*

\* write a program to find the max consecutively

\* repeated element of the array.

\* For example.

\* int[] arr = { 1,2,2,1,3,1};

\* output: max consecutively repeated element=2

\* times = 2

\* int[] arr = { 1, 2, 2, 3, 3, 3, 3, 2, 2, 2, 1 };

\* output: max consecutively repeated element = 3

\* times = 4

\*/

**public** **class** Main {

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

**int**[] arr = { 1, 2, 2, 2, 1, 1, 3, 1 };

*maxConsecutivelyRepeated*(arr);

}

**private** **static** **void** maxConsecutivelyRepeated(**int**[] arr) {

**int** count = 1;

**int** maxi = 0, element = 0;

**for**( **int** i = 0; i < arr.length - 1; i++ ) {

**if** ( arr[i] == arr[i+1]) {

count++;

}

**else** {

count = 1;

}

**if**( count > maxi ) {

maxi = count;

element = arr[i];

}

}

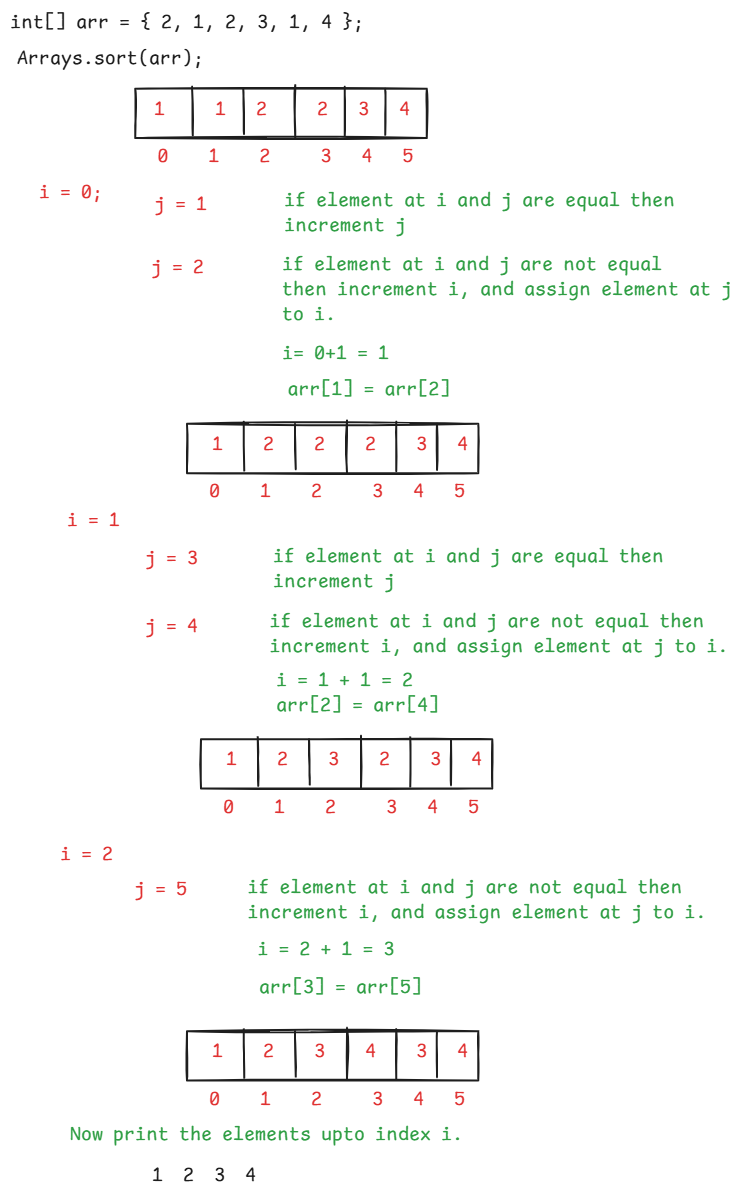
System.***out***.println("max consecutively repeated element : " + element);

System.***out***.println("times : " + maxi);

}

}

Remove the duplicate elements from an array.



/\*

\* write a program to remove the duplicate elements

\* from the given array.

\*

\* For ex:

\* int[] arr = { 2, 1, 2, 3, 1, 4}

\* output: 1 2 3 4

\*/

**import** java.util.Arrays;

**public** **class** Solution {

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

**int**[] arr = { 2, 1, 3, 1, 4, 2, 5 };

*removeDuplicates*(arr);

}

**private** **static** **void** removeDuplicates(**int**[] arr) {

// **TODO** Auto-generated method stub

//sort the array

Arrays.*sort*(arr);

System.***out***.println("Array elements before removing duplicates");

**for**(**int** k=0; k < arr.length; k++) {

System.***out***.print(arr[k] + " ");

}

**int** i = 0;

**for**(**int** j = 1; j < arr.length; j++) {

**if**( arr[i] == arr[j] ) {

**continue**;

}

i++;

arr[i] = arr[j];

}

System.***out***.println("\n======================");

System.***out***.println("Array elements after removing duplicates");

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

System.***out***.print(arr[k] + " ");

}

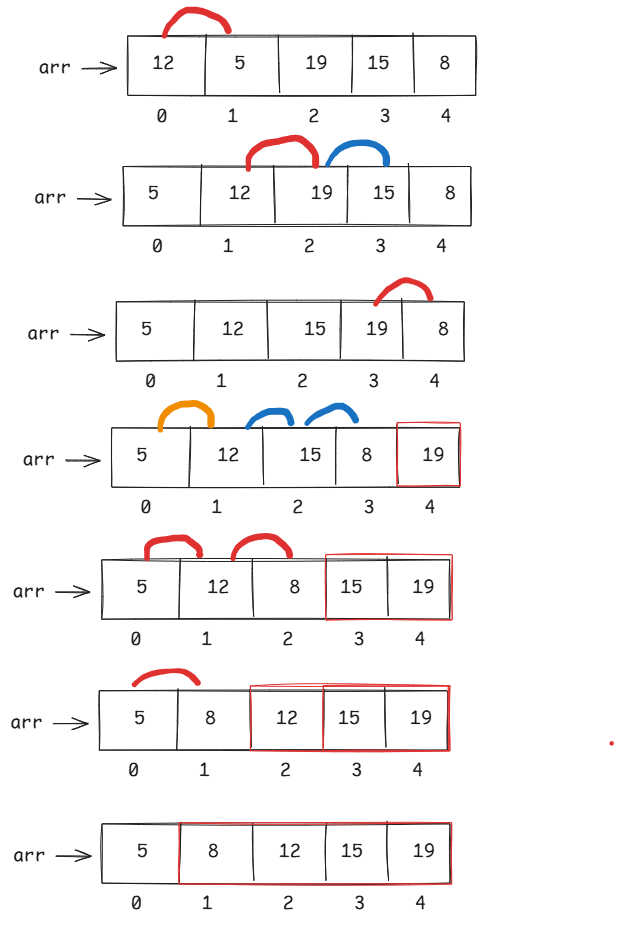
}

}

Bubble sort :

* The bubble sort technique works like below.

1. start from the beginning of the array
2. compare first element with second element and if first element is greater then swap them.
3. Move to the next pair, and repeat the same for all the array elements.
4. After one pass, the largest element is moved to the end of the array. It means, the largest element is bubbled.
5. Now again start from the beginning of the array, repeat the same process upto last but one element, so that second largest element is bubbled.
6. Repeat the process until n-1 elements are bubbled.



/\*

\* write a program to sort the elements of

\* an array with bubble sort.

\*/

**public** **class** Solution {

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

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

*bubbleSort*(arr);

}

**private** **static** **void** bubbleSort(**int**[] arr) {

// **TODO** Auto-generated method stub

**int** n = arr.length;

System.***out***.println("Array elements before sorting");

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

System.***out***.print(arr[i] + " ");

}

System.***out***.println(" \n ==============================");

**for**(**int** i = 0 ; i < n - 1; i++ ) {

**for**( **int** j = 0; j < n - 1 -i; j++ ) {

**if** ( arr[j] > arr[j+1] ) {

**int** temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

}

System.***out***.println("Array elements after sorting");

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

System.***out***.print(arr[i] + " ");

}

}

}

Two Dimentional array (2-D Array):

datatype[][] arrayname = new datatype[rows][cols];

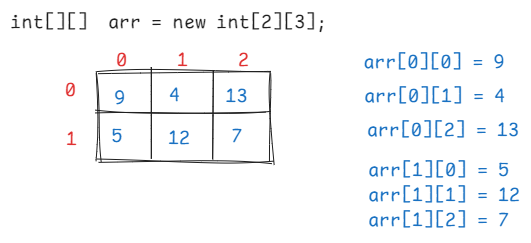
(or)

datatype arrayname[][] = new datatype[rows][cols];

(or)

datatype[] arrayname[] = new datatype[rows][cols];

for example:



* A static 2-D array can be created like below.

int[][] arr = { {5,6,1}, {2,8,0}, {4,5,9} };

/\*

\* write a program to read and print the elements

\* of two dimentional array(matrix)

\*/

**import** java.util.Scanner;

**public** **class** MainClass {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter the no of rows");

**int** rows = scan.nextInt();

System.***out***.println("enter the no of cols");

**int** cols = scan.nextInt();

//create 2-d array

**int**[][] arr = **new** **int**[rows][cols];

//read the elements

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

**for**( **int** j = 0; j < cols; j++ ) {

System.***out***.println("enter value for " + i + ", " + j);

arr[i][j] = scan.nextInt();

}

}

System.***out***.println("The array elements : ");

//print the elements

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

**for**( **int** j = 0; j < cols; j++ ) {

System.***out***.print(arr[i][j] + " ");

}

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

}

}

}

/\*

\* write a program to find the sum of even elements and odd elements

\* of the 2d array separately.

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

// **TODO** Auto-generated method stub

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter the no of rows");

**int** rows = scan.nextInt();

System.***out***.println("enter the no of cols");

**int** cols = scan.nextInt();

//create 2-d array

**int**[][] arr = **new** **int**[rows][cols];

//read the elements

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

**for**( **int** j = 0; j < cols; j++ ) {

System.***out***.println("enter value for " + i + ", " + j);

arr[i][j] = scan.nextInt();

}

}

*findEvenOddSum*(arr, rows, cols);

}

**private** **static** **void** findEvenOddSum(**int**[][] arr, **int** rows, **int** cols) {

// **TODO** Auto-generated method stub

**int** evenSum = 0;

**int** oddSum = 0;

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

**for**( **int** j = 0; j < cols; j++ ) {

**if** ( arr[i][j] % 2 == 0 )

evenSum += arr[i][j];

**else**

oddSum += arr[i][j];

}

}

System.***out***.println("Even sum : " + evenSum);

System.***out***.println("Odd sum : " + oddSum);

}

}

/\*

\* write a program to find the addition of

\* two 2-d arrays.

\*/

**public** **class** MainClass {

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

**int**[][] m1 = { {2, 1, 3}, {4, 7, 0} };

**int**[][] m2 = { {5, 3, 9}, {7, 6, 1} };

*addTwoMatrices*(m1, m2);

}

**private** **static** **void** addTwoMatrices(**int**[][] m1, **int**[][] m2) {

// **TODO** Auto-generated method stub

**if**(m1.length != m2.length || m1[0].length != m2[0].length) {

System.***out***.println("The matrices have different size, so we can't add");

**return**;

}

**int**[][] m3 = **new** **int**[m1.length][m1[0].length];

**for**(**int** i = 0; i < m1.length; i++) {

**for**(**int** j = 0; j < m1[0].length; j++) {

m3[i][j] = m1[i][j] + m2[i][j];

}

}

System.***out***.println("The first matrix");

**for**(**int** i = 0; i < m1.length; i++) {

**for**(**int** j = 0; j < m1[0].length; j++) {

System.***out***.print(m1[i][j]+ " ");

}

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

}

System.***out***.println("The second matrix");

**for**(**int** i = 0; i < m2.length; i++) {

**for**(**int** j = 0; j < m2[0].length; j++) {

System.***out***.print(m2[i][j]+ " ");

}

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

}

System.***out***.println("The addition matrix");

**for**(**int** i = 0; i < m3.length; i++) {

**for**(**int** j = 0; j < m3[0].length; j++) {

System.***out***.print(m3[i][j]+ " ");

}

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

}

}

}

**/\***

**\* find sum of left and right diagonal elements**

**\*/**

**public** **class** Solution {

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

// **TODO** Auto-generated method stub

**int**[][] arr = { {1, 5, 3}, {5, 0, 8}, {6, 2, 7} };

*findDiagonalSum*(arr);

}

**private** **static** **void** findDiagonalSum(**int**[][] arr) {

// **TODO** Auto-generated method stub

**int** leftSum = 0, rightSum = 0;

**if**( arr.length != arr[0].length) {

System.***out***.println("The matrix is not a square matrix. So, we can't find diagonals");

**return**;

}

**for**(**int** i=0; i < arr.length; i++) {

**for**(**int** j=0; j < arr[0].length; j++) {

**if**( i == j ) {

leftSum += arr[i][j];

}

**if**( i+j == arr.length - 1 ) {

rightSum += arr[i][j];

}

}

}

System.***out***.println("The matrix");

**for**(**int** i=0; i < arr.length; i++) {

**for**(**int** j=0; j<arr[0].length; j++) {

System.***out***.print(arr[i][j]+ " ");

}

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

}

System.***out***.println("left diagonal sum : " + leftSum);

System.***out***.println("right diagonal sum : " + rightSum);

}

}

/\*

\* write a Java program to find the multitplication

\* of two matrices.

\*

\*/

**public** **class** Solution {

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

**int**[][] a = { {1, 8}, {2, 6}, {3, 4} };

**int**[][] b = { {1, 6, 0}, {9, 0, 7} };

*multiplyMatrices*(a, b);

}

**private** **static** **void** multiplyMatrices(**int**[][] a, **int**[][] b) {

// **TODO** Auto-generated method stub

**if**( a[0].length != b.length ) {

System.***out***.println("you can't multiply the matrices");

**return**;

}

//create third matrix for result

**int**[][] c = **new** **int**[a.length][b[0].length];

**for** ( **int** i = 0; i < a.length; i++ ) {

**for** ( **int** j = 0; j < b[0].length; j++ ) {

c[i][j] = 0;

**for**( **int** k = 0; k < b.length; k++ ) {

c[i][j] += a[i][k] \* b[k][j];

}

}

}

System.***out***.println("The first matrix");

**for**( **int** i = 0; i < a.length; i++ ) {

**for**(**int** j = 0; j < a[0].length; j++ ) {

System.***out***.print(a[i][j]+" ");

}

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

}

System.***out***.println("The second matrix");

**for**( **int** i = 0; i < b.length; i++ ) {

**for**(**int** j = 0; j < b[0].length; j++ ) {

System.***out***.print(b[i][j]+" ");

}

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

}

System.***out***.println("The Multiplication matrix");

**for**( **int** i = 0; i < c.length; i++ ) {

**for**(**int** j = 0; j < c[0].length; j++ ) {

System.***out***.print(c[i][j]+" ");

}

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

}

}

}

String handling

* A string is a sequence of characters enclosed within double quotes.
* A string can be created in 2 ways in Java.
* 1. string literal
* 2. string object
* String is a class in java.lang package. It is not a primitive datatype. So, if you create in a literal form or with new keyword, both are string objects only.

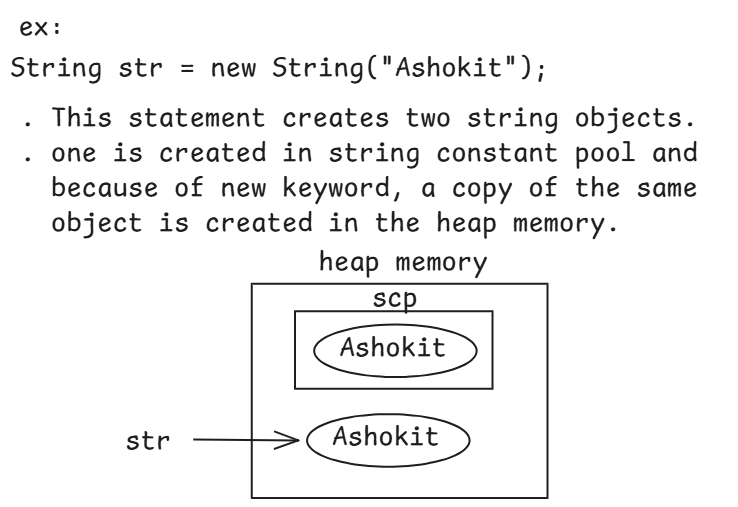
ex:

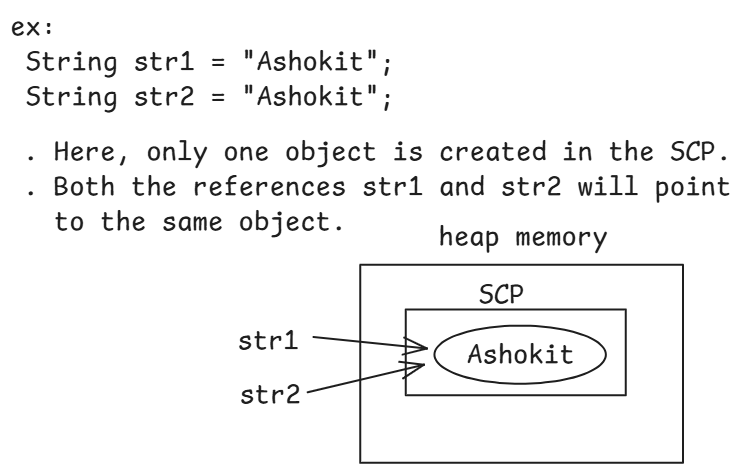
String str1 = “Hello”; //literal

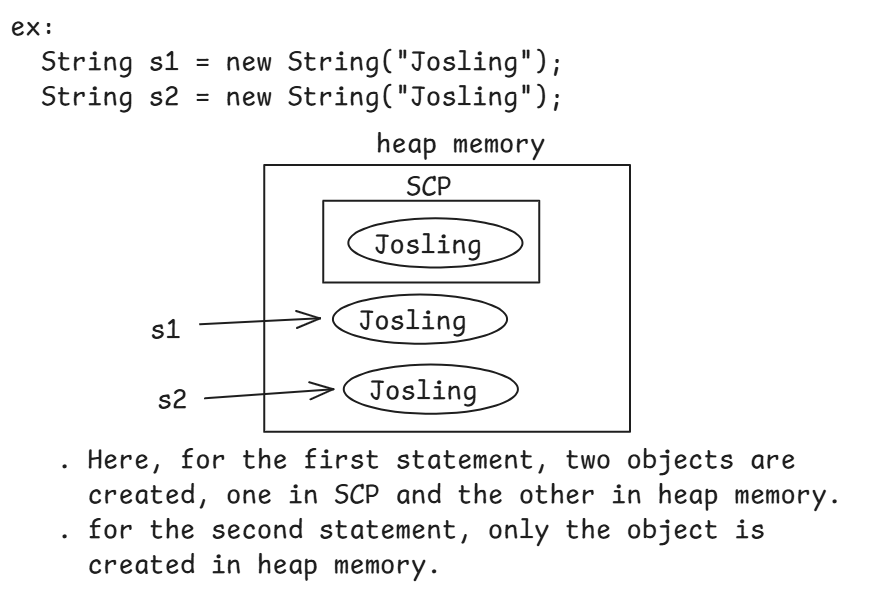
String str2 = new String(“Hello”); //object



* when you create in string object in literal form, the object is created in a special memory area in the heap, called scp.
* when you create a string object with new keyword, then the object is created directly in the heap memory.







finding the length of a string:

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

length (attribute) – finds the size of an array

length() (method) -- finds the size of a string

ex1: String str = new String(“Core Java”);

S.o.p( str.length ); //error

S.o.p( str.length() ); // 9

ex2: String str[] = { “John”, “Jim”, “Jackson” };

S.o.p( str.length ); //3

S.o.p( str[2].length ); //error

S.o.p( str[2].length() );//7

ex3:

int[] arr = { 3, 6, 1, 5};

S.o.p( arr.length() ); //error

S.o.p( arr.length ); //4

strings comparision:

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

. strings can be compared in 4 ways.

1. using == operator
2. using equals() method
3. using equalsIgnoreCase() method
4. using compareTo() method

. The == operator checks whether the two variables are pointing to the

same object or not. If yes, then it returns true. Otherwise, returns

false.

ex1:

String str1 = “Hello”;

String str2 = “Hello”;

S.o.p( str1 == str2 ); //true

ex2:

String str1 = “Java”;

String str2 = “Ashokit”;

S.o.p( str1 == str2 ); //false

ex3:

String str1 = “Java”;

String str2 = new String(“Java”);

S.o.p( str1 == str2 ); // false

. The equals() method compares the content of the two strings. If they

are same, returns true. Otherwise, returns false.

ex1:

String str1 = “Ashokit”;

String str2 = “ashokit”;

S.o.p( str1 == str2 ); //false

S.o.p( str1.equals(str2) ); //false

ex2:

String str1 = new String(“James”);

String str2 = new String(“James”);

S.o.p( str1.equals(str2) ); // true

. The equalsIgnoreCase() method, compares the content of the two strings

case-insensitively. If they are same, returns true. Otherwise returns

false.

ex:

String str1 = “Laptop”;

String str2 = “laptop”;

S.o.p( str1 == str2 ); // false

S.o.p( str1.equals(str2) ); //false

S.o.p( str1.equalsIgnoreCase(str2) ); //true

. The compareTo() method compares the two strings lexicographically. It

means, in the natural order. If they are same then returns 0.

If the first string is less than second then returns negative value.

Otherwise returns positive value.

( lexicographically is a comparision based on ASCII values ).

ex1:

String str1 = “Ashokit”;

Stirng str2 = “ashokit”;

S.o.p( str1.compareTo(str2) ); // negitive value(-32)

ex2:

String str1 = “Ashokit”;

String str2 = new String(“Ashokit”);

S.o.p( str1.compareTo(str2) ); // 0

ex3:

String str1 = "Ashokit";

String str2 = "Ashokti";

System.***out***.println(str1.compareTo(str2)); // -11

charAt(index) & indexOf(char):

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

.charAt(index) returns a character at the given index.

.if the given index is out of bounds then StringIndexOutOfBoundsException will be thrown.

For ex:

String str = “Ashokit”;

S.o.p( str.charAt(2) ); // h

S.o.p( str.charAt(7) ); //StringIndexOutOfBoundsException

. indexOf(char) returns the index of the first occurrence of the given character.

. if the given character doesn’t exist then it returns -1.

For ex:

String str = “cat sat”;

S.o.p ( str.indexOf(‘a’) ); //1

S.o.p ( str.indexOf(‘k’) ); // -1

S.o.p( str.indexOf(“sat”) ); // 4

lastIndexOf(char) : returns the index of last occurrence of the given character.

Ex: String str = “cat sat”;

S.o.p( str.lastIndexOf(‘a’) ); //5

indexOf(char, beginIndex) : returns the index of the character, starting from the begin index.

Ex: String str = “cat sat on mat”;

S.o.p( str.indexOf(‘a’) ); // 1

S.o.p( str.lastIndexOf(‘a’) ); // 12

S.o.p( str.indexOf(‘a’, 3) ); // 5

substring(beginIndex, endIndex) : It returns a portion of a string from beginIndex to endIndex-1.

For ex:

String str = “cat sat on mat”;

String newStr = str.substring(4, 14);

S.o.p(newStr); //sat on mat

substring(beginIndex) : returns a string from the beginIndex to end of the string.

Ex:

String str = “Ashokit Core Java”;

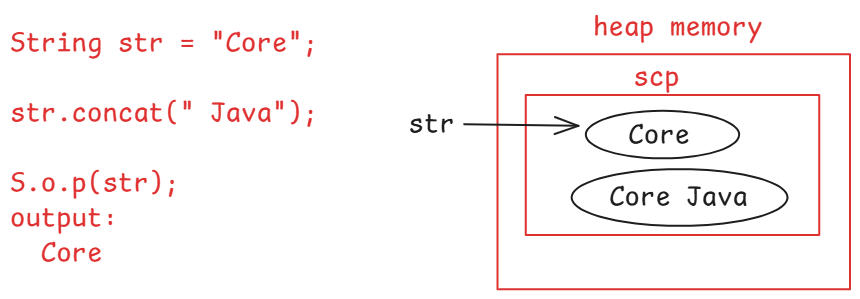
String newStr = str.substring( 8 );

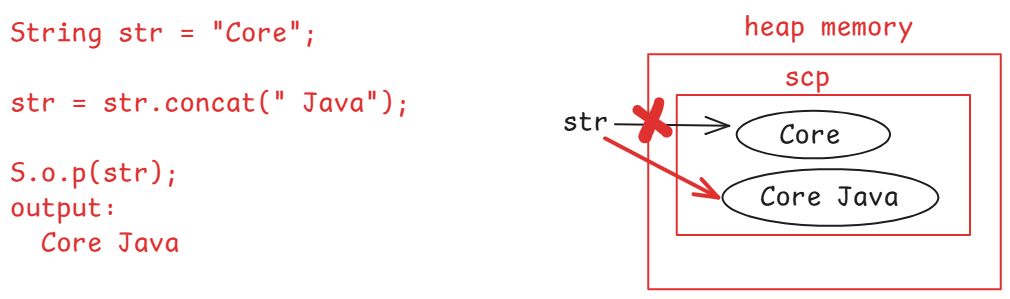
S.o.p(newStr); // Core Java

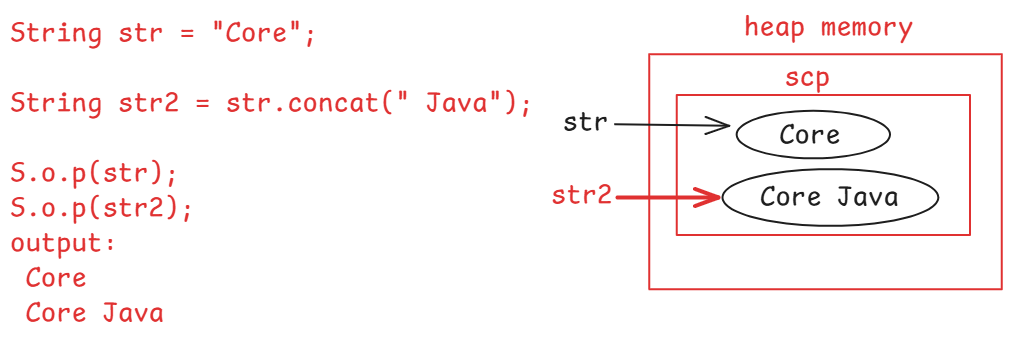
String immutable:

. A String object is an immutable object. It means, once the object is created, it does not allow to make changes to its data.

. So, if we make any changes to the data, a new string object is created to store the result.







toUpperCase() & toLowerCase() :

ex:

String str = “Ashokit”;

str.toUpperCase();

S.o.p(str); // Ashokit

EX:

String str = “Ashokit”;

S.o.p( str.toUpperCase() ); // ASHOKIT

S.o.p( str ); // Ashokit

ex:

String str = “ashokit”;

str.toLowerCase();

S.o.p( str ); // ashokit

ex:

String str = “ashokIT”;

S.o.p( str.toLowerCase() ); //ashokit

S.o.p( str ); // ashokIT

split() : It will split one string into multiple strings at the given character/delimeter and returns an array of strings.

For ex:

String url = “http://localhost:8080/index”;

String[] strings = url.split(“:”);

for ( int i = 0; i < strings.length; i++ ) {

S.o.p( strings[i] );

}

output:

http

//localhost

8080/index

for each loop:

. for each loop is used to iterate over the elements of an array or a Collection object.

syntax:

for ( datatype variable : array or collection )

{

statements;

}

. Here, the datatype should match with the type of the array or the type of the elements of the Collection object.

Ex:

for ( String str : strings ) {

S.o.p(str);

}

output:

http

//localhost

8080/index

ex2:

int[] arr = { 4, 6, 9, 1, 2 };

int sum = 0;

for ( int x : arr ) {

sum += x;

}

S.o.p(sum); // 22

join() method: It will combine multiple strings together into a

single string.

It is a static method, so we need to call this

method with String classname.

ex:

String strings[] = { “The”, “Sky”, “is”, “Blue” };

String joinedString = String.join(“ “, strings);

S.o.p(joinedString);

output:

The Sky is Blue

String joinedString2 = String.join(“-“, strings);

S.o.p(joinedString2);

output:

The-Sky-is-Blue

replace() method: It replaces all occurrences of the old string

with a new string.

ex1:

String str = “The sky is blue”;

String replacedString = str.replace(“blue”, “white”);

S.o.p(replacedString);

output:

The sky is white

ex2:

String str = “This is java. JAVA is powerful”;

String replacedStr = str.replace(“java”, “programming”);

S.o.p(replacesStr);

output:

This is programming. JAVA is powerful

replaceAll() method: It replaces all occurrences of the old string

which matches the given regular expression,

with new string.

ex1:

String str = "Hello John123, Good Morning!";

//replaces each digit with empty string

String replacedStr = str.replaceAll("[0-9]", "");

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

output: Hello John, Good Morning!

ex2:

String str = "Hello John123 @#!$, Good Morning!";

//replaces each digit and special character with empty string

String replacedStr = str.replaceAll("[^A-Za-z, ]", "");

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

output:

Hello John , Good Morning

ex3:

String str = "This is java. JAVA is powerful";

//replaces java(case-insensitive) with Programming

String replacedStr = str.replaceAll("(?i)java", "Programming");

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

output:

This is Programming. Programming is powerful

ex4:

String creditCard = "1234-5678-9876-5432";

String maskedCreditCard = creditCard.replaceAll("[^-](?=.{4})", "x");

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

output:

xxxx-xxxx-xxxx-5432

trim() method: It removes white spaces from a string at before start

of the first character or after the end of the last

character.

ex:

String str = " Th e sky is blue ";

String newStr = str.trim();

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

output:

Th e sky is blue

ex2:

String username = "John ";

**if**( username.equals("John") ) {

System.***out***.println("Hey! You are John");

}

**else** {

System.***out***.println("Hey! You are not John");

}

output: Hey! You are not John

ex3:

String username = "John ";

**if**( username.trim().equals("John") ) {

System.***out***.println("Hey! You are John");

}

**else** {

System.***out***.println("Hey! You are not John");

}

output: Hey! You are John

startsWith() & endsWith():

* startsWith() checks whether a string starts with a given characters or not. If yes, returns true. Otherwise returns false.
* endsWith() checks whether a string ends with a given characters or not. If yes, returns true. Otherwise, returns false.

ex1:

String str = “The cat sat on mat”;

S.o.p( str.startsWith(“cat”) ); // false

S.o.p( str.endsWith(“mat”) ) ; //true

ex2:

String email = “test@gmail.com”;

if( email.endsWith(“yahoo.com”) )

S.o.p(“You are a yahoo user”);

else

S.o.p(“You are not a yahoo user”);

isEmpty() & isBlank() :

* isEmpty() finds the length of the string. If it is zero, then returns true, otherwise returns false.
* isBlank() first trims the string, then finds the length of the string. If it is zero, then returns true, otherwise returns false.

ex1:

String user=" ";

System.***out***.println(user.isEmpty());

System.***out***.println(user.isBlank());

output: false

true;

ex2:

String user="";

System.***out***.println(user.isEmpty());

System.***out***.println(user.isBlank());

output: true

true

ex3:

String user="John";

System.***out***.println(user.isEmpty());

System.***out***.println(user.isBlank());

output: false

false

intern() method:

* This intern() method will move the cursor from the heap memory object to the object created in the SCP.

ex:

String user1="john";

String user2 = **new** String("john");

user2 = user2.intern();

System.***out***.println(user1 == user2);

output: true

format() method:

* This method returns a formatted string for the given string with the arguments.
* It is a static method, so we have to call this method with the String classname.

ex1:

String s = String.*format*("This guy is : %s and his age is : %d and his weight is : %.2f kg", "John", 35, 75.8);

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

output:

This guy is : John and his age is : 35 and his weight is : 75.80 kg

ex2:

**int** empno = 7788;

String ename = "Scott";

**double** sal = 5000.0;

String formattedString = String.*format*("Empno : %d %nEname : %s %nSal : %.3f", empno, ename, sal);

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

output:

Empno : 7788

Ename : Scott

Sal : 5000.000

valueOf() method:

* This method converts a value from other data types like int, float, boolean, char, etc.. into a string type.
* It is a static method, so we have call this method with String class name.

ex1:

**int** x = 50;

**int** y = 50;

System.***out***.println(String.*valueOf*(x) + String.*valueOf*(y));

output: 5050

ex2:

S.o.p(String.valueOf(1)+String.valueOf(true)+String.valueOf(0)+ String.valueOf(false));

output: 1true0false

=========================================================================

/\*

\* write a program to count the vowels and consonents separately,

\* in a given string.

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter a string");

String str = scan.nextLine();

*countVowelsAndConsonents*(str);

}

**private** **static** **void** countVowelsAndConsonents(String str) {

// **TODO** Auto-generated method stub

**int** vCount = 0, cCount = 0;

//convert the string into lowercase

str = str.toLowerCase();

//convert the string into character array

**char** ch[] = str.toCharArray();

**for** ( **int** i = 0; i < ch.length; i++ ) {

**char** c = ch[i];

//check is this character is a letter or not

**if**( Character.*isLetter*(c) ) {

**switch**( c ) {

**case** 'a':

**case** 'e':

**case** 'i':

**case** 'o':

**case** 'u': vCount++;

**break**;

**default**: cCount++;

}

}

}

System.***out***.println("vowels count : " + vCount);

System.***out***.println("consonents count : " + cCount);

}

}

/\*

\* write a program to find the length of a string

\* without using length() method or length attribute

\*/

**import** java.util.Scanner;

**public** **class** Main {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter a string");

String input = scan.nextLine();

//convert the string into character array

**char** ch[] = input.toCharArray();

**int** count = 0;

**for**( **char** c : ch ) {

count++;

}

System.***out***.println("length of the string is : " + count);

}

}

/\*

\* write a program to swap the two strings without

\* using a third variable

\*/

**public** **class** MainClass {

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

String s1 = "Hello";

String s2 = "Ashokit";

System.***out***.println("The strings before swapping");

System.***out***.println("string1 : " + s1);

System.***out***.println("string2 : " + s2 );

s1 = s1 + s2;

s2 = s1.substring(0, s1.length() - s2.length());

s1 = s1.substring(s2.length());

System.***out***.println("The strings after swapping");

System.***out***.println("string1 : " + s1);

System.***out***.println("string2 : " + s2 );

}

}

/\*

\* write a program to check whether the given

\* two strings are anagrams or not.

\* Anagrams: If one string in a permutation of another,

\* then they are anagram strings.

\* It means, the two strings must be formed with

\* the same set of characters, but the order could

\* be different.

\* examples:

\* 1. s1 = "Listen", s2 ="Silent"

\* 2. s1= "The eyes", s2 = "They see"

\* 3. s1= "Debit card" s2= "Bad credit"

\*/

**import** java.util.Arrays;

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter first string");

String str1 = scan.nextLine();

System.***out***.println("enter second string");

String str2 = scan.nextLine();

**boolean** flag = *checkAnagrams*(str1, str2);

**if**( flag == **true** )

System.***out***.println("The strings are anagrams");

**else**

System.***out***.println("The strings are not anagrams");

}

**private** **static** **boolean** checkAnagrams(String str1, String str2) {

**boolean** flag = **true**;

**if**( str1.trim().length() != str2.trim().length() ) {

flag = **false**;

**return** flag;

}

**char** ch1[] = str1.trim().toLowerCase().toCharArray();

**char** ch2[] = str2.trim().toLowerCase().toCharArray();

Arrays.*sort*(ch1);

Arrays.*sort*(ch2);

**for**( **int** i = 0; i < ch1.length; i++ ) {

**if** ( ch1[i] != ch2[i] ) {

flag = **false**;

**break**;

}

}

**return** flag;

}

}

/\*

\* write a program to check whether a given string

\* is a palindrome or not

\* example:

\* str = "liril", reverse = "liril" -- palindrome

\* str = "madam", reverse = "madam" -- palindrome

\* str = "ashokit", reverse = "tikohsa" -- not a palindrome

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter a string");

String str = scan.nextLine();

**boolean** flag = *isPalindrome*(str);

**if** ( flag == **true** )

System.***out***.println(str + " : is palindrome");

**else**

System.***out***.println(str + " : is not a palindrome");

}

**private** **static** **boolean** isPalindrome(String str) {

**boolean** flag = **true**;

str = str.toLowerCase();

**for** ( **int** i = 0, j = str.length() - 1; i <= j; i++, j--) {

**if** ( str.charAt(i) != str.charAt(j) ) {

flag = **false**;

**break**;

}

}

**return** flag;

}

}

/\*

\* write a program to find the frequency of each

\* character in a given string.

\* For example:

\* str = "Missisipi"

\* output:

\* M - 1

\* i - 4

\* s - 3

\* p - 1

\*/

**import** java.util.Arrays;

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter a string");

String str = scan.nextLine();

*findFrequency*(str);

}

**private** **static** **void** findFrequency(String str) {

//convert the string into lowercase

str = str.trim().toLowerCase();

//convert the string into character array

**char**[] ch = str.toCharArray();

//sort the character array

Arrays.*sort*(ch);

**for**( **int** i = 0; i < ch.length; i++ ) {

**int** count = 0;

**for** ( **int** j = i; j < ch.length; j++ ) {

**if**( ch[i] == ch[j] ) {

count++;

i = j;

}

**else**

**break**;

}

System.***out***.println( ch[i] + " ------- " + count);

}

}

}

StringBuffer class:

* StringBuffer is a class provided in java.lang package.
* StringBuffer is not a child class of String class, they are two different classes, but they works on string values.
* When a StringBuffer object is created, the object is created in the heap memory, but not in the String Constant Pool(SCP).
* A StringBuffer object can be created only with new keyword.
* A StringBuffer object is a mutable and a thread safe object.
* Mutable object means, the object will allow to make the changes to its content, after the object is created.
* When a StringBuffer object is created, additionally 16 characters space is allocated, to allow the changes to the content.

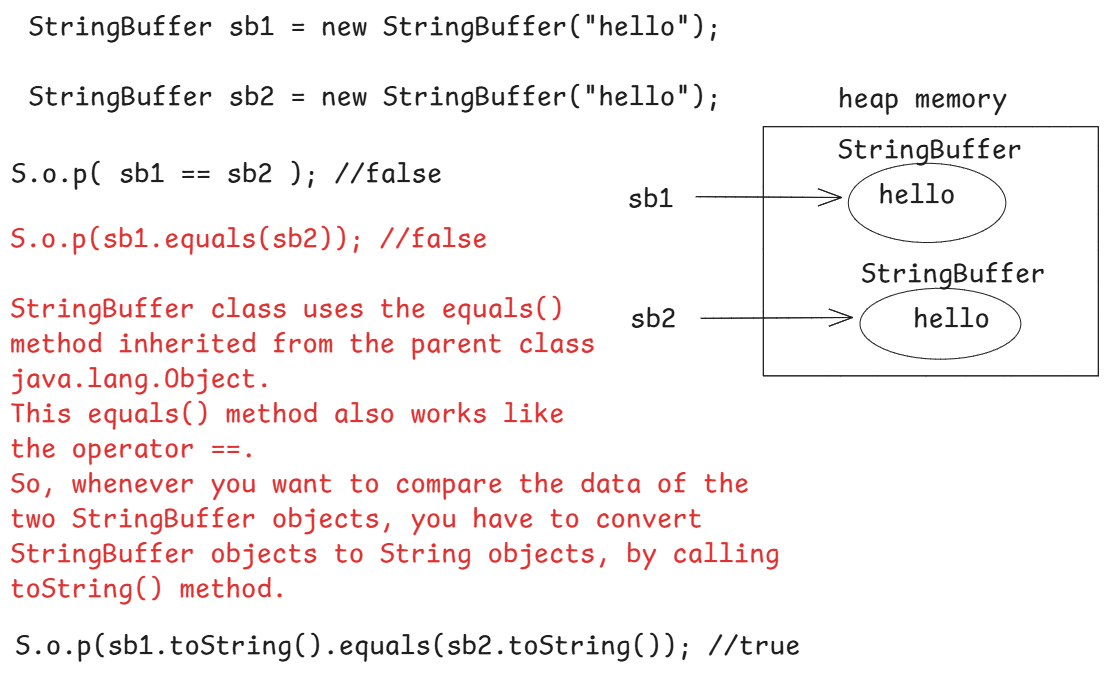
ex:

StringBuffer sb = “hello”; //error

StringBuffer sb = new StringBuffer(“hello”); //correct

S.o.p(sb.length()); // 5

S.o.p(sb.capacity()); //21 ( 5 + 16 )



/\*

\* example code on StringBuffer with insert,append, delete

\* and reverse methods

\*/

**public** **class** Main {

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

StringBuffer sb1 = **new** StringBuffer("The is blue");

System.***out***.println("The Initial StringBuffer value : " + sb1);

System.***out***.println("The StringBuffer value after inserting 'sky' ");

sb1.insert(4, "sky ");

System.***out***.println(sb1);

System.***out***.println("The StringBuffer value after appending 'at morning' ");

sb1.append(" at morning");

System.***out***.println(sb1);

System.***out***.println("The StringBuffer value after deleting 'at ' ");

sb1.delete(16, 19);

System.***out***.println(sb1);

System.***out***.println("The StringBuffer value after reverse ");

sb1.reverse();

System.***out***.println(sb1);

}

}

output:

The Initial StringBuffer value : The is blue

The StringBuffer value after inserting 'sky'

The sky is blue

The StringBuffer value after appending 'at morning'

The sky is blue at morning

The StringBuffer value after deleting 'at '

The sky is blue morning

The StringBuffer value after reverse

gninrom eulb si yks ehT

StringBuilder class:

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

* StringBuilder is a class provided in the java.lang package.
* StringBuilder object is a mutable and not a thread-safe object.
* StringBuilder object is created in the heap memory, but not SCP.
* StringBuilder is recommended to use in a single thread applications.
* StringBuffer and StringBuilder classes have the same methods. But StringBuffer methods are synchronized methods and StringBuilder methods are not synchronized methods.
* A StringBuilder object can be created only with new keyword.

ex:

StringBuilder builder = “hello”; //error

StringBuilder builder = new StringBuilder(“hello”); //correct

Q) what is the difference between String and StringBuffer?

A) String is a immutable and thread-safe object, where as StringBuffer is mutable and thread-safe object.

Q) what is the difference between StringBuffer and StringBuilder?

A) StringBuffer is a mutable and thread-safe object, where as StringBuilder is mutable and not a thread-safe object.

Q) What is the difference between String and StringBuilder?

A) String is immutable and thread-safe object, where as StringBuilder is mutable and not a thread-safe object.

For example, if you want to check, a string is palindrome or not with using StringBuffer/StringBuilder, then the code looks like below.

String str = "malayalam";

StringBuffer sb = **new** StringBuffer(str);

StringBuffer sb2 = sb.reverse();

**if**(sb2.toString().equals(str)) {

System.***out***.println("The given string is palindrome");

}

**else** {

System.***out***.println("The given string is not a palindrome");

}

/\*

\* write a program to remove the duplicate characters

\* from a given string

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter a string");

String str = scan.nextLine();

String newStr = *removeDuplicates*(str);

System.***out***.println("Original string : " + str);

System.***out***.println("After removing duplicates : " + newStr);

}

**private** **static** String removeDuplicates(String str) {

//create a StringBuilder to append the characters

StringBuilder builder = **new** StringBuilder();

//create a boolean array, assuming ASCII character set

//Indexes are from 0 to 255, and the default value

//stored at each index is false.

**boolean**[] seen = **new** **boolean**[256];

//convert the string into character array.

**char**[] ch = str.toCharArray();

//Iterate the loop for each character in the array.

**for** ( **char** c : ch ) {

**if** ( seen[c] == **false** ) {

builder.append(c);

seen[c] = **true**;

}

}

**return** builder.toString();

}

}

/\*

\* write a program for string compression

\* example:

\* input = "aaabbcad";

\* ouput = "a3b2c1a1d1"

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter a string");

String str = scan.nextLine();

String newStr = *compress*(str);

System.***out***.println("Original string : " + str);

System.***out***.println("Compressed string : " + newStr);

scan.close();

}

**private** **static** String compress(String str) {

//create a StringBuilder object

StringBuilder builder = **new** StringBuilder();

**int** count = 1;

//start the for loop

**for** ( **int** i = 0; i < str.length(); i++ ) {

**if** ( i+1 < str.length() && str.charAt(i) == str.charAt(i+1) ) {

count++;

}

**else** {

builder.append(str.charAt(i)).append(count);

count = 1; //reset the count

}

}

**return** builder.toString();

}

}

/\*

\* write a program to convert a given string into

\* title case.

\* For example,

\* str = "The sky is blue"

\* output:

\* The Sky Is Blue

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter a string");

String str = scan.nextLine();

String newStr = *convertToTitleCase*(str);

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

}

**private** **static** String convertToTitleCase(String str) {

//convert the given string into lowercase

str = str.toLowerCase();

//split the string to get string array at whitespace

String[] words = str.split(" ");

//create a StringBuilder object

StringBuilder builder = **new** StringBuilder();

//iterate the string array with for each loop

**for** ( String word : words ) {

**if**( word.length() > 0 ) {

builder.append(String.*valueOf*(word.charAt(0)).toUpperCase()).append(word.substring(1)).append(" ");

}

}

**return** builder.toString();

}

}

/\*

\* write a program to reverse each word in given string

\* For example,

\* str = "the cat sat on mat"

\* output:

\* eht tac tas no tam

\*/

**import** java.util.Scanner;

**public** **class** Solution {

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

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter a string");

String str = scan.nextLine();

String newStr = *reverseEachWord*(str);

System.***out***.println("Original string : " + str);

System.***out***.println("After reversing each word : " + newStr);

}

**private** **static** String reverseEachWord(String str) {

//create a StringBuilder object

StringBuilder builder = **new** StringBuilder();

//split the string into string array

String[] words = str.split(" ");

//for each loop

**for** ( String word : words ) {

**if** ( word.length() > 0 ) {

**for**(**int** i = word.length() - 1; i >= 0; i-- ) {

builder.append(word.charAt(i));

}

builder.append(" ");

}

}

**return** builder.toString();

}

}

OOPS

(Object Oriented Programming System)

* To develop a software application, we need to choose a programming model.
* The available programming models are,

1. Procedure Oriented Programming model
2. Object Oriented Programming model

* In Procedure Oriented Programming(POP) model, the software is developed by creating functions.
* If application need more functionalities then more functions were developed.
* The data is maintained at main function and passed to the other functions. In this way, it is difficult to find out, which fucntion is using which data and which function is modifying which data.
* So, for developing software applications for nowadays requirements, POP model doesn’t work.
* The OOP model was introduced to develop large scale applications, by combining the data and the related functionality together at one place with the help of classes.
* OOPS has defined 4 principles in the software development.
* 1. abstraction
* 2. encapsulation
* 3. inheritance
* 4. polymorphism

abstraction:

. abstraction is a process of providing the essential details/data by hiding un-essential details, based on the user of a system.

. abstraction can be implemented in a Java application using abstract classes and interfaces.

. For example, When you receice a phone call, as a user of mobile, the number and the location from where the call is connected will be shown, but the internals like towers, routers and their connectivity is not shown. This is called abstraction.

encapsulation:

. encapsulation is a process of combining the data(attributes) and the related functionality(behaviour) together into a single unit.

. encapsulation can be implemented in Java, by creating classes with private variables and public setter and getter methods.

Inheritance:

* Inheritance is a process of creating new classes by deriving the properites and the behaviors from the existing classes.
* The new class created is called as child class or sub class or derived class.
* The existing class from which the new class is created, is called as parent class or super class or base class.
* For example, You have an existing class Vehicle and you want to create a new class Car. In this case, you can create Car class by deriving the properties and behaviours from the existing Vehicle class. This is called inheritance
* With inheritance, the advantages are,

1. code reusability
2. reduces code redundency
3. simplifies maintenance
4. improves productivity

polymorphism:

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

* Polymorphsim word is a combination of 2 Greek words called poly(many) and morphos(forms).
* Polymorphism allows an application to define one task in multiple forms.
* Polymorphism allows an object to perform a task in many ways.
* For example, a Bank object can open a new account by taking adhaar card or voterid or pancard as input. This is polymorphism.
* Polymorphism can be implemented in the software, by using the mechanisms called method overloading and method overriding.

classes & objects:

* In OOP model, we have to consider everything in the real world as an object.
* For example, employees of an IT company are objects or customers of a bank are objects.
* The objects of different groups will have different properties and different behaviors.
* So, first we should create a template to define the properties and behaviors for a group of objects.
* This template is called a class in OOP.
* So, you can define a class as, “A class is a template or blue print with the required properites and behaviors for a group of objects”.
* properties means fields or variables or data members and behaviors means methods.
* So, a class is a collection of variables and methods.
* class is a keyword derived from an english word, “classification”.

syntax of creating a class:

<access modifier> class <classname>

{

variables;

methods;

}

For example:

class Authentication

{

//fields

String username;

String password;

//methods

boolean authenticate()

{

//logic

}

boolean authorize()

{

//logic

}

}

* an object is an instance of a class.
* For a class, we can create multiple objects.

syntax:

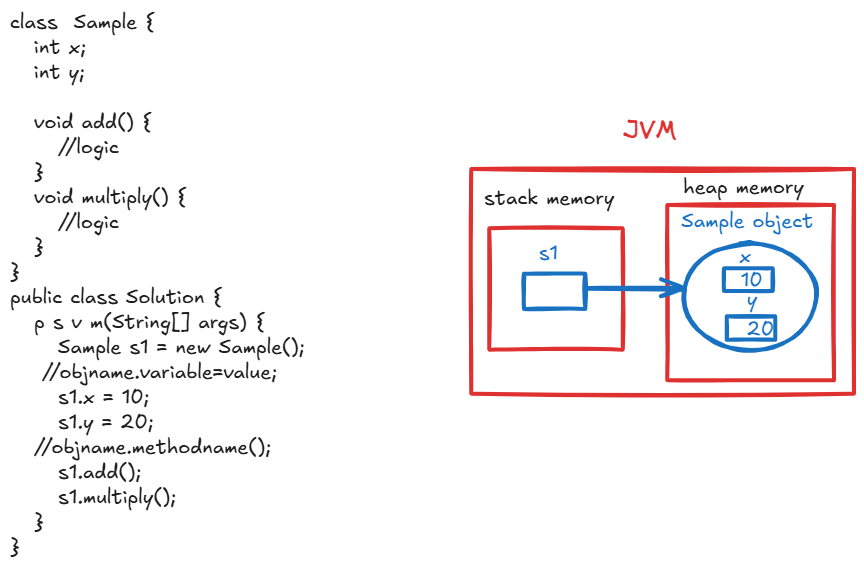
classname objectname = new classname();

for example,

Authentication auth1 = new Authentication();

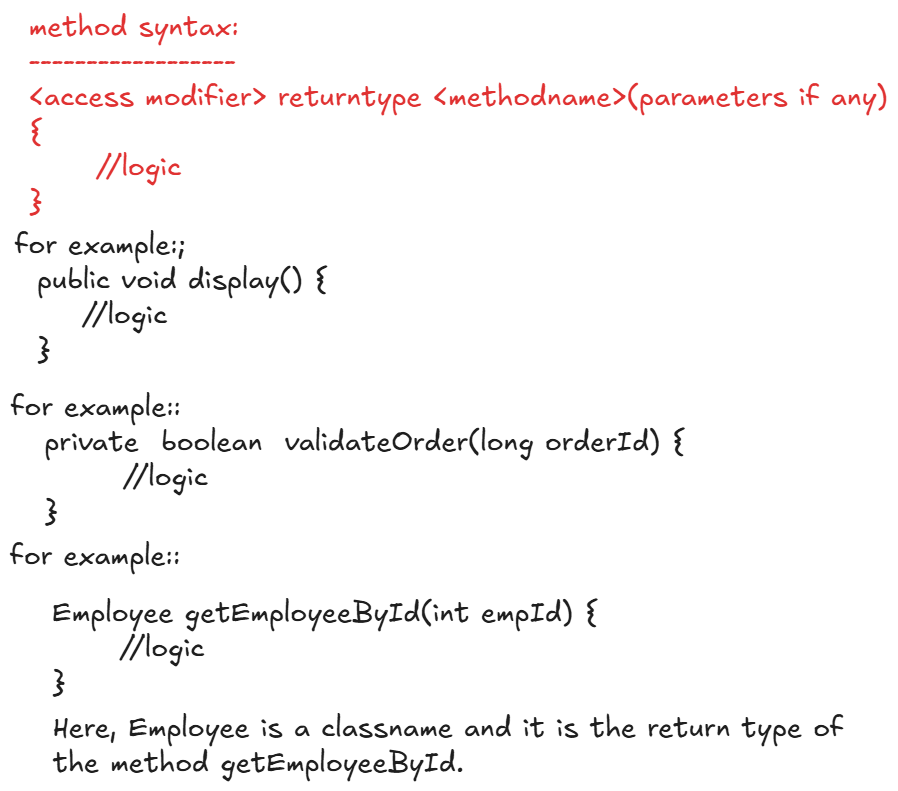
Authentication auth2 = new Authentication();

* When you create an object for a class, the memory for an object is created in the heap memory of JVM, and the objectname, also called as reference variable is created in the stack memory.



\* When you create an object, the objectname will be stored in stack memory and the object is created in heap memory.

\* When an object is created, the memory is allocated for the instance variables/fields inside the object.



/\*

\* create a class Product with fields id, name and price.

\* create two methods to update the price and to display

\* product details.

\*/

**class** Product {

//fields/instance variables

**int** id;

String name;

**double** price;

//methods

**void** updatePrice(**double** newPrice) {

price = newPrice;

}

**void** display() {

System.***out***.println("id : " + id);

System.***out***.println("name : " + name);

System.***out***.println("price : " + price);

}

}

**public** **class** Solution {

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

//creating object

Product p = **new** Product();

//assign the values

p.id = 101;

p.name = "Mobile";

p.price = 14599.0;

//update the price

p.updatePrice(16899.0);

//display the product

p.display();

}

}

Access modifiers:

* Access modifiers are used to control the visibility of a class or a variable or a method or a constructor.
* we have 4 access modifiers.

1. private
2. default/package-private
3. protected
4. public

private:

* private keyword can be used with variables, methods and constructors. But can not be used with classes.
* If you declare a variable/method/constructor as private then they are visible only within the same class in which they are created.

ex1:

private class Employee { //error

int empno;

String empName;

}

ex2:

class Employee {

private int empno;

public static void main(String[] args)

{

Employee e1 = new Employee();

e.empno = 7171; //valid

}

}

ex3:

class Employee {

private int empno;

}

class Main {

public static void main(String[] args)

{

Employee e1 = new Employee();

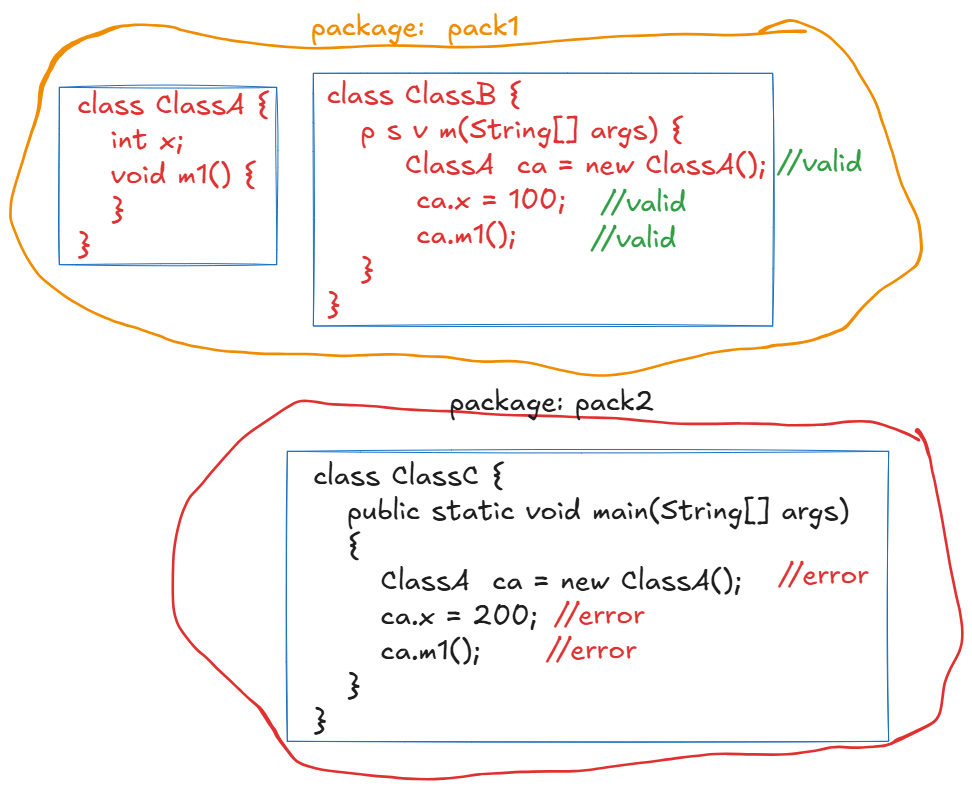
e1.empno = 7272; //error

}

}

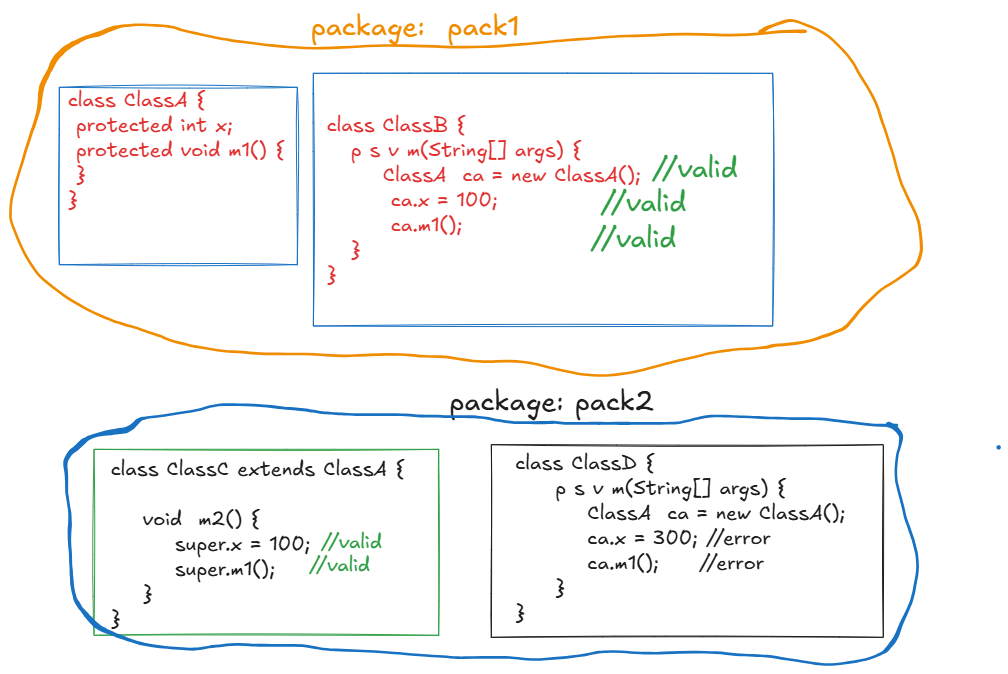
default:

* If we don’t specify any access modifier to a class/variable/method/constructor then it has default access modifier by default.
* The members with default access modifier are visible with in the same package. At outside of the current package, they are not accessible.



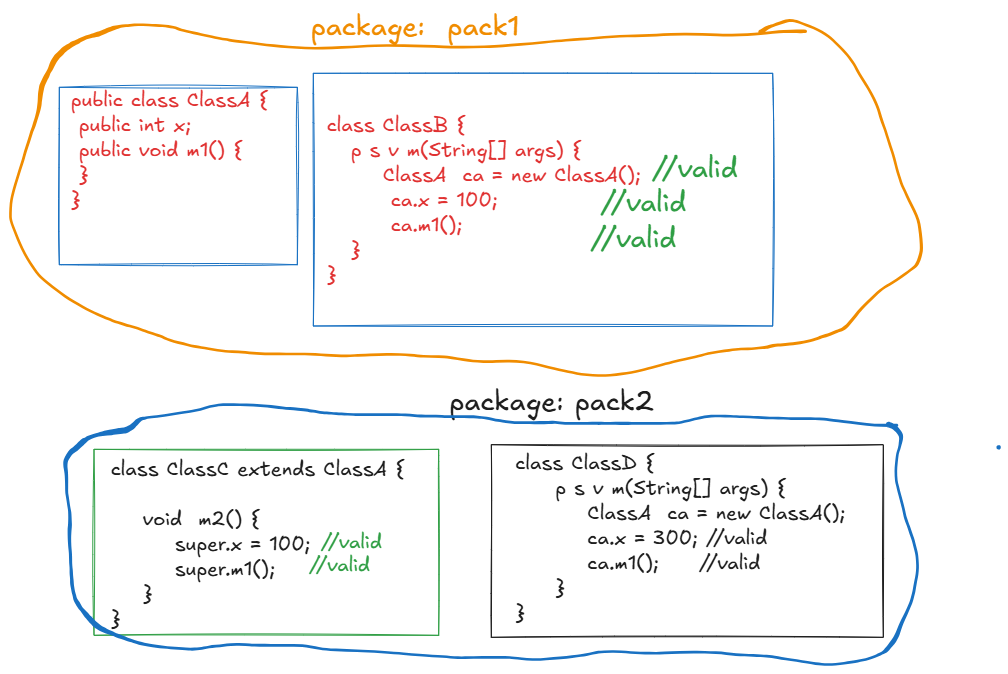
protected:

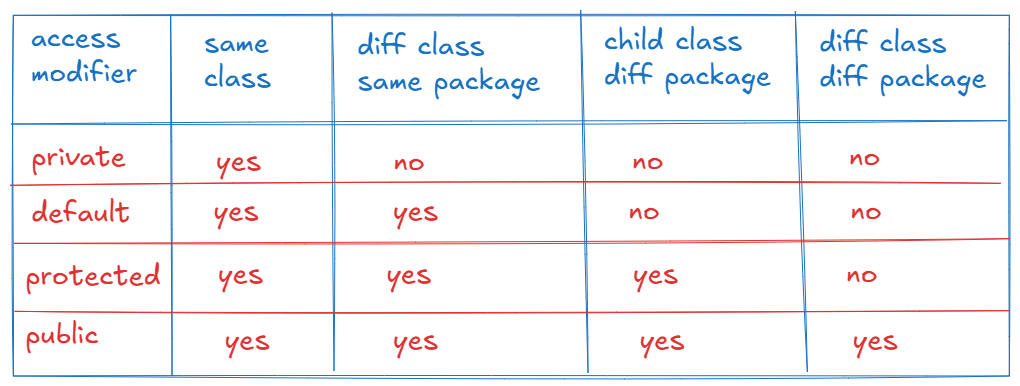
* protected keyword can be used with variables or methods or constructors. But can’t be used with classes.
* A protected member is visible with in the classes of the same package and also visible in the child classes of different package.



public:

* public keyword can be used with classes or variables or methods or constructors.
* public access modifier represents global visibility.
* The public members are visible to the classes of the same package or different packages also.
* private access modifier represents local visibility and public access modifier represents global visibility.

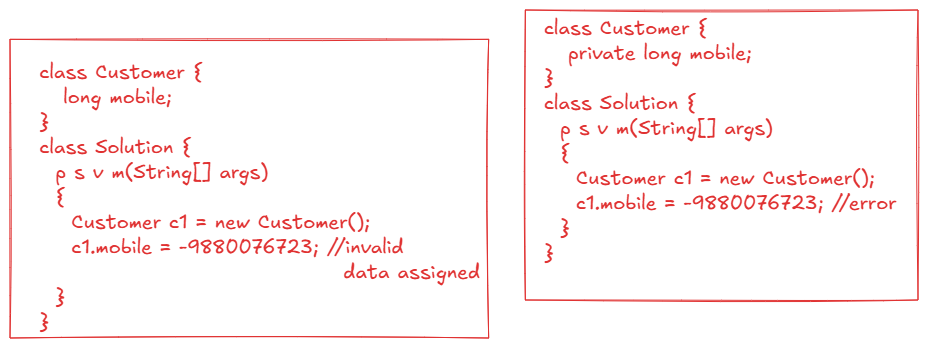




===================================================================

Defining setter and getter methods:

* In a class, we declare the instance variables as private, to prevent storing invalid data into the variables by creating an object at outside the class.



* If a variable is declared as private, then you have to define setter and getter methods in the class, to set the value or read the value of a variable.
* we have to follow a naming convention, to define the setter and getter methods.

ex:

if variable is,

private int xxx;

then you have to defined setter and getter methods like below.

public void setXxx(int xxx) {

this.xxx = xxx;

}

public int getXxx() {

return xxx;

}

//example code

**class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** salary;

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setSalary(**double** salary) {

**if**(salary < 0)

**throw** **new** RuntimeException();

**this**.salary = salary;

}

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** String getEname() {

**return** ename;

}

}

**public** **class** Solution {

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

Employee e1 = **new** Employee();

e1.setEmpno(7171);

e1.setSalary(10000.0);

e1.setEname("SCOTT");

System.***out***.println("empno = " + e1.getEmpno());

System.***out***.println("ename = " + e1.getEname());

System.***out***.println("salary = " + e1.getSalary());

}

}

constructor:

* A constructor is a special method in a class, which is used to initialize the object, whenever the object is created.
* Initializing an object means, assigning the values to the data members of an object.
* Suppose, if you don’t create a constructor in a class, then Java compiler creates a default constructor, and this default constructor will initialize the data member with the default values based on their datatypes.

For example:

**class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** salary;

**private** **boolean** hasPassport;

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setSalary(**double** salary) {

**this**.salary = salary;

}

**public** **boolean** isHasPassport() {

**return** hasPassport;

}

**public** **void** setHasPassport(**boolean** hasPassport) {

**this**.hasPassport = hasPassport;

}

}

**public** **class** Solution {

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

Employee e1 = **new** Employee();

System.***out***.println("empno = " + e1.getEmpno());

System.***out***.println("ename = " + e1.getEname());

System.***out***.println("salary = " + e1.getSalary());

System.***out***.println("hasPassport = " +e1.isHasPassport());

}

}

output:

empno = 0

ename = null

salary = 0.0

hasPassport = false

* In the above example, we have Employee class, but we didn’t write any constructor. So, Java compiler has defined default constructor to initialize the data members with default values.

. Suppose, if you want to initialize an object with a meaningful state, whenever the object is created then, you have to manually define a constructor in a class.

. To define a constructor, we have to follow the below rules.

1. a constructor can have access modifier
2. a constructor does not have a return type
3. a constructor name must be same as classname.

For example:

**class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** salary;

**private** **boolean** hasPassport;

//constructor

**public** Employee() {

empno=7171;

ename="John";

salary = 5500;

hasPassport=**true**;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setSalary(**double** salary) {

**this**.salary = salary;

}

**public** **boolean** isHasPassport() {

**return** hasPassport;

}

**public** **void** setHasPassport(**boolean** hasPassport) {

**this**.hasPassport = hasPassport;

}

}

**public** **class** Solution {

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

Employee e1 = **new** Employee();

System.***out***.println("empno = " + e1.getEmpno());

System.***out***.println("ename = " + e1.getEname());

System.***out***.println("salary = " + e1.getSalary());

System.***out***.println("hasPassport = " + e1.isHasPassport());

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

Employee e2 = **new** Employee();

System.***out***.println("empno = " + e2.getEmpno());

System.***out***.println("ename = " + e2.getEname());

System.***out***.println("salary = " + e2.getSalary());

System.***out***.println("hasPassport = " + e2.isHasPassport());

}

}

output:

empno = 7171

ename = John

salary = 5500.0

hasPassport = true

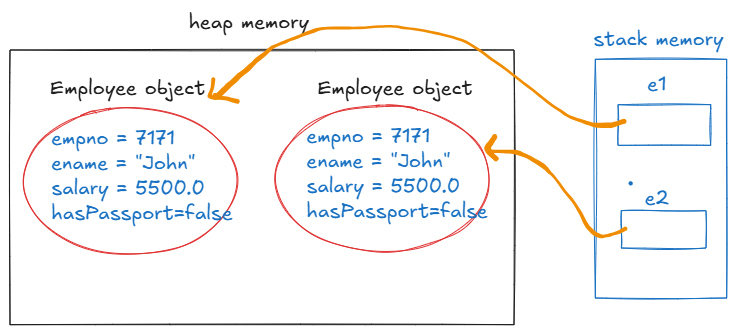
============================

empno = 7171

ename = John

salary = 5500.0

hasPassport = true



* In the above example, we have created a constructor in the Employee class, with out parameters.
* So, this constructor will initialize the objects with the same state.
* If you want to initialize each object with a different state, whenever the object is created, then you have to define a constructor with parameters.

For example,

**class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** salary;

**private** **boolean** hasPassport;

//constructor(with parameters)

**public** Employee(**int** empno, String ename, **double** salary, **boolean** hasPassport) {

**this**.empno = empno;

**this**.ename = ename;

**this**.salary = salary;

**this**.hasPassport = hasPassport;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setSalary(**double** salary) {

**this**.salary = salary;

}

**public** **boolean** isHasPassport() {

**return** hasPassport;

}

**public** **void** setHasPassport(**boolean** hasPassport) {

**this**.hasPassport = hasPassport;

}

}

**public** **class** Solution {

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

Employee e1 = **new** Employee(7676, "Allen", 8900.0, **false**);

System.***out***.println("empno = " + e1.getEmpno());

System.***out***.println("ename = " + e1.getEname());

System.***out***.println("salary = " + e1.getSalary());

System.***out***.println("hasPassport = " + e1.isHasPassport());

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

Employee e2 = **new** Employee(7989, "John", 9700.0, **true**);

System.***out***.println("empno = " + e2.getEmpno());

System.***out***.println("ename = " + e2.getEname());

System.***out***.println("salary = " + e2.getSalary());

System.***out***.println("hasPassport = " + e2.isHasPassport());

}

}

output:

empno = 7676

ename = Allen

salary = 8900.0

hasPassport = false

============================

empno = 7989

ename = John

salary = 9700.0

hasPassport = true

* In the above example, we have created a constructor with parameters, so the different objects of the class are initialized with different state.

Types of constructors:

1. constructor without parameters (or)

parameter-less constructor

1. constructor with parameters (or)

parameterized constructor

* If you don’t write any constructor in a class, then Java compiler adds a default constructor to the class and this default constructor also belongs to parameter-less constructor category.
* default constructor initialized the different objects of a class with default state.
* If you want to initialize the different objects of a class with same state, then you have to define parameter-less constructor.
* If you want to initialize the different objects of a class with different state, then you have to define parameterized constructor.

Q) what do you mean by state of an object?

A) At the current moment, whatever the data available in the data members of an object is called state of an object.

Q) can we define multiple constructors in a class?

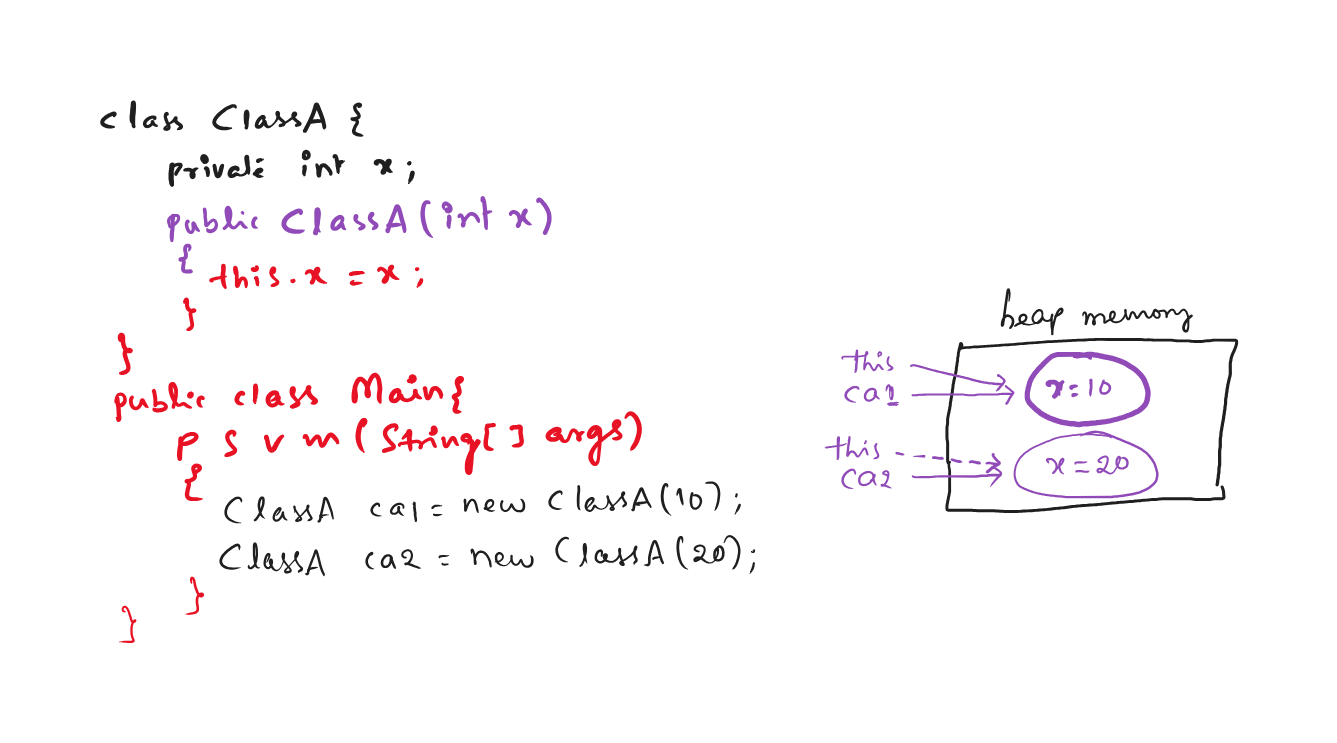
A) Yes.

this keyword:

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

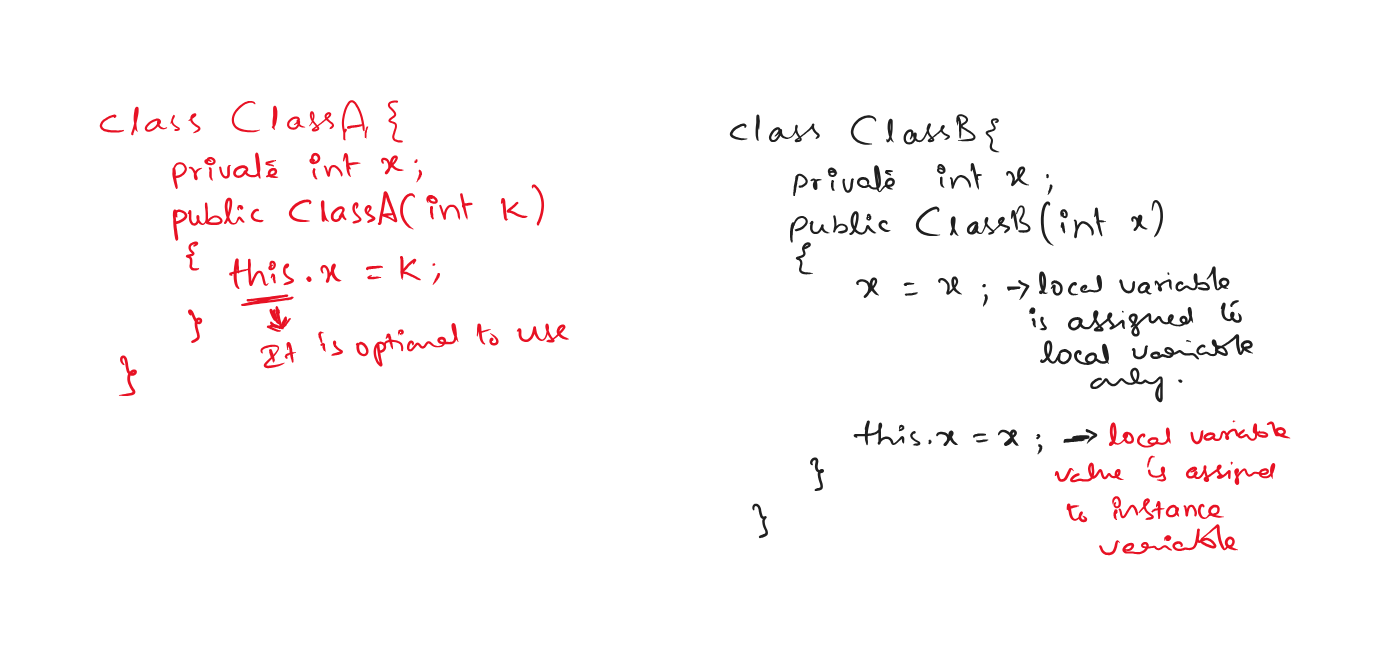
* this is a keyword in Java, which always refers the current object of a class.
* Suppose, if two objects are created for a class then this refers to the first object, if a method/constructor is called with the first object and this refers to the second object, if a method/constructor is called with the second object.

For ex:



* If a local variable name is matching with instance variable name then to distinguish between the two variables, we use this keyword.

For example:

this



this() call: It is used to call one constructor from another

constructor of the same class.

this() call must be a first statement in the

constructor.

this keyword can be used as any statement in the

constructor.

this keyword can be used in a constructor or in a

method also. But this() call can be used only in a

constructor.

this() call can be used to call the parameterized

constructor from the parameter-less constructor and

vice-versa.

example:

**class** ClassA {

**private** **int** x, y;

**public** ClassA() {

**this**(20, 30);

**this**.x = 1;

**this**.y = 2;

}

**public** ClassA(**int** x) {

**this**.x = x;

}

**public** ClassA(**int** x, **int** y) {

**this**(40);

**this**.x = x;

**this**.y = y;

}

**public** **void** display() {

System.***out***.println("x = " + **this**.x);

System.***out***.println("y = " + **this**.y);

}

}

**public** **class** Main {

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

ClassA ca = **new** ClassA();

ca.display();

}

}

output:

x = 1

y = 2

Inner classes

* If you define one class with in the boundary of the another class then it is called inner class.
* Suppose, if you have a class for which there is only one dependent class and if that class can’t work independently then you can define that class as a inner class, in the dependent class.
* For example, we have a class called Sim, and another class Mobile, which is a dependent class. The class Sim can’t work independently without Mobile class, so in this case we can define Sim as a inner class under Mobile class.

class Mobile { //outer class

//variables

//methods

class Sim { //inner class

//variables

//methods

}

}

* outer class private variables are visible to the inner class, but inner class private variables are not visible to the outer class.

example:

**class** OuterClass {

**private** **int** x;

**public** **void** outerClassMethod() {

System.***out***.println("x = " + x);

System.***out***.println("y = " + y); //error

}

**class** InnerClass {

**private** **int** y;

**public** **void** innerClassMethod() {

System.***out***.println("x = " + x); // correct

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

}

}

}

* For outer class, we can use only public or default access modifiers. But for inner class, we can use any access modifier.

for example:

private class OuterClass { //error

private class InnerClass { //correct

}

}

* Inner class object can be created in the outer class, or in a different class also.
* To create the inner class object in a different class, the syntax is,

outerclass.innerclass obj = outerclassobject.new innerclass();

example:

**class** OuterClass {

**private** **int** x;

//InnerClass ic = new InnerClass();

**public** **void** outerClassMethod() {

System.***out***.println("x = " + x);

}

**class** InnerClass {

**private** **int** y;

**public** **void** innerClassMethod() {

System.***out***.println("x = " + x); // correct

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

}

}

}

**public** **class** Main {

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

OuterClass oc = **new** OuterClass();

OuterClass.InnerClass ic = oc.**new** InnerClass();

oc.outerClassMethod();

ic.innerClassMethod();

}

}

Types of inner classes:

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

1. non-static inner class
2. static inner class
3. local inner class
4. anonymous inner class

* a local inner class means, you have to create a class inside a method.
* a local inner class can have default access modifier. But we can’t write private/protected/public.
* For a local inner class, object can be created only with in that method.

ex:

**class** OuterClass {

**private** **int** x;

**public** OuterClass(**int** x)

{

**this**.x = x;

}

**public** **void** outerClassMethod()

{

**class** LocalInnerClass

{

**private** **int** x;

**public** **void** m1() {

System.***out***.println("x = " + x);

}

}

LocalInnerClass lic = **new** LocalInnerClass();

lic.m1();

}

}

* anonymous inner classes are used to provide an implmentation for an interface, by without creating a class.

(Hint: anonymous means, name-less)

Q) what is anonymous object?

A) If you create an object, with out a name, then it is anonymous object.

ex:

new ClassA(); //anonymous object(name-less object)

* If you want to call a single method and just for once, of a class then you can create anonymous object.

example:

**class** OuterClass {

**private** **int** x;

**public** OuterClass(**int** x)

{

**this**.x = x;

}

**public** **void** outerClassMethod()

{

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

}

}

**public** **class** Main {

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

**new** OuterClass(10).outerClassMethod();

}

}

static keyword:

* static keyword represents that a member is belongs to class, not belongs to an instance of a class.
* we can create,

. static variables

. static blocks

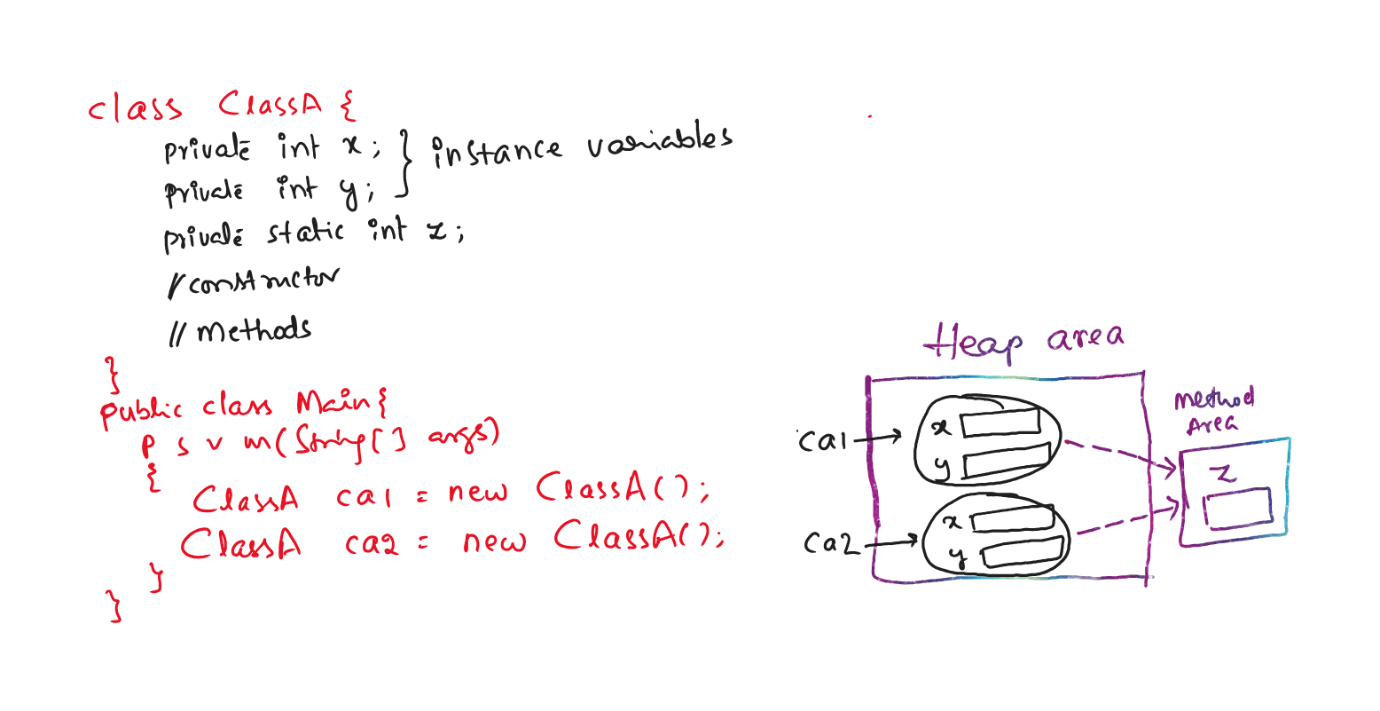
. static methods

. static inner classes

static variable:

* when you want to share a common data across all instances of a class then you have to create a variable as a static variable.
* In a class, you can define non-static and static variables.
* non-static variables are also called instance variables.
* For instance variables, the memory is allocated for each object of a class, inside that object.
* For static variables, the memory is allocated at outside of the objects in the heap memory, at method area.

For example:



For example, if you are creating a class Student, each student has a different id and name, but they have collegeName as common. So, we have to delcare the collegeName as a static variable, to share the common data across all the student instances.

ex:

class Student {

private int sid;

private String sname;

private static String collegeName;

//constructors

//methods

}

* static variables are class-level variables and instance variables are object-level variables.

Q) what is the difference between static and non-static/instance variable?

1) static variable is shared commonly across all instances of a class. But instance varible is separate for each instance of a class.

2) The memory is allocated for a static variable when the class is loaded into the JVM, by a class loader. But the memory is allocated for instance variable, when the object is created. It means, static variable is early loaded and instance variable is lazy loaded.

3) For a static variable, the memory is created in method area and for an instance varibale, the memory is allocated in heap area.

4) if a static variable’s value is modified then it effects on all instances of a class. But it an instance variable’s value is modifier then it effects on only that specific instance.

5) static variables can be called with classname and instance variables can be called with object name.

(Note: if you call a static variable with object name, internally JVM calls it with the class name only).

static block:

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

* It is used to initialize the static variables of a class.
* We can also initialize the static variables in a constructor. But for each instance, the static variable will be re-initialized. So, we create a static block.
* Sometimes, multiple lines of code should be executed to initialize a static variable. This code, we can define in a static block.
* A static block will be executed for only once, when the class is loaded into the JVM by the class loader.
* Based on requirement, we can define multiple static blocks also in a class. The static blocks are executed, in the same order that they have defined in a class.

for example:

**class** ClassA {

**int** x;

**static** **int** *y*;

**public** ClassA(**int** x) {

System.***out***.println("ClassA(int) :: constructor");

}

**static** {

System.***out***.println("static block :: block1");

}

**static** {

System.***out***.println("static block :: block2");

}

}

**public** **class** Main {

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

ClassA ca1 = **new** ClassA(10);

}

}

output:

static block :: block1

static block :: block2

ClassA(int) :: constructor

Q) what is the difference between constructor and a static block?

A) write your answer here

static method:

* static method belongs to a class, not belongs to any instance of a class.
* When you want to share a common logic across all instances of a class, then you have to create a static method.
* static methods can be directly with the classname, without creating an object.
* static methods are generally created to define helper methods or utility methods like for calculations, validations, etc.

//Example

/\*

\* we are creating User class, Validator class and Main class.

\* In the Validator class, we are creating static methods

\* to act like helper methods to check the email is valid

\* and to check the password is strong.

\*/

**class** User

{

**private** String email;

**private** String password;

**public** User(String email, String password) {

**this**.email = email;

**this**.password = password;

}

**public** String getEmail() {

**return** email;

}

**public** **void** setEmail(String email) {

**this**.email = email;

}

**public** String getPassword() {

**return** password;

}

**public** **void** setPassword(String password) {

**this**.password = password;

}

}

**class** Validator {

**public** **static** **boolean** isValidEmail(User user) {

**return** user.getEmail()!=**null** && user.getEmail().contains("@") && user.getEmail().contains(".");

}

**public** **static** **boolean** isStrongPassword(User user) {

**return** user.getPassword().length() >= 8 && user.getPassword().matches(".\*[A-Z].\*");

}

}

**public** **class** Main {

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

User user = **new** User("john@gmail.com", "teSt@123");

**if** ( Validator.*isValidEmail*(user) && Validator.*isStrongPassword*(user) ) {

System.***out***.println("User object is strong");

}

**else** {

System.***out***.println("User object is not strong");

}

}

}

Q) what is the difference between instance method and static method?

A) write your answer here.

Q) can we use this keyword in a static method?

A) No.

Q) can you call a static method with the object name?

A) Yes. But internally JVM calls the static method with the class of that object.

Q) what is the output of the below code.

**class** A {

**static** **void** m1() {

System.***out***.println("static method :: m1");

}

}

**public** **class** Main {

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

A a = **null**;

a.*m1*();

}

}

1. static method :: m1

Q) can we create a private static method?

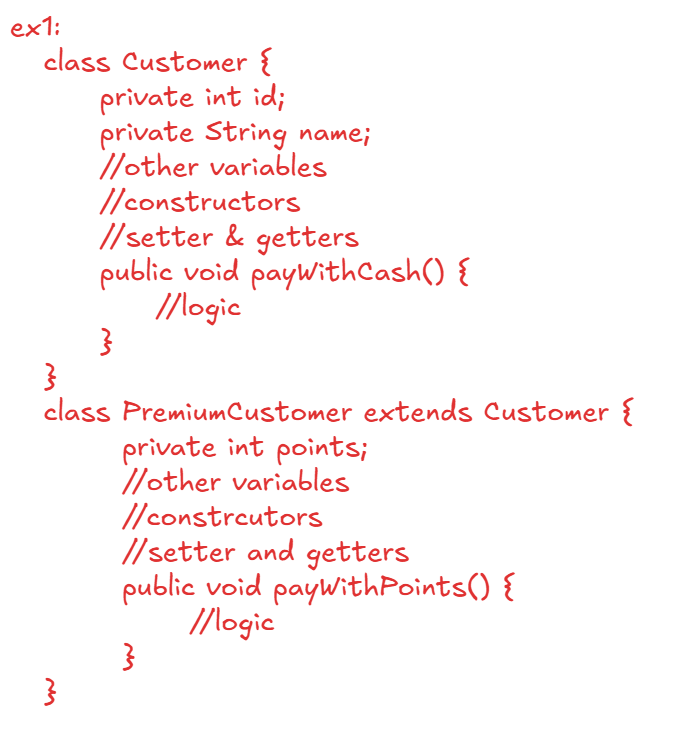
A) Yes

Inheritance

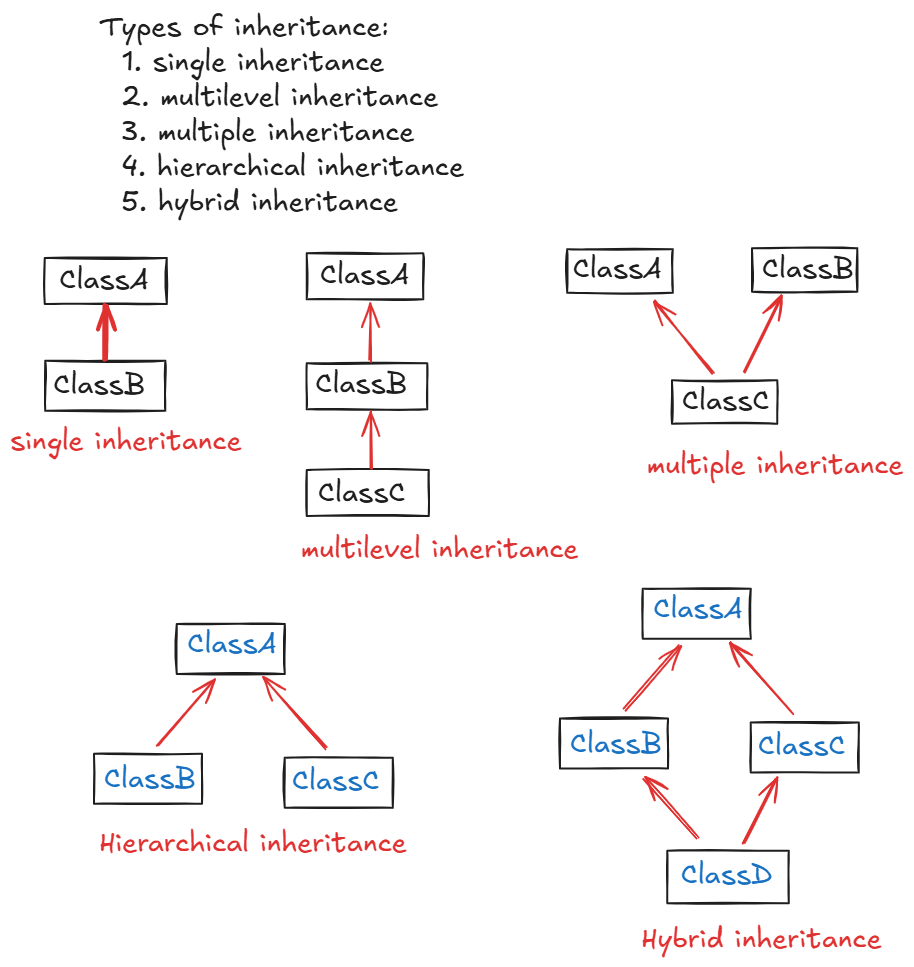
* Inheritance is a process of creating a new class from an existing class, by inheriting the properties and behaviour of an existing class.
* The existing class is called parent class, and the new class is called child class.
* By creating a new class from an existing class, we will get,

1. code reusability
2. reduces duplication of code(reduces code redundency)
3. improves productivity
4. code maintainance is easy

* The keyword “extends” is used to create a child class from the parent class.



* Here, PremiumCustomer is a child class and Customer is a parent class.
* PremiumCustomer has inherited the Customer functionality and also, added a new functionality called payWithPoints.
* Since, child class has more functionality, we create an object for the child class.



Single: creating a child class from a parent class.

multilevel: creating a child class from a child class.

multiple: creating a child class from more than one parent class.

hierarchical: creating mulitple child classes from a parent class.

hybrid: it is a combination of hierarchical and multiple.

Q) why multiple inheritance is not supported at classes level?

A) 1. when a child class object is created, the child class constructor calls the parent class constructor.

2. In multiple inheritance, a child class has more than one parent class.

3. So, the JVM gets an ambiguity that which parent class constructor it has to call. Because of this ambiguity, multiple inheritance is not supported.

Note: Multiple inheritance is possible with interfaces.

At classes level, hybrid inheritance is also not supported.

super keyword:

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

* super can be used as a keyword or as a super call.
* super keyword can be used to call a parent class variable or method.
* super call can be used to call a parent class constructor.
* In each constructor of a class, by default the first statement is a super call.

ex1:

**class** ClassA {

**public** ClassA() {

System.***out***.println("ClassA :: PARAMETERLESS constructor");

}

}

**class** ClassB **extends** ClassA {

**public** ClassB() {

System.***out***.println("ClassB :: PARAMETERLESS constructor");

}

}

**public** **class** Solution {

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

ClassB classB = **new** ClassB();

}

}

output:

ClassA:: PARAMETERLESS constructor

ClassB:: PARAMETERLESS constructor

ex2:

**class** ClassA {

**public** ClassA() {

System.***out***.println("ClassA :: PARAMETERLESS constructor");

}

**public** ClassA(**int** x) {

System.***out***.println("ClassA :: PARAMETERIZED constructor");

}

}

**class** ClassB **extends** ClassA {

**public** ClassB() {

**super**(10); //super call

System.***out***.println("ClassB :: PARAMETERLESS constructor");

}

}

**public** **class** Solution {

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

ClassB classB = **new** ClassB();

}

}

output:

ClassA :: PARAMETERIZED constructor

ClassB :: PARAMETERLESS constructor

ex3:

**class** ClassA {

**public** ClassA() {

System.***out***.println("ClassA :: PARAMETERLESS constructor");

}

**public** ClassA(**int** x) {

**this**();

System.***out***.println("ClassA :: PARAMETERIZED constructor");

}

}

**class** ClassB **extends** ClassA {

**public** ClassB() {

**this**(10); //this call

System.***out***.println("ClassB :: PARAMETERLESS constructor");

}

**public** ClassB(**int** x) {

System.***out***.println("ClassB :: PARAMETERIZED constructor");

}

}

**public** **class** Solution {

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

ClassB classB = **new** ClassB();

}

}

output:

ClassA :: PARAMETERLESS constructor

ClassB :: PARAMETERIZED constructor

ClassB :: PARAMETERLESS constructor

Q) can we write this() call and super() call in a single constructor?

A) No. Because, this() call or super() call must be first statement only.

* we can store child class object in a parent class reference, but we can’t store parent class object in child class reference.
* with parent class reference, we can only call parent class methods, but not child class methods.

example:

**class** ClassA {

**public** ClassA() {

System.***out***.println("ClassA :: PARAMETERLESS constructor");

}

**protected** **void** m1() {

System.***out***.println("ClassA :: m1() ");

}

}

**class** ClassB **extends** ClassA {

**public** ClassB() {

System.***out***.println("ClassB :: PARAMETERLESS constructor");

}

**public** **void** m2() {

System.***out***.println("ClassB :: m2()");

}

}

**public** **class** Solution {

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

//child class object stored in child class reference

ClassB classB = **new** ClassB();

classB.m1();

classB.m2();

//parent class object stored in child class reference

//error

ClassB classB = **new** ClassA();

//child class object stored in parent class reference

ClassA classA = **new** ClassB();

classA.m1();

classA.m2(); //error

}

}

/\*

\* example on hierarchical inheritance

\*/

**class** Employee {

**private** **int** empno;

**private** String ename;

**public** Employee(**int** empno, String ename) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

}

**class** FulltimeEmployee **extends** Employee {

**private** **double** sal;

**public** FulltimeEmployee(**int** empno, String ename, **double** sal) {

**super**(empno, ename);

**this**.sal = sal;

}

**public** **void** showDetails() {

System.***out***.println("empno : "+ getEmpno());

System.***out***.println("ename : "+ getEname());

System.***out***.println("sal : "+sal);

}

}

**class** ParttimeEmployee **extends** Employee {

**private** **double** hoursWorked;

**private** **double** ratePerHour;

**public** ParttimeEmployee(**int** empno, String ename, **double** hoursWorked, **double** ratePerHour) {

**super**(empno, ename);

**this**.hoursWorked = hoursWorked;

**this**.ratePerHour = ratePerHour;

}

**public** **void** showDetails() {

System.***out***.println("empno : "+ getEmpno());

System.***out***.println("ename : "+ getEname());

System.***out***.println("sal : " + (hoursWorked \* ratePerHour) );

}

}

**public** **class** Solution {

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

FulltimeEmployee fulltime = **new** FulltimeEmployee(101, "Scott", 5000.0);

fulltime.showDetails();

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

ParttimeEmployee parttime = **new** ParttimeEmployee(102, "Miller", 120, 40.0);

parttime.showDetails();

}

}

output:

empno : 101

ename : Scott

sal : 5000.0

=================================

empno : 102

ename : Miller

sal : 4800.0

polymorphism

* polymorphism is an ability of an object to perform similar action for different inputs.
* polymorphism denotes many forms of an action/task.
* polymorphism can be implemented into a software application using two mechanisms called method overloading and method overriding.

method overloading:

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

* Defining the same method for more than once in a class, with a difference in number of parameters or type of parameters or sequence of parameters is called method overloading.
* In method overloading, method name must be same but there must be a difference in the parameters.

ex1:

class ClassA {

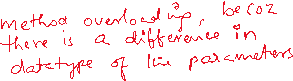
void m1(int x, int y)

{

//logic



}



void m1(int a, double b)

{

//logic

}

}

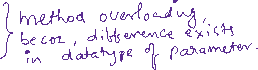
ex2:

class EmployeeService {

public Employee findEmp(int empno) {

//logic

}



public Employee findEmp(String email) {

//logic

}

}

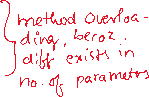
ex3:

class Recharge {

void doRecharge(long mobile,double amount)

{

//logic



}

void doRecharge(long mobile,double amount,String couponCode)

{

//logic

}

}

* method overloading does not depends on return type and access modifier.

ex:

public class ClassA {

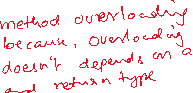
public double m1(double x, double y) {



//logic



}



private void m1(int a, int b) {

//logic

}

}

ex:

public class ClassA {

private double m1() {



//logic



}



public void m1() {

//logic

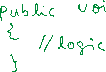
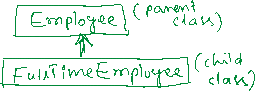
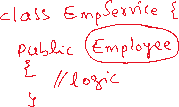
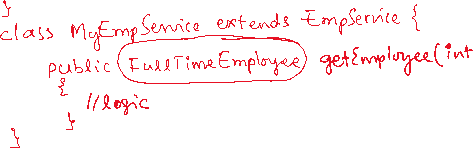
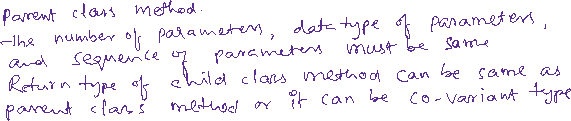
}

}

method overriding:

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

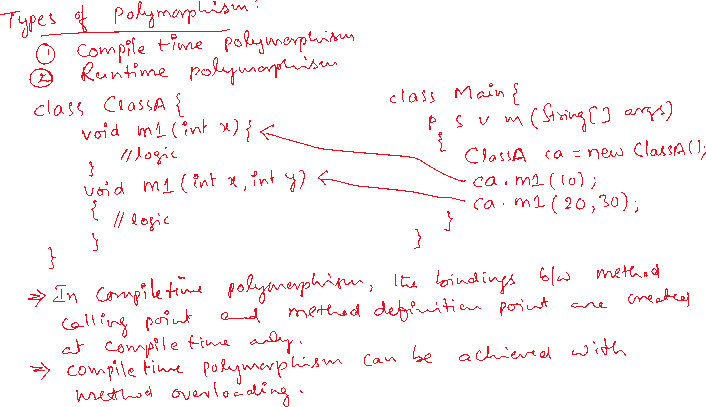
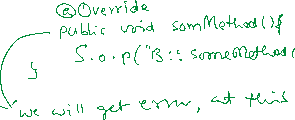
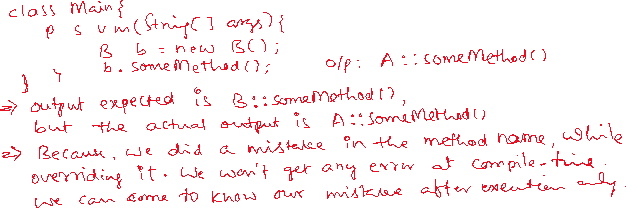
* method overriding is nothing but, re-defining a parent class behaviour in the child class.
* For method overriding, inheritance is required.
* If a method exists in parent class, but that method definition is not suitable for the child class requirement, then the child class can override that method.



@Override annotation:

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

* This annotation we can use at method level only.
* When this annotation is added, the compiler will check, whether this method overridden correctly or not. If not, the compiler will generate an error.
* It is not mandatory to use this annotation while overriding a method.
* If not used, and if we did a mistake in the method name or parameter types then we can identify our mistake after executing the program, by observing the result.
* So, it is recommended to use this annotation at method level, while overriding a parent class method in the child class.



Q) can we overload a constructor?

A) YES.

Q) can we override a constructor?

A) NO

Q) can we overload a method in the same class?

A) YES

Q) can we override a method in the same class?

A) NO

Q) can we override a method in the child class?

A) YES

Q) can we overload a method of parent class in the child class?

A) YES.

Q) can we overload a private method?

A) With in the same class, we can overload the private method.

Q) can we overload a static method?

A) YES, We can overload with in the same class.

Q) can we override a static method?

A) NO.

Q) when do you create a private method in a class?

A) when you want to create a helper method for another method, that helper method we create as private method.

Q) can we call non-static method in a static method directly in a class?

A) No. If you want to call non-static method in a static method then you have to create the object of the class.

ex:

**class** ClassA {

**void** m1() {

System.***out***.println("m1()");

}

**static** **void** m2() {

System.***out***.println("static m2()");

m1(); //error

}

}

solution:

**class** ClassA {

**void** m1() {

System.***out***.println("m1()");

}

**static** **void** m2() {

System.***out***.println("static m2()");

new ClassA().m1();

}

}

Q) can we call a static method in a non-static method directly in a class?

A) Yes.

ex:

**class** ClassA {

**void** m1() {

System.***out***.println("m1()");

m2(); //correct

}

**static** **void** m2() {

System.***out***.println("static m2()");

}

}

Q) can we use this or super keywords in a static method?

A) No.

Q) write the differences between overloading and overriding?

A) 1. In overloading, method name must be same but difference must exist in parameters(number/data type/sequence).

In overriding, methd name and the parameters must be same

2.For overloading, inheritance is not mandatory.

For overriding, inheritance is mandatory.

3.private and static methods can be overloaded, but can’t be overridden.

4. In overloading, the return type can be same or different type.

In overriding, the return type must same or a co-variant type.

5.constructors can be overloaded, but can’t be overridden.

6.Using overloading, we can achieve compile-time polymorphism.

Using overriding, we can achieve runtime polymorphism.

final keyword in Java:

* final is an important non-access modifier in Java, that can be used with variables, methods and classes, with a distinct purpose.

final variables:

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

* final variables are used to create constants in a program.
* For a final variable, we can assign a value for only once. we can’t reassign the value.
* final variables are 3 types.

1. instance final variables
2. static final variables
3. local final variables

instance final variable:

\* instance final variable is an object-level constant.

\* It means, for an object, the value assigned to the final variable is fixed. We can’t modify the value of that final variable with in that object scope.

\* You must assign the value to a final instance variable either at the variable declaration or in the constructor.

\* For a final variable, you can define a getter method, but you can’t define a setter method.

ex:

**import** java.util.Random;

**class** BankAccount {

**private** **final** **long** accountId;

**private** String accountHolderName;

**public** BankAccount() {

accountId=**new** Random().nextLong(376234198);

accountHolderName="john";

}

**public** **long** getAccountId() {

**return** accountId;

}

**public** **void** setAccountId(**long** accountId) {

**this**.accountId = accountId; //error

}

}

**public** **class** Solution {

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

BankAccount account1 = **new** BankAccount();

BankAccount account2 = **new** BankAccount();

System.***out***.println(account1.getAccountId());

System.***out***.println(account2.getAccountId());

` }

}

static final variable:

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

* static final variable is a class-level constant.
* a static final variable can be assigned with a value either at the variable declaration or in a static block. But not allowed in the constructor.
* A static final variable is a constant for all objects of the class.

ex:

**class** BankAccount {

**private** **final** **long** accountId;

**private** String accountHolderName;

**private** **static** **final** String ***ifsc***;

**static** {

***ifsc*** = "SBIN0001234";

}

**public** BankAccount() {

accountId=**new** Random().nextLong(376234198);

accountHolderName="john";

***ifsc*** = "SBIN0001234"; //error

}

**public** **long** getAccountId() {

**return** accountId;

}

**public** **void** setAccountId(**long** accountId) {

**this**.accountId = accountId; //error

}

}

local final variable:

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

* local final variable can be created with in a method or in a block.
* A local final variable can be assigned with a value for only once and it can’t be reassigned again in that method/block.
* The only modifier that could be used in a method is final.

ex:

**class** Student {

**private** **final** **int** studentId;

**public** Student(**int** studentId) {

**this**.studentId = studentId;

}

**public** **void** calculatePercentage(**int** totalMarks) {

**final** **int** maxMarks = 600;

**double** marksPercentage = ( totalMarks / (**double**) maxMarks) \* 100;

System.***out***.println("percentage = " + marksPercentage + "%");

}

}

**public** **class** Solution {

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

Student student = **new** Student(209981);

student.calculatePercentage(579);

}

}

Note: If you create a class level constant(static final), then according to Java naming conventions, the variable name should be written in upper case.

ex:

public static final int MAX\_USERS = 100;

final method:

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

* If you want to prevent a method from overriding by the child classes then you have to declare that method as a final method.
* a final method in a class indicates that the functionality is fixed/constant and the child classes can’t alter the functionality.
* If a method is private then it is not visible to the child classes. So, there is no use to make a private method as a final method.
* If you make a private method as final then you will not get any compile-time error. But there is no use.

ex:

**class** Student {

**private** **final** **int** studentId;

**public** Student(**int** studentId) {

**this**.studentId = studentId;

}

**public** **final** **void** calculatePercentage(**int** totalMarks) {

**final** **int** maxMarks = 600;

**double** marksPercentage = ( totalMarks / (**double**) maxMarks) \* 100;

System.***out***.println("percentage = " + marksPercentage + "%");

}

}

**class** DayScholar **extends** Student {

**public** DayScholar(**int** studentId) {

**super**(studentId);

}

@Override

**public** **void** calculatePercentage(**int** totalMarks) { //error

//logic

}

}

Q) can we create static final methods?

A) Yes, but anyhave static methods can’t be overridden. So, there is no use of making a static method as final.

Q) can we overload a final method?

A) Yes.

Q) which methods we can’t override?

A) 1. private methods

2. static methods

3. final methods

final class:

-----------

* If a class is declared as final then you can’t create child classes for that class.
* If a class is final, other classes can use this final class by creating an object, but they can’t extend this class.
* final variables can’t be re-assigned.
* final methods can’t be overridden.
* final classes can’t be extended.
* For example, java.lang.String is a final class, so we can’t create a child class for this String class.

Q) can we create a final object?

A) Yes. It means, we can’t re-assign another object.

example:

class A {

...

}

class Main {

public static void main(String[] args) {

final A a = new A(); //correct

a = new A(); //error

}

}

Q) can we create a final constructor?

A) No.

Q) Can we create a static constructor?

A) No.

Note: We can’t use any other modifier except public/private/protected with a constructor.

abstract classes and interfaces

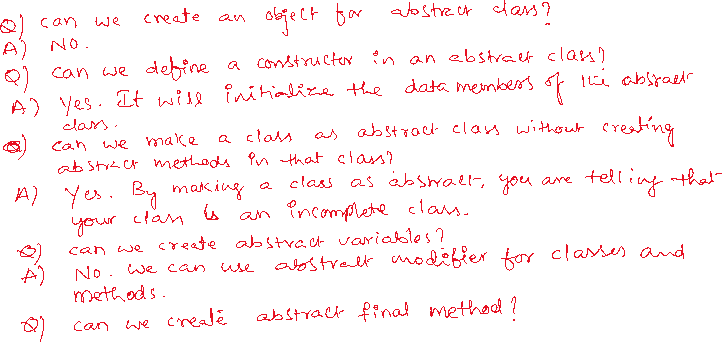
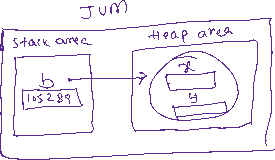
* In Java, we can implement abstraction principle using abstract classes and interfaces.
* abstraction is a process of providing the necessary details and hiding the unrequired information from the user of an application.



* while creating a parent class with child classes, in parent class we will define the common functionalities required to the child classes.
* Suppose, for any common functionality, each child class has to define the implementation in its own way, then we have to declare that method as abstract method in the parent class.
* For example, if we are creating a parent class Shape with child classes Circle and Rectangle. Here, calculateArea is a common functionality for the child classes, but the two child classes have to define this functionality in its own way. So, we have to declare the calculateArea as abstract method in the parent class.
* If a class has atleast one abstract method, then we must declare that class as abstract class.
* If a method has declaration and definition also in a class, then it is concrete method.
* If a method has only declaration/signature in a class, then it is abstract method.



* For an abstract class we can’t create an object.
* We can create a reference variable to the abstract class and we can store the child class object.



//abstract class example.

//InsurancePolicy.java

**public** **abstract** **class** InsurancePolicy {

**private** **double** basePremium;

**public** InsurancePolicy(**double** basePremium) {

**super**();

**this**.basePremium = basePremium;

}

**public** **abstract** **double** calculateRiskFactor();

**public** **final** **double** calculateFinalPremium() {

**return** basePremium + calculateRiskFactor();

}

**public** **double** getBasePremium() {

**return** basePremium;

}

**public** **void** setBasePremium(**double** basePremium) {

**this**.basePremium = basePremium;

}

}

//HealthInsurancePolicy.java

**public** **class** HealthInsurancePolicy **extends** InsurancePolicy {

**private** **int** age;

**private** **boolean** isSmoker;

**public** HealthInsurancePolicy(**double** basePremium, **int** age, **boolean** isSmoker) {

**super**(basePremium);

**this**.age = age;

**this**.isSmoker = isSmoker;

}

@Override

**public** **double** calculateRiskFactor() {

**double** riskFactor = 0;

**if** ( age >= 60 ) {

riskFactor = getBasePremium() \* 0.35;

}

**if**( age >= 60 && isSmoker ) {

riskFactor = getBasePremium() \* 0.50;

}

**return** riskFactor;

}

}

//Solution.java

**public** **class** Solution {

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

InsurancePolicy health = **new** HealthInsurancePolicy(27999.0, 62, **true**);

**double** finalAmount = health.calculateFinalPremium();

System.***out***.println("Base Premium : " + health.getBasePremium());

System.***out***.println("Final Premium : " + health.calculateFinalPremium());

}

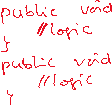
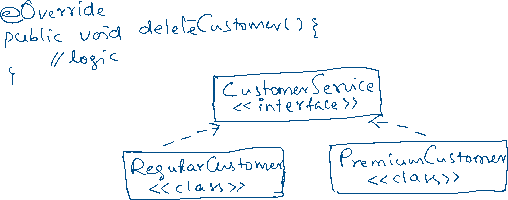
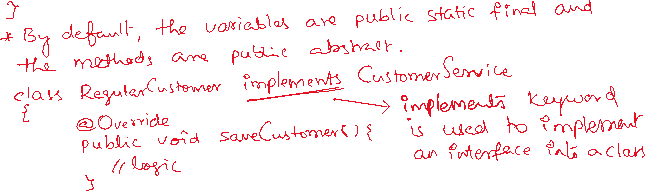
}

====================================================================

Interface:

----------

* In OOP terms, a family of classes means, the classes which are developed to provide a similar functionality to the users.
* While creating a family of classes, suppose if you want to define a contract that the classes must provide a certain functionalities with their own implementation, then you have to create an interface in the application.
* Interfaces are used to define a contract for a group of classes that they have to implement certain methods.
* The keyword to create an interface is “interface”.
* An interface can contain public static final variables and public abstract methods.



* One interface can extend one or multiple interfaces also.

interface I1 {

void m1();

}

interface I2 {

void m2();

}

interface I3 extends I1, I2 {

void m3();

}

* By default, an interface has default access modifier.
* You can make an interface as public also.
* public interface must be defined in a separate java file.
* If you are creating inner interfaces then you can use any access modifier(public/private/protected/default).

public interface Inter1 {

private interface Inter2 { //correct

.....

}

}

private interface Inter1 { //error

public interface Inter2 {

. . . .

}

}

Q) can we create an object for an interface?

A) No.

Q) can we define a constructor in interface?

A) No.

Q) If a class is not extending another class, then what is its super class?

A) java.lang.Object

Q) If an interface is not extending another interface, then what is its super?

A) nothing.

Q) can a class inherit another class and interface at a time?

A) Yes.

ex: interface I1 {

void m1();

}

class A {

void m2() {

//logic

}

}

class B extends A implements I1 {

@Override

public void m1() {

//logic

}

}

Note: extends followed by implements is correct, but implements followed by extends is wrong.

Q) If two interfaces have same abstract method, then how many times we need to override that method in the class?

A) one time.

ex:

interface I1 {

void m1();

}

interface I2 {

void m1();

}

class A implements I1, I2 {

@Override

public void m1() {

//logic

}

}

Types of interfaces:

====================

1. normal interface

2. marker interface

3. functional interface

\* From Java8 version, interface can also contain default methods

and static methods.

\* A normal interface can contain any number of public static

final variables, public abstract methods, default methods and

static methods.

\* A marker interface is an empty interface, it does not contain

any thing.

ex: interface MyInter {

}

\* When you want to add some special behaviour to the objects of

a class, not for the objects of all classes, then you create

a marker interface and implement it into that class.

\* For example, java.io.Serializable is a marker interface and if

a class implements this interface, then the objects of that class

are added with a special behaviour that they can be transferred

over a network.

class Employee implements java.io.Serializable {

//variables

//setter and getters

}

class Customer {

//variables

//setter and getters

}

\* Here, the objects of Employee class can be transferred over a

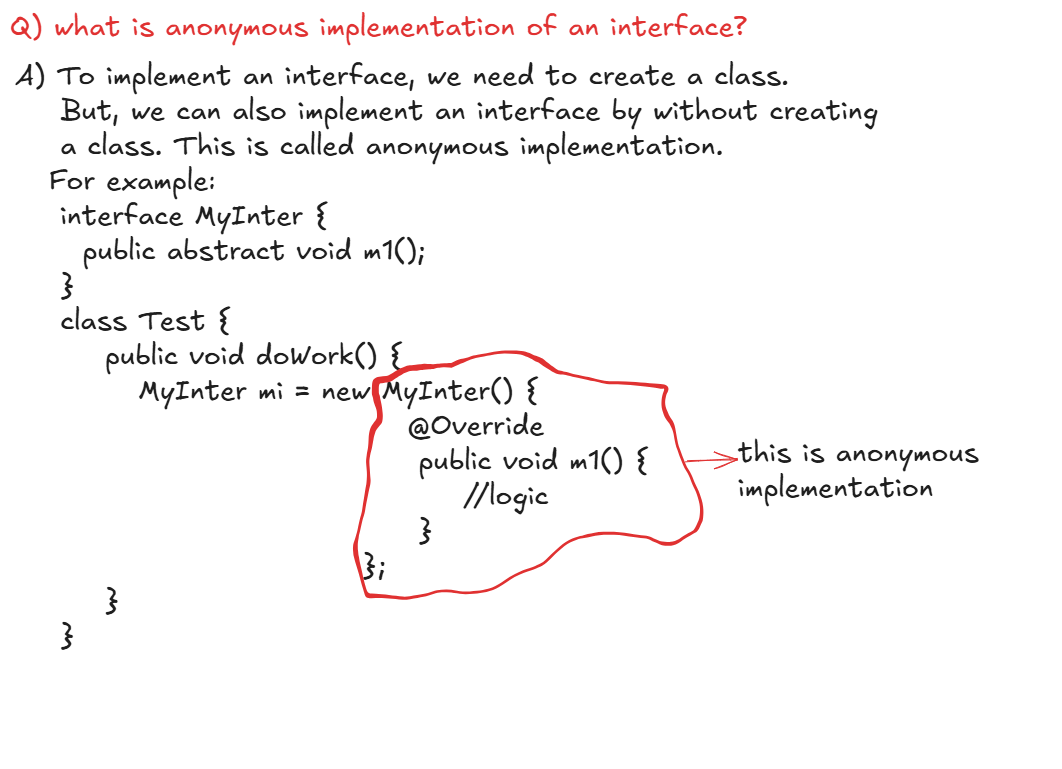
network, but not the objects of Customer class.

\* A functional interface can contain any number public static

final variables, any number of default methods, any number of

static methods, but it must contain a single abstract method.

\* functional interfaces concept is newly added in Java 8 version.



//Example: an interface with 2 implementations

**interface** Vehicle {

**void** start();

**void** stop();

**void** go();

}

**class** Bike **implements** Vehicle {

@Override

**public** **void** start() {

System.***out***.println("Bike :: start()");

}

@Override

**public** **void** stop() {

System.***out***.println("Bike :: stop()");

}

@Override

**public** **void** go() {

System.***out***.println("Bike :: go()");

}

}

**class** Car **implements** Vehicle {

@Override

**public** **void** start() {

System.***out***.println("Car :: start()");

}

@Override

**public** **void** stop() {

System.***out***.println("Car :: stop()");

}

@Override

**public** **void** go() {

System.***out***.println("Car :: go()");

}

}

**public** **class** Solution {

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

Vehicle v1 = **new** Bike();

v1.start();

v1.go();

v1.stop();

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

Vehicle v2 = **new** Car();

v2.start();

v2.go();

v2.stop();

}

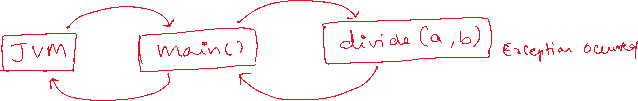
}

Exception Handling

* In a program, there can be 3 types of errors.

1. compile-time error
2. runtime error
3. logical error

* compile-time errors are the syntax errors, where the Java compiler will notify the errors to the programmer.
* runtime errors occur, because of the input values given by the user. These are also called as exceptions.
* logical errors doesn’t occur because of the syntax or the input.
* It occurs, because of the mistake did in the logic. This can be identified based on the input and the output.
* The default behaviour of an application, when an exception occurs is the application’s execution is terminated and the exception stack trace will be displayed to the user.



* Exception handling is a mechanism through which an application can handle the exceptions and avoids sudden termination of the application and also it can notify the user about the error, in a user understandable format.
* By adding exception handling, we can able to make an application to run smoothly, even though an exception occurs.

Exception handling keywords:

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

1.try

2.catch

3.finally

4.throws

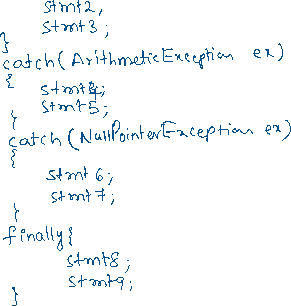
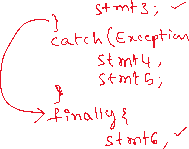
5.throw



* try block contains a group of statements from which an exception may occur.
* catch block contains a group of statements, which can handle the exception.
* finally block contains a group of statements, which must be executed irrespective of the exception.
* Suppose, if an exception occurs in the try block statement, then the control immediately jumps to the nearest catch block, after execution of the catch, the control then jumps to the finally block.



* Suppose, if exception doesn’t occur in try block, then the control will directly jump to the finally block without executing the catch block.



Q) can we define only try block without catch and finally blocks?

A) No

Hint: The allowed combinations are,

try + catch

try + finally

try + catch + finally

Q) can we define any statements/code in between try and catch blocks?

A) No. It is a compile-time error.

try{

stmt1;

stmt2;

}

stmt3; ---->compile-time error

catch(Exception ex) {

stmt4;

stmt5;

}

Q) can we define any statements/code in between catch and finally blocks?

A) No. It is a compile-time error.

try{

stmt1;

stmt2;

stmt3;

}

catch(Exception ex) {

stmt4;

stmt5;

}

stmt6; ----🡪 compile-time error

finally {

stmt7;

stmt8;

}

Q) can we write any java code after the finally block?

A) Yes.

Q) what is the rule to be followed while creating multiple catch blocks?

A) catch blocks for specific exceptions, followed by catch block for generic Exception is correct. But catch block for generic Exception, followed by catch block for specific exceptions is compile-time error.

try { try {

. . . . . . . .

. . . . . . . .

} }

catch(ArithmeticException ex) { catch(Exception ex) {

. . . . . .

. . . . . .



} }

catch(Exception ex) { catch(ArithmeticException ex) {



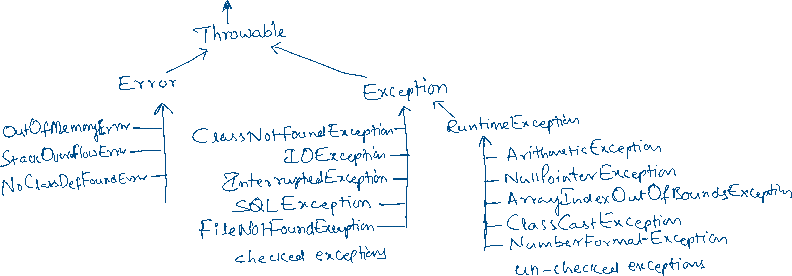
. . . . . .



. . . . . .



} }



Q) How can you identify that an exception class belongs to checked or un-checked exception?

A) If the super class of an exception class is Exception, then it is a checked exception. If the super class of an exception class is RuntimeException, then it is un-checked exception.

Q) what is the difference between checked and un-checked exception?

A) checked exceptions are those for which the compiler checks to see whether the programmer is handling the exception with try and catch blocks or propagating the exception with throws keyword or not. If not done, then compiler generates an error.

unchecked exceptions are those for which the compiler doesn’t check to see whether the programmer is handling the exception with try and catch blocks or propagating the exception with throws keyword or not. The compiler does not generate any error.

For checked exceptions, the compiler will enforce the programmer to handle the exception with try and catch or propagate the exception with throws keyword.

For uncheched exceptions, the compiler will not enforce the programmer to handle the exception with try and catch or propage the exception with throws keyword.

Q) In how many ways we can display the exception details?

A) 3 ways.

1.We can display the exception classname and the exception message

2.We can display only the exception message

3.We can display the complete stack trace of the exception.

\*To display the exception classname and the exception message, you have

to display the exception object.

System.out.println(ex);

\*To display only the exception message, you have to call getMessage()

method.

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

\*To display the complete stack trace, you have to call printStackTrace()

method.

ex.printStackTrace();

User-defined exceptions:

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

* As per the business requirements, in a Java project we need user-defined exceptions also.
* An user-defined exception class can be created in two ways.

1. by extending Exception class
2. by extending RuntimeException class

* If you want to make an user-defined exception class as a checked exception then extend Exception class.
* If you want to make an user-defined exception class as un-checked exception then extend RuntimeException class.
* While creating a user-defined exception class, you can make the class as empty or you can define a parameterized constructor.
* Suppose, if you want to throw an exception without a message, then create that exception class as empty.
* Suppose, if you want to throw an exception with message, then create that exception class with parameterized constructor.

ex:

public class UserNotFoundException extends Exception {

}

ex:

public class CustomerNotFoundException extends Exception {

public CustomerNotFoundException(String message)

{

super(message);

}

}

throw & throws :

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

* throw is a keyword, which is used to throw an user-defined exception from a method.



* throw keyword is used inside the method body.
* throw keyword can throw an object of an exception class.

ex:

class BankAccount {

void withdraw(double amount) {

try {

if(amount > balance) {

throw new InsufficientBalanceException();

}

else {

//logic

}

}

catch(AccountNotFoundException ex) {

ex.printStackTrace();

}

}

}

throws keyword:

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

* throws keyword is used in the method header/signature and it is used to propagate the exceptions from the current method to the caller method.
* If there is any chance of checked exception in a method and if it is not handled with try and catch blocks then that method must propagate the exception to the caller method, by using throws keyword.
* If a method is not handling a checked exception with try and catch blocks, or not propagating the exception to the caller with throws keyword then compiler generates an error.
* We can declare a list of exceptions in the method header with throws keyword by separating with comma(,).
* If there is a chance of unchecked exception in a method and if it is not handled with try and catch blocks then propagating that exception to the caller method with throws keyword is optional. Because, Java will automatically propagate the unchecked exceptions.

ex1:

class ClassA {

void m1()

{

int x = 5 / 0;

S.o.println(x);

}

}

\* Here, in m1() method ArithmeticException will occur. But try and catch blocks are not included to handle it and throws keyword is not used to propagate the exception to the caller. But still, we don’t get any compiletime error.

\* Because, ArithmeticExcpeption is unchecked exception, and it is automatically propagated to the caller method by Java.

ex2:

class ClassA {

void m1()

{

Class.forName(“java.ashokit.Demo”);

}

}

\* Here, in m1() method, ClassNotFoundException may be thrown, and it is a checked exception.

\* m1() method has not included try and catch blocks to handle it, or throws keyword is not used to propagate the exception to the caller method. So, compiler generates an error.

\* So, we need to modify the code like below.

class ClassA {

void m1() throws ClassNotFoundException

{

Class.forName(“java.ashokit.Demo”);

}

}

Q) can we throw mulitple exceptions at a time with throw keyword in a method?

A) No

class A {

void m1() {

if(somecondition) {

throw new MyException1(), new MyException2(); //error

}

}

}

Q) can we propagate unchecked exceptions with throws keyword?

A) Yes, but it is optional.

Q) if try block has return statement then will finally block execute?

A) Yes.

Q) if try block has exit statement then will finally block execute?

A) No.

Q) can we handle more than one exception in a single catch block?

## A) Yes. It is called multi exception catch block.

when you want to handle mulitple exceptions with the same code, then you can define multi exception catch block.

try {

stmt1;

stmt2;

stmt3;

}

catch(ArithmeticException | NullPointerException ex) {

stmt4;

stmt5;

}

Q) what is rethrowing an exception?

A) If you throw an exception from catch block, using throw keyword then it is called rethrowing an exception.

ex:

**void** m1(**int** a, **int** b) {

**try** {

**double** c = a / b;

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

}

**catch**(ArithmeticException ex) {

System.out.println(ex);

**throw** **new** ArrayIndexOutOfBoundsException();

}

**catch**(ArrayIndexOutOfBoundsException ex) {

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

}

}

Q) will method overloading depends on the execeptions declared in throws keyword?

A) No. The overloaded methods can throw any exceptions with throws keyword.

void m1() throws ClassNotFoundException {

//logic

}

void m1(int x) throws IOException {

//logic

}

Q) will method overriding depends on the exceptions declared in throws keyword?

A) Yes. we need to follow the below rules.

1) For checked exceptions,

1.1)The overriding method can throw the same exceptions

1.2)The overriding method can throw the exceptions that are

subclasses of the exceptions declared in the super class

method.

1.3)The overriding method can not throw new checked exceptions.

2) For un-checked exceptions, the overriding method can throw any

exceptions using throws keyword. Java doen not enforce any

restrictions.

ex1:

**class** ClassA {

**void** m1() **throws** ReflectiveOperationException {

Class.*forName*("java.lang.String");

}

}

**class** ClassB **extends** ClassA {

@Override

**void** m1() **throws** ClassNotFoundException { //correct

System.***out***.println("I am in subclass");

}

}

ex2:

**class** ClassA {

**void** m1() **throws** ReflectiveOperationException {

Class.*forName*("java.lang.String");

}

}

**class** ClassB **extends** ClassA {

@Override

**void** m1() **throws** IOException { //error

System.***out***.println("I am in subclass");

}

}

ex3:

**class** ClassA {

**void** m1() **throws** ReflectiveOperationException {

Class.*forName*("java.lang.String");

}

}

**class** ClassB **extends** ClassA {

@Override

**void** m1() **throws** ArithmeticException, NullPointerException { //correct

System.***out***.println("I am in subclass");

}

}

TEST :

1)

Implement a program to find the number of rabbits and chickens in a farm. Given the number of heads and legs of the chickens and rabbits in a farm, identify and display the number of chickens and rabbits in the farm.

If the given input cannot make a valid number of rabbits and chickens, then display an appropriate message.

Sample input output

Heads=150, legs=500 chickens=50

Rabbits = 100

Heads=3, legs=11 chickens and rabbits can not be found

2)

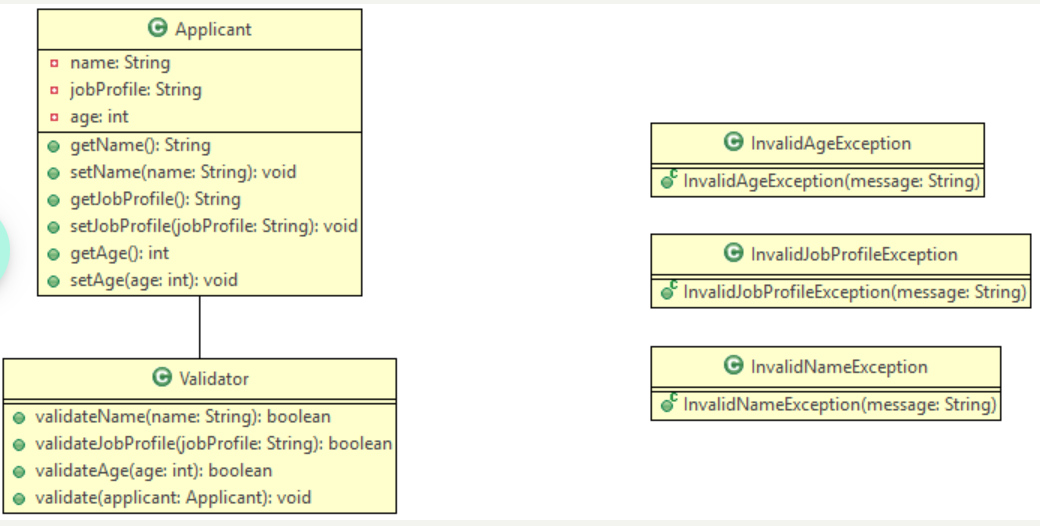
Implement a program to find out whether a number is a seed of another number.

A number X is said to be a seed of number Y if multiplying X by its every digit equates to Y.

E.g.: 123 is a seed of 738 as 123\*1\*2\*3 = 738

3)

A bank wants to conduct examinations for recruitment. You need to develop an application for the applicants to submit their details.



Validator

Method Description

validateName(String name)

Validate that the name is not null or empty. If the name is null or empty, return false, else return true.

validateJobProfile(String jobProfile)

Validate that the jobProfile is either 'Associate' or 'Clerk' or 'Executive' or 'Officer'. If the jobProfile is valid, return true, else return false. Perform case-insensitive comparison.

validateAge(int age)

Validate that the age is between 18 and 30 (both inclusive). If the age is valid, return true, else return false

validate(Applicant applicant)

Validate the details of the applicant by calling the appropriate methods. If any validation fails, throw user defined exceptions based on the below description.

4)

 create a Booking class with customerEmail, seatsRequired, isBooked and seatsAvailable attributes. Make only seatsAvailable attribute as static. Create setter and getters.

Define a constructor with customerEmail and seatsRequired as arguments.

Define a static block to initialize static attribute with value 400

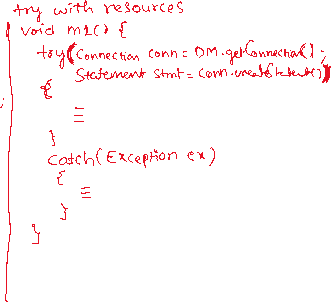
If the required number of seats are available, then set isBooked to true and update the seatsAvailable accordingly.

If the required number of seats are not available, then set isBooked to false.

Test the application by creating at least two objects in main method.

try with resources:

* try with resources means, we can define a try block with parameters.
* The parameters of try block, must be AutoCloseable objects.
* try block it self will clean up/close the resources which are given as parameters , once the try block is executed successfully or by throwing an exception.
* For example, if your try block contains the code to open a database connection then whether exception occurs or does not occurs, the connection must be closed. For this, we define finally block to clean up the resources.
* if you use try with resources, the try block only will clean up the given resources, so you no need to define finally block to clean up the resources.
* For a try block, we can pass more than one parameter also by separating with semi-colon.
* AutoCloseable objects means, the objects of the classes that implements AutoCloseable interface.



Multithreading

=====================

process:

-------

* A process is a program/an application in execution.
* Suppose, when you start Google chrome browser, as you know Google chrome is an application and when you start the browser, at Operating System level, a process is created and stored into RAM(tempory/primary memory).
* Each process will execute on a cpu core of a processor.
* If you have one cpu core processor and two processes to execute, then

cpu core will execute each process for a time period, also called time slice.

* It means, the cpu core executes process1 for approx 100ms, then it will switch to process2 and executes for approx 100ms, then it will switch back to process1 for approx 100ms, and so on.
* If you have two cpu core processor and two processes to execute, each process will execute on a separate cpu core, parallelly.

thead:

-----

* An application/process contains mulitple tasks.
* If the tasks are executed sequentially then application completion will take more time and the performance will be decreased.
* If the tasks are executed parallely then application will be completed fastly and performance will be improved.
* A thread is a group of statements which are executed independently to complete task, from the rest of the application execution.
* Multithreading is dividing a big task into smaller, independent tasks that can run parallely, so that improves efficiency and reducing the time.
* A general example for multithreading is cooking a meal for dinner.

task1: prepare pasta (thread1)

task2: prepare veg curry (thread2)

task3: prepare Grill chicken (thread3)

* A programmatic example for multithreading is university appln.

task1: upload the students details to the database (thread1)

taks2: generate the reports for the students (thread2)

Q) what is the difference between process and thread?

A) 1. A process contains one or more threads. A thread does not contain process, but it a part of a process.

2. Switching from process-to-process or thread-to-thread by the CPU is called context-switch. This context-switch between the processes will take more time and between the threads takes less time.

creating theads:

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

* In a Java application, threads can be created in two ways.

1. create a class by extending Thread class.
2. create a class by implementing Runnable interface.

* In both cases, you have to implement a thread’s logic by overrding run() method.
* Runnable is a functional interface, which has single abstract method called public void run() method.
* Thread class implements Runnable interface, and has overridden run() method, but its logic is a dummy logic.
* So, when you are creating your thread class, you have to override the run() method to define the logic.

ex1: public class MyThread extends Thread {

@Override

public void run() {

//thread’s logic

}

}

ex2: public class MyThread implements Runnable {

@Override

public void run() {

//thread’s logic

}

}

Q) which is the better way to create a thread, extending Thread class or implementing Runnable interface?

A) implementing Runnable interface is better option than extending Thread class. Because, if you implement Runnable interace, still the class can extend a parent class.

//A sample example on multiple threads

**package** com.ashokit.thread;

**class** MyThread1 **extends** Thread {

@Override

**public** **void** run() {

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

System.***out***.println("i = " + i);

**try** {

Thread.*sleep*(2000); //2 sec

}

**catch**(InterruptedException ex) {

ex.printStackTrace();

}

}

}

}

**class** MyThread2 **implements** Runnable{

@Override

**public** **void** run() {

**for**(**int** j = 11; j <= 20; j++ ) {

System.***out***.println("j = " + j);

**try** {

Thread.*sleep*(2000); //2 sec

}

**catch**(InterruptedException ex) {

ex.printStackTrace();

}

}

}

}

**public** **class** Main {

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

MyThread1 t1 = **new** MyThread1();

//wrapping Runnable object into a Thread object

//we are wrapping, because Runnable object

//doesn't have a start() method.

Thread t2 = **new** Thread(**new** MyThread2());

t1.start();

t2.start();

}

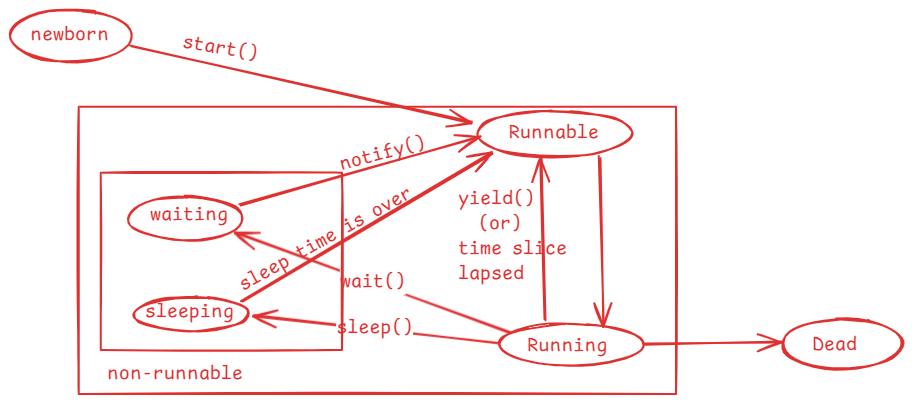
}

Thread life cycle:

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

* Thread life cycle means, from thread creation to thread destruction, it will undergo different phases and these phases forms the life cycle.

1. Newborn
2. Runnable
3. Running
4. Non-Runnable
5. Dead



* when a thread object is created then the thread enters into newborn state.
* when start() method is called on a thread object then the thread is moved to the Runnable state.
* mulitple threads can sit in Runnable state at a time.
* Thread schedular will pick up one thread from Runnable state and moved

it into Running state.

* In Running state, run() method is called.
* When the time slice is lapsed/when yield() method is called, a thread is moved back from Running to Runnable state by the schedular.
* when sleep() is called, a thread is moved from Running state to the non-runnable state.
* once sleep time is completed, a thread is moved to Runnable state.
* when wait() is called, a thread is moved from Running state to non-runnable state.
* when notify()/notifyAll() method is called, then a waiting thread goes to Runnable state.
* If a thread’s execution is finished, then it goes to dead state.

yield() method:

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

yield() method tells the schedular that a thread is willing to come out of Running state and it wants to give chance to other thread to enter into running state.

yield() is a static method, so it can be called with classname.

Thread.yield();

//example on yield() method

**package** com.ashokit.thread;

**class** MyThread1 **extends** Thread {

@Override

**public** **void** run() {

//Thread.currentThread() returns a current thread reference

//getName() returns the name of the current thread.

System.***out***.println("Inside " + Thread.*currentThread*().getName());

**for**(**int** i=1; i<=5; i++) {

System.***out***.println("i = "+i);

Thread.*yield*();

}

System.***out***.println( Thread.*currentThread*().getName() + " finished");

}

}

**class** MyThread2 **extends** Thread {

@Override

**public** **void** run() {

System.***out***.println("Inside " + Thread.*currentThread*().getName());

**for**(**int** j=1; j<=5; j++) {

System.***out***.println("j = "+j);

Thread.*yield*();

}

System.***out***.println( Thread.*currentThread*().getName() + " finished");

}

}

**public** **class** MainClass {

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

MyThread1 t1 = **new** MyThread1();

MyThread2 t2 = **new** MyThread2();

t1.setName("First-Thread");

t2.setName("Second-Thread");

t1.start();

t2.start();

}

}

joining the threads:

* If one thread wants to wait until the other threads execution is completed then we need to join the threads.
* If t1 and t2 are the threads, and if we join t2 with t1 then t1 will wait until t2’s execution is completed.
* For example, thread1 is generating a report and thread2 is proofreading a report, then thread2 can proofread after generating the report. So thread2 has to wait until thread1 has generated the report. For this, we can join thread1 with thread2.

//example1

/\*

\* This example has two threads, main thread and other thread

\* we have joined other thread with the main thread

\* the main thread execution is suspended until the other thread

\* execution is finished.

\*/

**package** com.ashokit.thread;

**class** MyThread **extends** Thread {

@Override

**public** **void** run() {

System.***out***.println("Inside " +Thread.*currentThread*().getName());

**for**(**int** i = 1; i <=50; i++) {

System.***out***.println(" i = "+i);

**try** {

Thread.*sleep*(2000);

}

**catch**(InterruptedException ex) {

ex.printStackTrace();

}

}

System.***out***.println(Thread.*currentThread*().getName() +" is completed");

}

}

**public** **class** MainClass {

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

Thread.*currentThread*().setName("Main Thread");

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

MyThread t1 = **new** MyThread();

t1.setName("Other Thread");

t1.start();

**for**(**int** j = 51; j <= 70; j++ ) {

System.***out***.println(" j = "+j);

**if**(j==55) {

t1.join();

}

Thread.*sleep*(2000);

}

System.***out***.println(Thread.*currentThread*().getName()+ " is completed");

}

}

//example2

/\*

\* This program has three threads, main thread, other thread1 and other

\* thread2

\* The other thread1 is joined with main thread and other thread2 is joined

\* with other thread1.

\*/

**package** com.ashokit.thread;

**class** MyThread1 **extends** Thread {

MyThread2 t2;

**public** MyThread1(MyThread2 t2) {

**this**.t2 = t2;

}

@Override

**public** **void** run() {

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

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

System.***out***.println(" i = " + i);

**try** {

**if** ( i == 6 )

{

t2.join();

}

Thread.*sleep*(2000);

}

**catch**(Exception e) {

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

}

}

System.***out***.println(Thread.*currentThread*().getName() + " is finished");

}

}

**class** MyThread2 **extends** Thread {

@Override

**public** **void** run() {

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

**for**(**int** j=11; j <= 20; j++) {

System.***out***.println(" j = " + j);

**try** {

Thread.*sleep*(5000);

}

**catch**(Exception e) {

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

}

}

System.***out***.println(Thread.*currentThread*().getName() + " is finished");

}

}

**public** **class** Solution {

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

Thread.*currentThread*().setName("Main thread");

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

MyThread2 t2 = **new** MyThread2();

t2.setName("Other thread2");

MyThread1 t1 = **new** MyThread1(t2);

t1.setName("Other thread1");

t1.start();

t2.start();

**for** ( **int** k = 101; k <= 111; k++) {

System.***out***.println("k = " + k);

**if**( k == 105 ) {

t1.join();

}

}

}

}

Thread priority:

* A thread priority is an integer value and its range is from 1 to 10.



* A thread priority is an indication to the schedular that which thread should get running state first. But it depends on thread schedular.
* The default priority of a thread is 5.
* In Thread class, 3 class level constants are defined to indicate the mininum, normal and the maximum priority that a thread can have.
* public static final int MIN\_PRIORITY = 1;
* public static final int NORM\_PRIORITY = 5;
* public static final int MAX\_PRIORITY = 10;
* a thread priority can be changed by calling setPriority() and can be retrieved by calling getPriority().
* if we set the priority for a thread < 1 or > 10 then at runtime, IllegalArgumentException will be thrown.

t1.setPriority(Thread.MAX\_PRIORIRY);

t2.setPriority(11); //exception

* You can set the priority for a thread, before you start the thread.
* You can retrieve the priority of a thread, by calling getPriority().
* For ex:
* Thread t = Thread.currentThread();
* sysout(“current thread priority : “ + t.getPriority());

Thread synchronization:

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

* If multiple threads are using a shared resource and trying to read/modify the data at the same time then we will get unpredictable results.
* Suppose, two people are withdrawing the money from the same bank account at the same time, then there is a chance of overdrawing the money.
* For example, balance available is 5000.0, and the two people at a time can see the balance as 5000.0 and if they withdraw 3000.0 each, then the total amount withdrawn is 6000.0. This is overdrawing the amount and leads to inconsistency.
* Threads synchronization controls access to a shared resource by multiple threads, by allowing only one thread to act on the shared resource at the same time.
* Threads synchronization can be implemented using synchronized keyword.
* with synchronized keyword(non-access modifier) we can create synchronized methods or synchronized blocks.
* synchronization works by acquiring the lock on an object by a thread.
* In Java, every object has a lock/monitor.
* A thread acquires the lock on the object, by calling synchronized method or synchronized block.
* Only one thread can acquire the lock on an object at a time.
* If two threads are calling a synchronized method at the same time, then one of the two threads acquires lock on the object.
* when the execution of a synchronized method/block is completed by a thread then the lock gets released.

/\*

\* In this example, Course is a shared resource for two threads

\* The two threads are trying to register for the same course

\* which has only one seat available.

\* If synchronization is not applied, there is a possibility that

\* the two threads registered for the same seat.

\* To avoid this inconsistency, we made the registerForCourse() method

\* as synchronized.

\* One thread at a time can execute the synchronized method, so that

\* we can avoid the inconsitency.

\*/

package com.ashokit.thread;

class Course {

private String courseName;

private int numOfSeatsAvailable;

public Course(String courseName, int numOfSeatsAvailable)

{

super();

this.courseName = courseName;

this.numOfSeatsAvailable = numOfSeatsAvailable;

}

public synchronized void registerForCourse(int rollno) {

try {

if ( this.numOfSeatsAvailable - 1 < 0 ) {

throw new Exception("Sorry, seats are not available!. Your rollno : " + rollno);

}

System.out.println("Booking successful!!! Your rollno is : " + rollno);

numOfSeatsAvailable -= 1;

System.out.println("Available seats now : " + this.numOfSeatsAvailable);

}

catch(Exception ex) {

System.out.println("ERROR : " + ex.getMessage());

}

}

}

class RegisterThread extends Thread {

Course course;

int rollno;

RegisterThread(Course course, int rollno) {

this.course = course;

this.rollno = rollno;

}

@Override

public void run() {

course.registerForCourse(rollno);

}

}

public class MainClass {

public static void main(String[] args) {

Course course = new Course("CSE", 1);

RegisterThread t1 = new RegisterThread(course, 101);

RegisterThread t2 = new RegisterThread(course, 102);

t1.start();

t2.start();

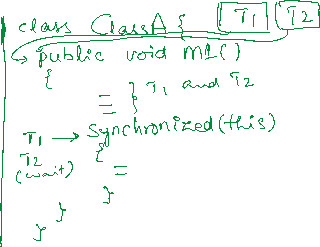
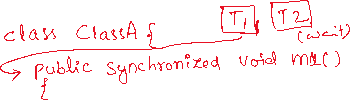
}

}

synchronized block:

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

* Suppose, you have a synchronized method but only some statements of the method if they are executed simultaneously by two threads will cause inconsistency.
* In this case, when one thread is executing synchronized method, the other thread has to wait until the entire method is executed by the thread one. It means, the other thread has to wait for more time unnecessarily.
* So, to reduce the unnecessary waiting time for the threads, we can use synchronized blocks inside a method.
* If one thread is executing synchronized block, the other thread has to wait only until the thread1 has executed the block. So, waiting time is decreased.



* To the synchronized block, we should pass an object, whose lock a thread has to acquire as a parameter.
* If it is a non-static method, we can pass “this” as a parameter.
* If it is a static method, we can pass Classname.class as a parameter.

Q) How many synchronized blocks can be created in a method?

A) zero or multiple

Q) Which can improve the performance of an application, synchronized method or synchronized block?

A) synchronized block.

Inter-thread communication:

* Inter-thread communication is possible between synchronized threads.
* If we want one synchronized thread to wait and wants to give chance to another synchronized thread to execute then we need inter-thread communication.
* For inter-thread communication, we have to use the following of the Object class.

1. wait() : moves current thread to waiting state, until another invokes notify()/notifyAll() method.
2. wait(millis) : moves current thread to waiting state until another thread invokes notify()/notifyAll() method, or time has lapsed.
3. notify(): notifies a single waiting thread, to resume
4. notifyAll(): notifies all waiting threads, to resume

/\*

\* In this example, Course is a shared resource for three threads

\* The two threads are trying to register for the same course

\* which has only one seat available and the third thread

\* is cancelling the seat.

\* The one thread gets booking success, and the other thread waits to

\* see for the cancellation of seat.

\* The cancellation thread cancels the seat and notifies the waiting thread.

\* so, the other thread also gets booking successful.

\*

\*/

**package** com.ashokit.thread;

**class** Course {

**private** String courseName;

**private** **int** numOfSeatsAvailable;

**public** Course(String courseName, **int** numOfSeatsAvailable) {

**super**();

**this**.courseName = courseName;

**this**.numOfSeatsAvailable = numOfSeatsAvailable;

}

**public** **synchronized** **void** registerForCourse(**int** rollno) {

**try** {

**if**(**this**.numOfSeatsAvailable - 1 < 0)

wait();

**if** ( **this**.numOfSeatsAvailable - 1 < 0 ) {

**throw** **new** Exception("Sorry, seats are not available!. Your rollno : " + rollno);

}

System.***out***.println("Booking successful!!! Your rollno is : " + rollno);

numOfSeatsAvailable -= 1;

System.***out***.println("Available seats now : " + **this**.numOfSeatsAvailable);

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

}

**catch**(Exception ex) {

System.***out***.println("ERROR : " + ex.getMessage());

}

}

**public** **synchronized** **void** cancelSeats() {

**try** {

**this**.numOfSeatsAvailable += 1;

System.***out***.println("Cancellation Successful");

System.***out***.println("Available seats now : " + **this**.numOfSeatsAvailable);

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

notify();

}

**catch**(Exception ex) {

ex.printStackTrace();

}

}

}

**class** RegisterThread **extends** Thread {

Course course;

**int** rollno;

RegisterThread(Course course, **int** rollno) {

**this**.course = course;

**this**.rollno = rollno;

}

@Override

**public** **void** run() {

course.registerForCourse(rollno);

}

}

**class** CancelThread **extends** Thread {

Course course;

CancelThread(Course course) {

**this**.course = course;

}

@Override

**public** **void** run() {

course.cancelSeats();

}

}

**public** **class** MainClass {

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

Course course = **new** Course("CSE", 1);

RegisterThread t1 = **new** RegisterThread(course, 101);

RegisterThread t2 = **new** RegisterThread(course, 102);

t1.start();

t2.start();

**try** {

Thread.*sleep*(8000);

}

**catch**(Exception ex) {

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

}

CancelThread t3 = **new** CancelThread(course);

t3.start();

}

}

Q) why wait(), notify() and notifyAll() methods are given in Object class, not in Thread class?

A) 1. When you synchronize on an object, the thread acquires lock

associated with the object.

2.In Inter-thread communication, the threads coordination happens by

acquiring and releasing the locks on the object. So, the wait(),

notify() and notifyAll() are ties to the lock of the object, not the

thread.

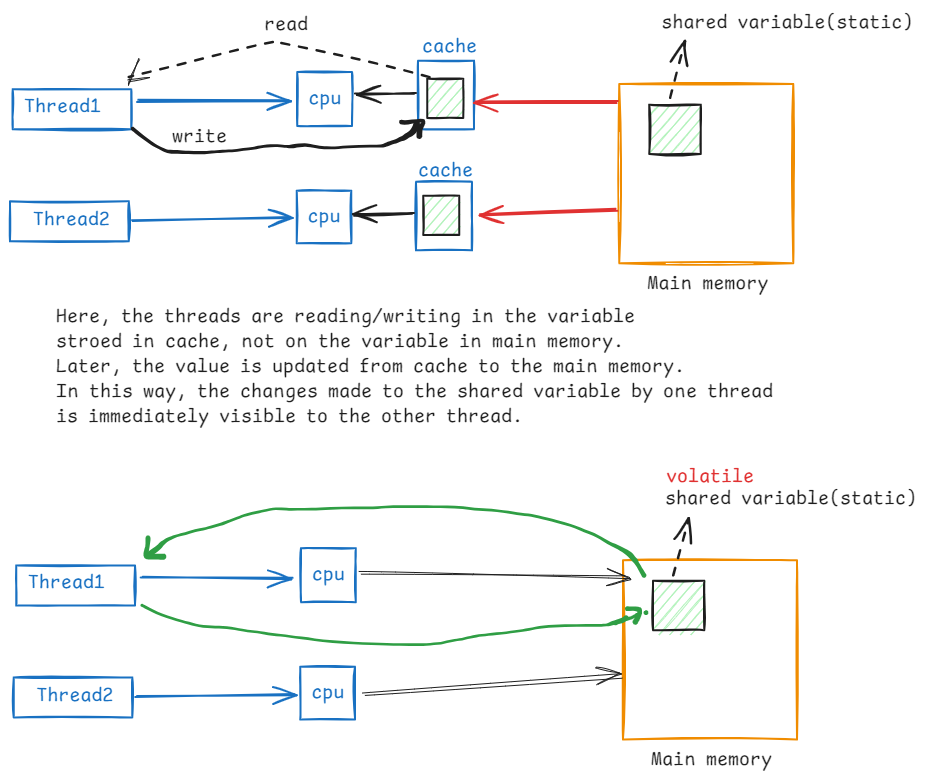
1. That’s why these methods are given in Object class, but not in Thread class.

Deadlock:

* Suppose, we have two threads and two resources.
* thread1 holds the lock on resource1 and waiting for lock on resource2 and thread2 holds the lock on resource2 and waiting for lock on resource1.
* In this case, thread1 is blocked for resource2 and thread2 is blocked for resource1.
* So, the two threads will never complete. This is called deadlock situation.
* Deadlocks can be prevented by making the two threads to acquire the locks on the resources in the same order.
* **package** pack1;
* **class** MyThread1 **extends** Thread
* {
* Object obj1;
* Object obj2;
* **public** MyThread1(Object obj1, Object obj2)
* {
* **this**.obj1=obj1;
* **this**.obj2=obj2;
* }
* @Override
* **public** **void** run() {
* **synchronized**(obj1) {
* System.***out***.println(Thread.*currentThread*().getName() +" has acquired lock on resource1");
* **try** {
* Thread.*sleep*(2000);
* }
* **catch**(InterruptedException ie) {
* System.***out***.println(ie);
* }
* System.***out***.println(Thread.*currentThread*().getName()+ " is waiting to acquire lock on resource2");
* **synchronized**(obj2) {
* System.***out***.println(Thread.*currentThread*().getName() +" has acquired lock on resource2");
* }
* }
* System.***out***.println(Thread.*currentThread*().getName()+" : is completed");
* }
* }
* **class** MyThread2 **extends** Thread
* {
* Object obj1;
* Object obj2;
* **public** MyThread2(Object obj1, Object obj2)
* {
* **this**.obj1=obj1;
* **this**.obj2=obj2;
* }
* @Override
* **public** **void** run() {
* **synchronized**(obj2) {
* System.***out***.println(Thread.*currentThread*().getName() +" has acquired lock on resource2");
* **try** {
* Thread.*sleep*(2000);
* }
* **catch**(InterruptedException ie) {
* System.***out***.println(ie);
* }
* System.***out***.println(Thread.*currentThread*().getName()+ " is waiting to acquire lock on resource1");
* **synchronized**(obj1) {
* System.***out***.println(Thread.*currentThread*().getName() +" has acquired lock on resource1");
* }
* }
* System.***out***.println(Thread.*currentThread*().getName()+" : is completed");
* }
* }
* **public** **class** DeadLock {
* **public** **static** **void** main(String[] args) {
* Object obj1 = **new** Object();
* Object obj2 = **new** Object();
* MyThread1 t1 = **new** MyThread1(obj1, obj2);
* t1.setName("Thread1");
* MyThread2 t2 = **new** MyThread2(obj1, obj2);
* t2.setName("Thread2");
* t1.start();
* t2.start();
* }
* }

volatile keyword:

* In Java, when a thread is executing on a CPU core, it allows a thread to cache variable for better performance.
* If there is a shared variable(static) for two threads, then the variable gets cached and the threads are performing read/write operations on the variable in cache.
* So, the changes made to the shared variable by one thread does not immediately reflect in the main memory. Hence, the other thread can’t access the updated value.
* To avoid this issue, volatile is provided and if we declare a variable as volatile, then a thread can not store it in cache and the threads can perform read/write operations directly on the shared variable in main memory.



Q) what are the types of threads?

A) 2 types of threads

1. user threads

2. daemon threads

\* by default, each thread is a user thread.

\* Main thread will not terminate the currently running JVM, until user threads are completed.

\* If make any thread as daemon thread, then Main thread does not wait for the completion of deamon thread.

\* If all the user threads in application are completed, then main thread will terminate the JVM. So, the deamon threads are also terminated.

\* setDeamon() method can make a thread as daemon thread.

\* isDeamon() method can be used to verify that a thread is daemon thread or user thread.

\* we create daemon threads in an application, to perform any background activities. For example, performance monitoring of an application.

\* In JVM, Garbage Collector thread is a daemon thread and it executes continously until JVM is terminated.

//Daemon thread demo

**package** com.ashokit.thread;

**class** MyThread1 **extends** Thread {

@Override

**public** **void** run() {

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

**for**(**int** i=1; i<=30; i++) {

System.***out***.println("i = " + i);

**try** {

Thread.*sleep*(2000);

}

**catch**(InterruptedException ex) {

System.***out***.println("Error : " + ex.getMessage());

}

}

System.***out***.println(Thread.*currentThread*().getName() + " : is finished");

}

}

**public** **class** MainClass {

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

Thread.*currentThread*().setName("Main Thread");

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

MyThread1 t1 = **new** MyThread1();

t1.setName("Daemon Thread");

t1.setDaemon(**true**);

t1.start();

**for**(**int** j=1; j<=10; j++) {

System.***out***.println("j = " + j);

**try** {

Thread.*sleep*(2000);

}

**catch**(InterruptedException ex) {

System.***out***.println("Error : " + ex.getMessage());

}

}

System.***out***.println(Thread.*currentThread*().getName() + " : is finished");

}

}

==================================================

Generics

* Suppose, we have a requirement that we need multiple classes with similar functionalities and they have to work on different types of objects.
* For example, we need classes with add and edit functionalities for different types of objects. So, we can create classes like below.

class EmployeeOperations {

void add(Employee e) {

}

void edit(Employee e) {

}

}

class CustomerOperations {

void add(Customer c) {

}

void edit(Customer c) {

}

}

class ProductOperations {

void add(Product p) {

}

void edit(Product p) {

}

}

* If we start creating classes like this, then the number of classes will be increased.
* So, one solution is, we can create a class that could perform the functionalities for all types of objects, like below.

class Operations {

void add(Object obj) {

}

void edit(Object obj) {

}

}

Here, the parameter of the methods is Object type. We know that Object is a super class for all classes in Java, so, these methods can perform the operations for any type of objects.

* In the methods of this class, type casting is required for performing the operation on actual object.
* For example, if you pass Employee object to the add() method, the method has to type cast the Object to the Employee type.

for example: Employee e = (Employee)obj;

* If you did any mistake in type casting like type casting to a wrong class type then You will get, ClassCastException.
* So, in Java, Generics concept is introduced.
* With Generics we can create classes, methods and interfaces that could be used for different types of objects and we can choose the object type at runtime.

For example:

public class Operations<T> {

void add(T t) {

}

void edit(T t) {

}

}

* The above is a generic class, where we need to pass the object type at runtime, within angle bracket(<>).

For example:

Operations<Employee> empOp = new Operations<Employee>();

Operations<Customer> custOp = new Operations<Customer>();

* Here, empOp object works for Employee class objects and custOp object works for Customer class objects. So, no type casting is required.
* So, Generics are introduced in Java, for 2 reasons.

1. To reduce the number of classes for similar operations.
2. To avoid type casting.

* While creating Generic classes/interfaces/methods, we use some special characters.

T --- Type

E --- Element

K --- Key

V --- Value

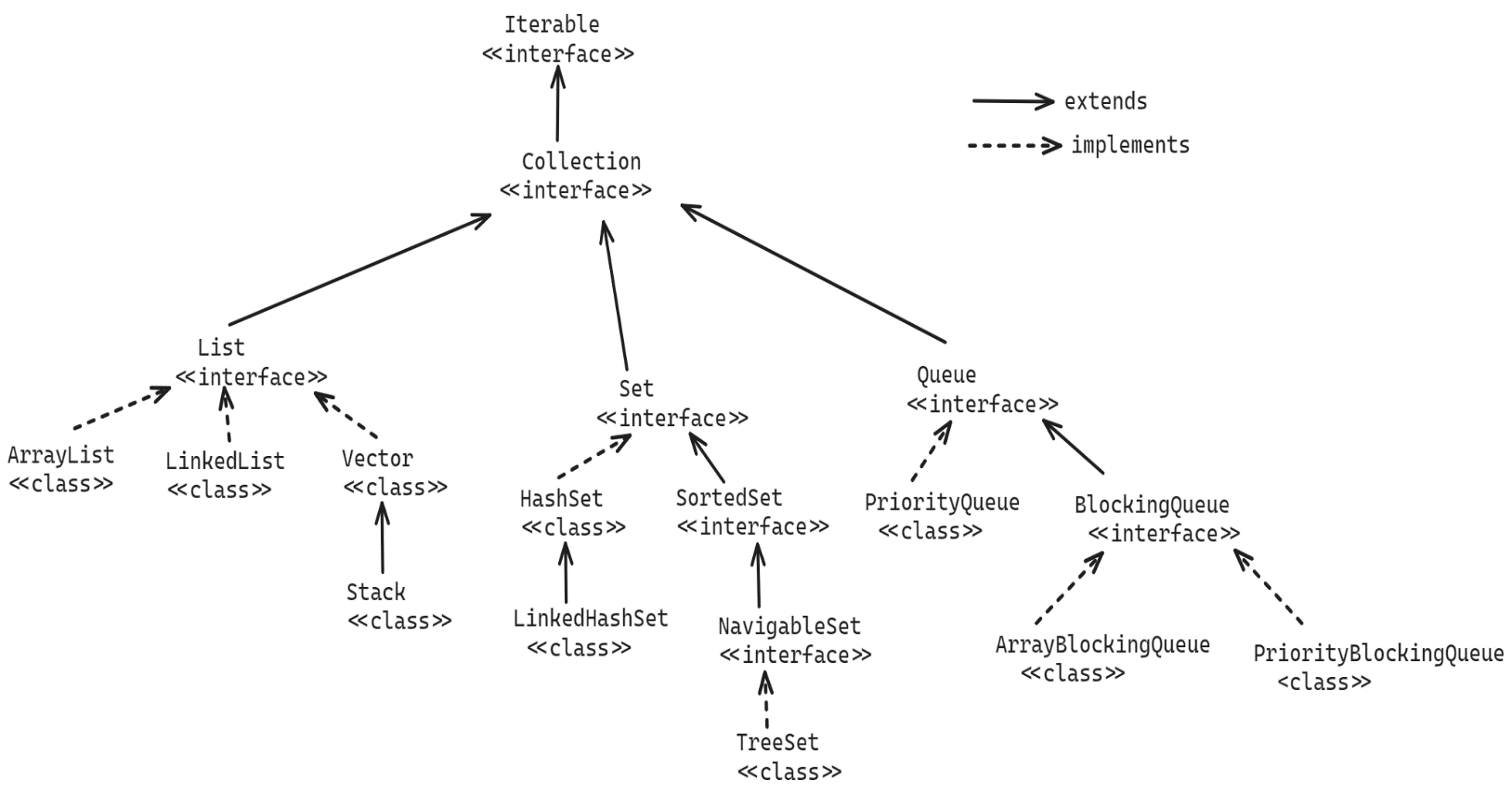
ID --- Primary key, etc.

* While creating generic classes/interfaces/methods, we also use some special patterns, which are called wild card patterns.

Collection Framework

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

* In realtime projects, mostly the data is in objects.
* In realtime projects, we have large number of objects like millions of objects.
* some thing is essential to store, manage and manipulate these groups of objects. So, we use collections.
* To manage the large amounts of objects, different data structure principles are used like LIFO(Last In First Out), FIFO(First In First Out), Linked lists, tree, graph, etc..
* In Java, we have Collection Framework provided, which contains built-in classes and interfaces with the implementation of Data structure principles. So, as a developer, we no need to define the logics manually to implement the data structure principles.
* The java.util.Collection is the main interface in the Java Collection Framework.



Q) what is the difference between an array and a collection?

A) 1. array has a fixed size, but collection has not fixed size.

2. array can store homogeneous elements, but collection can store

heterogeneous elements.

3.array has faster performance, but collection has slow performance.

4. array has basic functionalities, but collection has rich

functionalities.

Methods of Collection interface:

1. add(E e) : Adds an element to the collection. If added successfully then returns true, otherwise returns false.
2. addAll(Collection c): Adds all the elements of the specified collection to this collection.
3. clear(): removes all the elements from this collection.
4. remove(E e) : removes a specified element from this collection. If removed successfully then returns true, otherwise returns false.
5. contains(Object o): checks whether a specified object exists in this collection or not. If exist, returns true. Otherwise, returns false.
6. iterator(): returns an Iterator object over this collection.
7. size(): returns the number of elements in this collection.
8. isEmpty(): returns true, if this collection is empty. Otherwise, returns false.
9. toArray(): returns an array for this collection.
10. removeAll(Collection c): removes the elements from this collection, that are also available in the specified collection.
11. retailAll(Collection c): Retains only the elements in this collection that are contained in the specified collection.

List<E> interface:

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

* When you want to store the elements in a sequence in a collection object then you have to use List object.
* In a List object, each element has position, which is also called index.
* Because the elements are stored in the List object in the order of insertion, we call List as a Ordered collection.
* In a List object, you can store duplicate elements also.
* The implementation classes of List interface are,

ArrayList

LinkedList and

Vector

* The methods of Collection interface are inherited to List interface and also List interface has provided the below key methods.

1. add(index, element): inserts an element at the given position.
2. get(index): returns the element from the given position.
3. indexOf(element): returns the position of the element.
4. lastIndexOf(element): returns the posistion of the last occurrence of the element.
5. remove(index): removes the element from the given position.
6. listIterator() : returns a ListIterator object.
7. set(index, element): replaces an element at the given position.

Traversing a List object:

========================

* Traversing the elements of a list means, visiting or obtaining the each element from a List object.
* Traversing a list could be done in 3 ways.

1. for/for each
2. Iterator
3. ListIterator

* For example, you have an ArrayList class object with some elements of type String and now you want to traverse the list object.

option-1:

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

{

S.o.println(list.get(i));

}

for(String str : list) {

S.o.println(str);

}

option-2:

Iterator<String> it = list.iterator();

while(it.hasNext()) {

S.o.println(it.next());

}

option-3:

ListIterator lit = list.iterator();

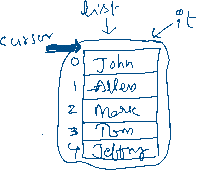
while(lit.hasNext()) {

S.o.println(lit.next());

}

. When a Iterator or ListIterator object is created, they will point to the exisitng collection object, but a copy of the collection object is not created.

. Iterator/ListIterator object maintains a cursor and it is by default positioned at before the first element of the collection object.



. ListIterator extends Iterator and it can move the cursor in forward

and backward direction.

. The methods of Iterator interface are,

1.hasNext()

2.next()

3.remove()

. The methods of ListIterator interface are,

1. hasNext()
2. next()
3. remove()
4. hasPrevious()
5. previous()
6. nextIndex()
7. previousIndex()
8. set(element)

ArrayList<E> class:

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

* It is an array implementation of List interface.
* It uses a resizable-array(dynamic array) to store the elements.
* When you create an ArrayList object by without specifying the capacity then ArrayList object is constructed with initial capacity as 10.
* ArrayList object can grow automatically. For example if 11th element is added to the ArrayList object then its capacity will be incremented by 50%. It means, the capacity becomes 15.
* Suppose, you have created an ArrayList object, by specifying the initial capacity.

ArrayList<Integer> arrList = new ArrayList<>(5);

* Here, when you add 6th element, the capacity will be grown to 7.
* //A sample code to perform ArrayList operations.
* **import** java.util.ArrayList;
* **import** java.util.ListIterator;
* **public** **class** Solution {
* **public** **static** **void** main(String[] args) {
* ArrayList<String> arrList = **new** ArrayList<>();
* arrList.add("Tom");
* arrList.add("John");
* arrList.add("Miller");
* arrList.add("Allen");
* arrList.add("Wilson");
* arrList.add("John"); //adding duplicate element
* arrList.add(**null**); //adding a null element
* arrList.set(2, "Lisa"); //replacing element at index 2
* System.***out***.println("Traversing the elements of the list with for each loop");
* System.***out***.println("=".repeat(40));
* **for**(String str : arrList) {
* System.***out***.println(str);
* }
* System.***out***.println("=".repeat(40));
* System.***out***.println("Traversing the elements of the list with ListIterator from first element to last element");
* ListIterator<String> iterator = arrList.listIterator();
* **while**(iterator.hasNext()) {
* String str = iterator.next();
* System.***out***.println(str);
* }
* System.***out***.println("=".repeat(40));
* System.***out***.println("Traversing the elements of the list with ListIterator from last element to first element");
* **while**(iterator.hasPrevious()) {
* String str = iterator.previous();
* System.***out***.println(str);
* }
* System.***out***.println("=".repeat(40));
* System.***out***.println("The size of the list : " + arrList.size());
* System.***out***.println("=".repeat(40));
* arrList.clear(); //removes all the elements of the list
* System.***out***.println("The size of the list after clear : " + arrList.size());
* }
* }

//Employee.java

**public** **class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** String gender;

**private** **double** experience;

**public** Employee(**int** empno, String ename, **double** sal, String gender, **double** experience) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.gender = gender;

**this**.experience = experience;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", gender=" + gender + ", experience="

+ experience + "]";

}

}

Solution.java

//A sample code to perform ArrayList operations.

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**public** **class** Solution {

**private** **static** List<Employee> getEmployees() {

ArrayList<Employee> lstEmp = **new** ArrayList<>();

lstEmp.add(**new** Employee(7221, "King", 5000.0, "Male", 3.2));

lstEmp.add(**new** Employee(7101, "Miller", 9000.0, "Male", 5.5));

lstEmp.add(**new** Employee(7809, "Lisa", 8000.0, "Female", 4.2));

lstEmp.add(**new** Employee(7176, "Edward", 14000.0, "Male", 8.9));

lstEmp.add(**new** Employee(7432, "Mary", 12000.0, "Female", 6.7));

lstEmp.add(**new** Employee(7788, "Scott", 25000.0, "Male", 11.6));

lstEmp.add(**new** Employee(7054, "Jenny", 3000.0, "Female", 2.3));

lstEmp.add(**new** Employee(7530, "Kathey", 7000.0, "Female", 3.9));

lstEmp.add(**new** Employee(7834, "Wilson", 20000.0, "Male", 9.4));

lstEmp.add(**new** Employee(7337, "Rose", 6000.0, "Female", 4.0));

lstEmp.add(**new** Employee(7838, "Mark", 4000.0, "Male", 2.1));

**return** lstEmp;

}

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

List<Employee> lstEmployees = *getEmployees*();

System.***out***.println("Female employees data");

System.***out***.println("\*".repeat(40));

Iterator<Employee> iterator = lstEmployees.iterator();

**while**(iterator.hasNext()) {

Employee e = iterator.next();

**if**(e.getGender().equalsIgnoreCase("FEMALE"))

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

}

System.***out***.println("\*".repeat(40));

System.***out***.println("Employees with experience > 5 years");

System.***out***.println("\*".repeat(40));

**for**(Employee e : lstEmployees)

{

**if**(e.getExperience() > 5)

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

}

}

}

Sorting the elements of an ArrayList object:

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

* To sort the elements of any List object, we have two ways.

1. by calling sort() method of List interface, which is newly added in Java 8 version.
2. by calling sort() method of java.util.Collections class.

* Collections class is provided in java.util package contains all static methods, which are utility methods for different type of Collection objects.
* To sort the elements using Collections class, we need to call sort() method with List parameter or with List and Comparator parameters.

sort(List<E> list)

sort(List<E> list, Comparator<E> c)

* To sort the elements using List interface method, we need to provide Comparator object parameter.

sort(Comparator<E> c)

* Collections.sort(arrayList); 🡪 Sorts the specified list into ascending order, according to the [natural ordering](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html) of its elements. All elements in the list must implement the [Comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html) interface.
* Comparable is a fuctional interface, which has a single abstract method called compareTo().
* Suppose, all elements in the list are integers or double values or strings, etc.. then they are automatically sorted in ascending order.
* Suppose, all elements in the list are custom object like Employee objects or Customer objects or Product objects,etc.. then the class has to implement Comparable interface.
* The compareTo() method of Comparable interface returns an int value.
* int compareTo([T](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html) o)
* Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
* Suppose, you want to sort the employees of a list in employee no’s ascending order, then you have to modify the Employee class like below.

public class Employee implements Comparable<Employee>

{

-----

-----

@Override

public int CompareTo(Employee o) {

return this.empno – o.empno;

}

}

* Suppose, you want to sort the employees of a list in the employee name’s order, then you have to modify the Employee class like below.

public class Employee implements Comparable<Employee>

{

----

----

@Override

public int compareTo(Employee o) {

return this.ename.compareTo(o.ename);

}

}

Note: Collections.sort() method internally uses merge sort algorithm.

//The below example is for sorting the elements of a list in asceding order of employee salaries.

**public** **class** Employee **implements** Comparable<Employee>{

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** String gender;

**private** **double** experience;

**public** Employee(**int** empno, String ename, **double** sal, String gender, **double** experience) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.gender = gender;

**this**.experience = experience;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", gender=" + gender + ", experience="

+ experience + "]";

}

@Override

**public** **int** compareTo(Employee o) {

**return** (**int**)(**this**.sal - o.sal);

}

}

//Solution.java

//A sample code to perform ArrayList operations.

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**public** **class** Solution {

**private** **static** List<Employee> getEmployees() {

ArrayList<Employee> lstEmp = **new** ArrayList<>();

lstEmp.add(**new** Employee(7221, "King", 5000.0, "Male", 3.2));

lstEmp.add(**new** Employee(7101, "Miller", 9000.0, "Male", 5.5));

lstEmp.add(**new** Employee(7809, "Lisa", 8000.0, "Female", 4.2));

lstEmp.add(**new** Employee(7176, "Edward", 14000.0, "Male", 8.9));

lstEmp.add(**new** Employee(7432, "Mary", 12000.0, "Female", 6.7));

lstEmp.add(**new** Employee(7788, "Scott", 25000.0, "Male", 11.6));

lstEmp.add(**new** Employee(7054, "Jenny", 3000.0, "Female", 2.3));

lstEmp.add(**new** Employee(7530, "Kathey", 7000.0, "Female", 3.9));

lstEmp.add(**new** Employee(7834, "Wilson", 20000.0, "Male", 9.4));

lstEmp.add(**new** Employee(7337, "Rose", 6000.0, "Female", 4.0));

lstEmp.add(**new** Employee(7838, "Mark", 4000.0, "Male", 2.1));

**return** lstEmp;

}

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

List<Employee> lstEmployees = *getEmployees*();

System.***out***.println("List of employees before sorting : ");

System.***out***.println("\*=".repeat(20));

**for**(Employee e : lstEmployees) {

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

}

System.***out***.println("\*=".repeat(20));

Collections.*sort*(lstEmployees);

System.***out***.println("List of employees after sorting on sal's : ");

System.***out***.println("\*=".repeat(20));

**for**(Employee e : lstEmployees) {

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

}

}

}

Comparator<E> interface:

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

* With Comparable interface, we can sort the elements of a list only on a single property/attribute/field.
* Suppose, In a single application, if we have a requirement to sort the list of employees in empno’s order and also in sal’s order then it is not possible.
* So, we can use Comparator<E> interface as a solution.
* Comparator<E> is a functional interface which has a single abstract method called compare() method.

int compare(T o1, T o2)

* Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.
* Suppose, if you want to sort the list of employees in empno ascending order then you have to create a class like below.
* **public** **class** EmpnoAscComparator **implements** Comparator<Employee> {
* @Override
* **public** **int** compare(Employee o1, Employee o2) {
* **return** o1.getEmpno() - o2.getEmpno();
* }
* }
* While calling Collections.sort() method, provide two parameters, one is list and the other is Comparator object.

Collections.sort(lstEmployees, new EmpnoAscComparator());

* suppose, if you want to sort the list of employees in sal’s descending order, then you have to create a class like below.
* **public** **class** SalDescComparator **implements** Comparator<Employee> {
* @Override
* **public** **int** compare(Employee o1, Employee o2) {
* **if**(o2.getSal() < o1.getSal())
* **return** -1;
* **else** **if**(o2.getSal() > o1.getSal())
* **return** 1;
* **else**
* **return** 0;
* }
* }
* You have to call sort() method like below.

Collections.sort(lstEmployees, new SalDescComparator());

Q) what is the difference between Comparable and Comparator interfaces?

A) 1. Comparable interface is from java.lang package and Comparator interface is from java.util package.

2. Comparable interface method is compareTo() and Comparator interface method is compare().

3. For Comparable, the logic should be defined in the same class, whose objects you want to sort. For Comparator, the logic should be defined in a separate class.

4. With Comparable, we can define a single sorting strategy. But with Comparator, we can define mulitple sorting strategies.

Q) is ArrayList object a thread-safe?

A) No, ArrayList object is not a thread-safe object.

Q) can we make ArrayList object as thread-safe?

A) Yes, we have to call synchronizedList() method of Collections class.

List<Employee> newList = Collections.synchronizedList(list);

Q) is ArrayList object a mutable?

A) Yes, bydefault, ArrayList object is mutable. So, you can perform add/update/delete operations on it.

Q) can we make ArrayList object as immutable?

A) Yes, we have to call unmodifiableList() method of Collections class.

lstEmployees = Collections.unmodifiableList(lstEmployees);

Q) can we copy the elemeents of one ArrayList object to another?

A) Yes, we have to call copy() of Collections class.

Collection.copy(destList, srcList);

======================================================================

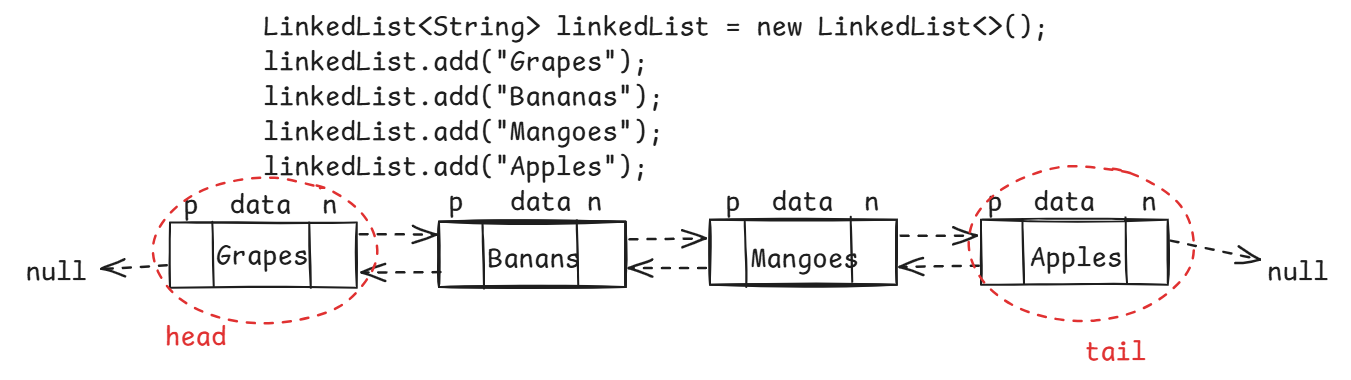
LinkedList class

* LinkedList is a class which implements List interface.
* In LinkedList, each element is stored as a node. So, a LinkedList is a collection of nodes.
* Each node contains 3 parts.

1. previous link/pointer
2. data
3. next link/pointer

* The element which is added to the LinkedList object will be stored as data in the node.
* The previous link holds the address of the previous node and the next link points to the address of the next node.
* The first node is called head/front and last node is called the tail/rear node.

For example,



* When frequent add/remove of elements is required then use LinkedList class object. Because these operations are faster in LinkedList object.
* When frequent data retrieval operation is required then use ArrayList class object.
* General examples of LinkedList are, Browser history and a Music Player play list.
* The additional operations on a LinkedList object are,

1. addFirst(E e)
2. addLast(E e)
3. getFirst()
4. getLast()
5. peekFirst()
6. peekLast()
7. pollFirst()
8. pollLast()
9. removeFrist()
10. removeLast()

* The difference between peek, poll and remove operations are,

peek 🡪 retrieves but does not remove the element. If the list is empty, returns null.

poll 🡪 retrieves and removes the element. If the list is empty, returns null.

remove 🡪 retrieves and remvoes the element. If the list is empty, throws NoSuchElementException.

//Main.java

**import** java.util.LinkedList;

**import** java.util.List;

**public** **class** Main {

**private** **static** LinkedList<Employee> getEmployees() {

LinkedList<Employee> lstEmp = **new** LinkedList<>();

lstEmp.add(**new** Employee(7221, "King", 5000.0, "Male", 3.2));

lstEmp.add(**new** Employee(7101, "Miller", 9000.0, "Male", 5.5));

lstEmp.add(**new** Employee(7809, "Lisa", 8000.0, "Female", 4.2));

lstEmp.add(**new** Employee(7176, "Edward", 14000.0, "Male", 8.9));

lstEmp.add(**new** Employee(7432, "Mary", 12000.0, "Female", 6.7));

lstEmp.add(**new** Employee(7788, "Scott", 25000.0, "Male", 11.6));

lstEmp.add(**new** Employee(7054, "Jenny", 3000.0, "Female", 2.3));

lstEmp.add(**new** Employee(7530, "Kathey", 7000.0, "Female", 3.9));

lstEmp.add(**new** Employee(7834, "Wilson", 20000.0, "Male", 9.4));

lstEmp.add(**new** Employee(7337, "Rose", 6000.0, "Female", 4.0));

lstEmp.add(**new** Employee(7838, "Mark", 4000.0, "Male", 2.1));

**return** lstEmp;

}

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

LinkedList<Employee> linkedListEmployees = *getEmployees*();

//read last element and remove it from the linked list

Employee e = linkedListEmployees.pollLast();

System.***out***.println("The last element removed from the linked list : " + e);

//fetch the current last element from the linked list

Employee emp = linkedListEmployees.peekLast();

System.***out***.println("The current last element in the linked list : " + emp);

//fetch the element from index 5

Employee e2 = linkedListEmployees.get(5);

System.***out***.println("The element at index 5 : " + e2);

System.***out***.println("----- clear the linked list ----------");

linkedListEmployees.clear();

System.***out***.println("Trying to get the first element");

**try** {

Employee e3 = linkedListEmployees.getFirst();

System.***out***.println("First element :" + e3);

}

**catch**(Exception ex) {

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

}

}

}

Q) is LinkedList object a thread-safe?

A) No, LinkedList object is not a thread-safe object.

Q) can we make LinkedList object as thread-safe?

A) Yes, we have to call synchronizedList() method of Collections class.

List<Employee> newList = Collections.synchronizedList(list);

Q) is LinkedList object a mutable?

A) Yes, bydefault, LinkedList object is mutable. So, you can perform add/update/delete operations on it.

Q) can we make LinkedList object as immutable?

A) Yes, we have to call unmodifiableList() method of Collections class.

lstEmployees = Collections.unmodifiableList(lstEmployees);

Q) can we copy the elemeents of one LinkedList object to another?

A) Yes, we have to call copy() of Collections class.

Collection.copy(destList, srcList);

toString(), equals() and hashCode() method:

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

toString(): You need to override this method in your class, if you want to display the content of an object as a string with System.out.println() statement.

* If you don’t override this method, then Object class toString() method is called and it returns a string in the below format.

classname@hexadecimal of object hashcode

without overriding toString() method:

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

class ClassA {

private int x;

public ClassA(int x) {

this.x=x;

}

}

public class Main {

p s v m(String[] args) {

ClassA ca = new ClassA(10);

System.out.println(ca);

}

}

output: ClassA@e457dcf6

with overriding toString() method:

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

class ClassA {

private int x;

public ClassA(int x) {

this.x=x;

}

@Override

public String toString() {

return “ClassA[ x = “ + x +” ]”;

}

}

public class Main {

p s v m(String[] args) {

ClassA ca = new ClassA(10);

System.out.println(ca);

}

}

output: ClassA[ x = 10 ]

equals():

* If you want to compare the two objects of your class with their data then you have to override equals() method in your class.
* If you don’t override the equals() method in your class, then Object class equals() method will be called and this method uses == operator for comparing the two objects.
* So, even though the two objects of a class have the same data, but still they are not equal.

without overriding equals():

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

**class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**public** Employee(**int** empno, String ename, **double** sal) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

}

**public** **class** Solution {

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

Employee e1 = **new** Employee(101, "Kumar", 5000.0);

Employee e2 = **new** Employee(101, "Kumar", 5000.0);

**if**(e1.equals(e2)) {

System.***out***.println("Both are equal objects");

}

**else** {

System.***out***.println("Both are not equal objects");

}

}

}

output: Both are not equal objects

with overriding equals():

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

**import** java.util.Objects;

**class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**public** Employee(**int** empno, String ename, **double** sal) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

Employee other = (Employee) obj;

**return** empno == other.empno && Objects.*equals*(ename, other.ename)

&& Double.*doubleToLongBits*(sal) == Double.*doubleToLongBits*(other.sal);

}

}

**public** **class** Solution {

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

Employee e1 = **new** Employee(101, "Kumar", 5000.0);

Employee e2 = **new** Employee(101, "Kumar", 5000.0);

**if**(e1.equals(e2)) {

System.***out***.println("Both are equal objects");

}

**else** {

System.***out***.println("Both are not equal objects");

}

}

}

output: Both are equal objects

hashCode() :

---------

* hash code of an object is an integer value
* If you don’t override hashCode() method in your class, then hashCode() method of Object class will be called.
* The hashCode() method of Object class returns the memory address of an object by converting it into integer value.
* If two objects of a class are equal, it means they have same data, but their hash codes are going to be different.
* We need the hash code as same if they are equal objects. So, we need to override hashCode() method in our class.
* This hash code of an object will play an important role while storing/retrieving the objects from a hashing collections.

without overriding hashCode():

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

**class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**public** Employee(**int** empno, String ename, **double** sal) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

}

**public** **class** Solution {

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

Employee e1 = **new** Employee(101, "Kumar", 5000.0);

Employee e2 = **new** Employee(101, "Kumar", 5000.0);

System.***out***.println(e1.hashCode());

System.***out***.println(e2.hashCode());

}

}

output: 1175962212

918221580

with overriding hashCode():

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

**import** java.util.Objects;

**class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**public** Employee(**int** empno, String ename, **double** sal) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

@Override

**public** **int** hashCode() {

**return** Objects.*hash*(empno, ename, sal);

}

}

**public** **class** Solution {

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

Employee e1 = **new** Employee(101, "Kumar", 5000.0);

Employee e2 = **new** Employee(101, "Kumar", 5000.0);

System.***out***.println(e1.hashCode());

System.***out***.println(e2.hashCode());

}

}

output: -950750608

-950750608

java.util.Set interface

* Set is an unordered collection of elements without duplicates.
* A Set object doesn’t store the duplicate elements. If a duplicate element is added, then it will skip the element.
* It is unordered collection, because it doesn’t store the elements in the same order internally.
* The implementation classes of Set interface are,

HashSet, LinkedHashSet and TreeSet.

* The set objects internally uses map objects, where map objects uses hash table data structure.
* hash table data structure stores the elements in key-value pairs.
* When you add an element to the set object, that element will be stored as a key and with value as PRESENT.
* Here, PRESENT is a dummy object, created for Object class.

HashSet<E> class:

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

* When you create a HashSet class object, it internally creates HashMap class object, to store the elements.
* HashMap class uses hash table data structure and the table is created with default initial capacity as 16 and load factor 0.75.
* While creating HashSet class object, you can also specify the intial capacity and load factor.

HashSet<String> set = new HashSet<>();

. initial capacity : 16

. load factor : 0.75

HashSet<String> set = new HashSet<>(10);

. initial capacity : 10

. load factor : 0.75

HashSet<String> set = new HashSet<>(10, 0.5f);

. initial capacity : 10

. load factor : 0.5

* The capacity of hash table structure will be doubled based on capacity and load factor.
* if capacity is 16 and load factor is 0.75 then the capacity will be doubled after adding 12(16 \* 0.75) pairs to the hash table.
* if the capacity is 10 and load factor is 0.5 then the capacity will be doubled after adding 5( 10 \* 0.5) pairs to the hash table.
* When you are adding an element to a set object, first hashcode is calculated and then based on the hashcode, bucket index is calculated and then the element will be stored into that bucket as a key with value as PRESENT.
* To access the elements of a set object, you have to use Iterator object or for each loop.

//example code

**import** java.util.HashSet;

**import** java.util.Iterator;

**import** java.util.Objects;

**class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**public** Employee(**int** empno, String ename, **double** sal) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

Employee other = (Employee) obj;

**return** empno == other.empno && Objects.*equals*(ename, other.ename)

&& Double.*doubleToLongBits*(sal) == Double.*doubleToLongBits*(other.sal);

}

@Override

**public** **int** hashCode() {

**return** Objects.*hash*(empno, ename, sal);

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + "]";

}

}

**public** **class** MainClass {

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

HashSet<Employee> hashSet = **new** HashSet<>();

hashSet.add(**new** Employee(7110, "Allen", 3000.0));

hashSet.add(**new** Employee(7054, "Kellen", 4000.0));

hashSet.add(**new** Employee(7231, "Border", 8000.0));

hashSet.add(**new** Employee(7110, "Allen", 3000.0));

hashSet.add(**new** Employee(7298, "Miller", 7000.0));

Iterator<Employee> iterator = hashSet.iterator();

**while**(iterator.hasNext()) {

System.***out***.println(iterator.next());

}

}

}

Q) is HashSet object a thread-safe?

A) No, HashSet object is not a thread-safe object.

Q) can we make HashSet object as thread-safe?

A) Yes, we have to call synchronizedSet() method of Collections class.

Set<Employee> newSet = Collections.synchronizedSet(set);

Q) is HashSet object a mutable?

A) Yes, by default, HashSet object is mutable. So, you can perform add/update/delete operations on it.

Q) can we make HashSet object as immutable?

A) Yes, we have to call unmodifiableSet() method of Collections class.

set = Collections.unmodifiableSet(set);

Q) can we copy the elemeents of one HashSet object to another?

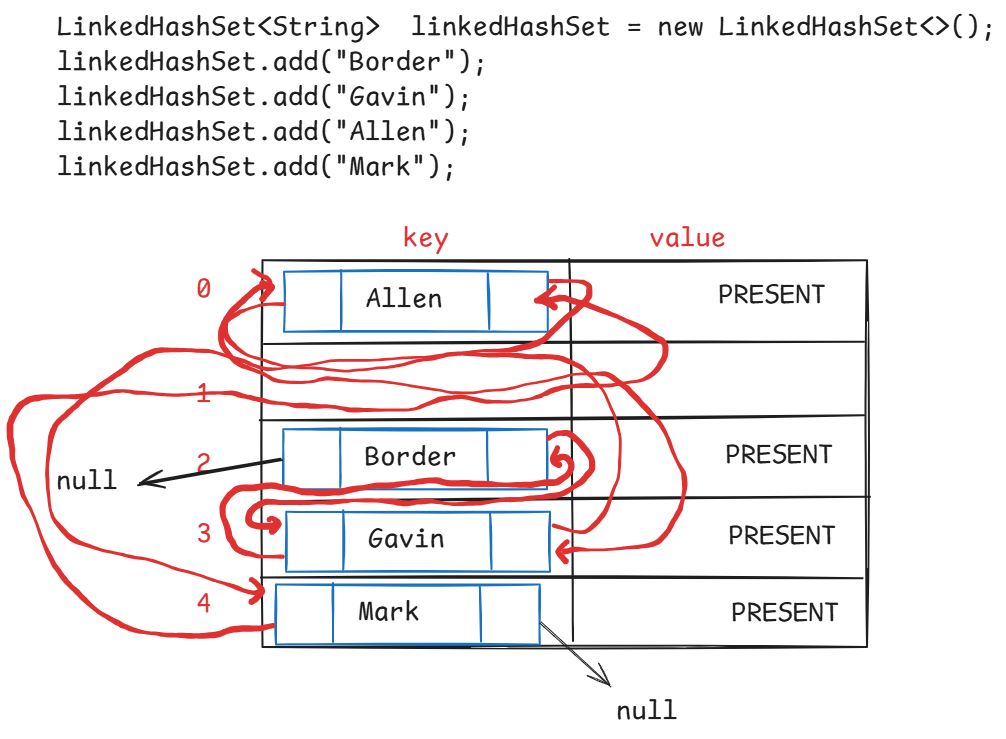
A) Yes, we have to call addAll() method of HashSet class.

set1.addAll(set2);

LinkedHashSet<E> class:

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

Hash table and linked list implementation of the Set interface, with predictable iteration order. This implementation differs from HashSet in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is the order in which elements were inserted into the set (*insertion-order*).



//sample code

**import** java.util.Iterator;

**import** java.util.LinkedHashSet;

**public** **class** Solution {

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

LinkedHashSet<String> linkedHashSet = **new** LinkedHashSet<>();

linkedHashSet.add("Border");

linkedHashSet.add("Miller");

linkedHashSet.add("Mark");

linkedHashSet.add("Allen");

linkedHashSet.add("Scott");

linkedHashSet.add("Miller");

Iterator<String> iterator = linkedHashSet.iterator();

**while**(iterator.hasNext()) {

System.***out***.println(iterator.next());

}

}

}

Q) is LinkedHashSet object a thread-safe?

A) No, LinkedHashSet object is not a thread-safe object.

Q) can we make LinkedHashSet object as thread-safe?

A) Yes, we have to call synchronizedSet() method of Collections class.

Set<Employee> newSet = Collections.synchronizedSet(set);

Q) is LinkedHashSet object a mutable?

A) Yes, by default, HashSet object is mutable. So, you can perform add/update/delete operations on it.

Q) can we make LinkedHashSet object as immutable?

A) Yes, we have to call unmodifiableSet() method of Collections class.

set = Collections.unmodifiableSet(set);

Q) can we copy the elemeents of one LinkedHashSet object to another?

A) Yes, we have to call addAll() method of HashSet class.

set1.addAll(set2);

SortedSet<E> interface methods:

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

1. first() : returns the first element of this set.
2. last() : returns the last element of this set
3. headSet(toElement) : returns the portion of this set from beginning to the toElement, exclusive.
4. tailSet(fromElement): returns the portion of this set from fromElement inclusive, until the last element.
5. subSet(fromElement, toElement): returns the portion of this set, from the fromElement inclusive to toElement exclusive.

NavigableSet<E> interface:

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

1. lower(element) : Returns the greatest element in this set strictly less than the given element, or null if there is no such element.
2. floor(element): Returns the greatest element in this set less than or equal to the given element, or null if there is no such element.
3. higher(element): Returns the least element in this set strictly greater than the given element, or null if there is no such element.
4. ceiling(element): Returns the least element in this set greater than or equal to the given element, or null if there is no such element.
5. pollFirst() : retrieves and removes the first element of this set, returns null if this set is empty.
6. pollLast() : retrieves and removes the last element of this set, returns null if this set is empty.
7. descendingSet(): returns reverse order of this set.

TreeSet<E> class:

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

* TreeSet class implements NavigableSet interface.
* TreeSet object maintains the elements in the natural sorting order, or the order defined with Comparator.
* TreeSet object, internally uses TreeMap object, to store the elements.
* To add the elements of a user-defined class(custom class) to the TreeSet object, the elements must be Comparable elements, to sort them in natural sorting order.
* To make them as comparable elements, the class must implement Comparable interface.
* If not implemented, at runtime, ClassCastException will be thrown.

//example code

import java.util.Objects;

import java.util.TreeSet;

class Employee implements Comparable<Employee> {

private int empno;

private String ename;

private double sal;

public Employee(int empno, String ename, double sal) {

super();

this.empno = empno;

this.ename = ename;

this.sal = sal;

}

public int getEmpno() {

return empno;

}

public void setEmpno(int empno) {

this.empno = empno;

}

public String getEname() {

return ename;

}

public void setEname(String ename) {

this.ename = ename;

}

public double getSal() {

return sal;

}

public void setSal(double sal) {

this.sal = sal;

}

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

Employee other = (Employee) obj;

return empno == other.empno && Objects.equals(ename, other.ename)

&& Double.doubleToLongBits(sal) == Double.doubleToLongBits(other.sal);

}

@Override

public int hashCode() {

return Objects.hash(empno, ename, sal);

}

@Override

public String toString() {

return "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + "]";

}

@Override

public int compareTo(Employee o) {

return this.getEmpno() - o.getEmpno();

}

}

public class Solution {

public static void main(String[] args) {

TreeSet<Employee> treeSet = new TreeSet<>();

treeSet.add(new Employee(7110, "Allen", 3000.0));

treeSet.add(new Employee(7054, "Kellen", 4000.0));

treeSet.add(new Employee(7231, "Border", 8000.0));

treeSet.add(new Employee(7110, "Allen", 3000.0));

treeSet.add(new Employee(7298, "Miller", 7000.0));

for(Employee e : treeSet) {

System.out.println(e);

}

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

System.out.println("First element : " + treeSet.first());

System.out.println("Last element : " + treeSet.last());

System.out.println("Lower element of empno : 7231 " + treeSet.lower(new Employee(7231, "Border", 8000.0)));

}

}

Q) is TreeSet object a thread-safe?

A) No, TreeSet object is not a thread-safe object.

Q) can we make TreeSet object as thread-safe?

A) Yes, we have to call synchronizedSet() method of Collections class.

Set<Employee> newSet = Collections.synchronizedSet(set);

Q) is TreeSet object a mutable?

A) Yes, by default, HashSet object is mutable. So, you can perform add/update/delete operations on it.

Q) can we make TreeSet object as immutable?

A) Yes, we have to call unmodifiableSet() method of Collections class.

set = Collections.unmodifiableSet(set);

Q) can we copy the elemeents of one TreeSet object to another?

A) Yes, we have to call addAll() method of HashSet class.

set1.addAll(set2);

Queue<E> interface:

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

* Queue orders elements in First In First Out(FIFO) manner, but not necessarily.
* A Queue represents a collection designed for holding the elements prior to processing.
* UseCase1: In Operating Systems, processes are added to the Queue as they arrive. The CPU picks a process from the front of the queue and executes it.
* UseCase2: In a Customer Service call center, incoming calls are placed in a queue. The first call is attended first.
* UseCase3: Employees submit print jobs, they are stored in a queue. The printer processes jobs one by one from the front of the queue.
* A queue maintains two pointers called head/fornt and tail/rear.
* The elements are inserted from the tail pointer and removed from the head pointer.
* The methods for inserting the element into the Queue are,

1. add(E e) : adds an element and returns true, if successful. Otherwise, throws IllegalStateException.
2. offer(E e): adds an element and returns true, if successful. Otherwise, returns false.

* The methods for removing element from the Queue are,

1. poll() : retrieves and removes the element from the head pointer. Returns null if the queue is empty.
2. remove(): retrieves and removes the element from the head pointer. Throws NoSuchElementException, if the queue is empty.

* The methods for retrieving the element from the Queue are,

1. element() : retrieves but does not remove the element from the head pointer. Throws NoSuchElementException, if the queue is empty.
2. peek() : retrieves but does not remove the element from the head pointer. Returns null, if the queue is empty.

Deque<E> interface:

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

* Deque<E> interface extends Quey<E> interface, and it is called double ended queue.
* It means, a Deque object allows addition/deletion/retrieval of elements from both head and tail pointers.
* The methods are,

1. addFirst(e)
2. addLast(e)
3. offerFirst(e)
4. offerLast(e)
5. getFirst()
6. getLast()
7. removeFirst()
8. removeLast()
9. pollFirst()
10. pollLast()
11. peekFirst()
12. peekLast()

* PriorityQueue class implements Queue interface, and LinkedList class implements Deque interface.

PriorityQueue<E> class:

* PriorityQueue class does not maintain the elements in the FIFO order. It maintains the elements in the natural sorting order or the order defined by the Comparator object.
* The default capacity of the PriorityQueue object is 11.
* The PriorityQueue object is an unbounded queue, which means, there is no size restrictions. We can add any number of elements.
* The elements must be Comparable elements, for sorting in natural sorting order.
* While constructing PriorityQueue object, if we pass Comparator object as a parameter then the elements may not be Comparable elements.
* we can not add null value to PriorityQueue object. If added, we will get NullPointerException.

//Example on PriorityQueue

**import** java.util.Iterator;

**import** java.util.PriorityQueue;

**class** Employee **implements** Comparable<Employee> {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**public** Employee(**int** empno, String ename, **double** sal) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + "]";

}

@Override

**public** **int** compareTo(Employee o) {

**return** **this**.getEmpno() - o.getEmpno();

}

}

**public** **class** MainClass {

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

PriorityQueue<Employee> priorityQueue = **new** PriorityQueue<>();

priorityQueue.add(**new** Employee(7110, "Allen", 3000.0));

priorityQueue.add(**new** Employee(7054, "Kellen", 4000.0));

priorityQueue.add(**new** Employee(7231, "Border", 8000.0));

priorityQueue.add(**new** Employee(7110, "Allen", 3000.0));

priorityQueue.add(**new** Employee(7298, "Miller", 7000.0));

System.***out***.println("The element at head pointer : " + priorityQueue.peek());

System.***out***.println("Elements of PriorityQueue :");

Iterator<Employee> iterator = priorityQueue.iterator();

**while**(iterator.hasNext()) {

System.***out***.println(priorityQueue.poll());

}

}

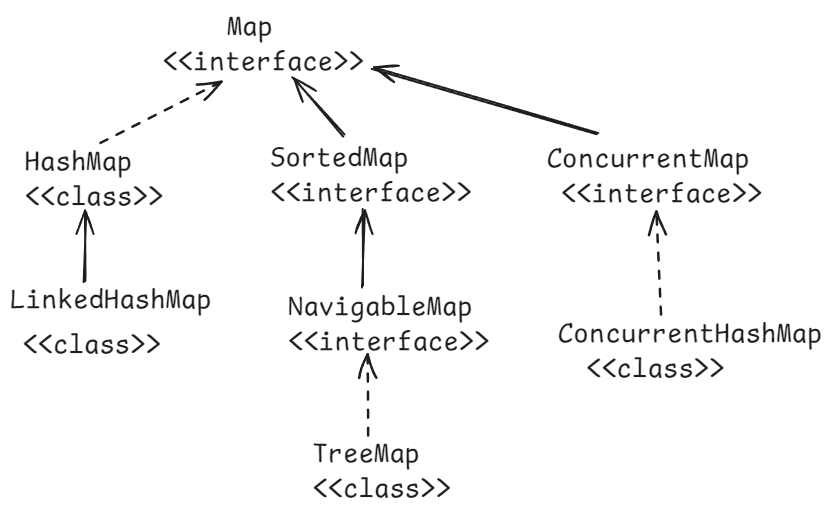
}

BlockingQueue<E> interface:

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

* It is a queue, which additionally supports blocking operations.
* A blocking queue will have a fixed capacity.
* If a thread wants to add an element, but the queue is full, then the thread has to wait until the space becomes available.
* If a thread wants to remove an element, but the queue is empty, then the thread has to wait until the queue becomes non-empty.
* The blocking operations are put() and take().
* When put() is called, if the queue is full, put() will wait for the space.
* when take() is called, if the queue is empty, take() will wait for the element.
* The implementation classes are, ArrayBlockingQueue and LinkedBlockingQueue.
* For creating producer-consumer applications, producer adds items to the queue and consumer remvoes items from the queue.
* If the queue is full, producer waits until space becomes available. If the queue is empty, consumer waits until item becomes available.
* For example, In a Job execution application, one thread is adding the job to the queue and the other thread is removing the job from the queue and executing it.
* If the queue if full, the first thread waits to add the job, until space becomes available. If the queue is empty, the other threads waits until a job is added.
* **import** java.util.concurrent.ArrayBlockingQueue;
* **class** MyThread **extends** Thread {
* **private** ArrayBlockingQueue<Integer> abq;
* **public** MyThread(ArrayBlockingQueue<Integer> abq) {
* **this**.abq = abq;
* }
* @Override
* **public** **void** run() {
* **try** {
* **for**(**int** i = 1; i <= 6; i++) {
* **if**(i == 6) {
* System.***out***.println("Element taken out from queue : " + abq.take());
* }
* abq.put(i);
* System.***out***.println("Element put into the queue : " + i);
* }
* }
* **catch**(Exception ex) {
* ex.printStackTrace();
* }
* }
* }
* **public** **class** Solution {
* **public** **static** **void** main(String[] args) {
* ArrayBlockingQueue<Integer> abq = **new** ArrayBlockingQueue<Integer>(5);
* MyThread myThread = **new** MyThread(abq);
* myThread.start();
* }
* }

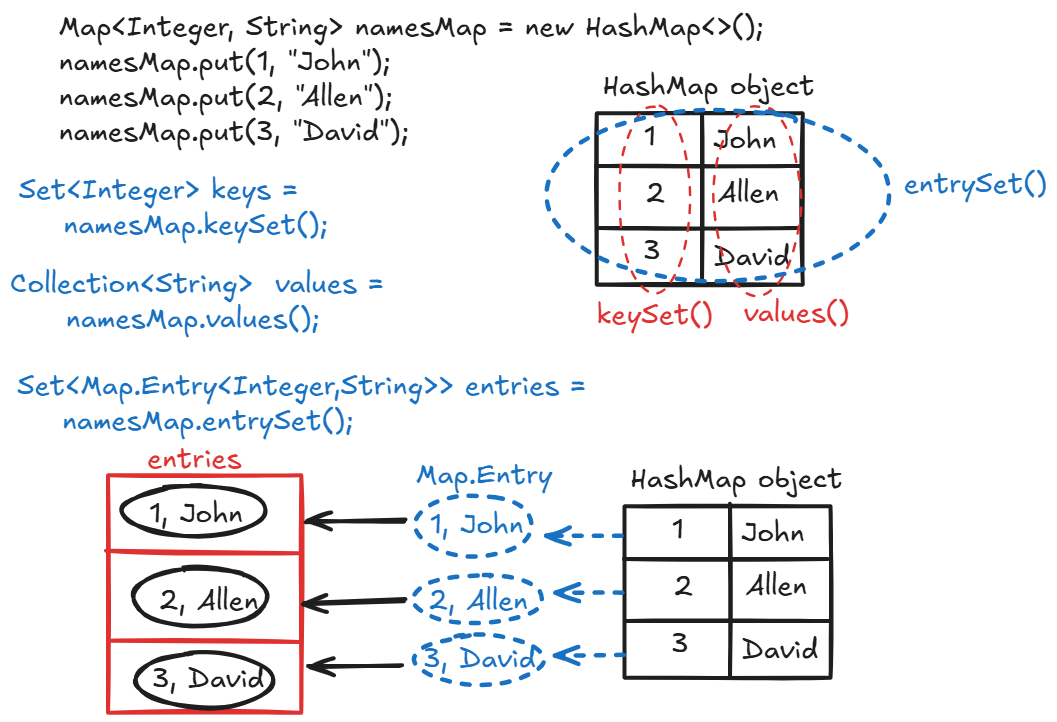
Map<E> interface:



* Map is a separate collection type in Java, given to store the elements in the form of key-value pairs.
* In a Map object, key can’t be duplicated, but value can be duplicated.
* we can insert one null key, and multiple null values into a Map object.

methods of Map interface:

* + 1. put(key,value): inserts the key-value to the map and returns the value.
    2. get(key): retruns the value of the specified key. Returns null if the key doesn’t exist.
    3. remove(key): removes the mapping from this map object for the key, and returns the value. If the map is empty, then returns null.
    4. replace(key,value): replaces the value for a key in this map, if the key exists. Otherwise returns null.
    5. containsKey(key): returns true, if this map object contains the specified key. Otherwise, returns false.
    6. containsValue(value): returns true, if this map object contains the specified value. Otherwise, returns false.
    7. keySet() : returns keys of this map object into a Set. Returns null if this map object is empty.
    8. values(): returns values of this map object into a set. Returns null if this map object is empty.
    9. entrySet(): returns the map entries as Map.Entry objects into a set. Returns null, if the map object is empty.
    10. size() : returns the number of mappings in this map object.
    11. clear(): removes all the mappings from this map object.
    12. isEmpty(): returns true, if this map is empty. Otherwise, returns false.
    13. putIfAbsent(key,value): puts this key-value, if the key is absent in this map object. If the key already exist, then returns its current value. But does not replace the value.



//Example on HashMap

**import** java.util.Collection;

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Set;

**public** **class** Main {

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

HashMap<String, String> hashMap = **new** HashMap<>();

hashMap.put("Manish", "A+");

hashMap.put("Anirudh", "A");

hashMap.put("Madhu", "B+");

hashMap.put("Sagar", "B");

hashMap.put("Lisa", "C");

System.***out***.println("Displaying the keys of the HashMap");

Set<String> keys = hashMap.keySet();

**for**(String key : keys) {

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

}

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

System.***out***.println("Displaying the values of the HashMap");

Collection<String> values = hashMap.values();

**for**(String value : values) {

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

}

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

System.***out***.println("Displaying the entries of the HashMap");

Set<Map.Entry<String, String>> entries = hashMap.entrySet();

**for**(Map.Entry<String, String> entry : entries) {

System.***out***.println(entry.getKey() + " ----> " + entry.getValue());

}

}

}

Q) is HashMap object a thread-safe?

A) No, HashMap object is not a thread-safe object.

Q) can we make HashMap object as thread-safe?

A) Yes, we have to call synchronizedMap() method of Collections class.

Map<Integer,Employee> newMap = Collections.synchronizedMap(map);

Q) is HashMap object a mutable?

A) Yes, by default, HashMap object is mutable. So, you can perform add/update/delete operations on it.

Q) can we make HashMap object as immutable?

A) Yes, we have to call unmodifiableMap() method of Collections class.

map = Collections.unmodifiableMap(map);

Q) can we copy the elemeents of one HashMap object to another?

A) Yes, we have to call putAll() method of HashMap class.

map1.putAll(map2);