Java Full Stack Development program



Trainer: Shekher

Working Professional

Duration : 6 Months

Fee: 30k

online + offline classes - class videos ( 1 year access)

- material - Job assistance

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

Full Stack Development = Front-end + Back-end + Database + Tools(DevOps)

Front-end developer : A programmer/developer who creates web pages

to allow the users to iteract with the business/website.

Front-end technologies : HTML + CSS + JavaScript + Bootstrap + Angular/React JS

Back-end developer : A programmer/developer who creates programs to connect

with a database, collects the data from database and

provides the data to the web pages.

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

Database : It is a storage area, whether all the information

about a business/products is stored.

\* A Full stack developer will develop web pages for front-end, will develop

functionalities for that business, and also maintains the data in Database.

\* A full stack developer can be fit for the job roles like

front-end developer, api developer, database developer,

back-end developer or fullstack developer.

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

Back-end development

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

\* You are going to learn,

Java,

Database(oracle/mysql)

DevOps tools

\* If you consider Java, you are going to learn,

1. Core Java( Java SE)

2. Advanced Java (Java EE)

3. Frameworks ( Hibernate, Spring and Spring Boot)

4. Microservices

5. Projects

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

Java Modules:

1. Java SE (Standard Edition)

2. Java EE (Enterprise Edition)

3. Java ME (Micro Edition)

\* Java SE module is also called as Core Java. It is used to

develop stand-alone/desktop applications.

\* For example, Calculator App, Notepad, Eclipse, MS-Excel, ... are

stand-alone applications.

\* Java EE module is used for developing web & enterprise applications.

\* Enterprise applications are nothing but large business applications.

\* For example, banking applications, telecom applications, irctc, etc..

are enterprise applications.

\* Java ME module is used for developing applications for Small scale devices

like mobiles, set-top boxes, etc..

\* Java back-end programmers have to learn Java SE module and Java EE module.

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

Programming language?

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

\* language - is used to convey something.

\* languages are 2 categories.

1. Natural language -- we can convey any information.

ex: English, Hindi, Telugu,.....

2. Formal language -- we can convey specific information.

ex: Accounts, programming languages, ....

\* Computer/machine can understand only binary language i.e., 0's and 1's.

\* But we can't develop the real-time applications in binary language.

\* So, we use high-level programming languages like Java, C#, Python, C, C++,...

to develop the real-time applications.

\* High-level programming languages are nothing but these languages statements

looks like english sentences.

\* There are translators(compilers & interpreters) to convert/translate high-level

language to binary language.

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

How to learn a programming language?

english: paragraph --- sentences --- words

+grammer

programming language : program --- statements --- tokens

+ syntax

\* In any programming language, there are 5 types of tokens only.

1. keywords

2. identifiers

3. literals

4. operators

5. strings

\* In any programming language, there are 5 types of statements only.

1. input statement

2. output statement

3. memory statement

4. arithmetic and logic statement

5. control statement

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

Core JAVA

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

\* In 1991, a programming language was developed called OAK.

\* In 1995, OAK was renamed to JAVA.

\* The software name of JAVA is JDK.(Java development kit).

\* Initially, JAVA is from Sun Micro systems.

\* In 2010, Oracle Corporation company has acquired Sun micro systems.

\* So, today, Java is from Oracle Corporation.

\* We call "James Gosling" as a father of Java.

Java Versions:

JDK1.0 ---- 1996

.

.

J2SE 5.0 ---- 2004

.

.

Java SE8 ---- 2014

.

.

Java SE11 ---- 2018

.

.

Java SE17 ---- 2021

.

.

Java SE22 ---- 2024 (current version)

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

Features of Java:

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

1. platform independent:

\* platform refers to operating system.

\* Let say, you have written a c-program on windows and compiled it.

\* Now, I would like to run the program on Linux operating system.

I will get an error.

\* The reason is, when you compile c-program on windows, the binary code

will be generated and it will only execute on windows OS only.

\* So, we call C and C++ programming languages as platform dependent.

\* Let say, you have written a Java program on windows and compiled it.

\* Now, I would like to run the program on Linux operationg system.

The program is executed.

\* The reason is, when you compile Java-program on windows, the byte code

will be generated and it will execute any operating system.

So, finally because of bye code in Java, we call Java as a platform independent

Programming language.

\* Java code follows a principle "Write Once Run Anywhere(WORA)".

2. Robust:

\* Java is a Robust programming language, because of two reasons.

1. memory managment

2. handling runtime errors.

\* Java has a Garbage Collector, which removes un-used memory automatically. So

the memory is managed efficiently.

\* Java has exception handling mechanism, with this we can handle the runtime

errors smoothly.

3. multithreaded:

\* Java is a multithreaded programming language. Because, Java allows a

program to perform multiple tasks simultaneously.

\* For example, In Ms-Word, when you type a paragraph, it will display the

content what you are typing and simultaneously it will check the spelling

mistakes. It means, Ms-Word is performing multiple tasks simultaneously.

This is called multi-threading.

4. Architecture Neutral:

\* Java is Architecture Neutral, because a program written on one processor

can execute on another processor also.

\* for example, if you write a java program on i5 processor then you can run

the program on i3 or i7 or silicon, etc.. processor also.

5. secured:

\* Java programs are executed in JVM(Java Virtual Machine).

\* In JVM, there is a component called byte code verifier.

\* When ever, byte code is going to execute, this verifier will check

whether it is modified or hacked by some one or not.

\* If modified then it won't allow the program to execute.

\* So, in this way, Java is a very secured programming language.

6. Object Oriented:

\* Java is an object oriented programming language.

\* Because, it allows the developers to implement the principles of Object Oriented Programming System(OOPS).

\* The pinciples of OOPs are,

1. abstraction

2. encapsulation

3. inheritance

4. polymorphism

7. Simple:

\* The syntax of writing the statements is very concise and

the statements are like english statements. That's why

we call Java is a simple programming.

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

JDK & JRE :

----------

\* JDK : Java Development Kit

\* JRE : Java Runtime Environment

\* JDK provides tools to compile and execute the java programs.

\* JRE provides JVM(Java Virtual Machine) and also some libraries

to run the program.

\* Every programming language provides some pre-defined programs to

develop the programs by the developers.

\* These pre-defined programs given by a language is called a library.

\* Java has provided a library, which is called class libarary to

develop the programs by the developers.

\* Before Java 8 version, when we install Java software, two folders are created.

1. JDK

2. JRE

\* From Java 8 version, when we install Java software, we will get only one

folder JDK and it in-built contains JRE.

installing jdk-17:

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

1: visit https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html

2: click on the link for Windows 64 MSI installer

3: jdk-17.0.11\_windows-x64\_bin.msi file is downloaded.

4: double click on the downloaded file, click on next buttons then finally close button.

5: Now Java software is installed at the below location.

C:\Program Files\Java\jdk-17

Path setting:

1: Goto, windows search, then type environment

2: select the option Edit system environment variables

3: click on environment variables button

4. Under system variables section, select Path variable and then click on Edit button.

5: next click on New button, and enter the value C:\Program Files\Java\jdk-17\bin

6: move up this java path to the top.

7: ok --> ok ---> ok.

How to verify the Java path?

1: open command prompt

2: type the command

java -version

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

structure of a Java program:

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

package statement;

import statement(s);

class <classname> {

variables;

methods;

}

\* package statement must be the first statement in a java program.

\* writing the package statement in a java program is optional.

\* maximum we can write only one package statement in a java program.

\* a java program can have zero or more import statements.

\* import statements must be after the package statement and before the class statement.

\* every java program must contain at least one class.

\* without a class, we can not write a java program.

Where we can we write a Java program?

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

1. Text editors (ex: Notepad, Notepad++, EditPlus, Sublime, etc..)

2. IDE's ( Integrated Development Environment)

\* IDE's are the softwares which helps the developers to write the programs easily.

ex: Eclipse, IntelliJ, STS(Spring Tool Suite), Visual Studio Code(VS Code), etc..

main() method in Java:

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

\* A java program compilation starts from line1 of the program.

\* A java program execution starts from the main() method.

\* JVM calls main() method to execute the program.

\* The syntax of writing the main() method in Java is,

public static void main(String[] args)

1 2 3 4 5

1: public : it is access modifier(which is a keyword)

2: static : it is non-access modifier( which is a keyword)

3: void : it is the return type of main method

4: main : it is the method name

5: String[] args : it is the parameter of the main method.

output statement in Java:

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

\* output statement in java is,

System.out.println();

Note: Java is a case-sensitive programming language.

It means, there is a difference in upper case letter and lower case letter.

Writing the first program to display a welcome message:

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

step-1: open notepad

step-2: enter the below java code

class HelloWorld

{

public static void main(String[] args)

{

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

System.out.println("Welcome to AshokIT");

}

}

step-3: save the program

File --> Save As ---> open D: drive --> open 9AM folder

change the save as type : All files

file name : HelloWorld.java

step-4: Goto D:\9AM folder and type cmd in the source bar

A new command prompt is opened.

D:\9AM> javac HelloWorld.java

step-5: execute the java program.

D:\9AM> java HelloWorld

Hello!

Welcome to AshokIT

Q) Can we execute a java program without compiling the program again?

A) Yes.

Q) If i want to execute a java program for 5 times, how many times i need to compile the program?

A) one time.

Q) If i have modified the code in my java program, do i need to compile the program again?

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

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

writing a second program in Java;

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

1) open notepad

2) enter the below code

class Addition

{

public static void main(String[] args)

{

int a = 10, b = 20;

int c = a + b;

System.out.println( "addition = " + c );

}

}

3) save the program

file --> save as --> open D:\9AM folder

change save as type: All files

file name: Addition.java

save

4) open command prompt

D:\9AM> javac Addition.java

5) execute the program.

D:\9AM> java Addition

addition = 30

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

Keywords and identifiers

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

keyword:

\* It is a token/word which has a pre-defined meaning in the programming language.

\* Java has around 55 keywords.

\* ex: class, interface, extends, implements, for, while, break, if, else, switch, case, .....

identifier:

\* identifiers are the names given to the variables, methods, classes, interfaces and packages.

ex:

class InsurancePolicy {

private int policyId;

private String policyHolderName;

public void displayPolicy() {

}

}

\* In the above example,

keywords are : class, private, int, String, public, void

identifiers are: InsurancePolicy, policyId, policyHolderName, displayPolicy

\* The rules to write the identifiers are,

1. identifier can have letters from a-z(both lowercase or uppercase), digits(0-9),

underscore(\_), or dollar($).

2. identifier should not start with a digit.

3. identifier is case-sensitive. Uppercase letter and lowercase letters are different.

4. Don't use a keyword as an identifier.

5. Don't use a whitespace in the identifier.

examples:

1. myVar //correct

2. My\_Var //correct

3. My Var //wrong

4. \_myVar //correct

5. my$var //correct

6. $\_myVar //correct

7. My$Var\_ //correct

8. 9MyVar //wrong

9. \_9MyVar //correct

10. My\_Var$9 //correct

11. @myVar //wrong

12. my@Var //wrong

13. my\_var@9 //wrong

memory statement in java:

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

\* Memory statement tells the JVM, about how much memory to be allocated and what is

the name of the memory location.

\* The syntax of a memory statement is,

datatype variablename;

(or)

datatype variablename = value;

\* ex1:

int a = 50;

\* This memory statement, tells the JVM that

allocate 4 bytes of memory block, give the name

to it as a and store the value in that memory as 50.

ex2:

int b = a;

\* This memory statenent tells the JVM that

allocate 4 bytes of memory block, give the name

to it as b, and fetch the value of memory block a

and store it in memory block b.

ex3:

System.out.println(a + b);

\* This statement tells the JVM that fetch the values

of memory blocks a and b, add the values and

display the output.

ex4:

int a = 20, b = 30; //correct

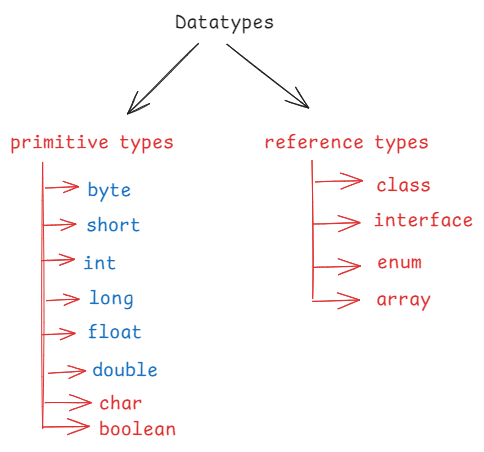
int a = 20, int b = 30; //wrong

int a = 20; int b = 30; //correct

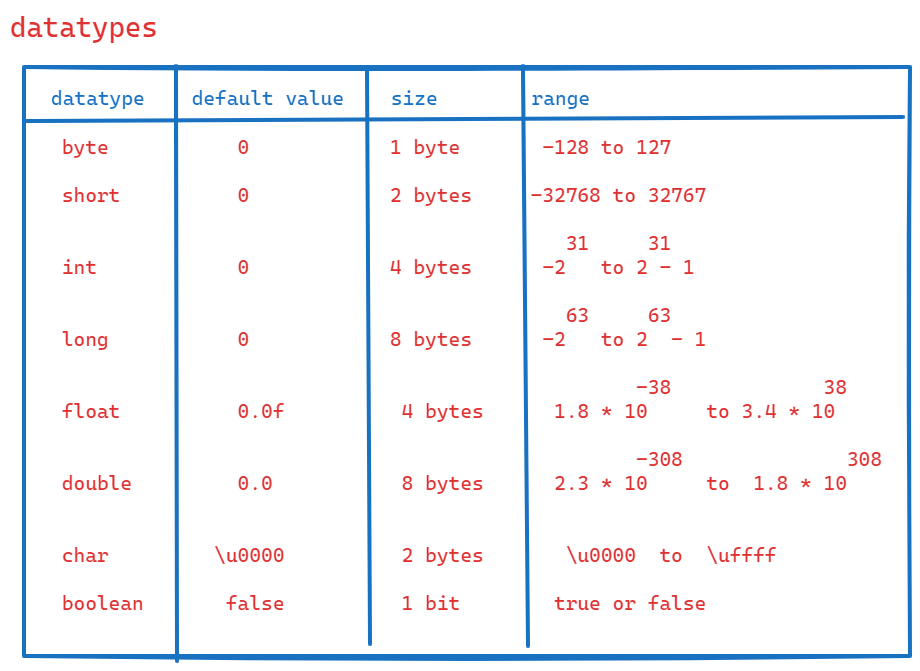
======================================================

|| Date: 30th July, 2024 ||

Datatypes in Java:



* Primitive types stores values and reference types stores objects.



* \u0000 is a Unicode for null value.
* Refer the below link to know th unicodes characters.
* <https://symbl.cc/en/unicode-table/>
* Unicode character set is a international character set which includes the characters of almost all the languages.
* The range of byte variable is -128 to 127.
* If you assign a number which is crossing the lower/upper limit, then we will get an error.

Ex: byte b = 128; // error

* To overcome this error, we can type cast the value into byte.

Ex: byte b = (byte)128; //correct

System.out.println(b); // -128

Ex: byte b = -129; //error

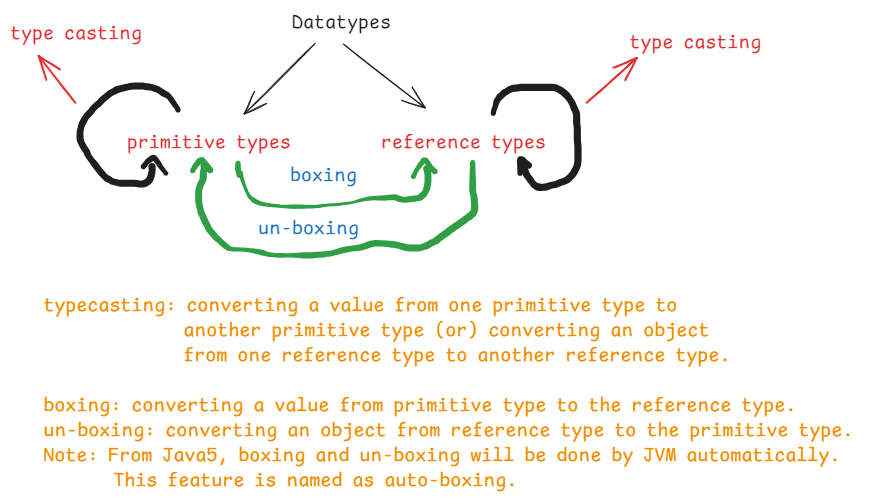
byte b = (byte) -129; //correct

System.out.println(b); // 127

Note: we can type cast this way for only byte and short data types.

TYPE CASTING:

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

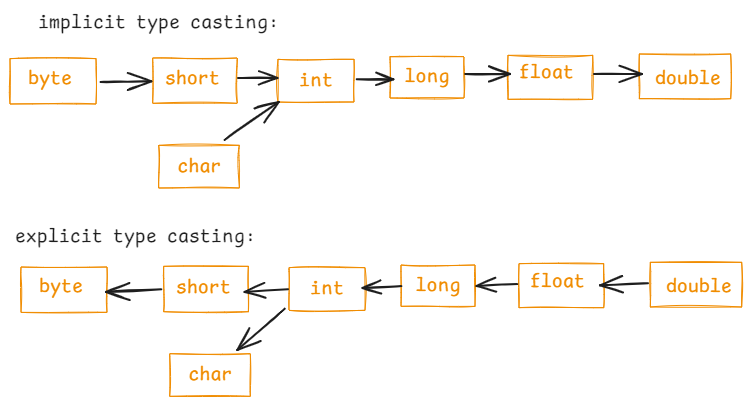


* Type casting is of 2 types.

1. Implicit type casting/widening/up casting
2. Explicit type casting/narrowing/down casting

|| Date: 31st July, 2024 ||

* Implicit type casting means, converting a value from low memory data type to high memory data type.
* Explicit type casting means, converting a value from high memory data type to low memory data type by compromising on the data loss.



Ex1:

byte b = 120;

int x = b; //implicit type casting

ex2:

long a = 298765L; // L represents long value

int b = a; //error

int b = (int) a; //correct, explicit type casting.

* For explicit type casting, we have to write the target data type with in a parenthesis.

Ex3:

char ch = ‘A’;

int x = ch; // implicit type casting.

System.out.println(x); // 65

Ex4:

int x = 68;

char ch = (char)x; //explicit type casting.

System.out.println(ch); // D

Ex5:

float f1 = 9.65f; // f represents float

double d = f1; // implicit type casting.

Note: a float variable can store upto 6 digits after the decimal point.

A double variable can store upto 15 digits after the decimal point.

Ex6:

boolean flag = false;

int x = flag; // error

int x = (int) flag; //error

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

Installing Eclipse IDE:

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

1. Visit <https://www.eclipse.org/downloads/>
2. Click on the link download packages
3. Eclipse IDE for Enterprise Java and Web Developers

Click on windows link

1. Eclipse-jee-2024-06-R-win32-x86\_64.zip file is downloaded.
2. Unzip/extract the file. The folder eclipse-jee-2024-06-R-win32-x86\_64

Is created.

1. Open the folder, you can find eclipse folder.
2. Open the eclipse folder, you can find eclipse application.

Note: To start the eclipse IDE easily, create a shortcut on desktop.

Writing the first program in eclipse:

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

* Before launching/starting the eclipse, create a folder which should act as a workspace.
* For example, In D:\ drive, create a new folder Workspace-9AM.
* A Workspace is a directory which stores the projects created by you.

1. Start eclipse.

Double click on eclipse shortcut on desktop.

1. In the eclipse launcher window, click on browse button, choose the

Folder Workspace-9AM

1. Click on the launch button.
2. Click on Fil menu 🡪 New 🡪 Project.. 🡪 Java project 🡪

Project name: TypeCastingProgram

Execution environment: Java SE-17

Click on Finish.

1. Expand project (TypeCastingProgram) 🡪 expand src folder 🡪 delete module-info.java
2. Right click on src folder 🡪 New 🡪 class 🡪

Name: Test 🡪 select check box for main method 🡪 finish.

1. Write the below code.

**public** **class** Test {

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

**byte** b = 120;

**int** x = b; // implicit

System.***out***.println("type casting byte to int : " + x);

**char** ch = 'A';

**int** y = ch; //implicit

System.***out***.println("type casting char to int : " + y);

**int** a = 68;

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

System.***out***.println("type casting int to char : " + ch2);

**long** m = 234401L;

**float** f1 = m; //implicit

System.***out***.println("type casting long to float : " + f1 );

}

}

1. Execute the program.

Right click on the code 🡪 RunAs 🡪 Java application.

(or)

Click on Run button(green play button on the top).

Output:

type casting byte to int : 120

type casting char to int : 65

type casting int to char : D

type casting long to float : 234401.0

|| DATE: 1st Aug, 2024 ||

Reading input from a user:

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

* If we write a program by assigning the values to the variables directly then that program will always gives same output.
* This is called a static program.
* To make a java program as dynamic, it should read the input from the user and it should display the output based on that input given by the user.
* Java programming language has provided a library which is also called as Java API(Application Programming Interface) for developing the applications using Java by the programmers.
* To read input from a user, a programmer has to use Scanner class given by the Java API.
* Java API consists multiple packages where a package is group of classes.
* Scanner class is given under a package java.util.
* To read the input from a user, a Java program has to first create an object for Scanner class.

Scanner scan = new Scanner(System.in);

* Here, System.in represents the standard input device(keyboard).
* Here, Scanner class has to read the input from the keyboard. That’s why we are passing System.in within parenthesis.
* In Scanner class, various methods are given to read the

different types of data.

1. nextInt() : reads int value
2. nextFloat() : reads float value
3. nextBoolean() : reads a boolean value
4. nextChar() : reads a char value
5. nextLine(): reads a string value, etc..

Java comments:

1. single line comment

// comment

1. multi line comment

/\*

\* line1

\* line2

\* line3

\*/

3. documentation comment

/\*\*

\* line1

\* line2

\* line3

\*/

/\*

\* write a program to find the sum of two numbers

\* by reading the input values from the user.

\*/

**import** java.util.Scanner;

**public** **class** SumOfTwoNumbers {

**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 );

}

}

/\*

\* write a program to take the input as forenheit value

\* then convert it into celcius value.

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

\*/

**import** java.util.Scanner;

**public** **class** ForenheitToCelcius {

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

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

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

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

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

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

}

}

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

|| DATE: 2nd Aug, 2024 ||

/\*

\* write a program to calculate emi by taking

\* input values as principle amount, rate of interest

\* and tenure in months.

\* Hint: formula n n

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

\*/

**import** java.util.Scanner;

**public** **class** \_EmiCalculator {

**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");

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

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

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

//converting rate of interest from per annum 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);

}

}

/\*

\* write a program to take the input as minutes

\* and convert into days : hours : minutes format.

\* ex: 3124 minutes = 2d : 4h : 4m

\*/

**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** days = minutes / 60 / 24;

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

**int** remainingMinutes = minutes % 60;

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

}

}

Operators:

* operator is a symbol which performs a well defined task.
* Operators works with operands.
* Operators are 3 types.

1. Unary operators
2. Binary operators
3. Ternary operator

* Unary operators works with a single operand.
* Binary operators works with two operands.
* Ternary works with 3 operands/3 parts.

Unary operators:

1. Increment operator(++)
2. Decrement operator( -- )

* Increment operator increments the operand value by 1.
* Decrement operator decrements the operand value by 1.

operand++ : post increment

++operand : pre increment

Operand-- : post decrement

--operand : pre decrement

Post increment : use the value then increment

Pre increment : increment the value then use it.

Post decrement : use the value then decrement.

Pre decrement : decrement the value then use it.

Ex:

int x = 10;

int y = x++;

S.o.p(x);

S.o.p(y);

Op: 11

10

Ex2:

int x = 10;

int y = ++x;

S.o.p(x);

S.o.p(y);

Op: 11

11

||Date: 5th Aug, 2024 ||

Ex3: int x = 19

int y = x--;

S.o.p(x);

S.o.p(y);

Op: 18

19

Ex4:

int x = 10;

int y = 20;

x = x++ + --y;

y = --x + y++;

S.o.p(x);

S.o.p(y);

Op:

28

47

Ex5:

int x = 10;

int y = x++ + ++x + x--;

S.o.p(x);

S.o.p(y);

Op: 11

34

Binary operators:

* Binary operators are divided into 5 types.

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Bitwise operators
6. Arithmetic operators:

+ addition

* Subtraction

\* multiplication

/ division

% modulus

Ex1:

int x = 5;

int y = x / 2;

int z = x % 2;

S.o.p(“ x = “ + x);

S.o.p(“ y = “ + y);

S.o.p(“ z = “ + z);

Op: x = 5

y = 2

z = 1

ex2:

double x = 5.0;

double y = x / 2;

double z = x % 2;

S.o.p(“ x = “ + x);

S.o.p(“ y = “ + y);

S.o.p(“ z = “ + z);

Op:

X = 5.0

Y = 2.5

Z = 1.0

Ex3:

int y = 4 / 0;

S.o.p(y);

Op: ArithmeticException: / by zero.

Ex4:

double y = 4 / 0.0;

S.o.p(y);

Op: Inifinity

Ex5:

double y = 4.0 / 0;

S.o.p(y);

Op: Infinity

Note: when you are dividing a number with zero, where either numerator or denominator is a double then the result is “Infinity”.

1. Relational operators:

* Relational operators are used to construct a preposition.
* Preposition is a condition which can return a boolean value either true or false.

< less than

> greater than

<= less than or equals

>= greater than or equals

== equals

!= not equals

Ex1:

int x = 9;

S.o.p ( x % 2 == 0 ); false

S.o.p ( x % 2 != 0 ); true

S.o.p ( x % 2 < 2 ); true

S.o.p ( x % 2 <= 1 ); true

S.o.p ( x % 2 >= 1 ); true

S.o.p( x % 2 > 1 ); false

1. Logical operators:

* These are used to combine two prepositions to a single preposition.

&& AND

|| OR

! NOT

C1 C2 C1 && C2 C1 || C2 !C1

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

true true true true false

true false false true false

false true false true true

false false false false true

* Logical AND(&&) operator returns true, if the two conditions are true.
* Logical OR(||) operator returns false, if the two conditions are false.
* Logical NOT(!) operator returns true, if the condition is false and vice-versa.

Ex1:

int x = 10;

if ( x++ > 12 && x-- < 10 )

S.o.p(“hi”);

Op: no output

Ex2:

int x = 10;

if ( x++ < 9 || x-- > 9 )

S.o.p(x);

Op:

10

* In && operator, if the first condition is false, then the control doesn’t check for the second condition.
* In || operator, if the first condition is true then the control doesn’t check for the second condition.

1. Assignment operators:

= assign a value

+= add the value to the current variable and store

the result back to the current variable.

-= subtract the value from the current variable and store

The result back to the current variable.

\*=

/=

%=

|| DATE: 6th Aug, 2024 ||

1. Bitwise operators:

* Bitwise operators are used in developing applications for device drivers, embedded systems, cryptography, etc..
* For web/enterprise applications development we don’t use bitwise operators.

& Bitwise AND

| Bitwise OR

^ Bitwise XOR

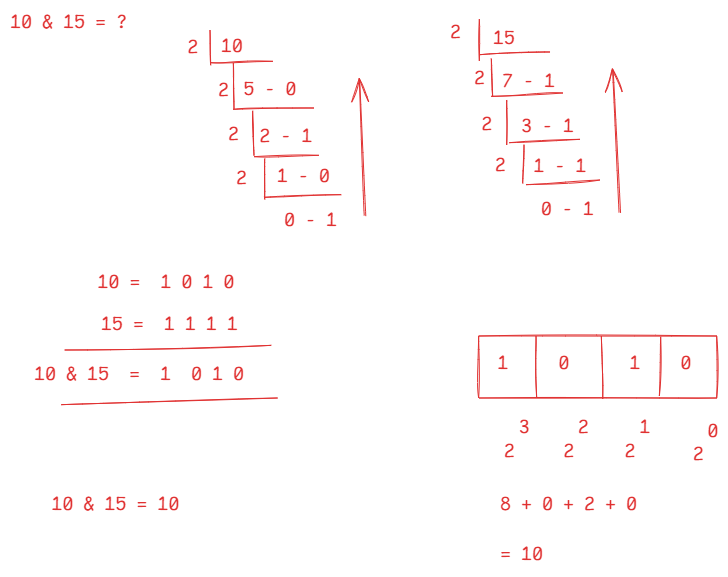
<< left shift

>> right shift

* Bitwise operators works on the bits of the operands.

Bitwise AND(&):

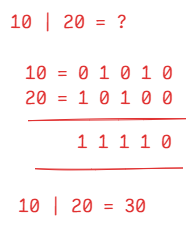
* It results a bit 1, if the corresponding bits are 1. Otherwise, the result is 0.



Bitwise OR(|):

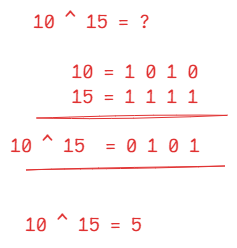
* It results a bit 0, if the corresponding bits are 0. Otherwise, the result is 1.

Ex:



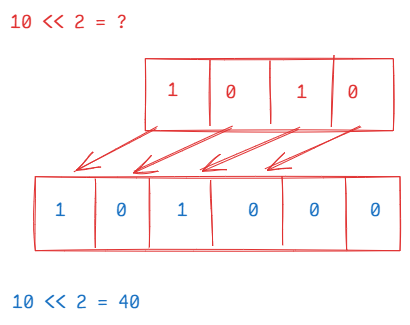
Bitwise XOR(^):

* It results a bit 1, if the corresponding bits are different. Otherwise, the result is 0.



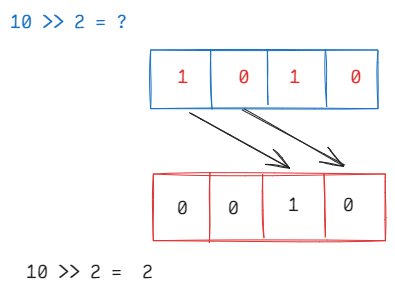
Left shift(<<):

* It shifts the bits of the operand to the specified number of positions to its left side.
* The empty bits at right side are replaced with 0’s.



Right shift(>>):

* Right shift operator shifts the bits of the operand to the right side by the specified positions.
* The empty bits created at left side are filled with 0’s.
* Left shift increases the operand value and right shift decreases the operand value.



Ternary operator:

* Ternary operator is represented with ?:
* Ternary operator works with 3 parts.

Part1: condition

Part2: value if true

Part3: value if false

part1 ? part2 : part3

ex1:

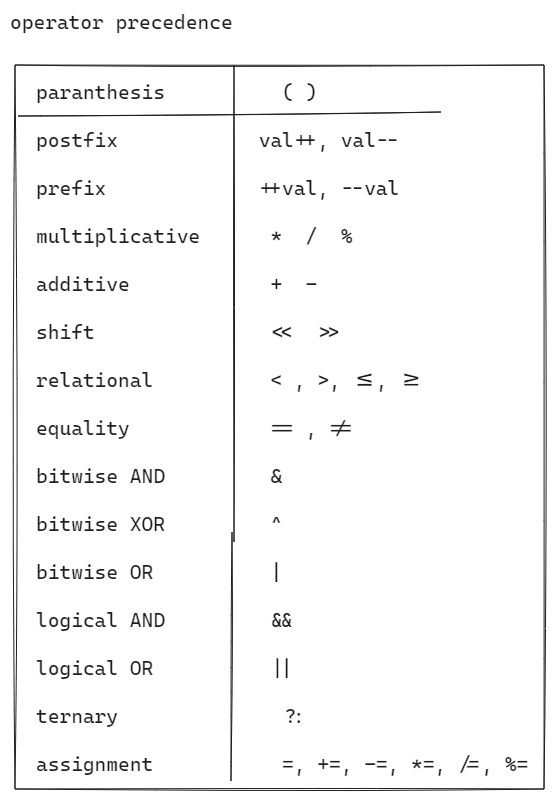
int k = 10 > 15 ? 10 : 15;

S.o.p(k); // 15

Ex2:

int k = (10>15 || 20<25) ? 10\*2 : 5\*3;

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



Ex1:

int x = 3 + 4 % 9 / 2 – 1 \* 8;

S.o.p(x); // -3

Ex2:

int x = 9 / 3 / 7 – 3 – 2 + 8 \* 3;

S.o.p(x); //19

Java Naming conventions:

1. Pascal case
2. Camel case
3. Upper case
4. Lower case

Pascal case : The first character of each word in an identifier should be in upper case. This pascal case naming convention is used for class name and interface name.

Ex: class EmployeeService { //class name is valid and followed pascal case

}

Ex: class employeeRepository { //class name is valid but not followed pascal case

}

Ex: class Employeecontroller { //class name is valid but not followed pascal case

}

Camel case: Except the first word, the first character of the next words in an identifier must be in upper case. The camel case should be used for variables and methods.

Ex: class Sample {

int totalPages; //valid, followed the camel case

void displayPages() { //valid, followed the camel case

}

}

Ex: class PolicyCalculator {

int policyid; //valid, but not followed camel case

String policyholdername; //valid, but bot followed camel case

void calculatePremium() { //valid, followed camel case

}

}

Upper case: constants in a java program should follow upper case.

In Java, we don’t have const keyword, but we use static and

final keywords to declare the constants.

Ex: class DaysToYearsConversion {

static final int DAYS\_PER\_YEAR = 365; //valid, followed upper case

}

Ex: class DaysToHoursConversion {

static final int hours\_per\_day = 24; //valid, but not followed upper case

}

Lower case : lower case should be used for keywords and package names.

Control statements:

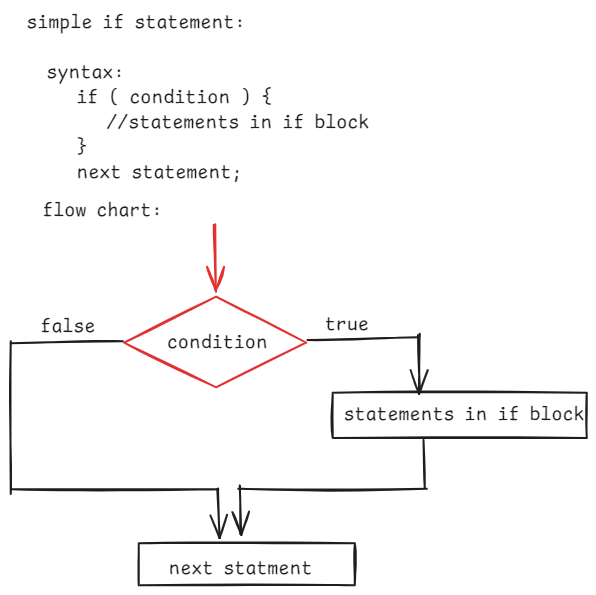
1. Selection control statements
2. Iteration control statements
3. Jumping control statements.

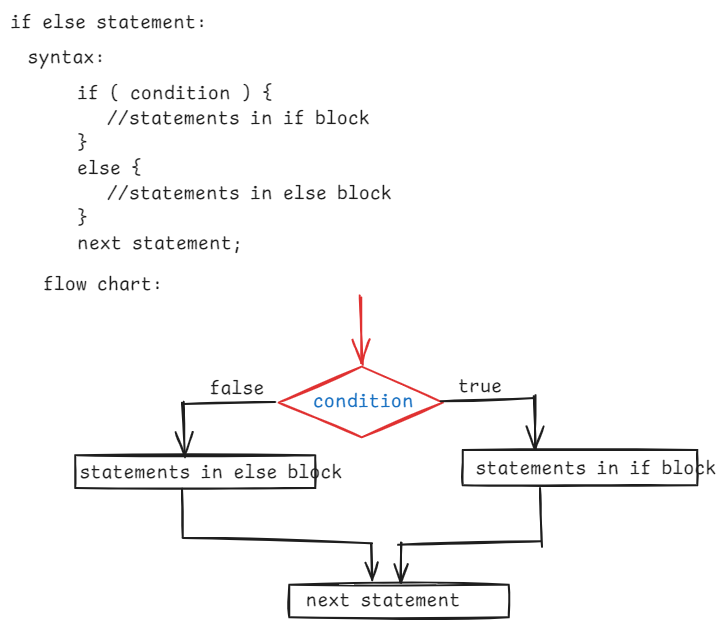
Selection control statements:

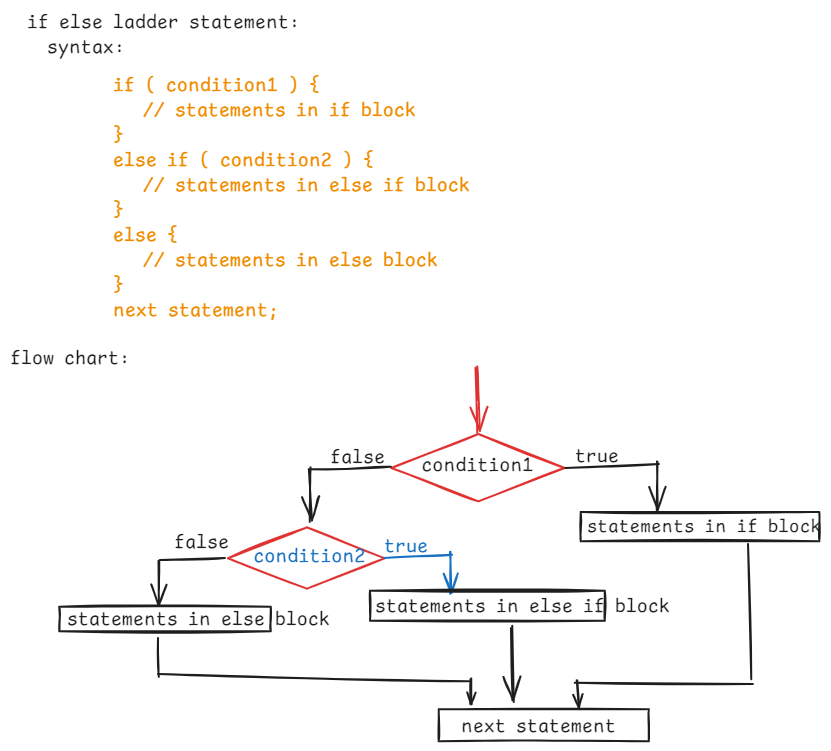
1. if statement
2. switch statement

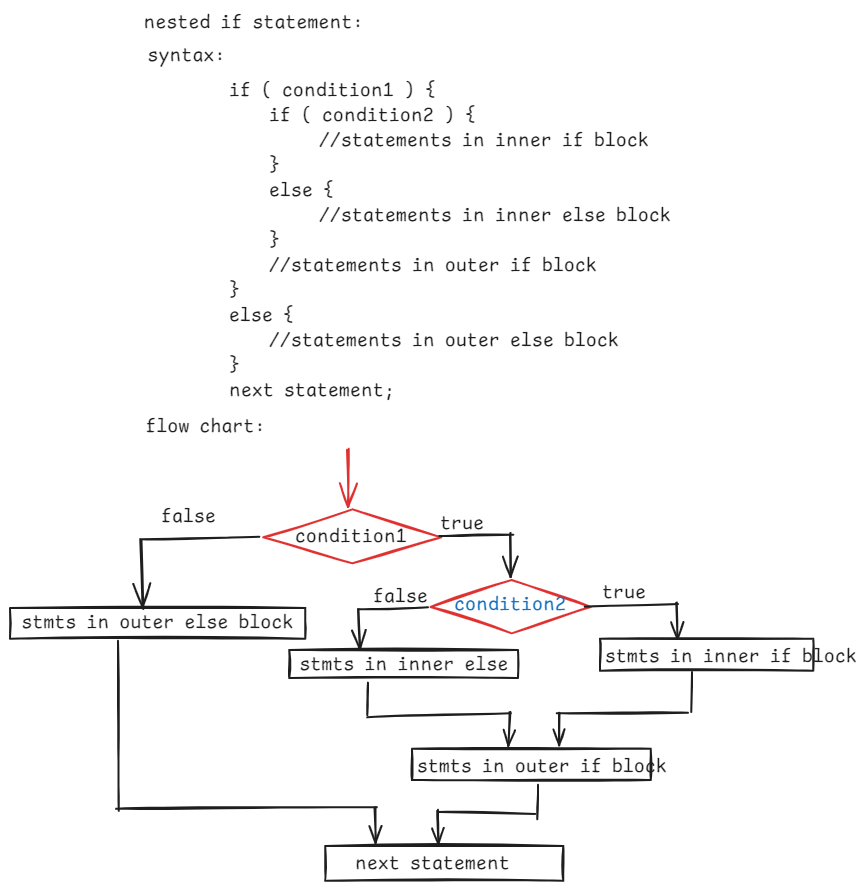
if statement:

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









|| DATE: 8th Aug, 2024 ||

/\*

\* write a program to check whether a given

\* number is even or odd.

\*/

**import** java.util.Scanner;

**public** **class** EvenOddClass {

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

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

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

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

**if** ( number % 2 == 0 ) {

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

}

**else** {

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

}

}

}

/\*

\* Write a program to take a number as input and do

\* the following.

\* If number is divisible by 3 then display "Zip"

\* If number is divisible by 5 then display "Zap"

\* If number is divisible by 3 and 5 then display "Jar".

\* Otherwise, display "Done".

\*/

**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();

**if** ( number % 3 == 0 && number % 5 == 0 )

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

**else** **if** ( number % 3 == 0 )

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

**else** **if** ( number % 5 == 0 )

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

**else**

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

scan.close();

}

}

/\*

\* write a program to find the biggest

\* of three numbers using nested if

\*/

**import** java.util.Scanner;

**public** **class** Biggest {

**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 && b == c) {

System.***out***.println("The three numbers are equal");

} **else** {

**if** (a > b) {

**if** (a > c)

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

**else**

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

} **else** {

**if** (b > c)

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

**else**

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

}

}

}

}

/\*

\* write a program for the following.

\* Take 3 numbers as input.

\* if first number is 7 then display product of the remaining numbers

\* if last number is 7 then display the product of first two numbers.

\* if 7 is in middle then display product of first and last numbers

\* if all the numbers are 7 then display product of three numbers.

\* if 7 is not in the given numbers then display the result as -1.

\*/

**import** java.util.Scanner;

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

**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 == 7 && b == 7 && c == 7 )

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

**else** {

**if**( a == 7 )

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

**else** **if** ( b == 7 )

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

**else** **if** ( c == 7 )

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

**else**

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

}

}

}

|| DATE: 9-Aug-24 ||

/\*

\* Write a program to take the two numbers as input

\* and do the following.

\* If the two numbers are same then display the sum

\* If they are different and one of the number is 9 then

\* display double of the sum

\* Otherwise, display -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 ) \* 2 );

}

**else** {

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

}

}

}

/\*

\* Write a program to take the input as distance in kms

\* and calculate the delivery fee as the following.

\* 1. for first 3 kms, delivery fee is 0

\* 2. for next 3 kms, delivery fee is 3 rs per km

\* 3. for remaining kms, delivery fee is 6 rs per km

\*/

**import** java.util.Scanner;

**public** **class** TestClass {

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

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

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

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

**int** deliveryFee = 0;

**if** ( distance < 0 ) {

System.***out***.println("distance can not be negitive");

}

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

deliveryFee = 0;

}

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

deliveryFee = ( distance - 3 ) \* 3;

}

**else** {

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

}

**if**( distance >= 0)

System.***out***.println("Delivery fee to pay in Rs. : " + deliveryFee);

}

}

/\*

\* Write a program to calculate the final price with the below

\* conditions.

\* 1. If item is shoe and material is leather add 14% GST to the price

\* 2. If material is not leather then add 9% GST.

\* 3. if item is mobile and brand is apple then add 12% GST and provide 5% discount.

\* 4. If brand is any other then add 6% GST

\* 5. For any other item, display the output as "invalid item".

\*/

**import** java.util.Scanner;

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

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

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

System.***out***.println("Enter the item (shoe/mobile) ");

String item = scan.nextLine();

**double** finalPrice = 0;

**if** ( item.equals("shoe") ) {

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

String material = scan.nextLine();

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

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

**if** ( material.equals("leather") ) {

**double** gst = price \* 0.14;

finalPrice = price + gst;

}

**else** {

**double** gst = price \* 0.09;

finalPrice = price + gst;

}

}

**else** **if** ( item.equals("mobile") ) {

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

String brand = scan.nextLine();

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

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

**if** ( brand.equals("apple") ) {

**double** gst = price \* 0.12;

**double** discount = price \* 0.05;

finalPrice = price + gst - discount;

}

**else** {

**double** gst = price \* 0.06;

finalPrice = price + gst;

}

}

**else** {

System.***out***.println("Invalid item");

System.*exit*(0); //terminate the execution of the program.

}

System.***out***.println("Item : " + item);

System.***out***.println("Final price : " + finalPrice);

scan.close();

}

}

|| DATE: 10-Aug-24 ||

/\*

\* write a program to take 3 sides of a triangle as input

\* and display the output like below.

\* 1. if 3 sides are equal then display Equilateral traingle

\* 2. if 2 sides are equal then display Isosceles traiangle

\* 3. if sides are not equal then display Scalene triangle

\*/

**import** java.util.Scanner;

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

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

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

System.***out***.println("enter side1");

**double** side1 = scan.nextDouble();

System.***out***.println("enter side2");

**double** side2 = scan.nextDouble();

System.***out***.println("enter side3");

**double** side3 = scan.nextDouble();

**if** ( side1 + side2 > side3 || side1 + side3 > side2 || side2 + side3 > side1 ) {

**if** ( side1 == side2 && side2 == side3) {

System.***out***.println("The three sides can form euqilateral triangle");

}

**else** **if** ( side1 == side2 || side2 == side3 || side1 == side3 ) {

System.***out***.println("The three sides can form isosceles triangle");

}

**else** {

System.***out***.println("The three sides can form scalane triangle");

}

}

**else** {

System.***out***.println("The given three sides can not form a triangle");

}

}

}

/\*

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

\* leap year or not, using nested if conditions.

\* 1. if a year is divisible by 4 and divisible by 100 then it must

\* be divisible by 400 also. Then it is a leap year.

\* 2. if a year is divisible by 4 and not divisible by 100 then

\* it is a leap year.

\* 3. if a year is not divisible by 4 then it is not a leap year.

\*/

**import** java.util.Scanner;

**public** **class** LeapYearTest {

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

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

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

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

**if** ( year % 4 == 0 ) {

**if** ( year % 100 == 0 ) {

**if** ( year % 400 == 0 ) {

System.***out***.println( year + " : is a leap year");

}

**else** {

System.***out***.println( year + " : is not a leap year.");

}

}

**else** {

System.***out***.println( year + " : is a leap year.");

}

}

**else** {

System.***out***.println( year + " : is not a leap year ");

}

}

}

Note:

Using ternary operator, we can find the leap year like below.

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

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

|| DATE: 12-Aug-24 ||

switch statement:

* when we want to compare same variable with multiple values and the comparison is a equality then we can use switch statement.
* switch statement has a parameter and that parameter must be either a variable or an expression.
* Under switch statement, we have to define one or more case statements.
* Each case has a value, and switch statement compares the value of the variable/expression with the case value.
* If a case value is matched, then the statements of that case will be executed.
* If that case doesn’t have a break statement, the control executes the next cases also until break statement occurs.
* If no case value is matched, then default case will be executed.
* Writing a default case is optional.

**Syntax:**

**switch ( variable or expression ) {**

**case value1 : statements;**

**break;**

**case value2 : statements;**

**break;**

**. .**

**. .**

**case valuen : statements;**

**break;**

**default: statements;**

**}**

**Ex1:**

**int** x = 1;

**switch**( x ) {

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

**break**;

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

**break**;

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

**break**;

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

}

**Output: ONE**

**EX2:**

**int** x = 2;

**switch**( x ) {

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

**break**;

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

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

**break**;

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

}

**OUTPUT: TWO**

**THREE**

**EX3:**

**int** x = 2;

**switch**( x \* x ) {

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

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

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

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

**break**;

}

Output: ZERO

ONE

TWO

THREE

* **A switch statement can use either integer(byte/short/int/long) or a character or a string variable.**

Ex:

**double** x = 2;

**switch**( x \* x ) {

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

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

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

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

**break**;

}

Output: error. Because we can’t use double in switch statement.

* A case value also must be either an integer or a character or a string value.

**Ex;**

**int** x = 2;

**switch**( x \* x ) {

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

**case** 1.0: System.***out***.println("ONE");

**case** 2.0: System.***out***.println("TWO");

**case** 3.0: System.***out***.println("THREE");

**break**;

}

**Output: error. Because, the case values are double and it is not allowed.**

**Ex:**

**int** x = 2;

**int** a = 1;

**int** b = 2;

**int** c = 3;

**switch**( x ) {

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

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

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

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

**break**;

}

**Output: error. Because, we can’t use variables in case statement.**

**Ex:**

**switch**( 2 ) {

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

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

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

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

**break**;

}

**Output: TWO**

**THREE**

* **IN switch statement, we can pass a variable or expression or a value also.**

**Ex:**

**int** a = 1;

**int** b = 2;

**int** c = 3;

**switch**( a, b ) {

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

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

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

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

**break**;

}

**Output: error. Because, in switch statement, we can pass only one expression or one variable or one value only.**

**Ex:**

**char** ch = 'x';

**switch**( ch ) {

**case** 'a':

**case** 'e':

**case** 'i':

**case** 'o':

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

**break**;

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

}

Output: CONSONENT

**Ex:**

**char** ch = 'i';

**switch**( ch ) {

**case** 97:

**case** 101:

**case** 105:

**case** 111:

**case** 117: System.***out***.println("VOWEL");

**break**;

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

}

**Output: VOWEL**

**Ex:**

String day = "Monday";

**switch**( day ) {

**case** "Monday": System.***out***.println("Monday's are lazy");

**break**;

**case** "Tuesday":

**case** "Wednesday":

**case** "Thursday": System.***out***.println("The day is a busy day");

**break**;

**case** "Friday": System.***out***.println("The day is so-so");

**case** "Saturday":

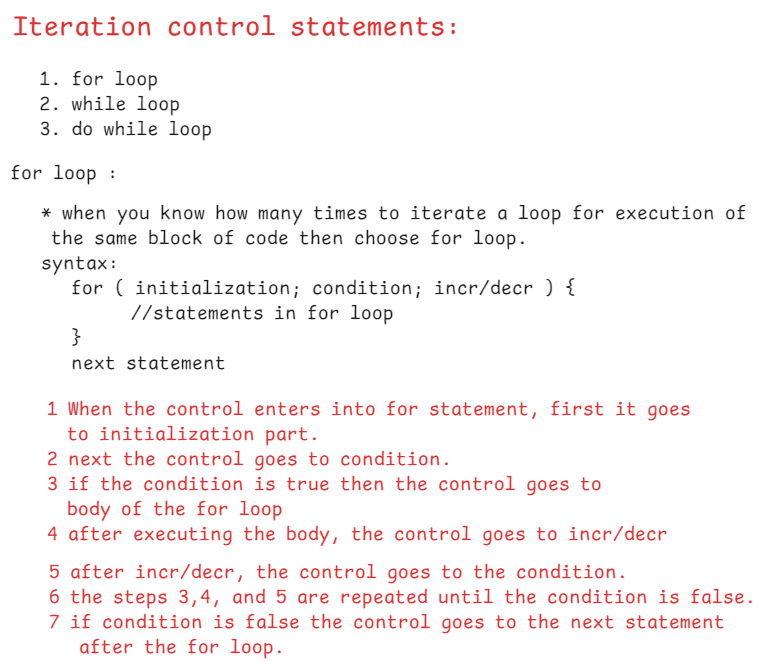
**case** "Sunday": System.***out***.println("Weekend is crazy");

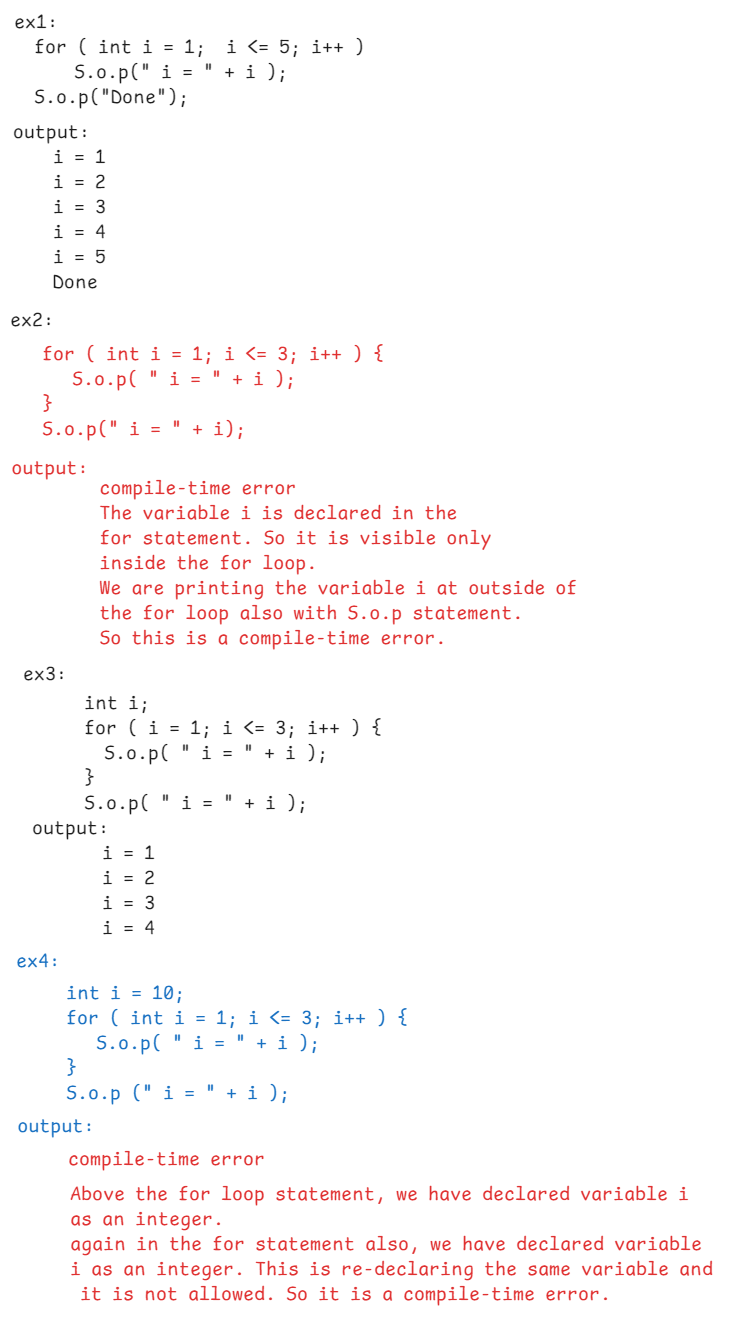
**break**;

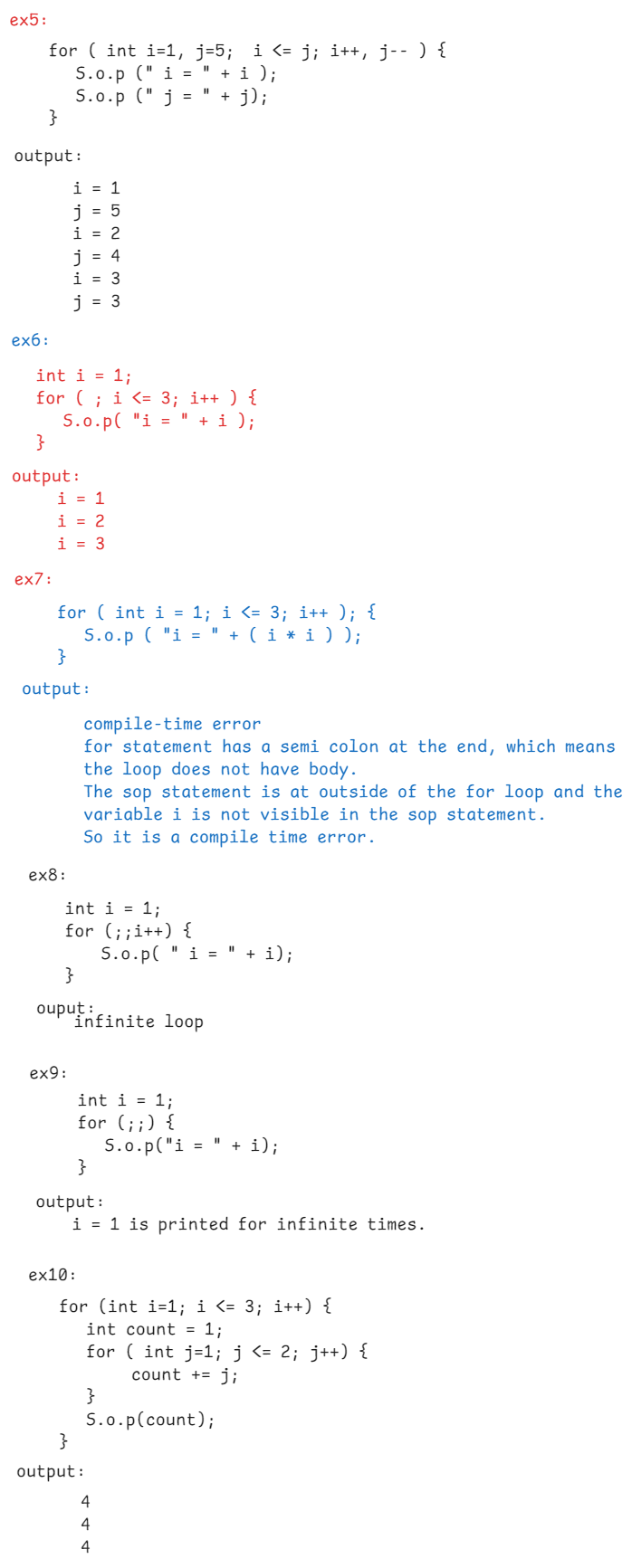
}

**Output: Monday’s are lazy**

**|| DATE: 13-Aug-24 ||**

****

****

****

**|| DATE: 14-Aug-24 ||**

/\*

\* write a program to find the sum of n

\* natural numbers.

\* ex: if n = 5,

\* sum = 15

\*/

**import** java.util.Scanner;

**public** **class** NaturalSumClass {

**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** n) {

**int** sum = 0;

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

sum = sum + i;

}

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

}

}

/\*

\* write a program to find the sum of

\* even and odd numbers separately upto the

\* given number.

\* ex: number = 10

\* output:

\* even sum = 30

\* odd sum = 25

\*/

**import** java.util.Scanner;

**public** **class** EvenOddSumClass {

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

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

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

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

*findEvenOddSum*(number);

}

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

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

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

**if** ( i % 2 == 0 ) {

evenSum += i; // evenSum = evenSum + i

}

**else** {

oddSum += i; // oddSum = oddSum + i

}

}

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

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

}

}

**|| DATE: 16-Aug-24 ||**

/\*

\* write a program to check whether a given

\* number is a prime number or not.

\*

\* prime number : A whole number which has only two factors(1 and itself)

\* is called as a prime number

\* examples: 3, 5, 7, 11, 13,...

\*/

**import** java.util.Scanner;

**public** **class** PrimeTest {

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

**int** count = 0;

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

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

count++;

**break**;

}

}

**if**( count == 0 )

**return** **true**;

**else**

**return** **false**;

}

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

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

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

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

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

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

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

**else**

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

}

}

/\*

\* 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

\* that number is equal to the same number

\* then it is a perfect number.

\* example: n = 6

\* 1 + 2 + 3 = 6

\* 6 is a perfect number

\* example: 24

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

\* 24 is not a perfect number

\* example: 28

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

\* 28 is a perfect number

\*/

**import** java.util.Scanner;

**public** **class** PerfectTest {

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

**int** sum = 0;

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

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

sum = sum + i;

}

}

**if** ( sum == n )

**return** **true**;

**else**

**return** **false**;

}

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

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

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

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

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

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

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

**else**

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

}

}

**|| Date: 17-Aug-24 ||**

/\*

\* write a program to find the factorial

\* of a given number.

\* ex: n = 5

\* output: 120

\*/

**import** java.util.Scanner;

**public** **class** FactorialTest {

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

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

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

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

**int** f = *findFactorial*(n);

System.***out***.println( n + "! = " + f);

}

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

**int** factorial = 1;

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

factorial = factorial \* i;

}

**return** factorial;

}

}

/\*

\* write a program to print the fibonacci series

\* of n terms.

\* 1. The first 2 terms of the series are 0 and 1

\* 2. the next term is sum of the previous 2 terms.

\* ex: n = 5

\* output: 0 1 1 2 3

\* ex: n = 8

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

\*/

**import** java.util.Scanner;

**public** **class** FibonacciTest {

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

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

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

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

*printFibonacciSeries*(n);

}

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

**int** firstTerm = 0;

**int** secondTerm = 1;

**if**( n <= 0 ) {

System.***out***.println("please enter a number > 0");

**return**;

}

**if** ( n == 1 ) {

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

}

**else** **if** ( n == 2 ) {

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

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

}

**else** {

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

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

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

**int** nextTerm = firstTerm + secondTerm;

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

firstTerm = secondTerm;

secondTerm = nextTerm;

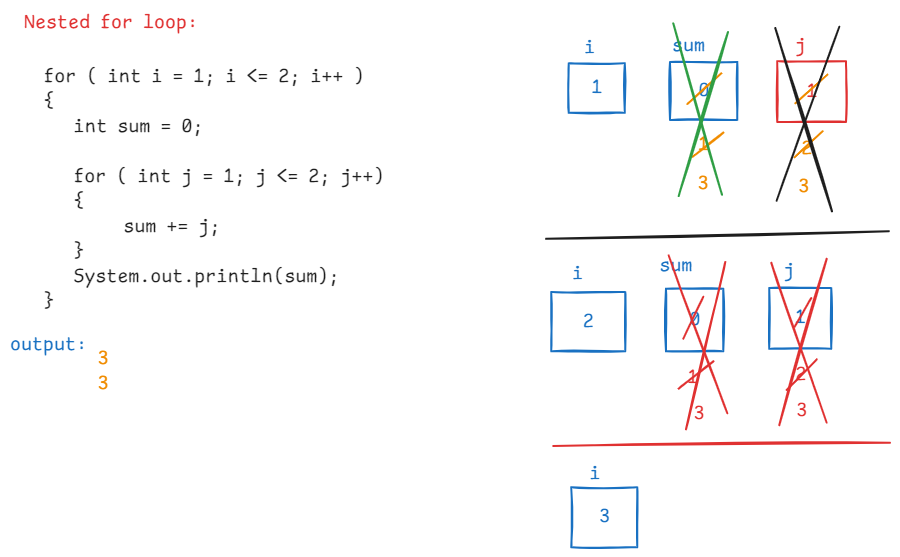
}

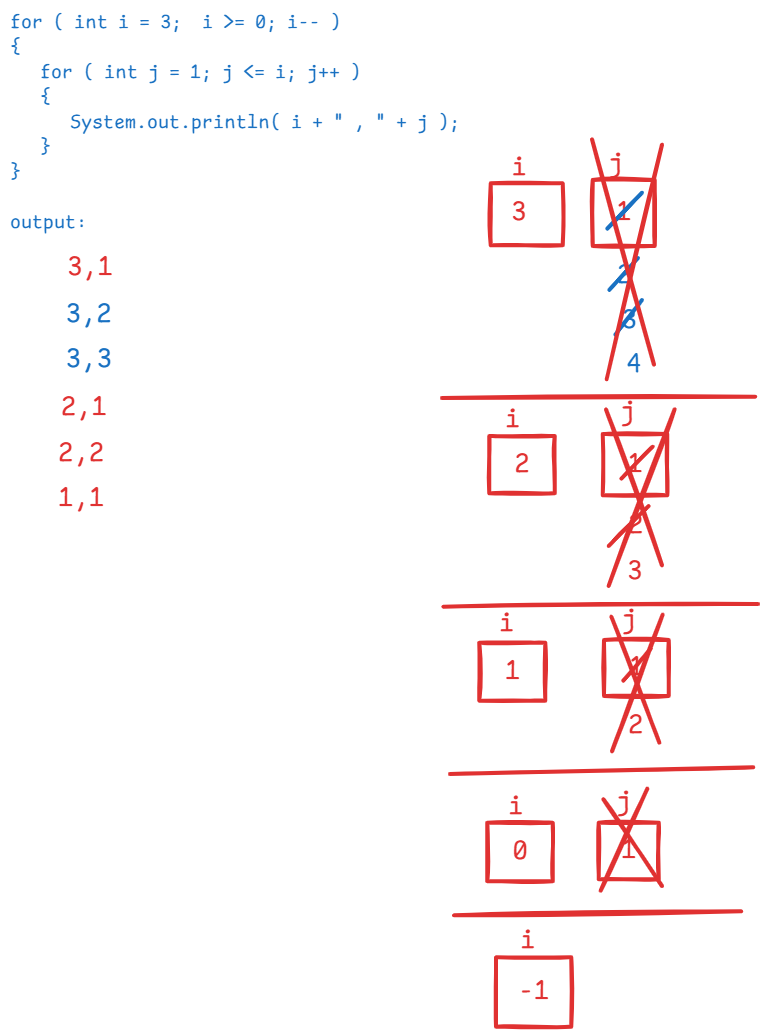
}

}

}

**|| DATE: 19-Aug-24 ||**

****

****

/\*

\* write a program to print all the

\* prime numbers between the given

\* range x and y.

\*/

**import** java.util.Scanner;

**public** **class** PrimesRangeTest {

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

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

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

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

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

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

*printPrimes*(x, y);

}

**private** **static** **void** printPrimes(**int** x, **int** y) {

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

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

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

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

flag = **false**;

**break**;

}

}

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

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

}

}

}

}

**|| DATE: 20-Aug-24 ||**

/\*

\* write a program to print the below pattern.

\* if n = 5,

\* output:

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

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

\* points:

\* 1. we always use nested loops for printing the patterns

\* 2. for the outer loop, we count the number of lines/rows

\* and loop for them.

\* 3. For the inner loop, we focus on the number of columns

\* and somehow we need to form the logic between rows and

\* columns.

\* 4. print \* inside the inner loop.

\*/

**import** java.util.Scanner;

**public** **class** Pattern1 {

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

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

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

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

*printPattern*(n);

}

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

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

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

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

}

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

}

}

}

/\*

\* write a program to print the below pattern.

\* if n = 5,

\* output:

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

\* \* \* \* \*

\* \* \* \*

\* \* \*

\* \*

\*/

**import** java.util.Scanner;

**public** **class** Pattern2 {

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

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

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

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

*printPattern*(n);

}

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

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

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

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

}

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

}

}

}

/\*

\* write a program to print the below pattern.

\* if n = 5,

\* output:

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

\* \* \* \* \*

\* \* \* \*

\* \* \*

\* \*

\*/

**import** java.util.Scanner;

**public** **class** Pattern3 {

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

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

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

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

*printPattern*(n);

}

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

//outer for loop for rows

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

//inner for loop1 for spaces

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

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

}

//inner for loop2 for stars

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

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

}

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

}

}

}

/\*

\* write a program to print the below pattern

\* if n = 5,

\* output:

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

\* \* \*

\* \* \*

\* \* \*

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

\*/

**import** java.util.Scanner;

**public** **class** Pattern4 {

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

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

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

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

*printPattern*(n);

}

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

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

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

**if**(i==1 || i==n || j==1 || j==n)

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

**else**

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

}

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

}

}

}

/\*

\* write a program to print the below pattern.

\* if n = 5,

\* output:

\* \*

\* \* \* \*

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

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

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

\*/

**import** java.util.Scanner;

**public** **class** Pattern5 {

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

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

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

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

*printPattern*(n);

}

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

//outer for loop for rows

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

//inner for loop1 for spaces

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

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

}

//inner for loop2 for stars

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

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

}

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

}

}

}

|| DATE: 22-Aug-24 ||

/\*

\* write a program to print the below pattern.

\* if n = 5

\* output:

\* \*

\* \* \* \*

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

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

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

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

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

\* \* \* \*

\* \*

\*/

**import** java.util.Scanner;

**public** **class** Pattern6 {

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

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

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

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

*printPattern*(n);

}

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

//outer for loop for rows

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

//inner for loop1 for spaces

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

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

}

//inner for loop2 for stars

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

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

}

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

}

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

//inner for loop1 for spaces

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

System.***out***.print(" "); // 1 space

}

//inner for loop2 for stars

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

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

}

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

}

}

}

/\*

\* write a program to print the below pattern

\* if n =5,

\* output:

\* 1

\* 0 1

\* 1 0 1

\* 0 1 0 1

\* 1 0 1 0 1

\*/

**import** java.util.Scanner;

**public** **class** Pattern7 {

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

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

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

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

*printPattern*(n);

}

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

**int** start=0;

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

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

start = 0;

**else**

start = 1;

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

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

start = 1 - start;

}

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

}

}

}

/\*

\* write a program to print the below pattern

\* if n = 5,

\* output:

\* \* \*

\* \* \*

\* \*

\* \* \*

\* \* \*

\*/

**import** java.util.Scanner;

**public** **class** Pattern8 {

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

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

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

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

*printPattern*(n);

}

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

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

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

**if** ( i == j || i +j == n+1 ) {

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

}

**else** {

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

}

}

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

}

}

}

/\*

\* write a program to print the below pattern.

\* if n = 5,

\* output:

\* 5 4 3 2 1

\* 4 3 2 1

\* 3 2 1

\* 2 1

\* 1

\*/

**import** java.util.Scanner;

**public** **class** Pattern9 {

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

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

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

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

*printPattern*(n);

}

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

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

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

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

}

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

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

}

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

}

}

}

|| Date : 23-Aug-24 ||

/\*

\* write a program to print the below pattern.

\* if n =5,

\* output:

\* A

\* A B

\* A B C

\* A B C D

\* A B C D E

\*

\*/

**import** java.util.Scanner;

**public** **class** Pattern10 {

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

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

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

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

*printPattern*(n);

}

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

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

**char** ch = 'A';

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

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

ch = (**char**) (ch + 1);

}

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

}

}

}

/\*

\* write a program to print the pascal traingle.

\* if n = 5

\* 1

\* 1 1

\* 1 2 1

\* 1 3 3 1

\* 1 4 6 4 1

\*/

**import** java.util.Scanner;

**public** **class** Pattern11 {

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

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

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

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

*printPattern*(n);

}

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

**for** ( **int** i = 0; i <= n - 1; i++ ) {

//inner for loop1 for spaces

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

System.***out***.print(" "); //1 space

}

//inner for loop2 for values

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

System.***out***.print(*nCr*(i,j) + " " );

}

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

}

}

**private** **static** **int** nCr(**int** i, **int** j) {

**return** *factorial*(i) / ( *factorial*(i-j) \* *factorial*(j) );

}

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

**int** fact = 1;

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

fact = fact \* k;

}

**return** fact;

}

}

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

**while loop:**

syntax:

while ( condition ) {

//statements in while loop

}

next statement;

/\*

\* write a program to find the sum of digits

\* of a given number.

\* ex:

\* number = 265

\* output: 2 + 6 + 5 = 13

\*/

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();

sumOfDigits(number);

}

private static void sumOfDigits(int number) {

int sum = 0;

if( number < 0 ) {

System.out.println("You have to enter positive integer only");

return;

}

while( number > 0 ) {

int r = number % 10;

sum += r;

number /= 10;

}

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

}

}

/\*

\* write a program to check whether a given

\* number is an armstrong number or not.

\*

\* Armstrong number:

\* If sum of n th power of each digit of a number is

\* equal to the same number then it is called Armstrong

\* number. Here n is the length of a number.

\* ex: number = 9

\* output: armstrong number

\* ex: number = 12

\* 2 2

\* output: 1 + 2 = 5

\* not an armstrong number

\* ex: number = 153

\* 3 3 3

\* output: 1 + 5 + 3 = 1 + 125 + 27 = 153

\* armstrong number

\* ex:

\* number = 1634

\* 4 4 4 4

\* output: 1 + 6 + 3 + 4

\* 1 + 1296 + 81 + 256

\* 1634

\* armstrong number

\*

\*/

**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 = *isArmstrong*(number);

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

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

}

**else** {

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

}

}

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

**int** temp = number;

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

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

**int** sum = 0;

**while**( number > 0 ) {

**int** r = number % 10;

sum = sum + (**int**) Math.*pow*(r, len);

number /= 10;

}

**if** ( sum == temp )

**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 equal to the same number then

\* it is palindrome.

\* ex:

\* number = 123

\* output: not a palindrome

\* ex:

\* number = 121

\* output : palindrome

\*/

**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 = *isPalindrome*(number);

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

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

}

**else** {

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

}

}

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

**int** temp = number;

**int** reverse = 0;

**while** ( number > 0 ) {

**int** r = number % 10;

reverse = reverse \* 10 + r;

number = number / 10;

}

**if** ( reverse == temp )

**return** **true**;

**else**

**return** **false**;

}

}

/\*

\* write a program to find the sum of even

\* and odd position digits of a number separately.

\* ex:

\* number = 24087

\* evenPositionSum = 12

\* oddPositionSum = 9

\* ex:

\* number = 214753

\* evenPositionSum = 11

\* oddPositionSum = 11

\*/

**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();

*evenOddPositionSum*(number);

}

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

//convert the number into a string

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

//find the length of the string

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

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

**int** evenPositionSum = 0, oddPositionSum = 0;

**if** ( len % 2 == 0 ) {

flag = **true**;

}

**while** ( number > 0 ) {

**int** digit = number % 10;

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

evenPositionSum += digit;

flag = **false**;

}

**else** {

oddPositionSum += digit;

flag = **true**;

}

number = number / 10;

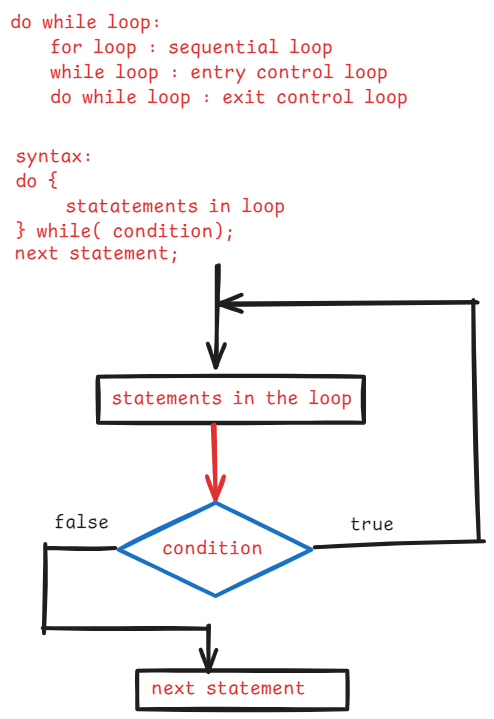
}

System.***out***.println("even position digits sum : " + evenPositionSum);

System.***out***.println("odd position digits sum : " + oddPositionSum);

}

}



/\*

\* write a program to do the following.

\* take a number between 0 to 10 from the user

\* generate a random number between 0 to 10.

\* verify random number and user input are matched.

\* if matched then display guess is correct, else

\* display guess is wrong.

\*/

import java.util.Random;

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

int guessedNumber=0;

do {

System.out.println("Guess a number between 0 to 10");

guessedNumber = scan.nextInt();

} while( guessedNumber < 0 || guessedNumber > 10 );

//generating random number between 0 to 10

Random r = new Random();

int randomNumber = r.nextInt(11); //it generates a random number b/w 0 to 10

if ( guessedNumber == randomNumber ) {

System.out.println("You guessed : " + guessedNumber);

System.out.println("Random number : " + randomNumber);

System.out.println("Yes, Your guess is correct!");

}

else {

System.out.println("You guessed : " + guessedNumber);

System.out.println("Random number : " + randomNumber);

System.out.println("No, Your guess is wrong!");

}

}

}

Jumping statements:

1. break
2. continue
3. return
4. exit

* break statement can be used in switch statement or in a loop.
* break statement is to stop the execution of a case or a loop.

for ex;

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

if(i==3) {

break;

}

S.o.p(i);

}

output: 1

2

ex:

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

{

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

{

if(i+j == 3) {

break;

}

S.o.p(j);

}

}

output: 1

1

2

3

ex:

lable1: for(int i=1; i<=3; i++)

{

lable2: for(int j=1; j<=3; j++)

{

if( i+j == 3) {

break label1;

}

S.o.p(j);

}

}

output: 1

continue:

* continue statement will skip the remaining statements execution in a loop and moves the control to the next iteration.

ex:

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

if(i==3)

continue;

S.o.println(i);

}

output: 1

2

4

5

ex:

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

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

if(j==2)

continue;

S.o.p(j);

}

}

output: 1

3

1

3

ex:

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

inner: for(int j=1; j<=3; j++) {

if(j==2)

continue outer;

S.o.p(j);

}

}

output: 1

1

return statement:

* return statement returns/moves the control from the current method to the calling point.
* return <value> returns the control and the value to the calling point.

ex1:

void m1() {

S.o.p(“hello”);

return; //valid

}

ex2:

int m2() { //error

S.o.p(“hello”);

}

ex3:

int m3() {

S.o.p(“hello”);

return true; //error

}

ex4:

boolean m4() {

S.o.p(“hello”);

return 1; //error

}

ex5:

int m5() { //error

S.o.p(1);

}

ex6:

int m6() {

return 1; //valid

}

ex7:

void m7() { //valid

S.o.p(“hi”);

}

exit statement:

* exit terminate the currently running JVM.
* exit() is a static method provided in System class.
* the status code in exit() can be zero/non-zero.
* status code zero indicates normal termination and non-zero indicates abnormal termination.

System.exit(0);

Arrays

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

* In a primitive variable, we can store one value at a time.
* If we want to store multiple values then we need to create multiple variables.
* If we more variables are created in a program, then complexity will be increased.
* The memory for the variables will not allocated sequentially in the JVM. So for each variable, to fetch the value, the JVM has to search the entire memory. This will decrease the performance of a program.
* So, we got a solution in the form of arrays.
* If we create an array variable then it can store multiple values. So we can reduce the number of variables. Hence, complexity is reduced.
* The memory is allocated for the array values sequentially in JVM. So performance will be improved.

creating an array:

1. array declaration
2. array creation

//array declaration

datatype[] array\_variable;

(or)

datatype array\_variable[] ;

//array creation

array\_variable = new datatype[size];

* we can also combine array declaration and array creation into a single statement.

datatype[] array\_variable = new datatype[size];

(or)

datatype array\_variable[] = new datatype[size];

ex:

int[] arr = new int[5];

* array index always starts at 0.
* if the array size is 5 then indexes of the elements are from 0 to 4.

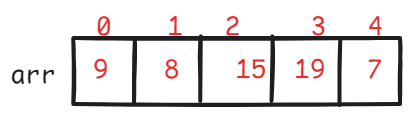
arr[0] = 9;

arr[1] = 8;

arr[2] = 15;

arr[3] = 19;

arr[4] = 7;



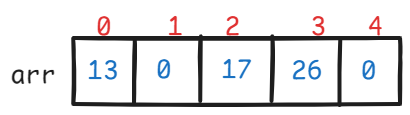
ex:

int arr[] = new int[5];

arr[0] = 13;

arr[2] = 17;

arr[3] = 26;



static array creation:

datatype[] array\_variable = new datatype[] { value1, value2, value3, . . . };

(or)

datatype[] array\_variable = { value1, value2, value3, . . . . };

ex:

int[] arr = { 29, 34, 11, 15, 86 };

Q) How to find the length of an array?

A) using length attribute, not using length() method.

ex: arr.length //correct

arr.length() //error

ex:

String str = “hello”;

str.length // error

str.length() // correct

Q) what are the limitations of an array?

A) 1. array has a fixed size. Once an array is created, we can’t increase/decrease the size.

2. array can store only homogenous elements(same type). But not hetrogeneous elements.(different type).

For ex, if we create an int[], it can only store the integer values.

/\*

\* write a program to find the sum of elements

\* of an array, by reading array size and the

\* elements from the user.

\*/

**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 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 value at arr[" + i +"]");

arr[i] = scan.nextInt();

}

*findSumOfArray*(arr);

}

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

**int** sum = 0;

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

sum = sum + arr[i];

}

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

}

}

/\*

\* write a program to find sum of even and odd

\* elements separately, by reading the array size

\* and the elements from the user.

\*/

**import** java.util.Scanner;

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

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

// **TODO** Auto-generated method stub

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

System.***out***.println("Enter the size of 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 value at arr[" + i +"]");

arr[i] = scan.nextInt();

}

*findSumOfEvenOddArray*(arr);

}

**private** **static** **void** findSumOfEvenOddArray(**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 array, by reading array size, elements and

\* the searching element from the user, using linear search

\*

\* linear search: It means, compare the searching element

\* with every element of the array. If

\* any element of the array is matched, then

\* element is found. Otherwise element is not

\* found.

\*/

**import** java.util.Scanner;

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

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

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

System.***out***.println("Enter the size of 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 value at arr[" + i +"]");

arr[i] = scan.nextInt();

}

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

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

*linearSearch*(arr, k);

}

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

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

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

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

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

flag = **true**;

**break**;

}

}

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

System.***out***.println("Element not found in the array!!!");

}

}

}

|| DATE: 2 – SEP- 24 ||

/\*

\* write a program to search for an element in the

\* array with binary search.

\* Binary search works like below:

\* 1. first sort the array in ascending order

\* 2. take the low and high indexes.

\* 3. calculate the mid index

\* 4. if element at mid point and searching element

\* are equal, then element is found and break the loop

\* 5. if element at mid point < searching element

\* then change low = mid + 1.

\* 6. if element at mid point > searching element

\* then change high = mid - 1.

\* 7. repeat steps 3 to 6, until element is found

\* or low > high.

\*/

**import** java.util.Arrays;

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

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

**int**[] arr = { 91, 32, 56, 29, 64, 12 };

//searching element

**int** k = 65;

*binarySearch*(arr, k);

}

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

//sort the array.

Arrays.*sort*(arr);

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

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

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

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

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

System.***out***.println("Element is found");

found = **true**;

**break**;

}

**else** **if**( arr[mid] < k ) {

low = mid + 1;

}

**else** {

high = mid - 1;

}

}

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

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

}

}

}

/\*

\* write a program to print the max consecutively

\* repeated element of the array.

\*

\* ex: int[] arr = { 1, 1, 2,2,2, 1,1, 5 };

\* output: element is : 2

\* max consecutively repeated for : 3 times

\*/

**public** **class** Test {

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

**int**[] arr = { 1, 1, 2, 2, 2, 1, 1, 5 };

*findMaxConsecutive*(arr);

}

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

**int** count = 1;

**int** maxi = 0, k = 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;

k = arr[i];

}

}

System.***out***.println("element is : " + k );

System.***out***.println("max consecutive times : " + maxi);

}

}

/\*

\* write a program to remove the duplicate

\* elements from the array.

\* example:

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

\* output:

\* { 3 , 1, 2 }

\*/

**import** java.util.Arrays;

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

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

**int**[] arr = { 10, 5, 10, 25, 5, 30, 10 };

*removeDuplicates*(arr);

}

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

Arrays.*sort*(arr);

**int** i = 0;

**for**(**int** j = 1; j < arr.length; j++ ) {

**if**( arr[i] != arr[j] ) {

i++;

arr[i] = arr[j];

}

}

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

System.***out***.println(arr[k]);

}

}

}

selection sort:

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

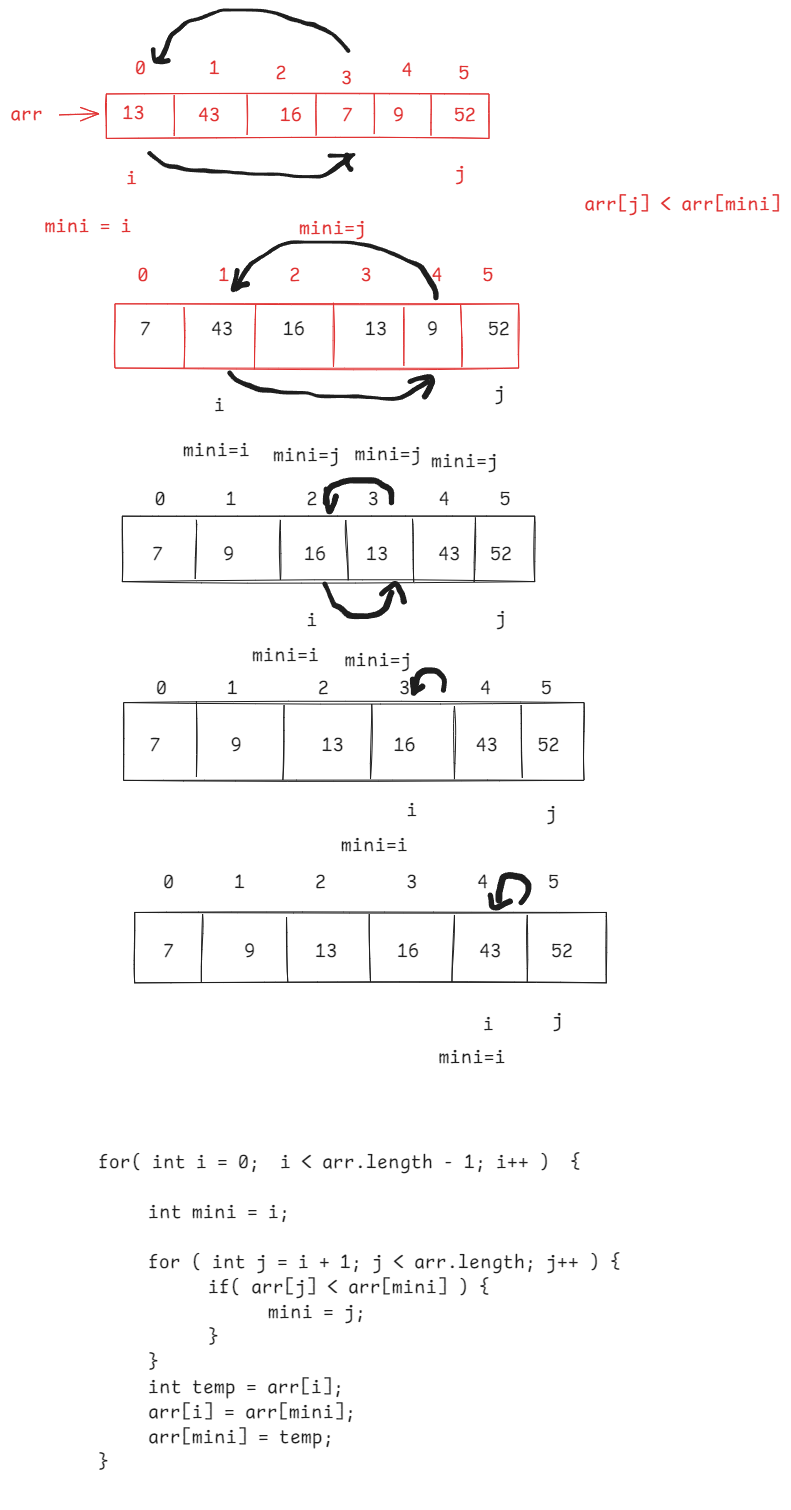
* In selection sort algorithm, first we need to find the

smallest element of the array, place it in the first position.

* next find the second smallest element of the array and place it

in the second position.

* continue this algorithm until n-1 elements are sorted.



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

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

**int**[] arr = { 34, 12, 15, 54, 8, 15 };

*selectionSort*(arr);

}

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

**for** ( **int** i = 0; i < arr.length - 1; i++ ) {

**int** mini = i;

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

**if** ( arr[j] < arr[mini] )

mini =j;

}

**int** temp = arr[i];

arr[i] = arr[mini];

arr[mini] = temp;

}

System.***out***.println("array elements after sorting........");

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

System.***out***.print( arr[i] + " ");

}

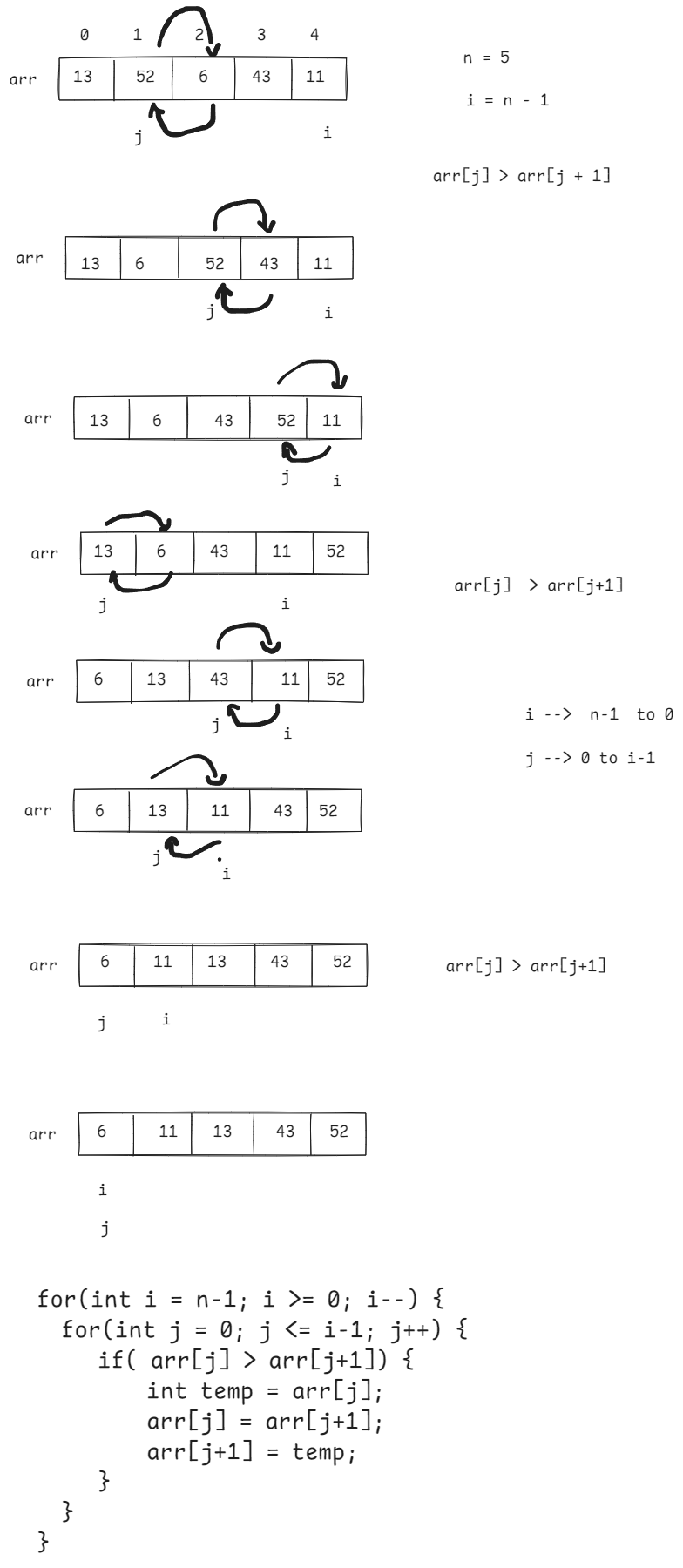
}

}

Bubble sort:

-----------

* In bubble sort, first find the highest element in array and place it at the last index.
* Next find the second highest element in array and place it at the second from last position.
* continue this, until all the elements are sorted.



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

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

**int** arr[] = { 13, 52, 7, 19, 29 };

*bubleSort*(arr);

}

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

**int** n = arr.length;

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

**for** ( **int** j = 0; j <= i - 1; j++ ) {

**if** ( arr[j] > arr[j+1] ) {

**int** temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

**if**( arr[i-1] > arr[i] ) {

**int** temp = arr[i-1];

arr[i-1] = arr[i];

arr[i] = temp;

}

}

System.***out***.println("array elements after sorting");

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

System.***out***.print(arr[i] + " ");

}

}

}

Two-dimentional array(Matrix):

datatype[][] arrayname = new datatype[rows][cols];

(or)

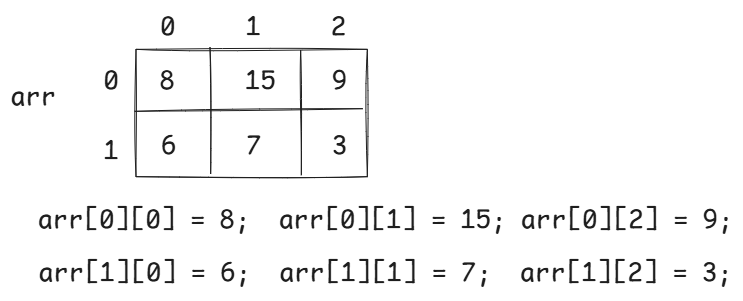
datatype arrayname[][] = new datatype[rows][cols];

(or)

datatype[] arrayname[] = new datatype[rows][cols];

for ex:

int[][] arr = new int[2][3];



for(int i = 0; i < 2; i++) {

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

System.out.print(arr[i][j]+” “);

}

System.out.println();

}

static 2-d array:

int[][] arr = {

{3, 0, 7},

{5, 9, 6},

{2, 7, 1}

};

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();

}

/\*

\* write a program to find the sum of

\* elements of the 2D array.

\*/

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

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

**int**[] arr[] = { {2, 3, 7}, {4, 5}, {1, 0, 9, 5} };

**int** sum = 0;

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

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

sum += arr[i][j];

}

}

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

}

}

/\*

\* write a program to find the sum of

\* left and right diagonal elements of

\* a 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 number of rows");

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

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

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

**if**( rows != cols ) {

System.***out***.println("Not a square matrix. So, diagonal"

+ "can't exist!!");

System.*exit*(0);

}

//create the array/matrix

**int**[][] arr = **new** **int**[rows][cols];

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

**for**( **int** j = 0; j < arr[0].length; j++) {

System.***out***.println("Enter the element at arr["+i+"]["+j+"]");

arr[i][j] = scan.nextInt();

}

}

*findDiagonalSum*(arr, rows, cols);

}

**private** **static** **void** findDiagonalSum(**int**[][] arr, **int** rows, **int** cols) {

**int** ldSum = 0, rdSum = 0;

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

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

**if** ( i == j ) {

ldSum += arr[i][j];

}

**if**( i + j == rows - 1 ) {

rdSum += arr[i][j];

}

}

}

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

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

}

}

/\*

\* write a program to find the sum of two

\* matrices.

\*/

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

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

**int**[][] a = { {3, 7, 5}, {1, 4, 8} };

**int**[][] b = { {1, 7, 4}, {2, 0, 9} };

**int**[][] c = **new** **int**[a.length][a[1].length];

*addMatrices*(a, b, c);

}

**private** **static** **void** addMatrices(**int**[][] a, **int**[][] b, **int**[][] c) {

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

**for** ( **int** j = 0; j < a[i].length; j++ ) {

c[i][j] = a[i][j] + b[i][j];

}

}

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

**for** ( **int** j = 0; j < c[i].length; j++ ) {

System.***out***.print(c[i][j] + " ");

}

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

}

}

}

/\*

\* write a program to find the multiplication

\* of two matrices.

\* multiplication condition: no of cols in matrix1 and no of rows in matrix2

\* must be same.

\*/

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

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

**int**[][] mat1 = { {2, 4, 0}, {7, 5, 1} }; // 2 \* 3 matrix

**int**[][] mat2 = { {1, 5}, {6, 2}, {0, 8} }; // 3 \* 2 matrix

*multiply*(mat1, mat2);

}

**private** **static** **void** multiply(**int**[][] mat1, **int**[][] mat2) {

**int** n = mat1.length; //no of rows in mat1

**int** m = mat1[0].length; //no of cols in mat1

**int** p = mat2.length; //no of rows in mat2

**int** q = mat2[0].length; //no of cols in mat2

**int**[][] mat3 = **new** **int**[n][q];

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

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

mat3[i][j] = 0;

**for** (**int** k = 0; k < p; k++) {

mat3[i][j] = mat3[i][j] + mat1[i][k] \* mat2[k][j];

}

}

}

System.***out***.println("Multiplication of the matrices : ");

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

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

System.***out***.print(mat3[i][j]+ " ");

}

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

}

}

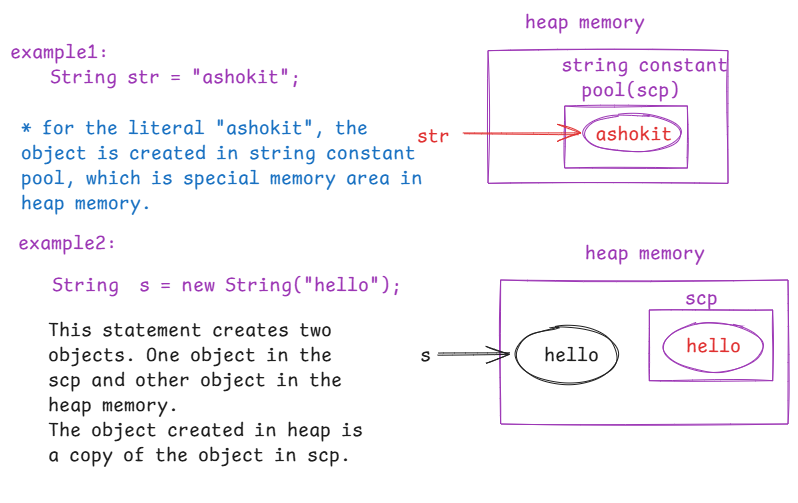
}

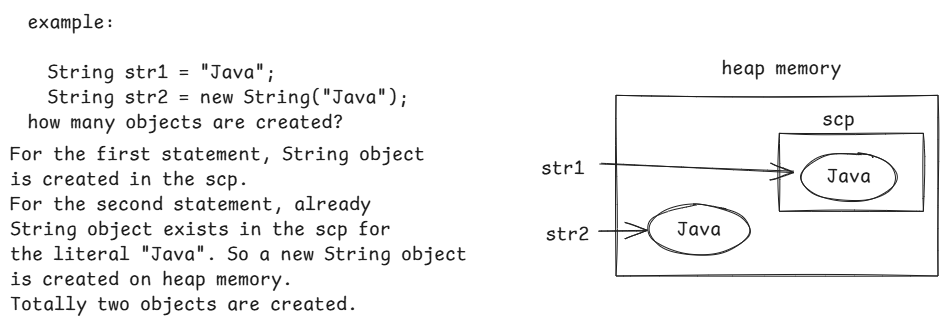
String handling

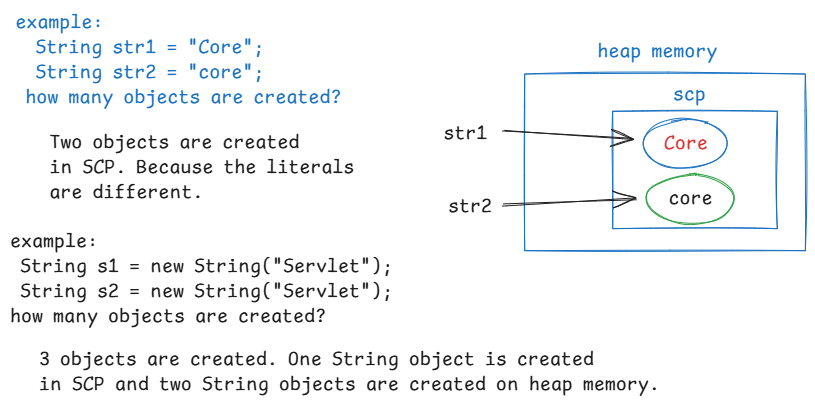
* A string is a sequence of characters enclosed within double quotes.
* A String object can be created in two ways.
* 1. using literal
* 2. using new keyword.

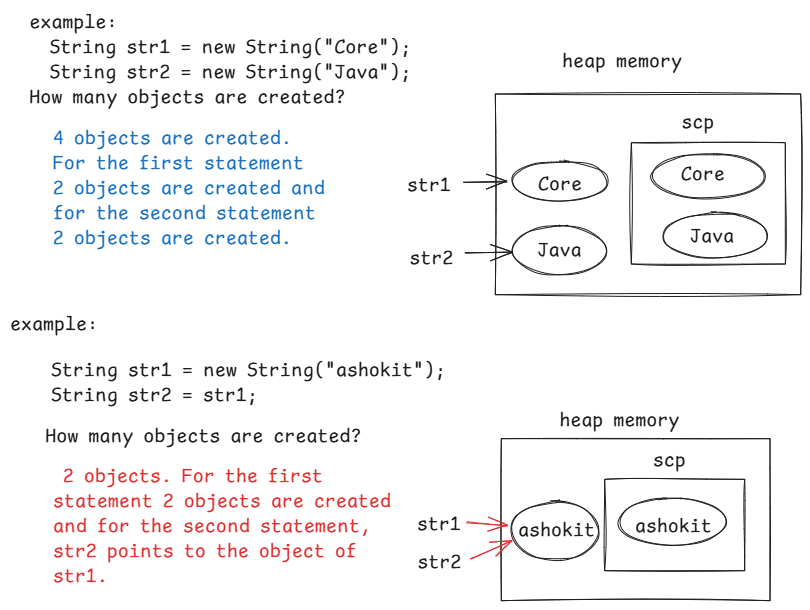
1. String str = "hello"; //using literal

2. String str = new String("hello"); //using new keyword









string operations:

1. length() : Returns no of characters in a string.

String str = new String(“Java”);

S.o.pln(str.length()); op: 4

S.o.pln(str.length); // error

String str[] = { “aaa”, “bbb”, “cccc” };

S.o.pln( str.length() ); //error

S.o.pln( str.length ); op: 3

S.o.pln( str[2].length ); //error

S.o.pln( str[2].length() ); op: 4

2. charAt(index): Returns character at the given

index.

If the given index is out of

range, then StringIndexOutOfBoundsException will be thrown.

ex:

String str = new String(“Core Java”);

S.o.pln(str.charAt(6)); // a

S.o.pln(str.charAt(-1)); //Exception

S.o.pln(str.charAt(11)); //Exception

3. indexOf(char): Returns the index of first occurance of a given character.

It returns -1, if the given character doesn’t exist.

ex:

String str = “ashokit”;

S.o.pln(str.indexOf(‘h’)); // 2

S.o.pln( str.indexOf(‘z’)); //-1

ex:

String str = “cat sat”;

S.o.pln( str.indexOf(“at”)); // 1

4. lastIndexOf(char): Returns the last occurrence of a given character. If not exist, returns -1.

String str = “cat sat”;

S.o.pln( str.lastIndexOf(“at”)); // 5

5. indexOf(char, beginIndex): Returns the index of a character, by starting the search from the beginIndex.

ex: String str = “cat sat on mat”;

S.o.pln( str.indexOf(‘a’, 3)); //5

S.o.pln( str.indexOf(‘a’)); // 1

S.o.pln( str.lastIndexOf(‘a’)); //12

6. substring(beginIndex, endIndex): Returns a portion of a given string. It returns a string from beginIndex to endIndex-1.

ex: String str = “ashokit Core Java session”;

S.o.pln(str.substring(8, 17)); //Core Java

7. substring(beginIndex): Returns a portion of a given string, from the begin index to the last character of the string.

ex: String str = “ashokit Core Java session”;

S.o.pln(str.substring(8)); //Core Java session

8. concat(string) : concatenates a given string to the existing string.

ex: Strig str = “Core”;

S.o.pln(str.concat(“Java”));//CoreJava

S.o.pln(str.concat(“ Java”));//Core Java

9. contains(string): retruns true, if the given string exists, in this string. Otherwise, returns false.

String str = “cat sat on mat”;

S.o.pln( str.contains(“dog”)); //false

S.o.pln(str.contains(“mat”)); //true

10. replace(oldChar, newChar): replaces old character with a new character in a given string.

String str = “tik tik”;

S.o.pln( str.replace(‘i’, ‘a’)); // tak tak

11. replace(oldString, newString): replaces old string with new string in a given string.

String str = “cat sat on mat. My cat is cute”;

S.o.pln(str.replace(“cat”, “dog”));

op: dog sat on mat. My dog is cute

12. replaceAll(regex, replacement): replaces the characters matching with regular expression, with the replacement.

ex:

String str = “admin@123”;

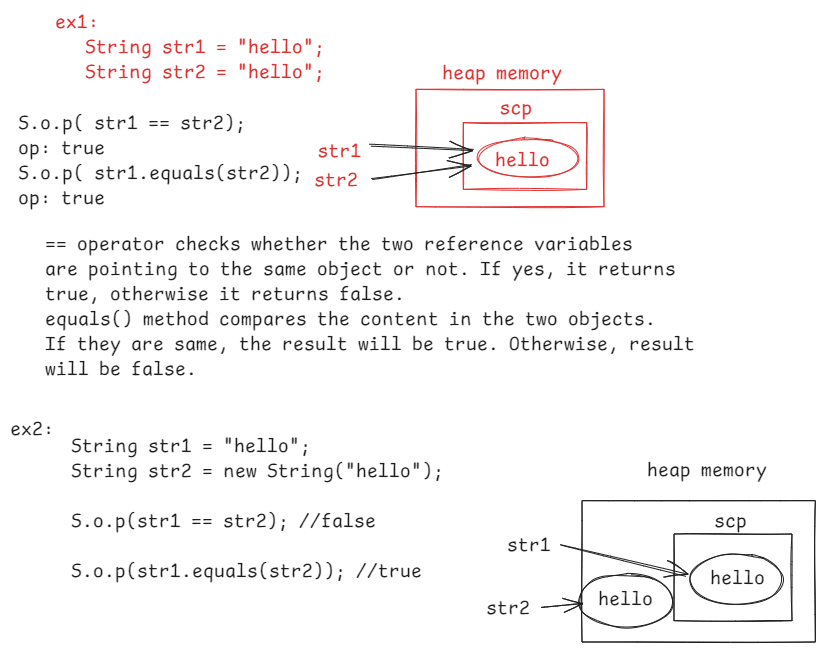
S.o.pln(str.replaceAll(“\\d”,”ashokit”));

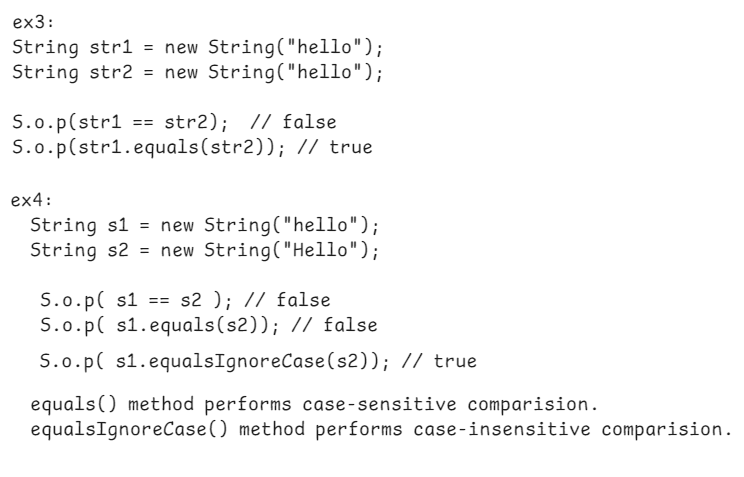
op: admin@ashokitashokitashokit

Note: \d : matches with 0 through 9

strings equality:

1. == operator
2. equals() method
3. equalsIgnoreCase() method
4. compareTo() method





compareTo() method compares the two strings lexicographically(alphabetical order).

If two strings are equals then returns 0.

if string1 is greater than string2 then returns >0

if string1 is less than string2 then retuns < 0.

String str1 = “hi”;

String str2 = “hi”;

if ( str1.compareTo(str2) == 0 )

S.o.p(“strings are equal”);

else

S.o.p(“strings are not equal”);

split() method:

* It will split/divide one string into multiple strings.

ex1:

String str1 = “hello, how are you”;

String str2[] = str1.split(“ “);

str2 ---> hello, how are you

0 1 2 3

String str3[] = str1.split(“,”);

str3 ----> hello how are you

0 1

ex2: String credentials = “username=admin;password=admin@123”;

String[] str = credentials.split(“;”);

str ----> username=admin password=admin@123

0 1

join() method:

* It combines multiple strings together into a single string.

ex: String[] strings = {“cat”, “sat”, “on”, “mat”};

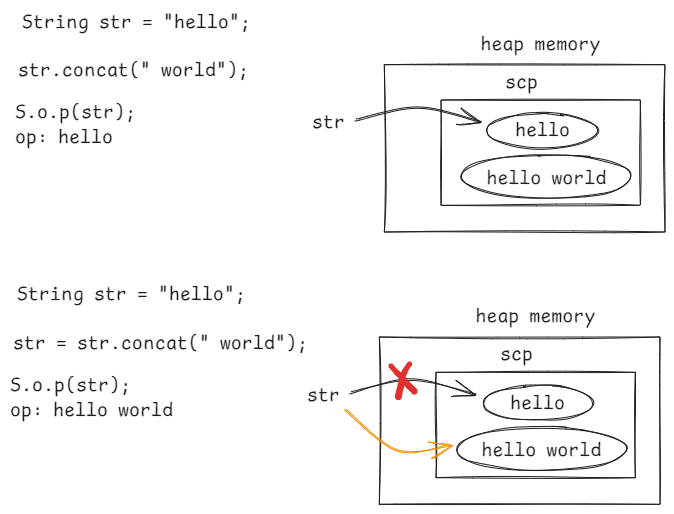
String str = String.join( “-“, strings);

S.o.p(str);

op: cat-sat-on-mat

string immutability:

* strings are immutable objects in Java.
* immutable object means, an object which does not allow any changes to its content, once it is created.
* If we make any changes like appending some value or deleting some characters or converting the case,etc.. on a string value, the result will be stored in new object.



toUpperCase() & toLowerCase() :

String str1 = “ashokit”;

S.o.p(str1.toUpperCase()); //ASHOKIT

S.o.p(str1); // ashokit

str1 = str1.toUpperCase();

S.o.p(str1); //ASHOKIT

trim() : removes white spaces before the first letter or after the last letter of a string value.

ex1: String str = “ ashokit”;

str = str.trim();

if(str.equals(“ashokit”)) {

S.o.p(“login success”);

}

else {

S.o.p(“login failed”);

}

ex2:

String str = “Spring Boot”;

str = str.trim();

S.o.p(str); //Spring Boot

toCharArray(): converts a string to character array.

ex: String str = “ashokit”;

char[] ch = str.toCharArray();

valueOf() :

* It is a static method of String class, and it converts a value from another datatype to the string type.

ex1: String str = String.valueOf(12) + String.valueOf(‘A’) + String.valueOf(3.75) + String.valueOf(true);

S.o.p(str);

op: 12A3.75true

String s = “175”;

int x = Integer.parseInt(s); //wrapping

wrapping converts a value from string type to primitive type.

primitive type wrapper class

int Integer

byte Byte

short Short

long Long

float Float

double Double

boolean Boolean

char Character

* To convert a String to a primitive type int,

call parseInt() method of Integer class.

* To convert a String to a primitive type float,

call parseFloat() method of Float class.

String s = “3.15f”;

float x = Float.parseFloat(s);

* To convert a String to a primitive type boolean, call parseBoolean() of Boolean class.

String str = “false”;

boolean b = Boolean.parseBoolean(str);

Note: we can’t convert a String to primitive type char, in this way.

contains() : Returns true, if the passed string exist in this string. Otherwise, returns false.

String str = “ashokit”;

str.contains(“it”); //true

str.contains(“java”); //false

startsWith() & endsWith() :

ex:

String str = “Java is awesome”;

S.o.p( str.startsWith(“java”)); //false

S.o.p( str.startsWith(“Java”)); //true

S.o.p(str.endsWith(“me”)); // true

S.o.p(str.endsWith(“m”)); //false

S.o.p(str.endsWith(“some”)); //true

/\*

\* write a program to swap the two strings,

\* without using a third variable.

\* ex: String str1 = "hello";

\* String str2 = "java";

\* S.o.p(str1); // java

\* S.o.p(str2); //hello

\*/

**import** java.util.Scanner;

**public** **class** SwappingStrings {

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

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

System.***out***.println("enter the first string");

String str1 = scan.nextLine();

System.***out***.println("enter the second string");

String str2 = scan.nextLine();

System.***out***.println("Strings before swapping");

System.***out***.println(" str1 = " + str1);

System.***out***.println(" str2 = " + str2);

//swapping without third variable

str1 = str1 + str2;

str2 = str1.substring(0, str1.length() - str2.length());

str1 = str1.substring(str2.length());

System.***out***.println("Strings after swapping");

System.***out***.println(" str1 = " + str1);

System.***out***.println(" str2 = " + str2);

}

}

/\*

\* write a program to count the number of

\* vowels and consonents in a given string

\* ex:

\* String str = "aDmin@123";

\* vowels count = 2

\* consonents count = 3

\* ex:

\* String str = "Missisipi";

\* vowels count = 4

\* consonents count = 5

\*/

**import** java.util.Scanner;

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

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

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

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

String str = scan.nextLine();

*findVowelsConsonentsCount*(str);

}

**private** **static** **void** findVowelsConsonentsCount(String str) {

str = str.toLowerCase();

**int** vCount = 0;

**int** cCount = 0;

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

**char** ch = str.charAt(i);

**if**(Character.*isLetter*(ch)) {

**if** ( ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') {

vCount++;

}

**else** {

cCount++;

}

}

}

System.***out***.println("vowels count = " + vCount);

System.***out***.println("consonents count = " + cCount);

}

}

/\*

\* write a program to check whether a given string

\* is palindrome or not.

\* ex1

\* str1 = "liril"

\* op: palindrome

\*

\* str2 = "ashokit"

\* op: not a palindrome

\*/

**import** java.util.Scanner;

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

**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) {

str = str.toLowerCase();

//convert the string to char array

**char**[] ch = str.toCharArray();

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

**for** (**int** i = 0, j = ch.length - 1 ; i < j ; i++, j-- ) {

**if** ( ch[i] != ch[j]) {

flag = **false**;

**break**;

}

}

**return** flag;

}

}

/\*

\* write a program to find the frequency of each

\* character in a given string.

\*

\* frequency means how many times the character

\* is repeated.

\*

\* for example,

\* str = "Missisipi"

\* output:

\* M - 1

\* i - 4

\* s - 3

\* p - 1

\*/

**import** java.util.Arrays;

**import** java.util.Scanner;

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

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

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

System.***out***.println("Please enter a string");

String str = scan.nextLine();

*findFrequency*(str);

}

**private** **static** **void** findFrequency(String str) {

//convert the string to lower case

str = str.toLowerCase();

//convert the string to character array

**char**[] ch = str.toCharArray();

//sort the array

Arrays.*sort*(ch);

**int** count;

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

count = 0;

**for** ( **int** j = i; j < ch.length; j++ ) {

**if**( ch[i] == ch[j]) {

i = j;

count++;

}

**else** {

**break**;

}

}

System.***out***.println(ch[i] + " - " + count );

}

}

}

/\*

\* write a program to check whether the given

\* two strings are anagrams or not.

\*

\* Anagrams means, the two strings shoule be permutation

\* of each other.

\* for example,

\* str1 = "listen"

\* str2 = "silent"

\* op: anagrams

\* str1 = "test"

\* str2 = "tast"

\* op: not anagrams

\*/

**import** java.util.Arrays;

**import** java.util.Scanner;

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

**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();

*checkAnagrams*(str1, str2);

scan.close();

}

**private** **static** **void** checkAnagrams(String str1, String str2) {

**if**( str1.length() == str2.length() ) {

str1 = str1.toLowerCase();

str2 = str2.toLowerCase();

**char** ch1[] = str1.toCharArray();

**char** ch2[] = str2.toCharArray();

Arrays.*sort*(ch1);

Arrays.*sort*(ch2);

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

**for**( **int** i = 0; i < ch1.length; i++ ) {

**if** ( ch1[i] != ch2[i] ) {

flag = **false**;

**break**;

}

}

**if**(flag) {

System.***out***.println("Both the strings are anagrams");

}

**else** {

System.***out***.println("Both the strings are not anagrams");

}

}

**else** {

System.***out***.println("Both the strings are not anagrams");

}

}

}

write a program to remove a character from a string at a given index.

ex:

str = “administrator”

index = 5

output: adminstrator

write a program to reverse each word of a given string.

ex:

str = “hello admin”

output:

olleh nimda

Q) why String is given as immutable?

A) String is designed in Java as immutable, for the below reasons.

1. memory efficiency

2. thread safety

3. security

\* Java uses, String Constant pool to store the strings. If a string already exists, then it reuses the existing string instead of creating a new one.

\* Strings can be shared between threads and since strings can not be modified, multiple threads can work with the same string object without causing any data inconsistency.

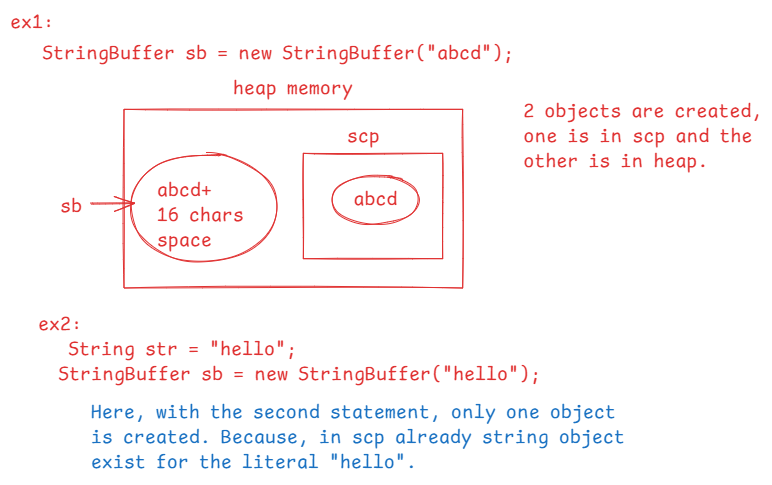
\* The sensitive informations like urls, username or passwords are all strings. If strings are mutable then some part of the code could alter these values during execution and it leads to some security problems. So, strings are given as immutable objects.

StringBuffer class:

* StringBuffer objects are mutable and thread safe objects.
* Whenever a StringBuffer object is created, additional 16 chars space is also created in the object, to allow the changes to the content.
* In StringBuffer class, the methods are synchronized methods, which means, they allow only one thread at a time to do the changes to the StringBuffer object.
* A StringBuffer object can be created with new keyword only.

StringBuffer sb = “abcd”; //error

StringBuffer sb = new StringBuffer(“abcd”);//ok



length and capacity:

length(): returns the number of characters in a StringBuffer object.

capacity(): returns the number of characters+16

ex:

StringBuffer sb =**new** StringBuffer("abcdef");

System.***out***.println(sb.length()); // 6

System.***out***.println(sb.capacity()); // 22

* Some of the methods of StringBuffer class,

1. append()
2. charAt(index)
3. indexOf(char)
4. lastIndexOf(char)
5. deleteCharAt(index)
6. delete(start, end)
7. substring(start,end)
8. reverse()
9. toString()

10.insert(index, value)

How do you compare the two StringBuffer objects?

StringBuffer sb1 = new StringBuffer(“hello”);

StringBuffer sb2 = new StringBuffer(“hello”);

S.o.p( sb1 == sb2); //false

S.o.p( sb1.equals(sb2) ); //error

we don’t have equals() method in StringBuffer class

S.o.p( sb1.toString().equals(sb2.toString());//true

StringBuilder class:

* StringBuilder objects are mutable and not a thread safe objects.
* Whenever a StringBuilder object is created, additional 16 chars space is also created in the object, to allow the changes to the content.
* In StringBuilder class, the methods are not synchronized methods, which means, they allow multiple threads at a time to do the changes to the StringBuilder object.

A StringBuilder object can be created with new keyword only.

StringBuilder builer=”abcd”; //error

StringBuilder builder=new StringBuilder(“abc”);

* This class is designed for use as a drop-in replacement for StringBuffer in places where the string buffer was being used by a single thread.
* StringBuffer and StringBuilder classes have exactly the same methods.

Q1) what is the difference b/w String and StringBuffer?

1. String is an immutable and a thread-safe object. But StringBuffer is a mutable and a thread-safe object.

Q2) what is the difference b/w StringBuffer and StringBuilder?

1. StringBuffer is mutable and thread-safe object, but StringBuilder is mutable and not a thread-safe object.

OOPS

(Object Oriented Programming System)

* To develop a s/w application, first we need to choose one of the two programming models.
* 1. Procedure Oriented Programming model.
* 2. Object Oriented Programming model.
* In Procedure Oriented Programming(POP) model, an application can be developed by creating a special function called main and the other functions.
* The data to the other functions will be provided from the main function.
* The data is in main function and the logics are in other functions. So, understanding the application is difficult.
* Finding the errors and fixing the bugs is also difficult and applications becomes platform dependent.
* POP model is suitable for creating small applications.
* The programming languages like C and Pascal are procedure oriented languages.
* In Object Oriented Programming(OOP) model, we can define the data and its related functionality at one place by creating classes.
* So, it is convinient for the programmers to understand the application.
* With OOP model, finding the mistakes and fixing the bugs is easy.
* For developing large applications, we have to choose OOP model.
* The programming languages like Java, Python, C++, .Net etc.. are OOP languages.
* The OOP System has defined 4 principles and if a language provides a way to implement these principles in the software, then it is called OOP language.
* 1. abstraction
* 2. encapsulation
* 3. inheritance
* 4. polymorphism

abstraction:

abstraction is a process of hiding unnecessary details and showing only essential details to the user of a system.

For example, When we receive a phone call, we will get the caller number and the location, which is essentail details. But the other details like tower locations are hidden and not shown to the user. This is called abstraction.

For example, when you want to call a pre-defined method of Java API, the details required are the method name, its return type and parameters are shown to the user. The logic of the methods are hidden. This is called abstraction.

while developing an application, the abstraction can be implemented in Java, using abstract classes and interfaces concept.

encapsulation:

Encapsulation is process of combining the data and the related functionality on the data together at one place and restrictiong un-authorized access to the data.

This encapsulation can be implemented by creating a class with private variables and public setter and getter methods.

Inheritance:

Inheritance is a process of creating new classes from the existing classes.

The new class is called child class and the existing class is called parent class.

The child class inherits the properties and the behaviour from the parent class.

With inheritance, we can get code reusability, the code duplication(redundancy) is reduced and improves a developers producitivity.

polymorphism:

* Poly means many and morphos means forms.
* Polymorphism denotes many forms of a task.
* A language has to provide a way to implement one task in many ways.
* polymorphism can be implemented by using 2 mechanisms.
* 1. method overloading
* 2. method overriding

class & object:

* class is keyword which is taken from another word called classification.
* a class is a classification for a group of objects with common attributes and the common behaviour.
* For example, if we take 10 employees, then have common attributes like empno,name,salary,.. and common behaviour like login, do the work, submit the work and logout. So, we can classify these employees into one class.
* For example, if we take 10 accounts, they have common attributes like acno, account holder, account type, balance and they have common behaviour like deposit and withdraw. So, we can classify them into one class.
* A class is a template or a blueprint for creating a group of objects with similar attributes and similar behaviour.

How to create a class:

class <classname> {

variables;

methods;

}

example:

class Account {

long accno;

double balance;

void deposit(double amount) {

//logic

}

void withdraw(double amount) {

//logic

}

}

object:

* every real-world entity is called an object.
* an object is an instance of a class.
* when an object is created, a class becomes to reality.
* By creating a class, we can maintain the attributes and the related operations together at one place.
* Every object contains state and behaviour.
* state means, the data stored into the attributes/variables of an object.
* behaviour means, operations that an object can perform.

How to create an object:

classname objectname = new classname();

ex: Account account1 = new Account();

* The right side part of the above syntax, creates an object in the heap memory and also invokes the constructor.
* At left side, we are storing the object into a reference variable, which is called object name.
* The classname at left side, represents type of the reference variable.
* Every class we create, is a user-defined data type.
* To perform operations on an object, it must be stored into a reference variable.

How to invoke a method?

referencevariable.methodname(parameters);

ex:

account1.deposit(10000);

/\*

\* write a program to create a class Account with

\* attributes accno and balanace.

\* Define two methods to deposit and withdraw the given amount.

\*/

**class** Account {

**long** accno;

**double** balance;

**void** deposit(**double** amount) {

**if**(amount > 0) {

balance = balance + amount;

System.***out***.println("After deposit, the new balance : "+ balance);

}

**else** {

System.***out***.println(amount + "can't be deposited");

}

}

**void** withdraw(**double** amount) {

**if**(amount <= 0 || amount > balance) {

System.***out***.println(amount + " can't be withdrawn");

}

**else** {

balance = balance - amount;

System.***out***.println("After withdraw, the new balance : " + balance);

}

}

} //end Account class

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

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

Account account1 = **new** Account();

account1.accno = 1010118L;

account1.balance = 8000.0;

account1.deposit(4000.0);

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

account1.withdraw(9000.0);

}

}

/\*

\* write a program to create a class Policy with

\* attributes policyId and basePremium

\* define a method calculateFinalPremium with age parameter

\* and add 10% of basePremium if age <= 50, otherwise add 5% of

\* basePremium. Display the final premium amount.

\*/

**class** Policy {

//data members or instance variables

**long** policyId;

**double** basePremium;

//method

**void** calculateFinalPremium(**int** age) {

//local variable

**double** finalPremium = 0;

**if** ( age <= 50 )

finalPremium = basePremium + basePremium \* 0.10;

**else**

finalPremium = basePremium + basePremium \* 0.05;

System.***out***.println("Final premium : " + finalPremium);

}

}

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

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

Policy p1 = **new** Policy();

p1.policyId = 1988901L;

p1.basePremium = 6890.0;

//call the method

p1.calculateFinalPremium(45);

}

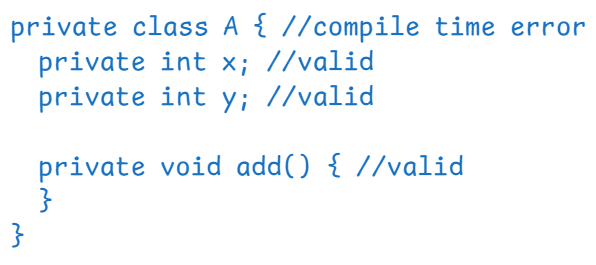
}

Access modifiers in Java:

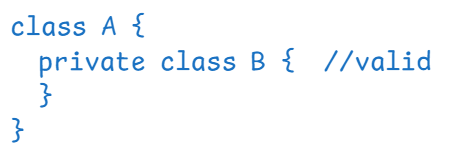
* Access modifier controls the visibility of a variable/method/class.
* We have 4 access modifiers.

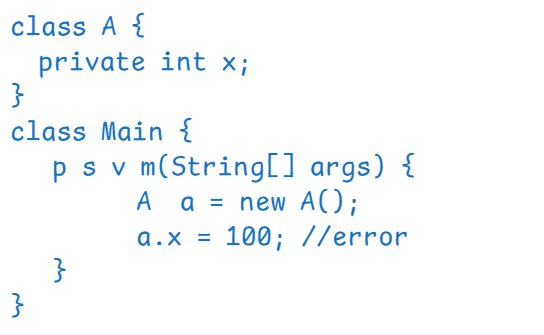
1. private
2. default / package-private
3. protected
4. public

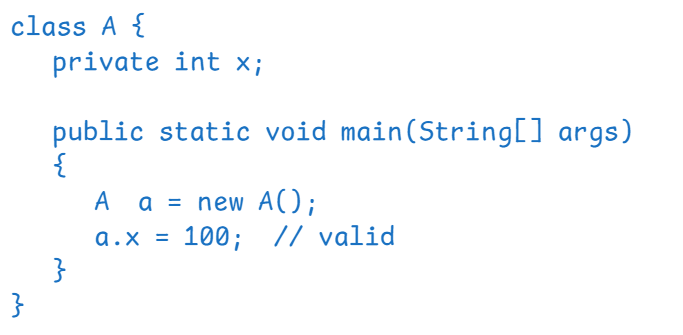
* private: If a variable/method/a class is declared as private, then it is visible within that class only.



* direcly, we can’t declare a class as private.

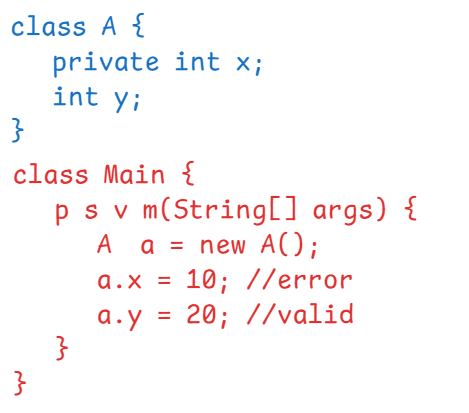


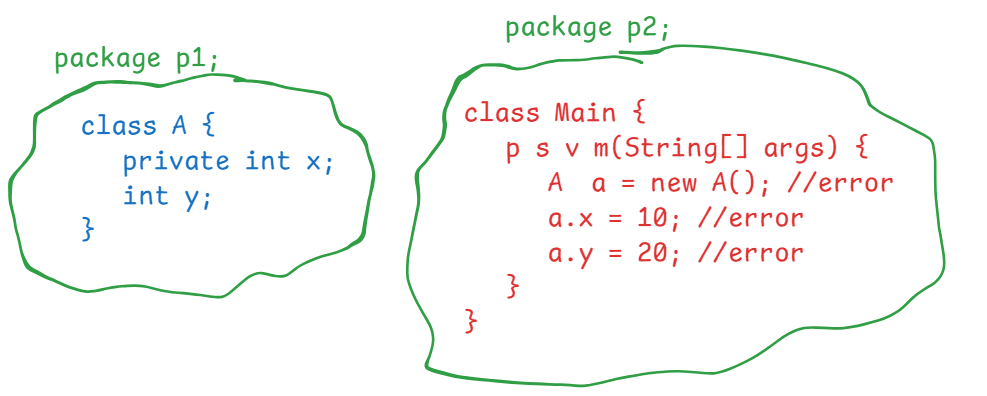




default: If a variable/method/class is declared as default then it is visible within the same package.

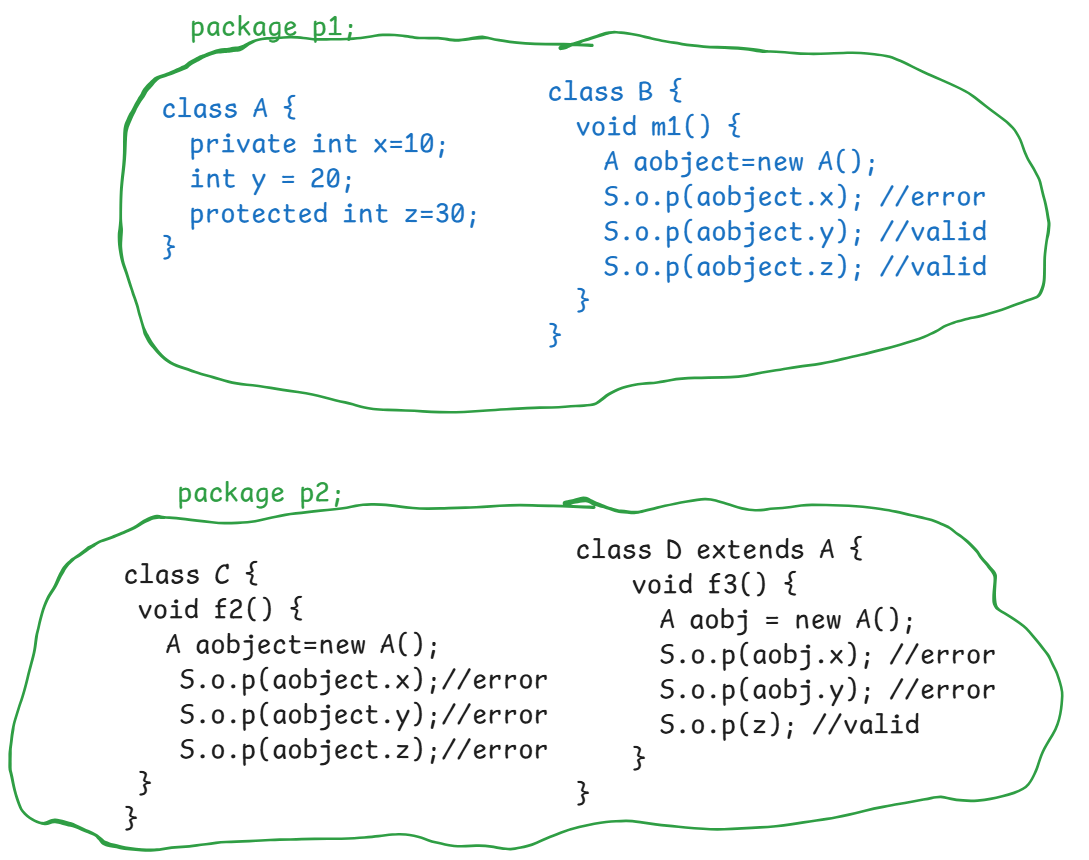
* If we don’t write any other access modifier for a variable/method/class then it is default.





protected:

If a variable/method/class is declared as protected then it is visible with in the same package and also visible in the child classes of other packages.

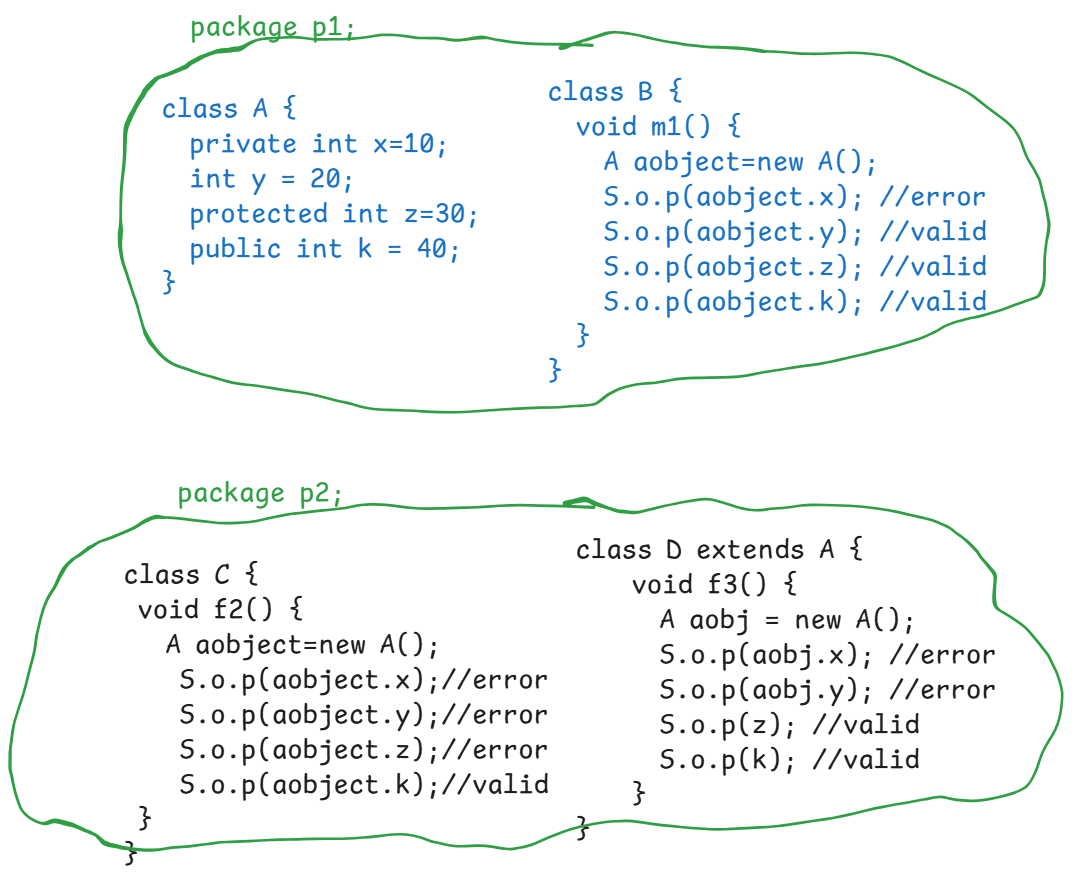


Note: we can’t add private or protected access modifiers for the outer class.

For outer class, the access modifier can be default or public.

For inner class, we can add either private/default/protected/public access modifier.

public: If a varaible/method/class is declared as public then it is globally visible in the same package classes and also in other package classes.



constructor

* A constructor is a special method of a class, which is used to initialize the data members of an object whenever the object is created.
* In a class, if we don’t write a constructor, then java compiler will add a default constructor and that default constructor will initialize the data members with default values based on their data type.
* If we write a constructor in a class, then java compiler will not add the default constructor.
* To write a constructor in a class,

1. classname and the constructor name both must be same.
2. constructor should not contain the return type.

ex1:

**class** Product {

**private** **int** id;

**private** String name;

**private** **double** price;

**void** showProduct() {

System.***out***.println("id :"+ id);

System.***out***.println("name :"+ name);

System.***out***.println("price :"+ price);

}

}

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

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

Product product = **new** Product();

product.showProduct();

}

}

output:

id : 0

name : null

price : 0.0

* In this example, we have not defined constructor in the Product class. So, the Java compiler has added a default constructor.

ex2:

**class** Product {

**private** **int** id;

**private** String name;

**private** **double** price;

//constructor

Product() {

id=101;

name="Mobile";

price=21000.0;

}

**void** showProduct() {

System.***out***.println("id :"+ id);

System.***out***.println("name :"+ name);

System.***out***.println("price :"+ price);

}

}

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

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

Product product = **new** Product();

product.showProduct();

}

}

output:

id : 101

name : Mobile

price : 21000.0

* In this example, we are writing a constructor, so Java compiler will not add a default constructor.
* In the main method, we have a statement

new Product(), it will create the object and also calls the constructor.

Types of constructors:

1. parameter-less constructor
2. parameterized constructor

* If you want to initialize multiple objects with same data or if you want to initialize an object with static information then you define parameter-less constructor.
* If you want to initialize an object with dynamic information, or if you want to initialize each object with different data then you define parameterized constructor.
* In a class we can define multiple constructors also.

ex:

**class** Product {

**private** **int** id;

**private** String name;

**private** **double** price;

//parameter-less constructor

Product() {

id=101;

name="Mobile";

price=21000.0;

}

//parameterized constructor

Product(**int** x, String y, **double** z) {

id = x;

name=y;

price=z;

}

**void** showProduct() {

System.***out***.println("id :"+ id);

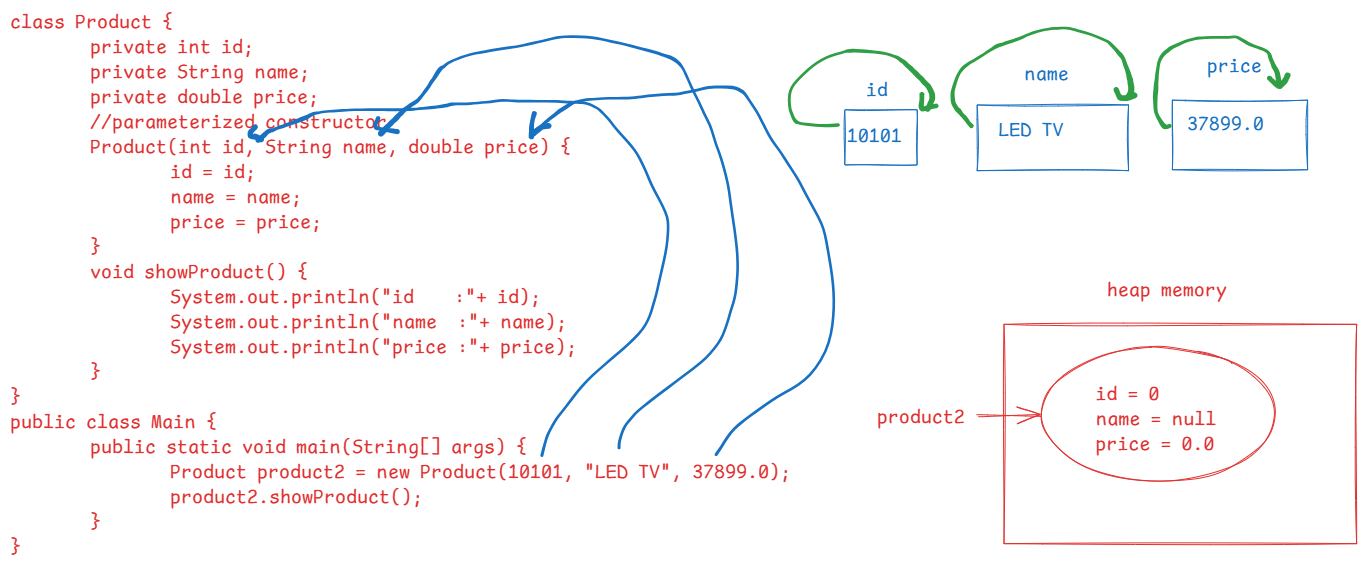
System.***out***.println("name :"+ name);

System.***out***.println("price :"+ price);

}

}

this keyword:



In the above diagram, the local varibles names and the instance varibles names are same.

In the constructor, the local variable value is again assigned to the local variable only.

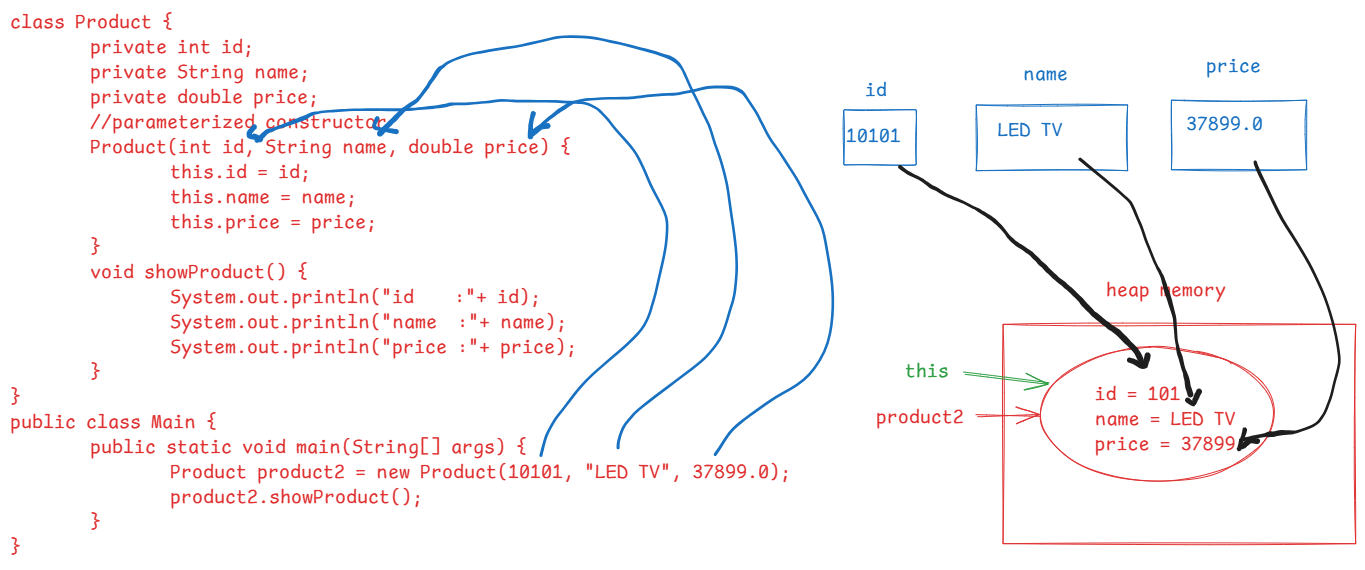
So, if we execute the above program, the output will be,

id : 0

name : null

price : 0.0

* If local variables and instance variables are having same names, then to distinguish the instance variable from the local variable, we use this keyword.
* The keyword this, refers the current object of a class.

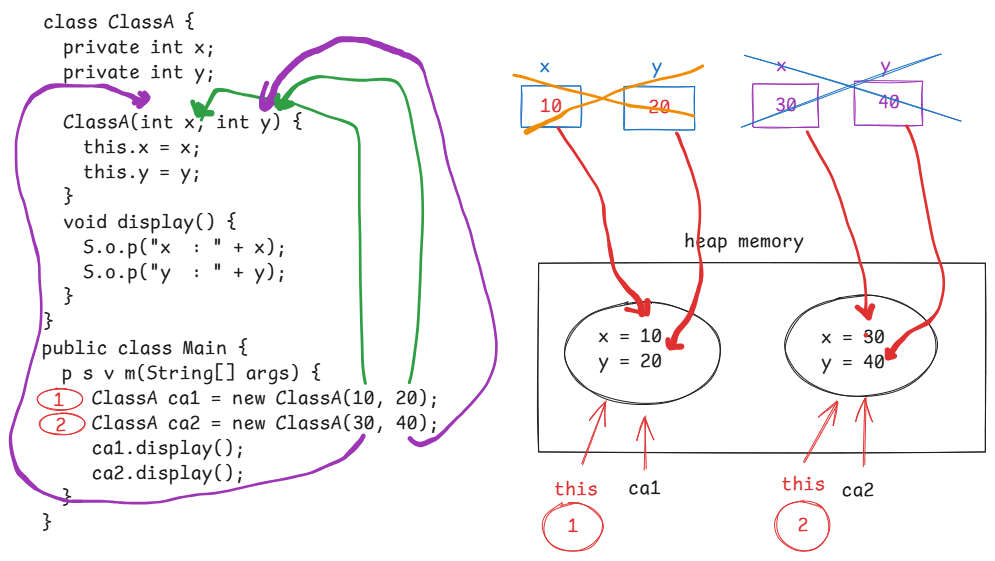


output:

id : 101

name : LED TV

price : 37899.0



this() call:

* It is used to call one constructor from another constructor of the same class.

**class** A {

**private** **int** x;

**private** **int** y;

//parameter-less constructor

A() {

**this**(15, 25); //this call statement is calling parameterized constructor

}

//parameterized constructor

A(**int** x, **int** y) {

**this**.x = x;

**this**.y = y;

}

**void** display() {

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

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

}

}

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

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

A aobject = **new** A();

aobject.display();

}

}

output:

x = 15

y = 25

**class** A {

**private** **int** x;

**private** **int** y;

//parameter-less constructor

A() {

System.***out***.println("I am parameter-less constructor");

}

//parameterized constructor

A(**int** x, **int** y) {

**this**(); // it is calling parameter-less constructor

**this**.x = x;

**this**.y = y;

}

**void** display() {

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

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

}

}

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

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

A aobject = **new** A(55, 65);

aobject.display();

}

}

output:

I am parameter-less constructor

x = 55

y = 65

How to define setter and getter methods:

ex1:

private int x;

public void setX(int x) {

this.x = x;

}

public int getX() {

return x;

}

ex2:

private String name;

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

ex3:

private int policyId;

public void setPolicyId(int policyId) {

this.policyId = policyId;

}

public int getPolicyId() {

return policyId;

}

ex4:

private boolean isActive;

public void setActive(boolean isActive) {

this.isActive = isActive;

}

public boolean isActive() {

return isActive;

}

Q) why should I declare variables/data members of a class as private?

A) To avoid storing invalid data into the instance variables from outside the class, we declare the variables as private.

Q) How to set/get the data from private variables?

A) By defining setter and getter methods.

Q) How to prevent setting invalid data by calling setter method?

A) by defining the logic in setter method.

For example:

**public** **void** setSalary(**double** salary) {

**if**(salary < 0) {

**throw** **new** RuntimeException();

}

**this**.salary = salary;

}

Inner classes:

* Defining one class in another class as a member is called inner class.
* If two classes are closely related classes and the other class can’t work independently without this class, then instead of writing them as separate classes, we can define the other class inside this class.

For ex:

class Mobile { //outer class

//data members

//methods

class Sim { //inner class

//data members

//methods

}

}

* By creating inner classes, we can organize the code, improves readability and we get better encapsulation.
* Outer class private variables are visible to the inner class, but inner class private variables are not visible to the outer class.

**class** OuterClass {

**private** **int** x = 10;

**class** InnerClass {

**private** **int** y = 20;

**public** **void** innnerClassShow() {

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

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

}

} //end of inner class

**public** **void** outerClassShow() {

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

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

}

} //end of outer class

* Inner class object can be created in outer class, or at outside of the outer class also, if the inner class is not a private class.

ex:

**class** OuterClass {

**private** **int** x = 10;

**class** InnerClass {

**private** **int** y = 20;

**public** **void** innnerClassMethod() {

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

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

}

} //end of inner class

**public** **void** outerClassMethod() {

//creating inner class object in outer class

InnerClass ic = **new** InnerClass();

}

} //end of outer class

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

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

OuterClass oc = **new** OuterClass();

//creating inner class object at outside of outer class

OuterClass.InnerClass ic = oc.**new** InnerClass();

}

}

Note: outer class can have default or public access modifier. But inner class can have any access modifier.

* Inner classes are four types.

1. non-static inner class
2. static inner class
3. local inner class
4. anonymous inner class.

* non-static inner class can access non-static and static data memebers of the outer class.
* static inner class can access only static data members of the outer class.

ex:

**class** OuterClass {

**private** **int** x = 10;

**private** **static** **int** *k* = 100;

**static** **class** InnerClass { //static inner class

**private** **int** y = 20;

**public** **void** innnerClassMethod() {

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

System.***out***.println("k = " + *k*); //valid

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

}

} //end of inner class

**public** **void** outerClassMethod() {

//creating inner class object in outer class

InnerClass ic = **new** InnerClass();

}

} //end of outer class

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

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

OuterClass oc = **new** OuterClass();

//creating inner class object at outside of outer class

OuterClass.InnerClass ic = **new** OuterClass.InnerClass();

}

}

* Local inner class is nothing but, a class which is defined inside the method of outer class.
* The local inner class is visible within that method only.

for example,

**class** OuterClass {

**public** **void** outerClassMethod() {

**class** InnerClass { //local inner class

**private** **int** a = 10;

**public** **void** display() {

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

}

} //end of local inner class

InnerClass ic = **new** InnerClass(); //inner class object

} //end of outerClassMethod

} //end of outer class

static keyword in Java:

* static is a non-access modifier and it can be used with,

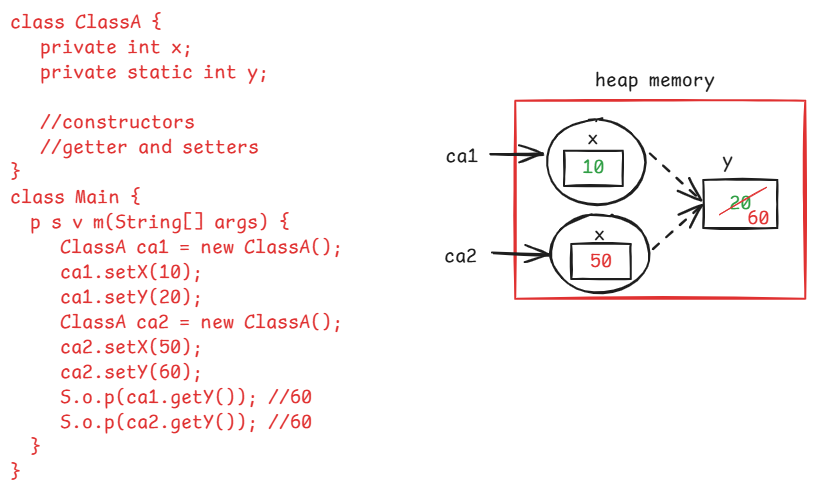
variables

methods

inner classes and

blocks.

* In a class, we can create instance variables and also static variables.
* if you want to share the data of a variable across multiple instances/objects of a class, then we have to create a static variable.
* If you create a static variable, then the memory for the variable is created at outside of the objects of a class, and it is shared to all the objects of that class.



* If a variable is static then the memory is allocated when the class is loaded into the JVM. But not at the time of object creation.
* If a variable is instance variable then the memory is allocated when the object is created for the class.
* If a static variable is modified through one object then the changed value is reflected to all the other objects.

static method:

* If a method is going to perform a task by without depending on the data of any object, then we can declared that method as a static method.
* If a method is a static method then it can be called with the classname directly.
* If we call a static method with the object name, then at runtime, JVM will replace the object name with the classname.
* A static method can also have any access modifier like private/default/protected/public.
* A static method allows only static variables of a class, but instance variables are not allowed.

/\*

\* write a program to count the number of instances

\* created for a class.

\*/

**class** User {

**int** userId;

**static** **int** *count*;

User(**int** userId) {

**this**.userId = userId;

*count*++;

}

**public** **static** **int** getCount() {

**return** *count*;

}

}

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

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

User user1 = **new** User(101);

User user2 = **new** User(102);

User user3 = **new** User(103);

User user4 = **new** User(104);

System.***out***.println("count of objects = " + User.*getCount*());

User user5 = **new** User(105);

User user6 = **new** User(106);

System.***out***.println("count of objects now = " + User.*getCount*());

}

}

static block:

* static block is used to initialize the static variables of a class.
* We can also initialize the static variables in a constructor. But, for each object creation, the static variable is re-initialized again and again.
* So, we can define a static block in a class.
* when the class is loaded into the JVM, static block is executed.
* static block can initialize only static variables, it doesn’t allow instance variables.
* some times, we need to execute some code to initialize the static variables. So, we can define that code in the static block.
* When a class is loaded into the JVM, first static variables memory is allocated and then after static block is executed.
* we can define multiple static blocks also in a class. In this case, the blocks are executed in the same order how they are defined in the class.

ex:

class A {

static {

S.o.p(“static block1”);

}

A() {

S.o.p(“constructor”);

}

static {

S.o.p(“static block2”);

}

}

class Main {

psvm(String[] args) {

}

}

output:

static block1

static block2

Q) what is the difference between a constructor and a static block?

A) 1. constructor is used to initialize the instance varaibles. But static block is used to initialize the static variables.

2. In a constructor I can use instance variables and static variables. But in a static block, I can use only static variables.

3. constructor is exectued when the object is created. But static block is executed when the class is loaded.

4. constructor can accept parameters. But static block can’t accept parameters.

5. this keyword is allowed in constructor. But it is not allowed in static block.

/\*

\* create a class called Booking with the below

\* attributes/fields.

\* bookingId

\* bookingDate

\* seatsRequired

\* seatsAvailable

\* Define methods to book/cancel the seats.

\* In main method, create one or more objects for

\* Booking class, invoke the methods.

\*

\*/

**import** java.time.LocalDate;

**import** java.time.LocalDateTime;

**import** java.util.Random;

**class** Booking {

**private** **long** bookingId;

**private** LocalDateTime bookingDate;

**private** **int** seatsRequired;

**private** **static** **int** *seatsAvailable*;

**static** {

//intiailizing the static variable here

*seatsAvailable* = 10;

}

//parameterized constructor

Booking(LocalDateTime bookingDate, **int** seatsRequired) {

**this**.bookingDate = bookingDate;

**this**.seatsRequired = seatsRequired;

}

**public** **void** bookSeats() {

//local variable

**boolean** isBooked = **false**;

**if** ( **this**.seatsRequired <= *seatsAvailable* ) {

*seatsAvailable* = *seatsAvailable* - seatsRequired;

isBooked = **true**;

}

**if** (isBooked) {

Random random = **new** Random();

**this**.bookingId = random.nextLong(28776199);

System.***out***.println("Booking successful! Your booking id : "+ **this**.bookingId);;

System.***out***.println("The seats remaining after booking : " + Booking.*fetchAvailableSeats*());

System.***out***.println("Booking Date&Time : " + **this**.bookingDate);

System.***out***.println("Seats booked : " + **this**.seatsRequired);

}

**else** {

System.***out***.println("Sorry, seats are not available!!");

}

}

**public** **void** cancelSeats() {

*seatsAvailable* = *seatsAvailable* + **this**.seatsRequired;

System.***out***.println("seats are cancelled!!");

System.***out***.println("The available seats after cancellation : " + Booking.*fetchAvailableSeats*());

}

**public** **static** **int** fetchAvailableSeats() {

**return** *seatsAvailable*;

}

}

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

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

//create Booking object

Booking booking1 = **new** Booking(LocalDateTime.*now*(), 6);

booking1.bookSeats();

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

//creating another Booking object

Booking booking2 = **new** Booking( LocalDateTime.*now*(), 7 );

booking2.bookSeats();

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

booking1.cancelSeats();

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

booking2.bookSeats();

}

}

output:

Booking successful! Your booking id : 26442144

The seats remaining after booking : 4

Booking Date&Time : 2024-09-27T10:28:26.064752400

Seats booked : 6

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

Sorry, seats are not available!!

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

seats are cancelled!!

The available seats after cancellation : 10

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

Booking successful! Your booking id : 24217047

The seats remaining after booking : 3

Booking Date&Time : 2024-09-27T10:28:26.080700900

Seats booked : 7

Q)How many types of variables are there in Java?

A) 3 types.

1. static variables

2. instance variables

3. local variables

Q) what is the difference between static variable and instance variable?

A)

Q) what is the difference between instance variable and a local variable?

A)

Inheritance

* Inheritance is a process of creating new classes from existing classes.
* Suppose, if you want to create class and if you find that some/all of the variables or methods are already exist in another class, then you can create the new class by inheriting from the existing class.
* This exising class becomes parent class/super class/base class and the new class becomes child class/sub class/derived class.
* With inheritance, we can achieve

. code reusability

. reduces redundancy

. improves productivity

. increases readability

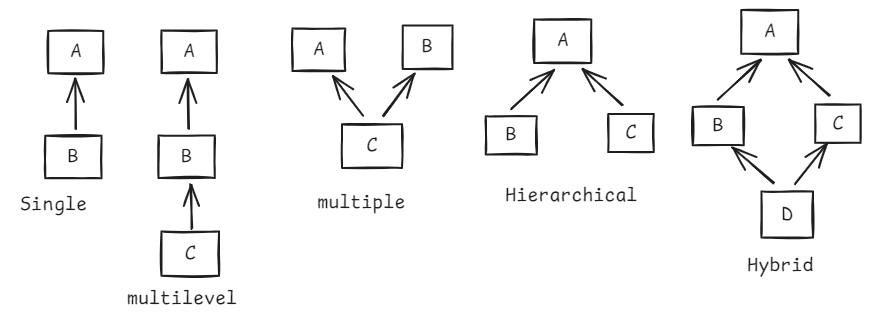
* A child class inherits properties and behaviour of a parent class.
* The keyword to create a child class from the parent class is “extends”.

class ChildClass extends ParentClass {

}

Types of inheritance:

1. single inheritance
2. multilevel inheritance
3. multiple inhertance
4. hierarchical inheritance
5. hybrid inheritance



* Generally, we create a child class object in inheritance. Because child class has properties and behaviour of parent class and also its own properties and behaviour.
* private variables and private methods of the parent class are not inherited to the child class.

example:

**class** ClassA {

**private** **int** privateVar = 40;

**protected** **int** protectedVar = 50;

**int** defaultVar = 60;

**public** **int** publicVar = 70;

**private** **void** privateMethod() {

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

}

**protected** **void** protectedMethod() {

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

}

**void** defaultMethod() {

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

}

**public** **void** publicMethod() {

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

}

}

**class** ClassB **extends** ClassA {

**void** testMethod() {

// System.out.println(privateVar);

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

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

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

// privateMethod();

protectedMethod();

defaultMethod();

publicMethod();

}

}

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

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

ClassB cb = **new** ClassB();

cb.testMethod();

}

}

* with parent class object we can call only parent class methods. But with child class object we can call both parent class methods and also the child class methods.

For example:

**class** ClassA {

**public** **void** m1() {

System.***out***.println("m1()");

}

}

**class** ClassB **extends** ClassA {

**public** **void** m2() {

System.***out***.println("m2()");

}

}

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

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

ClassA ca = **new** ClassA();

ClassB cb = **new** ClassB();

ca.m1();

//we can't call child class method

//with parent class object.

//ca.m2(); //error

//we can call parent class method

//with child class object

cb.m1();

cb.m2();

}

}

Q) can we store child class object in parent class reference variable?

A) YES.

ClassA ca = new ClassB();

Q) can we store parent class object in child class reference variable?

A) No

ClassB cb = new ClassA(); //error

Q) what is difference between storing child class object in parent class reference variable and child class reference variable.

A) If it is stored in parent class reference variable, it can only access parent class methods.

If it is stored in child class reference variable, it can access both parent class and child class methods.

For example,

**class** ClassA {

**public** **void** m1() {

System.***out***.println("m1()");

}

}

**class** ClassB **extends** ClassA {

**public** **void** m2() {

System.***out***.println("m2()");

}

}

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

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

//child class object is stored

//in parent class reference variable

ClassA ca = **new** ClassB();

ca.m1();

ca.m2(); //error

//child class object is stored

//in child class reference variable

ClassB cb = **new** ClassB();

cb.m1();

cb.m2();

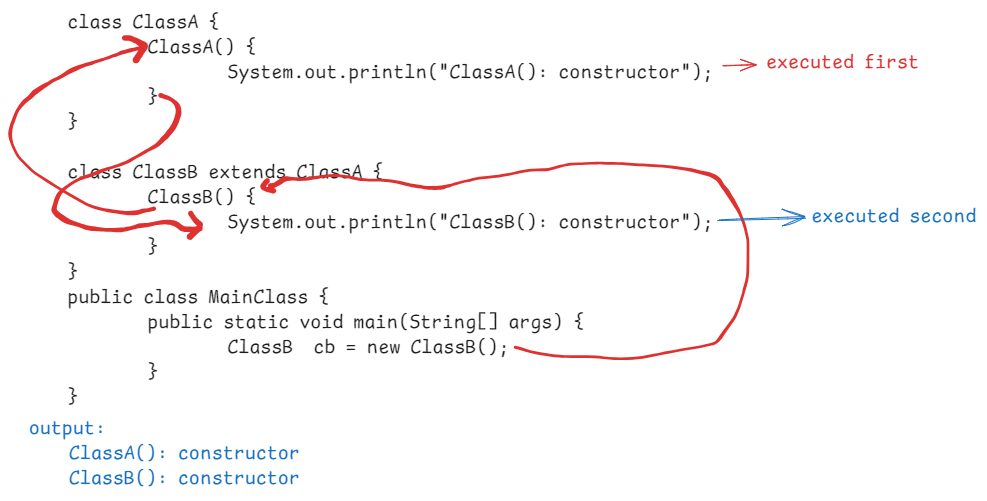
}

}

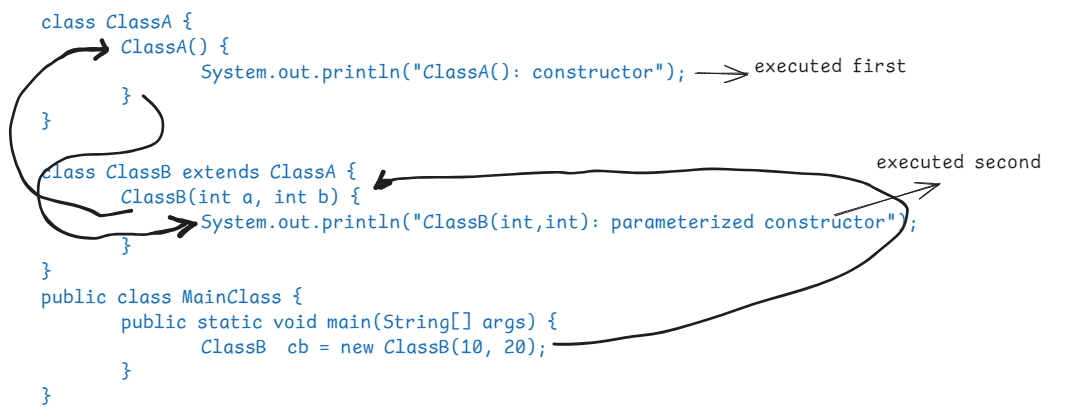
constructor execution in inheritance:

* By default, the first statement in a constructor is super() and it will call parameter-less constructor of the parent class.

ex1:



ex2:

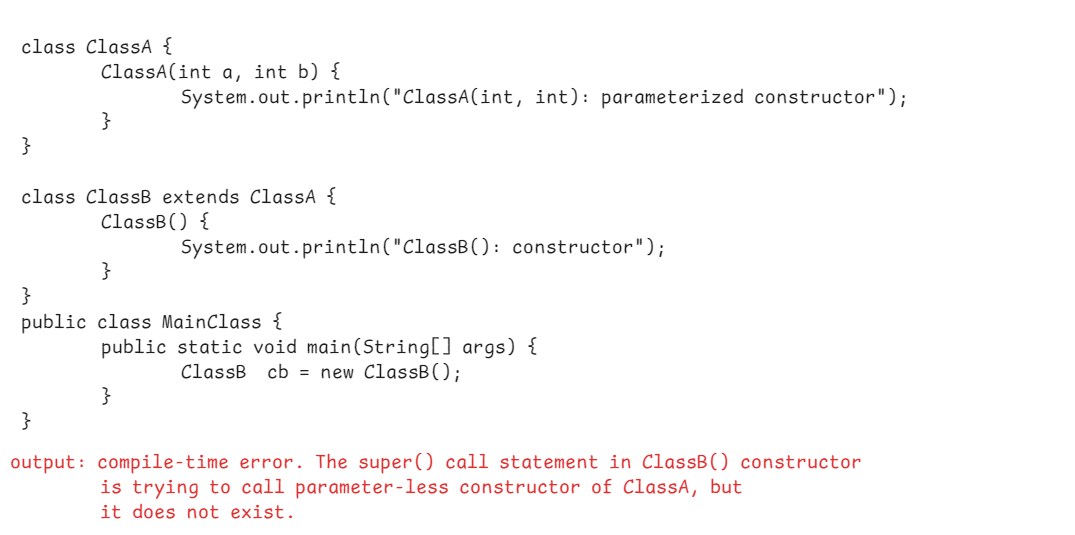


output:

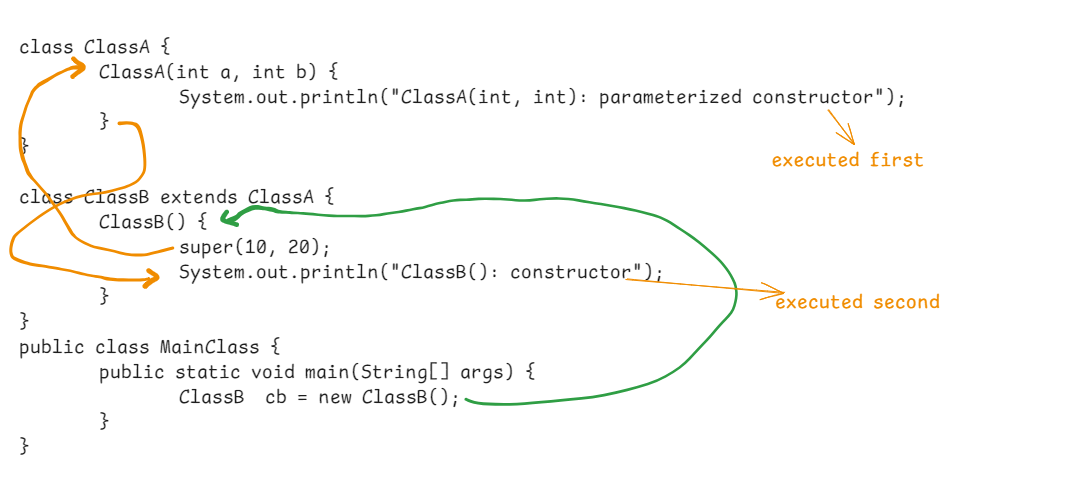
ClassA():constructor

ClassB(int,int):parameterized constructor

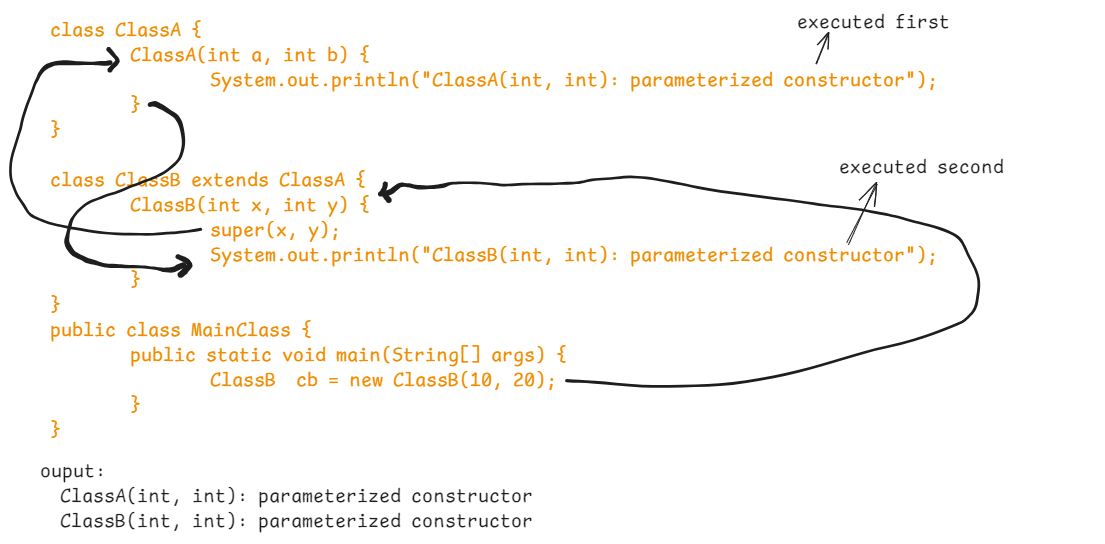
ex3:



ex4:



ex5:



ex6:

**class** ClassA {

ClassA() {

System.***out***.println("ClassA(): constructor");

}

ClassA(**int** a, **int** b) {

System.***out***.println("ClassA(int, int): parameterized constructor");

}

}

**class** ClassB **extends** ClassA {

ClassB(**int** x, **int** y) {

**super**(x, y);

System.***out***.println("ClassB(int, int): parameterized constructor");

}

ClassB() {

**this**(10, 20);

System.***out***.println("ClassB(): constructor");

}

}

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

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

ClassB cb = **new** ClassB();

}

}

output:

ClassA(int, int):parameterized constructor

ClassB(int, int): parameterized constructor

ClassB(): constructor

Q) can we write this() and super() calls at a time?

A) No, because both statements can’t be written as a

first statement.

Q) Why multiple inheritance is not supported at classes level?

A) because of ambiguity problem.

Q) can we create a method in child class, which is matching with the signature in a method in the parent class?

A) YES. This is called method overriding.

Q) can we call super class method in sub class with super keyword?

A) YES.

Q) what is the diff between super() call and super keyword?

A) super() call : calls the constructor of the super class

super keyword: calls the member of the super class.

super() call: should be the first statement in a constructor.

super keyword: we can write at any line.

super() call: we can’t use it a method. We can only use it in constructor.

super keyword: we can use in constructor or in methods.

Q) can we use the super keyword directly in a class?

A) No.

//example on hierarchical inheritance

**class** Employee {

**private** **int** empno;

**private** String ename;

**private** String email;

**private** **long** mobile;

**public** Employee(**int** empno, String ename, String email, **long** mobile) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.email = email;

**this**.mobile = mobile;

}

**public** **void** display() {

System.***out***.println("Empno : " + empno);

System.***out***.println("Ename : " + ename);

System.***out***.println("Email : " + email);

System.***out***.println("Mobile: " + mobile);

}

}

**class** FullTimeEmployee **extends** Employee {

**private** **double** salary;

**public** FullTimeEmployee(**int** empno, String ename, String email, **long** mobile, **double** salary) {

**super**(empno, ename, email, mobile);

**this**.salary = salary;

}

**public** **void** showFullTimeEmployee() {

**super**.display();

System.***out***.println("Salary : "+ salary);

}

}

**class** PartTimeEmployee **extends** Employee {

**private** **int** hoursWorked;

**private** **double** ratePerHour;

**public** PartTimeEmployee(**int** empno, String ename, String email, **long** mobile, **int** hoursWorked, **double** ratePerHour) {

**super**(empno, ename, email, mobile);

**this**.hoursWorked = hoursWorked;

**this**.ratePerHour = ratePerHour;

}

**public** **void** showPartTimeEmployee() {

**super**.display();

System.***out***.println("Hours Worked : " + hoursWorked);

System.***out***.println("Rate per Hour : " + ratePerHour);

}

}

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

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

FullTimeEmployee fte = **new** FullTimeEmployee(7178, "SCOTT", "scott@gmail.com",8990076654L, 6000.0);

fte.showFullTimeEmployee();

System.***out***.println("=".repeat(40));

PartTimeEmployee pte = **new** PartTimeEmployee(7101, "ALLEN", "allen@gmail.com", 9701231235L, 120, 1370.0);

pte.showPartTimeEmployee();

}

}

ex:

**class** ClassA {

**protected** **int** a;

}

**class** ClassB **extends** ClassA {

**super**.a = 50; //error

ClassB() {

**super**.a = 50; //valid

}

}

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

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

ClassB cb = **new** ClassB();

}

}

Q) can we use this keyword in a static method?

A) NO.

Q) can we use super keyword in a static method?

A) NO.

Q) can we use this or super keyword in main() method?

A) No

ex:

**class** ClassA {

**protected** **int** a=40;

}

**class** ClassB **extends** ClassA {

**int** a = 50;

}

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

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

ClassB cb = **new** ClassB();

System.***out***.println(cb.a); // 50

System.***out***.println(((ClassA)cb).a); //40

System.***out***.println(((ClassB)(ClassA)cb).a); //50

}

}

ex:

**class** ClassA {

**protected** **int** a=40;

}

**class** ClassB **extends** ClassA {

**int** a = 50;

**int** b = 60;

}

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

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

ClassA ca = **new** ClassA();

System.***out***.println(ca.a); //40

System.***out***.println(((ClassB)ca).a); //no compile-time error, but we

//get ClassCastException at runtime

}

}

Q) can we type cast child class object to the parent class type?

A) Yes.

ClassB cb = new ClassB();

ClassA ca = (ClassA)cb;

Q) can we type cast parent class object to child class type?

A) No. we will not get compiletime error, but we will

get exception at runtime.

ClassA ca = new ClassA();

ClassB cb = (ClassB)ca;

Polymorphism

* Polymorphsim denotes many forms.
* A method or an operator can perform a well-defined task.
* Polymorphism allows an object to exhibit the same behavior in different ways or allows different objects to exihit the same behavior in different ways.
* To implement polymorphism in an application, two mechanisms are used.

1. method overloading
2. method overriding

* with method overloading, one object can exhibit the same behavior in different ways.
* with method overriding, different objects can exhibit the same behavior in different ways.
* for example, a BankAccount object can exhibit the open account behavior in different ways like opening account with adhaar, opening account with pan card or opening account with voter id.
* for example, a Security object is exhibiting configuration behavior to allow all users to access the project and a WebSecurity object is exhibiting the same configuration behaviour to allow only admin users to access the project.

method overloading:

* method overloading means, defining the same method for more than once in a class, with a difference in parameters.
* The difference in paramters could be,

1. number of parameters
2. type of parameters
3. sequence of parameters

For example:

class Deliver {

boolean isDeliverable(String place) {

//logic

}

boolean isDeliverable(int zipcode) {

//logic

}

}

* Here, the method name is isDeliverable and defined for 2 times, with difference in type of the parameters. So it is method overloading.

For example:

class Product {

public double getPrice(int itemid) {

//logic

}

public double getPrice(String brand, String itemname, String size) {

//logic

}

}

* Here, the method name is getPrice and defined for 2 times, with difference in number of parameters. So, it is method overloading.

For example:

class UserLogin {

public boolean isValid(String username,String password) {

//logic

}

public boolean isValid(long mobile, String password) {

//logic

}

}

* Here, the method name is isValid and defined for 2 times with difference in type of parameters. So, it is method overloading.

For example:

class ClassA {

void m1(int x, double y) {

//logic

}

void m1(double a, int b) {

//logic

}

}

* Here, the method name m1 is defined for 2 times with a difference in sequence of parameters. So, it is method overloading.

For example:

class ClassA {

void m1(int a, int b) {

//logic

}

void m1(int x, int y) {

//logic

}

}

* Here, the method name is m1 defined for 2 times, but there is no difference in parameters. So, it is a compile-time error.

For example,

class Distance {

int findDistance(String source, String destination) {

//logic

}

void findDistance(int zipcode1, int zipcode2)

{

//logic

}

}

* Here, the method, findDistance() is defined for 2 times, with a difference in types of parameters.
* Here, the return types of the methods are different. But is method overloading only. Becoz, method overloading doesn’t depend on return types.

For example,

class FileOperations

{

public static void openFile(String fileName)

{

//logic

}

public static void openFile(String fileName, String filePath) {

//logic

}

}

* Here, openFile is a static method and we can overload the static methods also.

Method overriding:

* For method overriding, we need parent class and a child class. It means, inheritance is required.
* A parent class method, we can override it in the child class.
* overriding means, redefining the same method with a different functionality.
* If a parent class method is not suitable to the child class requirement then we can override the parent class method in the child class.
* In mehod overridng, the method name and its parameters must be same in the parent class and also in the child class.
* In method overloading or overriding, don’t compare the parameters names. They may be same or different.
* The return type of the overridden method in the child class can have the same return type of the parent class method, or it can be the co-variant type of the parent class method return type.
* The overridden method in the child class should contain either same level of access or higher level of access.

Example1:

class Customer {

boolean validate() {

//logic

}

}

class RegularCustomer extends Customer {

@Override

public boolean validate() {

//logic

}

}

* Here, in the parent class, the method has default access modifer, but in the child class, the method has public access modifier.
* public is higher level than default. So it is allowed.

Example:

**class** Employee {

}

**class** FullTimeEmployee **extends** Employee {

}

**class** A {

**public** Employee m1() {

**return** **new** Employee();

}

}

**class** B **extends** A {

@Override

**public** FullTimeEmployee m1() {

**return** **new** FullTimeEmployee();

}

}

* Here, in parent class(A), the m1() method return type is Employee, and in the child class(B), the m1() method return type is FullTimeEmployee.
* FullTimeEmployee is a sub type(co-variant type) of Employee. So, the above example is valid.

Example:

**class** Employee {

}

**class** FullTimeEmployee **extends** Employee {

}

**class** A {

**public** Employee m1() {

**return** **new** FullTimeEmployee();

}

}

**class** B **extends** A {

@Override

**public** FullTimeEmployee m1() {

**return** **new** FullTimeEmployee();

}

}

* in parent class, the m1() method has return type Employee, but it is returning FullTimeEmployee object.
* It means, if return type is parent class, that method can return either parent class object or child class object.

Example:

**class** A {

**public** **static** **void** m1() {

System.***out***.println("m1() : static method in parent class");

}

}

**class** B **extends** A {

@Override

**public** **static** **void** m1() { //error

System.***out***.println("m1(): static method in child class");

}

}

* In this example, we got error because, static method can not be overridden.
* When we write @Override annotation, the compiler will check whether the method can be overridden or not. If not, the compiler generates an error.

example:

**class** A {

**public** **static** **void** m1() {

System.***out***.println("m1() : static method in parent class");

}

}

**class** B **extends** A {

**public** **static** **void** m1() {

System.***out***.println("m1(): static method in child class");

}

}

* In this example, we are not writing @Override annotation.
* In child class, it is looking like, we are overriding the static method of the parent class. But actually, the static method in child class is hiding the static method of the parent class.
* static methods can’ be overridden. If you create a static method in the child class, looking same as static method in parent class, then this concept is not method overriding. It is method hiding.
* This method hiding is applicable for only static methods.

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) what are the types of polymorphism?

A) two types.

1. compile-time polymorphism.

2. runtime polymorphism.

method overloading is a compile-time polymorphism and method overriding is a runtime polymorphism.

* In compile-time polymorphism, the bindings between method call and method signature is done by the compiler at compile-time.
* In runtime polymorphism, the bindings between method call and the method signature is done at runtime by the JVM, based on type of the object.

Q) write all the differences between method overloading and method overriding?

Answer: fill here

final keyword:

* final means finalized.
* final keyword is used in Java, for variables, methods and classes.
* final variables represents constants in java programming.
* if a variable is declared as final, its value can not be modified once it is assigned.
* we can create instance final, static final and local final variables.

ex1:

class Customer {

//instance final

private final int customerId;

//static final

private static final int MAX\_CUSTOMERS=100;

void applyDiscount() {

//local final

final double discount = 0.25;

//some logic

}

}

* As per Java Naming conventions, it is recommended to define the static final variable name in UPPER CASE.
* For final variables, we can define getters, but we can’t define setters.
* If you declare a method as final, then you can’t override that method in the child classes.
* A final method can be overloaded.

ex1:

class Order {

public final void processOrder() {

//retrieve the cart items

//get the shipping address

//process the payment

//confirm the order

}

public final double calculateTotal() {

//logic

}

public final double calculateTotal(double discountRate) {

//logic

}

}

class PremiumOrder extends Order {

//trying to override the final method

public void processOrder() { //error

//some logic

}

}

* a final class can not be extended. It means, you can not create a child class for a final class.

public final class String {

}

public class MyString extends String { //error

}

//Patient.java

**public** **final** **class** Patient {

**private** **final** **int** patientId;

**private** **final** String patientName;

**private** **final** String gender;

**private** **final** String diagnosis;

**public** Patient(**int** patientId, String patientName, String gender, String diagnosis) {

**super**();

**this**.patientId = patientId;

**this**.patientName = patientName;

**this**.gender = gender;

**this**.diagnosis = diagnosis;

}

**public** **int** getPatientId() {

**return** patientId;

}

**public** String getPatientName() {

**return** patientName;

}

**public** String getGender() {

**return** gender;

}

**public** String getDiagnosis() {

**return** diagnosis;

}

}

//PatientUtils.java

**public** **final** **class** PatientUtils {

**public** **static** **final** **void** displayPatientInfo(Patient p) {

System.***out***.println("id : " + p.getPatientId());

System.***out***.println("name : " + p.getPatientName());

System.***out***.println("gender : " + p.getGender());

System.***out***.println("diagnosis : " + p.getDiagnosis());

}

**public** **static** **final** **double** calculateBMI(**double** weight, **double** height) {

**return** weight / ( height \* height);

}

}

//Solution.java

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

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

Patient patient = **new** Patient(109901, "John", "Male", "Cancer");

PatientUtils.*displayPatientInfo*(patient);

System.***out***.println("BMI = " + PatientUtils.*calculateBMI*(80, 7));

}

}

abstract classes:

* Suppose, we are creating a family of classes, which means, the classes are exhibiting the similar behavior.
* In this family of classes, if we find that some common functionality exists and also for some behaviour, the different implementations are required in different classes, then we have to create an abstract parent class.
* For example, we are creating classes like Circle and Rectangle and both of them are exibiting the simlar behaviour.
* Let’s say, fillColor and findArea are the similar functionalities. But here, fillColor implementation should be same in both the classes, but for findArea, implementation should be different in the two classes.
* In this case, we have to create an abstract class, to share the common behaviour and also to enforce the child classes that they must provide the implementation in their own way for the other behaviour.
* So, here, we can create an abstract parent class called Shape with fillColor as concrete method and findArea as an abstract method.
* a concrete method is a method which has the method header/signature and the body in the same class.
* an abstract method is a method which has only the method header in the parent class.

for example:

abstract class Shape {

public void fillColor(Strin color) //concrete

{

//logic

}

public abstract void findArea(); //abstract

}

class Circle extends Shape {

@Override

public void findArea() {

//logic

}

}

class Rectangle extends Shape {

@Override

public void findArea() {

//logic

}

}

* In a class, if we create atleast one abstract method, then we must declare that class as an abstract class.
* abstract keyword can be used with classes and with methods.
* abstract keyword can’t be used with constrcutors and variables.
* If a child class is not overriding all the abstract methods of the parent class, then we must declare that child class also as abstract class. Otherwise, it is a compile-time error.

ex:

**abstract** **class** ClassA {

**void** m1() {

//logic

}

**abstract** **void** m2();

**abstract** **void** m3();

}

**abstract** **class** ClassB **extends** ClassA {

@Override

**void** m3() {

// **TODO** Auto-generated method stub

}

}

Q) can we make a class as abstract without abstract methods?

A) Yes. Suppose, you want to tell that your class has concrete methods but their implementation logic is partial, not upto the mark, then you can declare the class as abstract class.

By making the class as abstract, you can inform the other developers that your class is not a fully implemented class.

Q) can we create an object for an abstract class?

A) No.

Q) can we create a constructor in abstract class?

A) Yes. We can’t create an object for the abstract class, but we can create an object for the child class.

When child class object is created, child class constructor calls, parent class constructor and this parent class constructor can initialize the instance variables, using the constructor.

Q) can we create an abstract class with all abstract methods?

A) Yes.

Q) can we create an abstract class with all the concrete methods?

A) Yes.

Q) a child class can extend how many abstract classes?

A) Only one.

/\*

\* a Demo code on abstract class

\*/

**abstract** **class** InsurancePolicy {

**private** **double** basePremium;

**public** InsurancePolicy(**double** basePremium) {

**this**.basePremium = basePremium;

}

**public** **abstract** **double** calculateRiskFactor();

**public** **double** calculateFinalPremium() {

**double** riskFactor = calculateRiskFactor();

**return** basePremium + riskFactor;

}

**public** **double** getBasePremium() {

**return** basePremium;

}

}

**class** HealthPolicy **extends** InsurancePolicy {

**private** **int** age;

**private** **boolean** isSmoker;

**public** HealthPolicy(**double** basePremium, **int** age, **boolean** isSmoker) {

**super**(basePremium);

**this**.age = age;

**this**.isSmoker = isSmoker;

}

@Override

**public** **double** calculateRiskFactor() {

**double** riskFactor = 0.0;

**if** ( isSmoker && age >= 60 ) {

riskFactor = getBasePremium() \* 0.45;

}

**else** **if**(isSmoker && age < 60) {

riskFactor = getBasePremium() \* 0.29;

}

**else** **if** ( isSmoker == **false** && age >=60 ) {

riskFactor = getBasePremium() \* 0.20;

}

**return** riskFactor;

}

}

**class** VehiclePolicy **extends** InsurancePolicy {

**private** String fuelType;

**private** **int** mfgYear;

**public** VehiclePolicy(**double** basePremium, String fuelType, **int** mfgYear) {

**super**(basePremium);

**this**.fuelType = fuelType;

**this**.mfgYear = mfgYear;

}

@Override

**public** **double** calculateRiskFactor() {

**double** riskFactor = 0.0;

**if**( fuelType.equalsIgnoreCase("Diesel") && mfgYear < 2019 ) {

riskFactor = getBasePremium() \* 0.52;

}

**else** **if**(fuelType.equalsIgnoreCase("Diesel") && mfgYear >= 2019) {

riskFactor = getBasePremium() \* 0.35;

}

**else** {

riskFactor = getBasePremium() \* 0.20;

}

**return** riskFactor;

}

}

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

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

InsurancePolicy healthPolicy = **new** HealthPolicy(24399.0, 69, **true**);

System.***out***.println("final premium to pay : " + healthPolicy.calculateFinalPremium());

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

InsurancePolicy vehiclePolicy = **new** VehiclePolicy(28979.0, "DIESEL", 2017);

System.***out***.println("final premium to pay : " + vehiclePolicy.calculateFinalPremium());

}

}

Q) can we create private abstract methods?

A) No.

Q) can we create static abstract method?

A) No

Q) can we create final abstract method?

A) No

Q) can we create an object for the abstract class?

A) No. But we can create a reference variable to store the

child class object.

Interfaces

* An interface is to define a contract for the classes to exihibit similar behaviour but with their own implementations.
* An interface is a collection public static final variables and public abstract methods.
* In an interface, by default the variables are public static and final. So, writing these modifiers is optional.
* In an interface, by default the methods are public and abstract. So, writing these modifiers is optional.
* interface keyword is used to create an interface.

interface <interfacename> {

public static final variables;

public abstract methods;

}

* A class, who is implementing the interface, has to override the abstract methods of the interface.

For ex:

interface PaymentGateway {

*boolean processPayment();*

*S*tring paymentStatus();

}

class PayPal implements PaymentGateway {

@Override

public boolean processPayment() {

//some logic

}

@Override

public String paymentStatus() {

//some logic

}

}

class Stripe implements PaymentGateway {

@Override

public boolean processPayment() {

//some logic

}

@Override

public String paymentStatus() {

//some logic

}

}

class MyClass {

//variables

//methods

}

* If a class implements an interface, that class has to override all the abstract methods. If it is overriding only some of the abstract methods, then you have to declare the class as abstract class.
* an interface does not have a constructor. So, we can not create an object for interface.

PaymentGateway pg = new PaymentGateway(); //error

PaymentGateway pg = new PayPal(); //valid

PaymentGateway pg = new Stripe(); //valid

PaymentGateway pg = new MyClass(); //error

* One interface can extend mulitple interfaces and one class can implement multiple interfaces.

For ex:

interface CrudRepository {

void save();

void delete();

}

interface SortingRepository {

void sortById();

void sortByName();

}

interface JpaRepository extends CrudRepository, SortingRepository {

void batchSave();

}

class EmployeeRepository implements CrudRepository,SortingRepository

{

@Override

public void save() {

}

@Override

public void delete() {

}

@Override

public void sortById(){

}

@Override

public void sortByName(){

}

}

* multiple inheritance can be implemented in Java through interfaces.
* if you want to create a class by inheriting a super class and also by implementing an interface, then extends keyword should come first then followed by implements keyword.

For example,

class B extends A implements I1 {

}

* If a class implements two interfaces and if same method with same return type and same parameters exist in two interfaces then the class has to override the method for only once.

ex:

**interface** I1 {

**void** m1();

}

**interface** I2 {

**void** m1();

}

**class** A **implements** I1, I2 {

@Override

**public** **void** m1() {

//logic

}

}

* For a class in Java, the default parent class is Object class. But for an interface in Java, there is no default parent interface.

Q) If a class implements an interface, can we define new methods in that class?

A) Yes.

example:

interface I1 {

void m1();

}

class A implements I1 {

@Override

public void m1() {

//logic

}

public void f1() { //new method

//logic

}

}

Q) what is the difference between class reference and an interface reference?

A) with class reference we can call the new methods of the class also. But with interface reference we can not call the new methods.

For example:

I1 i1 = new A();

i1.m1(); //correct

i1.f1(); //error

A a = new A();

a.m1(); //correct

a.f1(); //correct

**interface** Vehicle {

**void** start();

**void** stop();

**void** accelerate(**int** speed);

}

**class** Car **implements** Vehicle {

@Override

**public** **void** start() {

System.***out***.println("Car is starting.....");

}

@Override

**public** **void** stop() {

System.***out***.println("Car has stopped");

}

@Override

**public** **void** accelerate(**int** speed) {

System.***out***.println("Car is accelerating to " + speed + " km/h");

}

}

**class** Bike **implements** Vehicle {

@Override

**public** **void** start() {

System.***out***.println("Bike is starting.....");

}

@Override

**public** **void** stop() {

System.***out***.println("Bike has stopped");

}

@Override

**public** **void** accelerate(**int** speed) {

System.***out***.println("Bike is accelerating to " + speed + " km/h");

}

}

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

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

Vehicle car = **new** Car();

car.start();

car.accelerate(120);

car.stop();

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

Vehicle bike = **new** Bike();

bike.start();

bike.accelerate(80);

bike.stop();

}

}

Types of interfaces:

1. normal interfaces
2. marker interfaces
3. functional interfaces

* A normal interface can have more than one abstract method and any number of fields.
* A marker interface does not contain any abstract methods and any fields. It is an empty interface.
* A functional interface can have only one abstract method and any number of fields. This is newly included from Java8.
* marker interfaces are used to mark a class that the objects of that class should be provided with additional/special behavior.
* marker interface itself does not provide the special behavior, but the Java Runtime or a framework or any another code can provide the special behavior.
* For example, the famous marker interface in Java is, Serializable interface.
* if a class implements Serializable interface then JVM converts the objects of that class into byte stream.
* Another marker interface in Java is Cloneable interface.

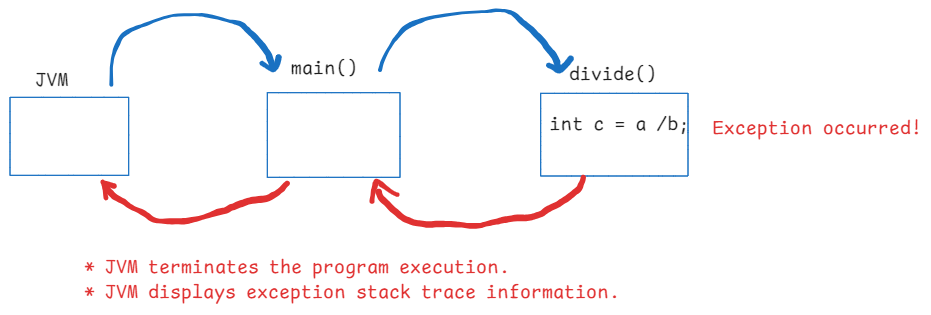
Exception Handling

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

* A program can have 3 types of errors

1. syntax error
2. runtime error
3. logical error

* syntax errors are identified at compile-time.
* runtime errors are identified at runtime.
* The compiler will display the syntax errors and the JVM will display the runtime errors.
* The logical errors can be identified by the input given and the output returned.
* syntax errors will occur, if the programmer doesn’t follow Java syntax while writing the code.
* runtime errors will occur, if the user doesn’t provide the valid data at the time of executing the program.
* These runtime errors are also called exceptions.
* Exceptions will occur not because of the code, it is because of the data given by the user at runtime.
* The default flow of an application is, if an exception occurs then suddenly the application gets crashed.
* With exception handling, we can prevent an application from sudden crash, if an exception occurs at runtime.
* So, at the time developing the application, a developer has to implement exception handling in the application code.
* For example, In an online purchase, if a user has selected the products, entered the delivery details and provided the card details for payment. But due to insufficient balance, the payment is declined and the application is suddenly crashed.
* But this should not happen in real-time application. So to handle this type of unexpected errors, application should contain exception handling.
* If an exception occurs in a method, that method propagete exception to the caller method, and the if the caller method is main(), then it propagates to JVM, and the JVM will terminate the program execution and will display the exception stack trace information.

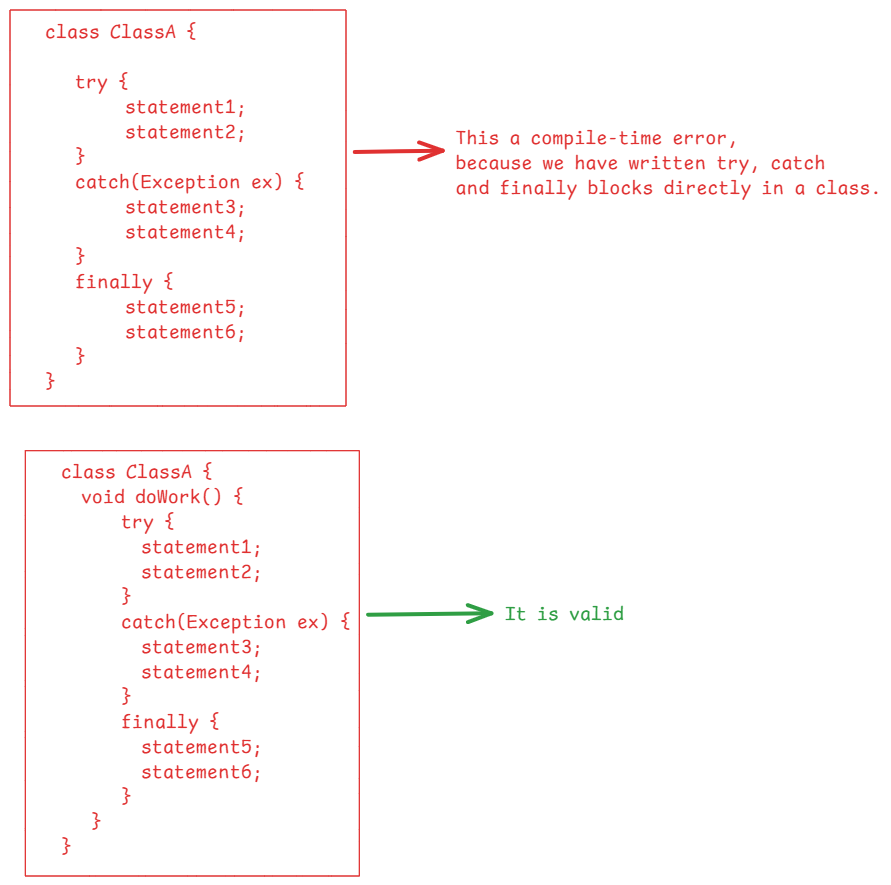


Exception handling keywords:

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

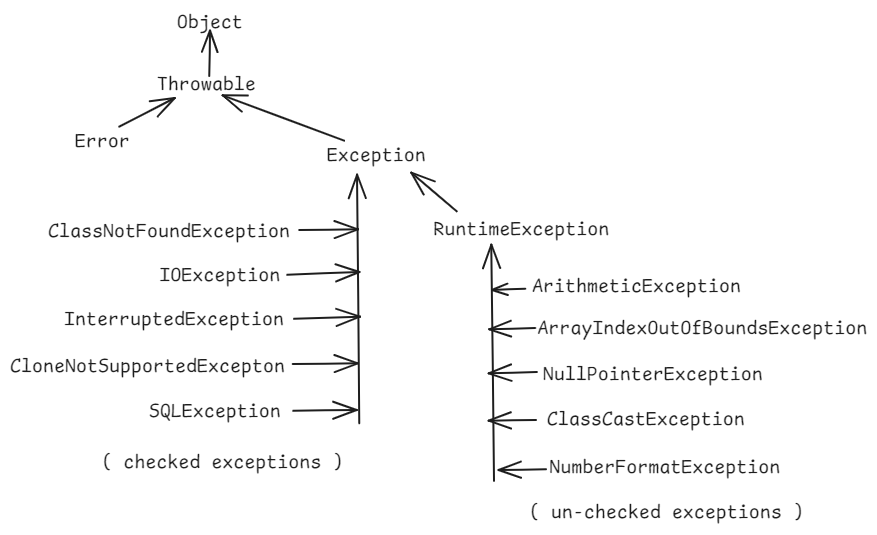
1. try
2. catch
3. finally
4. throw
5. throws

* try is a keyword used to define a block.
* try block should contain a set of statements which may cause an exception at runtime.
* catch block should contain a set of statements which can send a notification to the user and also which can prevent the application from sudden crash.
* finally block should contain a set of statements that must be executed irrespective of whether exception is occurred or doesn’t occur in the try block.
* try, catch and finally blocks can be defined in a method, but can’t be defined directly in a class.



* try block followed by catch block followed by finally block is correct.
* try block followed by catch block is correct
* try block followed by finally block is correct.
* only try block without a corresponding catch/finally block is an error.
* You can’t write a catch/finally block without a try block.

Exception class hierarchy:



types of exceptions:

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

1. checked exceptions
2. un-checked exceptions

* checked exceptions are known to the java compiler.
* un-checked exceptions are unknown to the java compiler.

checked exception:

* During compilation, if there is a chance of getting a checked exception from the code, the compiler will verify whether the programmer is handling the exception with try and catch blocks or

atleast propagating that exception to the caller method or not.

If not done anything then the compiler will generate an error message.

un-checked exception:

* During compilation, if there is a chance of getting an un-checked exception from the code, the compiler does not verify whether the programmer is handling the excepiton with try and catch blocks or atleast propagating that exception to the caller method or not.

ex1:

1.class ClassA {

2. void m1() {

3. Class.forName(“java.lang.Runtime”);

4. }

5.}

\* In this example, at line3, there is a chance of getting ClassNotFoundException. It is a checked exception, so, the compiler will verify whether the programmer is handling it with try and catch blocks. But not done. Now the compiler will check whether this exception is propagated to the caller method or not with throws keyword. But not done.

\* That’s why the compiler will generate an error at line3.

ex2:

1. class ClassA {
2. void m1() {
3. String str = null;
4. System.out.println(str.length());
5. }
6. }

\* In this example, at line4, there is a chance of getting NullPointerException. It is un-checked exception, so, the compiler does not verify whether the programmer is handling it with try and catch blocks or propagting it to the caller method with throws keyword or not. The compiler does not generate any error, because it is an unchecked exception.

try block with multiple catch blocks:

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

* In try block, if there is an exception raised at some line then the remaining lines of the try blocks are skipped and immediately the control jumps to the nearest catch block.
* The control checks the exception type in catch block is matching or not, if not then the contorl jumps to the next catch block.
* If there are multiple catch blocks but no one is matching, then the exception is propagated to the caller method.

for example:

1. **void** m1() {

2. **int**[] arr = **new** **int**[5];

3. **try** {

4. **int** x = 10 / 5;

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

6. arr[5] = 90;

7. System.***out***.println("Done");

8. }

9. **catch**(ArithmeticException ex) {

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

11. }

12. **catch**(ArrayIndexOutOfBoundsException ex) {

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

14. }

15. }

\* In this code, at line6, ArrayIndexOutOfBoundsException will be thrown.

\* The control skips the line7, and jumps to line9.

\* The catch block at line9 is not matching with the exception type.

\* So, the control jumps to the line12. Here, the exception type is matched, so the line13 is executed.

Ex:

1. **try** {

2. **int** x = 10 / 5;

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

4. String str = **null**;

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

6. arr[5] = 90;

7. System.***out***.println("Done");

8. }

9. **catch**(ArithmeticException ex) {

10. System.***out***.println("ArithmeticException occurred");

11. }

12. **catch**(ArrayIndexOutOfBoundsException ex) {

13. System.***out***.println("ArrayIndexOutOfBoundsException occurred");

14. }

15. **catch**(Exception ex) {

16. System.***out***.println("Exception occurred!!");

17. }

. In this example, at line 5, NullPointerExeception will be raised.

. The control jumps to line 9. But it is a catch block for ArithmeticException. So, the control jumps to line 12. But it is a catch block for ArrayIndexOutOfBoundsException. Not matched.

. The control jumps to line 15, it is a catch block for Exception. We know that Exception is the super class for all the exceptions, so, this catch block will handle the NullPointerException.

Ex:

try {

* - - -
* - - -
* - - -

}

catch(Exception ex) {

}

catch(ArithmeticException ex) {

}

catch(ArrayIndexOutOfBoundsException ex) {

}

* In this example, we have a compiletime error called unreachable catch blocks.
* Because, the catch block for Exception will handle all the exceptions. So the below catch blocks becomes unreachable.

\*\*\*\* When writing multiple catch blocks, the catch block for specific exception types then followed by catch block for Exception is allowed.

But catch block for Exception followed by catch blocks for specific exceptions is not allowed.

Different ways to display an exception:

1. we can display exception class name and its message
2. we can display only the message
3. we can display the complete stack trace information.

. To display exception classname with message, you have to print the object.

System.out.println(ex);

. To display only the exception message, you to call getMessage() method.

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

. To display the complete stack trace information, you to call printStackTrace()

ex.printStackTrace();

. The stack trace contains 4 details.

1. exception classname
2. message
3. method call stack
4. location(line numbers)

For ex:

java.lang.ArithmeticException: / by zero

at ClassA.m1(Main.java:9)

at Main.main(Main.java:35)  
========================================================================

Nested try blocks:

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

try {

statement1;

statement2;

statement3;

try {

statement4;

statement5;

statement6;

}

catch(ArithmeticException ex) {

//some logic

}

}

catch(Exception ex) {

//some logic

}

* If there is an exception occurs in the inner try block then the control jumps to the inner catch block. If it is unable to handle the exception, then the control jumps to the outer catch block.
* If there is an exception occurs in the outer try block then the control directly jumps to the outer catch block.

Q) if there are any statements after finally block, will they execute?

A) Yes.

Q) can we write any statements between try and catch blocks?

A) No.

Q) can we write any statements between catch and finally blocks?

A) No.

Q) if try block contains return statement and there is no exception raised in the try block, then finally block will execute or not?

A) finally block will be executed, but if any statements are there after the finally block, they are not executed.

example:

**try** {

**int** x = 10 / 2;

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

return;

}

**catch**(ArithmeticException ex) {

ex.printStackTrace();

}

**finally** {

System.***out***.println("I am finally");

}

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

output: 5

I am finally

Q) if try block contains exit statement and there is no exception raised in the try block, then finally block will execute or not?

A) finally block is not executed and even the statements after the finally block are also not executed.

example:

**try** {

**int** x = 10 / 2;

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

System.*exit*(0);

}

**catch**(ArithmeticException ex) {

ex.printStackTrace();

}

**finally** {

System.***out***.println("I am finally");

}

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

output: 5

Q) can we handle multiple exceptions in a single catch block?

A) Yes. We have to use a pipe ( | ) to separate the exception classes.

example:

**try** {

**int** x = 10 / 0;

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

arr[5]=80;

System.***out***.println(arr[5]);

}

**catch**(ArithmeticException | ArrayIndexOutOfBoundsException ex) {

ex.printStackTrace();

}

**catch**(NullPointerException npe) {

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

}

User-defined exceptions:

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

. we can create an user-defined exception in a project, either by extending Exception class or by extending RuntimeException class.

. If you create an exception class by extending Exception, then it becomes checked exception.

. If you create an exception class by extending RuntimeException, then it becomes un-checked exception.

ex:

class UserNotFoundException extends RuntimeException {

}

. while throwing this exception, you can’t pass any message. Because, this class doesn’t have a constructor.

ex:

class UserNotFoundException extends Exception {

public UserNotFoundException(String message) {

}

}

. while throwing this exception, you can pass the message. But you can’t access the message while displaying the exception.

. Because, you are not passing the message to super class.

ex:

class UserNotFoundException extends Exception {

public UserNotFoundException(String message) {

super(message);

}

}

. while throwing this exception, you can pass the message. You can also access the message while displaying the exception.

throw keyword:

. It is used to throw an exception from a method.

. throw keyword raises an exception object.

. we use throw keyword in a method to throw the user-defined

excepitons.

//example on throw keyword

//InsufficientBalanceException.java

**public** **class** InsufficientBalanceException **extends** Exception {

**public** InsufficientBalanceException(String message) {

**super**(message);

}

}

//BankAccount.java

**public** **class** BankAccount {

**private** **double** balance;

**public** BankAccount(**double** balance) {

**this**.balance = balance;

}

**public** **void** withdraw(**double** amount) {

**try** {

**if** ( amount > balance ) {

**throw** **new** InsufficientBalanceException("existing balance : "+balance+", given amount : "+ amount+", withdraw not possible");

}

balance = balance - amount;

System.***out***.println("amount withdrawn successfully");

System.***out***.println("Remaining balance : " + balance);

}

**catch**(InsufficientBalanceException ex) {

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

}

}

}

//MainClass.java

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

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

// **TODO** Auto-generated method stub

BankAccount account = **new** BankAccount(50000);

account.withdraw(60000);

}

}

throws keyword:

. throws keyword is used in the method signature, but not in the method body.

. The purpose of throws keyword is to propagate an exception from the current method where the exception is occurred to the caller method.

. If a checked exception is thrown from the current method then that method should handle that checked exception by writing try and catch blocks or it must propagate the exception to the caller method, using throws keyword.

. If not done, then compiler generates an error.

. If an unchecked exception is thrown from the current method and if the current method is not handling it with try and catch blocks or if it is not propagating the excetion to the caller method using throws keyword, the compiler doesn’t generates any error. The JVM will take care of this.

. with throws keyword, both checked and un-checked exceptions can be propagated to the caller method, from the current method.

. with throws keyword, you can declare multiple exceptions in the method signature by using a delimeter comma(,).

ex:

public void withraw(int accno, double amount) throws InsufficientBalanceException, InterruptedException {

//logic

}

Q) what is the difference between throw and throws?

A) 1. throw keyword is used inside the method body.

throws keyword is used in the method signature.

1. throw keyword can throw only one exception at a time.

with throws keyword we can declare mulitple exceptions also.

Q) is method overloading depends on throws keyword?

A) No.

. In method overloading, the methods can propagate the same or different exceptions with throws keyword.

ex1:

class ClassA {

void m1() throws ClassNotFoundException {

//logic

}

void m1(int x) { //correct

//logic

}

}

ex2:

class ClassA {

void m1() throws ClassNotFoundException {

//logic

}

//correct

void m1(int x) throws InterruptedException, NullPointerException {

//logic

}

}

Q) is method overriding depends on throws keyword?

A) Yes.

. The overridden method in the child class can propagate atmost the same checked exceptions, propagted by the parent class method. But it can’t propagate more than that.

. For unchecked exceptions, there is no restriction.

ex1:

class ClassA {

void m1() throws ClassNotFoundException {

//logic

}

}

class ClassB extends ClassA {

@Override

void m1() throws ClassNotFoundException, IOException { //error

//logic

}

}

ex2:

class ClassA {

void m1() throws ClassNotFoundException {

//logic

}

}

class ClassB extends ClassA {

@Override

void m1() throws ClassNotFoundException, ArithmeticException{ //valid

//logic

}

}

ex3:

class ClassA {

void m1() throws ClassNotFoundException {

//logic

}

}

class ClassB extends ClassA {

@Override

void m1() { // valid

//logic

}

}

Q) Tell me the rules of method overriding?

A) 1. method name in the child class should be same as method name

in the parent class.

2. The number and type and sequence of parameters in the child

class should be same as parent class method.

1. return type of the child class method should be either same or a co-variant type of the parent method return type.
2. The access modifier of the child class method should be either same level or higher level than the parent method.
3. The child class method can atmost propagate the same checked exceptions propagated by the parent class method.

try with resources:

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

. try with resources means, we can pass parameters to the try

block.

. The parameter should be an AutoCloseable object.

. AutoCloseable is an interface and an object of a class should

implement AutoCloseable interface.

. try block will automatically close the given resources after executing the try block.

. if exception occurs or doesn’t occur in the try block, it will automcatically close the given resources.

. So, you no need to define finally block, to close these resources.

. So, using try with resources, you can avoid or you can minimize the finally block.

ex:

try( Connection conn = DriverManager.getConnection(url,uname,pwd),

Statement stmt = conn.createStatement() ) {

statement1;

statement2;

statement3;

}

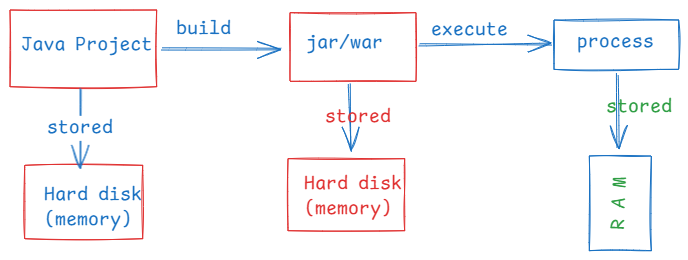
catch(Exception ex) {

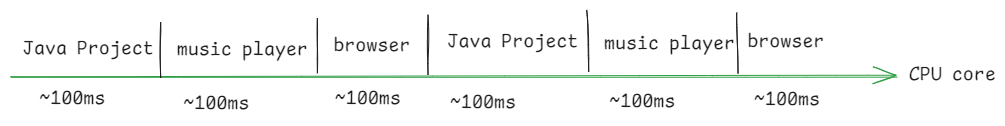
//logic

}

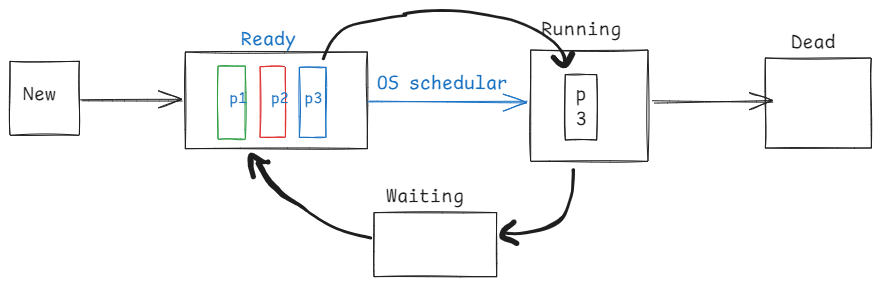
Multithreading

Process:

* an instance of a program/application loaded into RAM.
* 
* Suppose, if you execute a Java project, if you start a browser and if you start a music player in your system, then Operating system creates 3 processes and will be stored into RAM.
* OS comes with a schedular, which will provide CPU time for a process to run on a CPU core.
* The OS schedular will provide a time slice, approximately 100ms for each process, if they are are executing on a single cpu core.



* A process from creation to completion, it goes into different states.
* These states represent the process life cycle.
* 1. New 2. Ready 3. Running 4. Waiting 5. Dead



Thread:

* Each process contains atleast one thread, called main thread.
* If a process has multiple independent tasks, then we can create multiple theads in a process, where each thread will perform a task.
* If threads are not created then the tasks are executed sequentially on a single cpu core, hence the process completion time will be more.
* If threads are created then the tasks are parallelly executed on different CPU cores and hence the process is completed fastly. The performance of an application will be increased.
* In a Java program, if you want to execute mulitple tasks parallelly then you have to use multithreading.
* For example, a Java program wants to insert the records into the database and also wants to generate the reports. In this case we can create a thread for inserting the records and other thread for generating the reports.
* A thread is a group of statements, which are executed independently from the rest of the program execution.

creating threads:

* There are 2 ways to create a thread.

1. by extending Thread class
2. by implementing Runnable interface

* The logic of a thread should be defined in run() method.

ex:

class MyThread extends Thread {

@Override

public void run() {

//define the logic here

}

}

(or)

class MyThread2 implements Runnable {

@Override

public void run() {

//define the logic here

}

}

* If a thread is created by extending Thread class, then it can be started directly.

MyThread t1 = new MyThread();

t1.start();

* If a thread is cretaed by implementing Runnable interface, then it can’t be started directly,
* First wrap the the Runnable object into Thread object, then start the thread.

MyThread2 t2 = new MyThread2(); // Runnable oject

t2.start(); //compile-time error.

Thread t3 = new Thread(t2); // wrapping

t3.start(); //correct

**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();

}

}

Q) can we start a thread for twice, by calling start() method?

A) if we write,

t1.start();

t1.start();

then

we won’t get any compile-time error, but at runtime, IllegalThreadStateException will be thrown.

Q) can we call run() method of a thread directly?

A) Yes. But the threads will execute sequentially, but not parallelly.

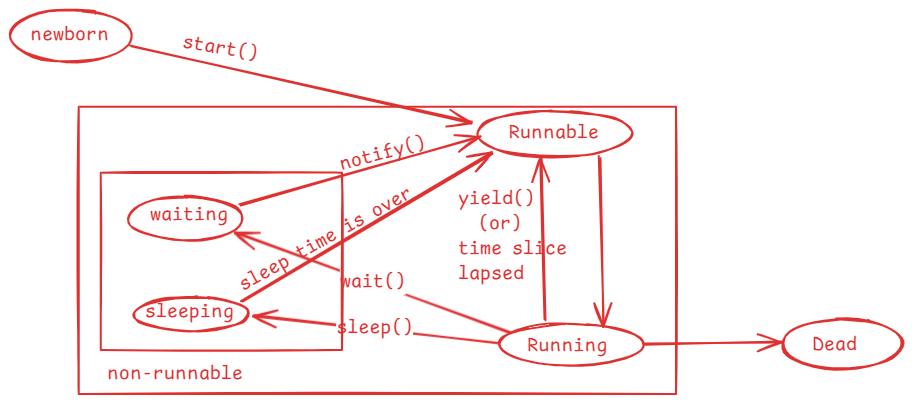
Q) which way is better to create a thread, either extending Thread class

or implmenting Runnable interface?

A) implementing Runnable interface is better approach, because still a thread class can inherit another parent class.

Thread life cycle:

* From object creation to termination, a thread under goes with different states. This is called life cycle.
* A thread life cycls has 5 states/stages.
* 1. newborn state
* 2. runnable state
* 3. running state
* 4. non-runnable state
* 5. dead state



* 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();

**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.start();

t2.start();

}

}

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

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 t1 with t2 then t2 will wait until t1’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();

}

}

}

}

Q) what is the difference between a process and a thread?

A) 1. process contains one or more threads.

a thread does not contain processes.

2.A process has a separate address space.

A thread does not have a separate address space.

3.process context switch is heavy. Because, cpu has to switch

from one address space to another address space.

thread context switch is light weight. Because, cpu no need to

switch, as the threads are part of the same process.

4.process to process communication requires additional resources

like sockets.

thread to thread communication does not require additional resources.

Threads 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 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, if few statements of a method leads to inconsistency, if they are executed by multiple threads at the same time, but if we make entire method as synchronized then it will increase waiting time for the other threads.
* So, instead of creating synchronized method, we can define a synchronized block inside the method with only the statements that cause inconsistency and by keeping the remaining statements at outside of the synchronized block.

For ex:

public void doWork() {

statement1;

statement2;

synchronized(this) {

statement3;

statement4;

statement5;

}

statement6;

statement7;

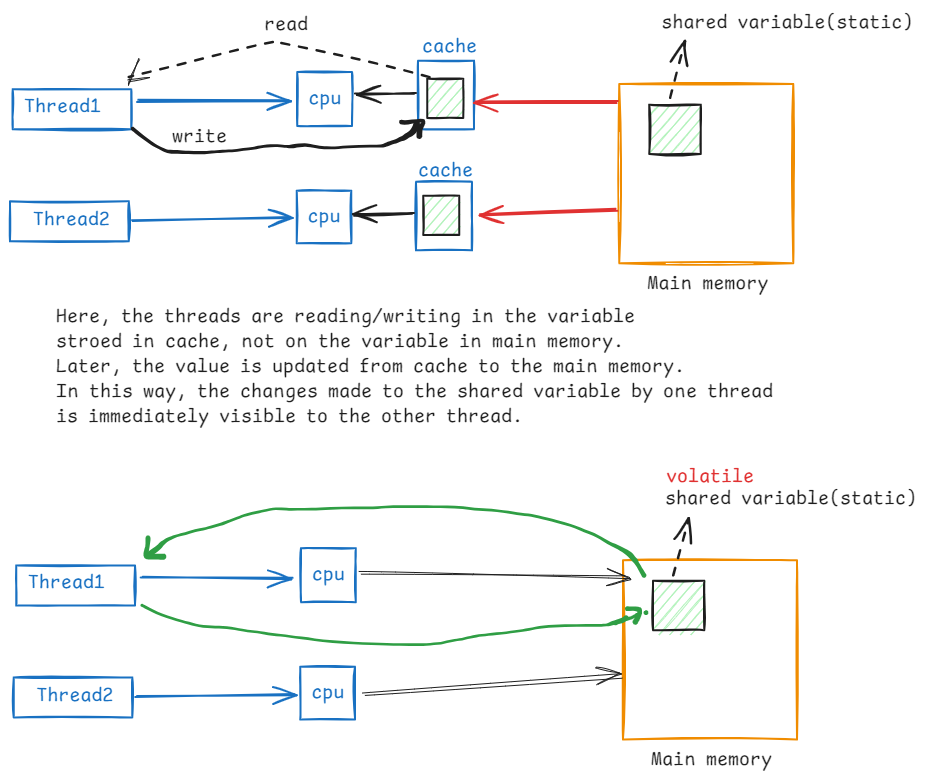
}

Q) synchronized method/block, which is better?

A) synchronized block, because it reduces waiting time for other threads.

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.



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** com.ashokit.thread;

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

**static** **final** Object ***res1*** =**new** Object();

**static** **final** Object ***res2*** = **new** Object();

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

Runnable task1 = () -> {

**synchronized**(***res1***) {

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**(***res2***) {

System.***out***.println(Thread.*currentThread*().getName() +" has acquired lock on resource2");

}

}

System.***out***.println(Thread.*currentThread*().getName()+" : is completed");

};

Runnable task2 = () -> {

**synchronized**(***res2***) {

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**(***res1***) {

System.***out***.println(Thread.*currentThread*().getName() +" has acquired lock on resource1");

}

}

System.***out***.println(Thread.*currentThread*().getName()+": is completed");

};

Thread t1 = **new** Thread(task1, "Thread1");

Thread t2 = **new** Thread(task2, "Thread2");

t1.start();

t2.start();

}

}

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");

}

}

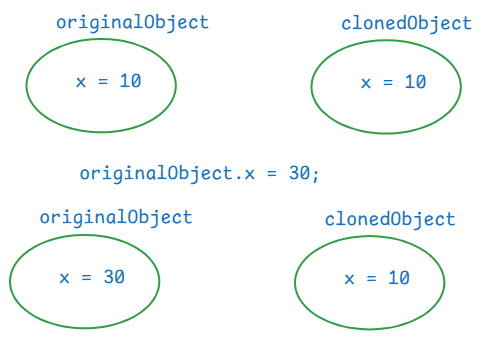
Cloning

. Cloning is a process of creating a duplicate object from an exisitng object.

. When creating complex objects, cloning is used.

. Instead of creating an object from scratch, we can clone exisiting objec to create a duplicate object.

. Once an object is cloned, the original and duplicate objects acts independently. The changes made to the original object, doesn’t effect on duplicate object and vice-versa.



. For example, in a gaming application, we need multiple enemies with the same characterstics. So, we create one enemy object, and we clone this object to create multiple enemies.

How to clone an object?

. first implement a class, whose objects requires cloning from java.lang.Cloneable marker interface.

. we have to override clone() method of Object class, in that class.

//example

**package** com.ashokit.cloning;

**class** ClassA **implements** Cloneable {

**private** **int** x;

**private** **int** y;

ClassA(**int** x, **int** y) {

// System.out.println("ClassA(int,int): constructor");

**this**.x = x;

**this**.y = y;

}

//Here, you are overriding clone() method of Object class

@Override

**protected** Object clone() **throws** CloneNotSupportedException {

**return** **super**.clone();

}

//Here, you are overriding toString() method of Object class

@Override

**public** String toString() {

**return** "ClassA [x=" + x + ", y=" + y + "]";

}

**public** **int** getX() {

**return** x;

}

**public** **void** setX(**int** x) {

**this**.x = x;

}

**public** **int** getY() {

**return** y;

}

**public** **void** setY(**int** y) {

**this**.y = y;

}

}

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

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

ClassA ca = **new** ClassA(10, 20);

ClassA clonedCa = (ClassA) ca.clone();

System.***out***.println("original object : " + ca);

System.***out***.println("cloned object : "+ clonedCa);

//make changes to original object

ca.setX(30);

ca.setY(40);

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

System.***out***.println("After making changes in original object");

System.***out***.println("original object : " + ca);

System.***out***.println("cloned object : "+ clonedCa);

}

}

output:

original object : ClassA [x=10, y=20]

cloned object : ClassA [x=10, y=20]

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

After making changes in original object

original object : ClassA [x=30, y=40]

cloned object : ClassA [x=10, y=20]

Types of cloning:

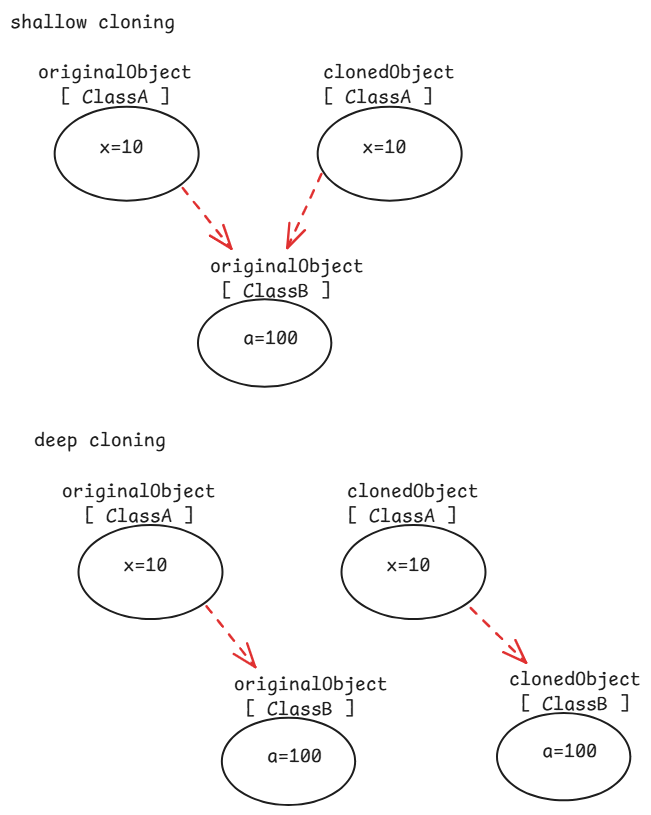
1. shallow cloning
2. deep cloning

. In shallow cloing, when a dependent object is cloned, its dependency object will not be cloned. It means, both original and cloned dependent objects will share the same dependency object.

. In deep cloning, when a dependent object is cloned, its dependency object is also cloned. So, both original and cloned dependent objects have separate dependency objects.

. In shallow cloning, dependent class implements Cloneable interface, but dependency class doesn’t.

. In deep cloning, both the classes implements Cloneable interface.



// shallow cloning example

**package** com.ashokit.cloning;

//dependent class

**class** ClassA **implements** Cloneable {

**private** **int** x;

**private** **int** y;

**private** ClassB classB;

ClassA(**int** x, **int** y) {

// System.out.println("ClassA(int,int): constructor");

**this**.x = x;

**this**.y = y;

classB = **new** ClassB(100, 200);

}

@Override

**protected** Object clone() **throws** CloneNotSupportedException {

**return** **super**.clone();

}

@Override

**public** String toString() {

**return** "ClassA [x=" + x + ", y=" + y + ", classB=" + classB + "]";

}

**public** **int** getX() {

**return** x;

}

**public** **void** setX(**int** x) {

**this**.x = x;

}

**public** **int** getY() {

**return** y;

}

**public** **void** setY(**int** y) {

**this**.y = y;

}

**public** ClassB getClassB() {

**return** classB;

}

**public** **void** setClassB(ClassB classB) {

**this**.classB = classB;

}

}

//dependency class

**class** ClassB {

**private** **int** a;

**private** **int** b;

ClassB(**int** a, **int** b) {

**this**.a = a;

**this**.b = b;

}

@Override

**public** String toString() {

**return** "ClassB [a=" + a + ", b=" + b + "]";

}

**public** **int** getA() {

**return** a;

}

**public** **void** setA(**int** a) {

**this**.a = a;

}

**public** **int** getB() {

**return** b;

}

**public** **void** setB(**int** b) {

**this**.b = b;

}

}

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

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

ClassA ca = **new** ClassA(10,20);

ClassA clonedCa = (ClassA) ca.clone();

System.***out***.println("original object : " + ca);

System.***out***.println("cloned object : " + clonedCa);

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

//making a change in the dependency object through original dependent object

ca.getClassB().setA(99);

ca.getClassB().setB(199);

System.***out***.println("After changing dependency object, through original dependent object");

System.***out***.println("original object : "+ ca);

System.***out***.println("cloned object : " + clonedCa);

}

}

output:

original object : ClassA [x=10, y=20, classB=ClassB [a=100, b=200]]

cloned object : ClassA [x=10, y=20, classB=ClassB [a=100, b=200]]

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

After changing dependency object, through original dependent object

original object : ClassA [x=10, y=20, classB=ClassB [a=99, b=199]]

cloned object : ClassA [x=10, y=20, classB=ClassB [a=99, b=199]]

//deep cloning example.

**package** com.ashokit.cloning;

//dependent class

**class** ClassA **implements** Cloneable {

**private** **int** x;

**private** **int** y;

**private** ClassB classB;

ClassA(**int** x, **int** y) {

// System.out.println("ClassA(int,int): constructor");

**this**.x = x;

**this**.y = y;

classB = **new** ClassB(100, 200);

}

@Override

**protected** Object clone() **throws** CloneNotSupportedException {

ClassB clonedB = (ClassB) classB.clone();

ClassA clonedA = (ClassA)**super**.clone();

clonedA.setClassB(clonedB);

**return** clonedA;

}

@Override

**public** String toString() {

**return** "ClassA [x=" + x + ", y=" + y + ", classB=" + classB + "]";

}

**public** **int** getX() {

**return** x;

}

**public** **void** setX(**int** x) {

**this**.x = x;

}

**public** **int** getY() {

**return** y;

}

**public** **void** setY(**int** y) {

**this**.y = y;

}

**public** ClassB getClassB() {

**return** classB;

}

**public** **void** setClassB(ClassB classB) {

**this**.classB = classB;

}

}

//dependency class

**class** ClassB **implements** Cloneable {

**private** **int** a;

**private** **int** b;

ClassB(**int** a, **int** b) {

**this**.a = a;

**this**.b = b;

}

@Override

**protected** Object clone() **throws** CloneNotSupportedException {

**return** **super**.clone();

}

@Override

**public** String toString() {

**return** "ClassB [a=" + a + ", b=" + b + "]";

}

**public** **int** getA() {

**return** a;

}

**public** **void** setA(**int** a) {

**this**.a = a;

}

**public** **int** getB() {

**return** b;

}

**public** **void** setB(**int** b) {

**this**.b = b;

}

}

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

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

ClassA ca = **new** ClassA(10,20);

ClassA clonedCa = (ClassA) ca.clone();

System.***out***.println("original object : " + ca);

System.***out***.println("cloned object : " + clonedCa);

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

//making a change in the dependency object through original dependent object

ca.getClassB().setA(99);

ca.getClassB().setB(199);

System.***out***.println("After making changes to the dependency object, through original dependent object");

System.***out***.println("original object : "+ ca);

System.***out***.println("cloned object : " + clonedCa);

}

}

output:

original object : ClassA [x=10, y=20, classB=ClassB [a=100, b=200]]

cloned object : ClassA [x=10, y=20, classB=ClassB [a=100, b=200]]

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

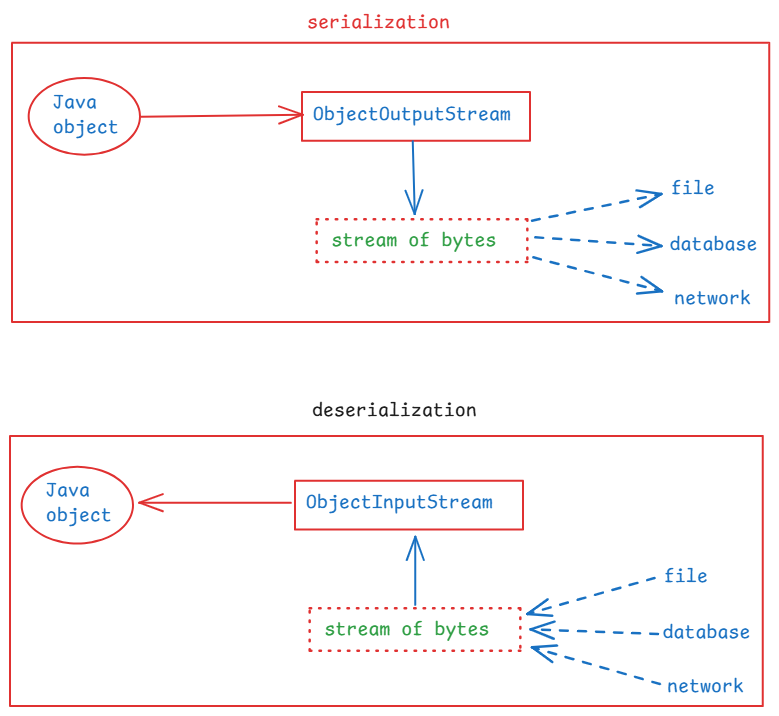
After making changes to the dependency object, through original dependent object

original object : ClassA [x=10, y=20, classB=ClassB [a=99, b=199]]

cloned object : ClassA [x=10, y=20, classB=ClassB [a=100, b=200]]

serialization & deserialization

* serialization is a process of converting a Java object into a stream of bytes, which can be easily saved in a file, transmitted over a network or stored in a database.
* deserialization is a reverse process of serialization. The stream of bytes are collected from a file, or network, or a database and converted back to a Java object.
* we serialize an object, when we want to save the state of an object into a file, or we want to send an object to another machine.
* To serialize a Java object, the class should implement a marker interface, java.io.Serializable.
* We use ObjectOutputStream class for serialization and ObjectInputStream class for deserialization.
* The writeObject() of ObjectOutputStream class converts a Java object into a stream of bytes.
* The readObject() of ObjectInputStream class converts a stream of bytes into a Java object.



// A demo on serialization

**package** com.ashokit.demo;

**import** java.io.FileInputStream;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutputStream;

**import** java.io.Serializable;

**class** Employee **implements** Serializable {

**int** empno;

String ename;

Employee(**int** empno, String ename) {

**this**.empno = empno;

**this**.ename = ename;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + "]";

}

}

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

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

Employee emp = **new** Employee(7878, "Scott");

//serialization

**try** ( FileOutputStream fileOut = **new** FileOutputStream("D:\\employee.ser");

ObjectOutputStream out = **new** ObjectOutputStream(fileOut)) {

out.writeObject(emp);

}

**catch**(IOException ie) {

ie.printStackTrace();

}

//deserialization

Employee deserializedEmp = **null**;

**try** ( FileInputStream fileIn = **new** FileInputStream("D:\\employee.ser");

ObjectInputStream in = **new** ObjectInputStream(fileIn)) {

deserializedEmp = (Employee) in.readObject();

}

**catch**(IOException | ClassNotFoundException ex) {

ex.printStackTrace();

}

System.***out***.println("Deserialized Employee : "+ deserializedEmp);

}

}

what is serialVersionUID?

. It is a unique identifier for each class that implements Serializable interface.

. It must be declared as a private static final long variable.

private static final long serialVersionUID = 1L;

. when an object is serialized, the serialVersionUID is also included in the stream of bytes.

. At the time of deserialization, JVM checks the id of the class and the id in the stream of bytes. If they match then only deserialization will proceed.

. Otherwise, JVM throws InvalidClassException.

what is transient keyword?

. If you want to stop any attribute of a class from serialization then you have to declare that variable as transient.

. when an object is serialized, the transient variable value will be stored as null, into the file.

. transient keyword can be used with only variables. Not with classes, interfaces or methods.

For example,

public class User implements Serializable {

private String username;

private transient String password;

//methods

}

( Please also refer Externalizable interface )

enum keyword:

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

. enum stands for enumeration.

. enum keyword is used to create a special data type.

. When you want to create a variable and that should be allowed to

store only one of the fixed set of constants as a value, then you

should create a special data type with enum keyword.

ex:

public enum Day {

MONDAY, TUESDAY, WEBNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY

}

. The constants in enum, are implicity static and final. So, we write the constants in upper case, but it is not a strict rule.

. You can create a variable of enum type as,

Day day;

. The variable can store one of the constants.

day = Day.MONDAY;

. Enums can also have fields, constructors and methods.

ex:

Gender.java

**public** **enum** Gender {

***MALE***("Male Person"), ***FEMALE***("Female Person"), ***OTHERS***("Other persons");

**private** String str; //field

Gender(String str) { //constructor

**this**.str = str;

}

**public** String getStr() { //method

**return** str;

}

**public** String sayHello() { //method

**return** "hello";

}

}

TestClass.java

**public** **class** TestClass {

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

Gender gender = Gender.***MALE***;

System.***out***.println(gender.sayHello()+ " : "+gender.getStr());

Gender gender2 = Gender.***FEMALE***;

System.***out***.println(gender2.sayHello()+ " : "+gender2.getStr());

//values(): returns an array of all enum constants

**for**(Gender g : Gender.*values*()) {

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

}

}

}

JVM internal components:

1. class loader
2. runtime data areas
3. execution engine
4. Java Native Interface(JNI)
5. Native Method libraries

. Class Loader is responsible for loading class files into JVM.

. Before JDK11, we have 3 types of class loaders.

1. Bootstrap class loader: loads the classes from Java API
2. Extension class loader: loads the classes from extensions

directory( jre/lib/ext).

1. Application class loader: loads the classes from the

application classpath

. From JDK11, we have 2 types of class loaders.

1. Bootstrap class loader
2. Application class loader

. Runtime data areas are,

1.method area: stores the class level information.

static variables, class structures and methods are

stored in this method area.

2.heap area: stores objects and arrays.

3.stack area: stores local variables, partial results of threads.

4.PC registers: Each thread has a Program Counter(PC), it holds the

address of the current instruction being executed by

the thread.

5.Native method stack: Manages native(non-Java) method calls.

. Execution Engine: It is responsible for reading byte code and executing it.

. Execution engine contains,

1. interpreter
2. Just-In-Time compiler(JIT)
3. Garbage Collector(GC)

. JNI allows the JVM to interact with native applications written in other languages like C/C++.

. Native method libraries are OS specific libraries and these will perform some tasks related system resources. JVM will access these libraries via JNI.

Generics

* Suppose, I have a Student class, now I want to create another class with methods, which can store or return a Student object.
* How to create this class?

public class StudentRecord {

Student[] records = new Student[100];

public void add(Student obj) {

//logic

}

public Student get(int id) {

//logic

}

}

* Suppose, I have a Book class, now I want to create another class with methods, which can store or return a Book object.
* How to create this class?

public class BookRecord {

Book[] records = new Book[100];

public void add(Book obj) {

//logic

}

public Book get(int id) {

//logic

}

}

* Suppose I have Staff class, now I want to create another class with methods, which can store or return a Staff object.
* How to create the class?
* Like the above classes, I need to create one more class StaffRecord with same functionalities.
* So, here we are creating multiple classes to implement similar functionality. Because, to work with different object types.
* So, if we create mulitple classes to implement similar functionalities, for different object types, the number of classes will be increased.
* So, to reduce the number classes, we create a single class and it can be used to different object types.
* This is possible with Generics.
* With Generics, we can create classes, interfaces and methods which can be used for different object types, but tied to a specific type.

For ex:

public class Record<T> {

T[] records = new T[100];

public void add(T obj) {

//logic

}

public T get(int id) {

//logic

}

}

* The above class is a generic class, and it can be used with different object types.
* For ex,

Record<Student> stuRecord = new Record<Student>();

This object stuRecord works with object type Student.

Record<Book> bookRecord = new Record<Book>();

This object bookRecord works with object type Book.

Record<Staff> staffRecord = new Record<Staff>();

This object staffRecord works with object type Staff.

* while creating Generic classes or Generic interfaces or Generic methods, some special characters are used like,

T -- Type

E -- Element

K -- Key

V -- Value

ID -- primary key type

* The advantage of Generics is,

1. reduces the need of creating number of classes
2. avoids type casting.

* While creating generic classes/methods, we can use 3 wild cards.

1. ? extends T : This wild card matches to the given type and

its sub types.

1. ? super T : This wild card matches to the given type and

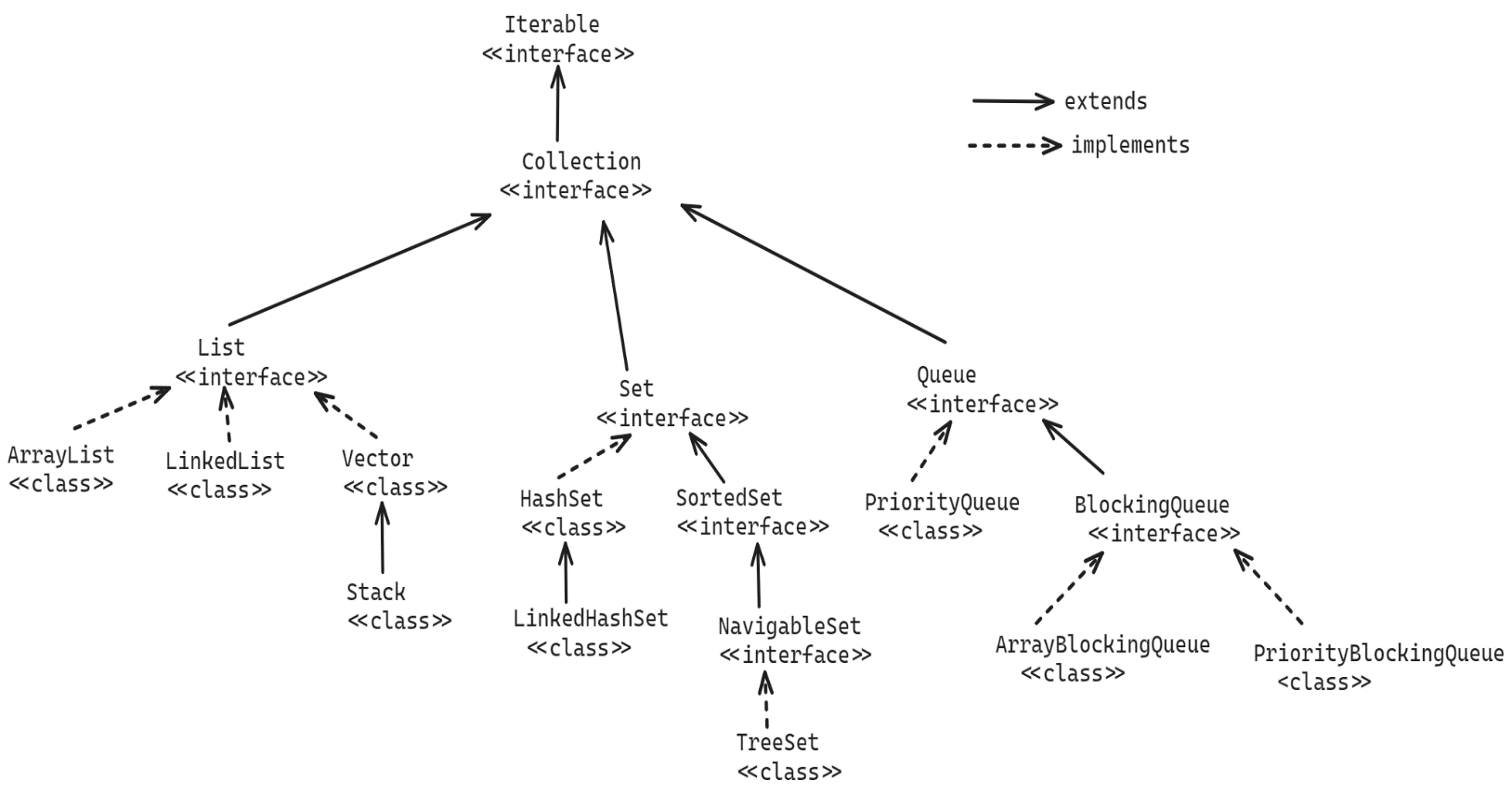
its super types.

1. ? : This wild card matches to the given type and its super

type and its sub type also.

Collections

* In realtime projects, mostly the data is in objects.
* In realtime projects, we have large number of objects like millions of objects.
* To some thing essential to store, manage and manipulate these groups of objects. So, we use collections.
* To manage the large amounts of data, 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.



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.

Methods of List interface:

* Since, List interface is inherited from Collection interface, the methods of Collection interface are also available in List interface.
* The List interface, also has some additional methods.

1. add(index, element) : inserts the specified element at the specified index.
2. get(index): returns the element from the specified position.
3. indexOf(element): returns the index of the first occurrence of the specified element. If not exist, then returns -1.
4. lastIndexOf(element): returns the last index of the specified element in this list. If not exist, then returns -1.
5. listIterator(): returns a ListIterator object over this list.
6. remove(index) : removes an element from the given index. It returns the removed element.
7. set(index, element): It will replace existing element with the new element at the given index.
8. sort(Comparator c): sorts the elements in this list, based on Comparator.
9. subList(fromIndex, toIndex): returns a list from fromIndex to toIndex-1.

ArrayList class:

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

* It is an resizable array implementation of List interface.
* A resizable array is also called a dynamic array and the array is not fixed, it will grow or shrink accordingly.
* If you want to store a group of elements and if we want to manage them or manipulate them or if you want to perform other operations like searching or sorting, then you need a collection object. Here that collection object ArrayList object.
* When you create an ArrayList object without specifying initial capacity, then the resizable array is created with default capacity as 10.

ex:

ArrayList<String> arrList = new ArrayList<String>();

* You can also specify the initial capacity, while creating the ArrayList object.

ex:

ArrayList<String> arrList = new ArrayList<String>(5);

* When an ArrayList object is added with all the elements, then its capacity will be increased to 50% of exisisting capacity.
* If exisiting capacity is 10, if we are adding the 11th element, then the capacity is increated to 15 ( 10 + 5).
* If we are adding 16th element, the capacity is increased to

22( 15 + 7 ).

* The type parameter fo a generic type should not be a primitive type.

ArrayList<int> a1 = new ArrayList<int>(); //error

ArrayList<Integer> a2 = new ArrayList<Integer>(); //valid

ArrayList<Employee> a3 = new ArrayList<Employee>(); //valid

ArrayList<Employee> a4 = new ArrayList<>(); //valid

ArrayList<> a5 = new ArrayList<Employee>(); //error

Iterating the ArrayList object:

ArrayList<String> arrList = new ArrayList<>();

arrList.add(“John”);

arrList.add(“Allen”);

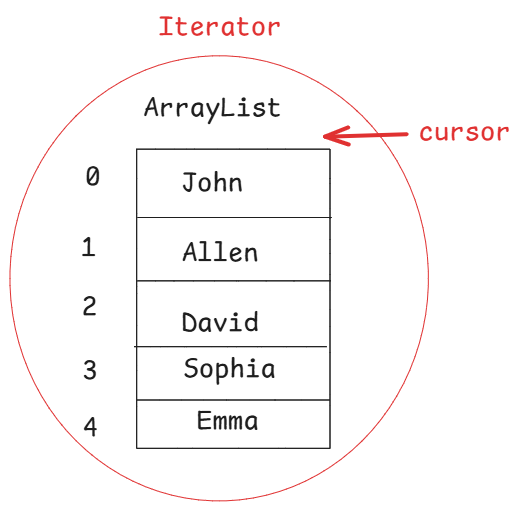
arrList.add(“David”);

arrList.add(“Sophia”);

arrList.add(“Emma”);

1. using Iterator object:

Iterator<String> iterator = arrList.iterator();



* when Iterator object is created, the object is created on the original collection object only and Iterator maintains a cursor and it is placed at before the first element initially.
* Iterator<T> is an interface, with 3 abstract methods.

1. hasNext() : checks for the next element after the cursor.

If exists returns true, otherwise returns false.

1. next() : moves the cursor to the next element and returns

that element.

1. remove(): removes the current element at the cursor.

while ( iterator.hasNext() ) {

S.o.println( iterator.next() );

}

1. using for each loop:

for ( String str : arrList ) {

S.o.println(str);

}

1. using ListIterator<E> object:

. ListIterator interface extends Iterator interface.

. using ListIterator, we can move the cursor in forward or

backward direction.

. Initially, the cursor is placed at before the first

element.

. To move the cursor in forward direction, the methods are

hasNext() & next(). To move the cursor in backward

direction, the methods are hasPrevious() & previous().

ListIterator<String> listIterator = arrList.listIterator();

/\*

\* This program prints the elements of the

\* ArrayList object in different approaches.

\*/

**package** com.pack;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.ListIterator;

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

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

ArrayList<String> arrList = **new** ArrayList<>();

arrList.add("John");

arrList.add("Allen");

arrList.add("David");

arrList.add("Sophia");

arrList.add("Emma");

arrList.add("David");

//using Iterator object

Iterator<String> iterator = arrList.iterator();

System.***out***.println("Printing the ArrayList using Iterator");

**while**( iterator.hasNext() ) {

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

}

//using for each loop

System.***out***.println("Printing the ArrayList using for each loop");

**for** ( String str : arrList ) {

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

}

//using ListIterator object

ListIterator<String> listIterator = arrList.listIterator();

System.***out***.println("Printing the ArrayList in reverse order using ListIterator");

**while**( listIterator.hasNext()) {

listIterator.next();

}

**while**( listIterator.hasPrevious() ) {

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

}

}

}

//ArrayList example with Employees.

Employee.java

**package** com.pack;

**public** **class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** **double** experience;

**private** **boolean** hasPassport;

**private** String gender;

**public** Employee(**int** empno, String ename, **double** sal, **double** experience, **boolean** hasPassport, String gender) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.experience = experience;

**this**.hasPassport = hasPassport;

**this**.gender = gender;

}

**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** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

**public** **boolean** isHasPassport() {

**return** hasPassport;

}

**public** **void** setHasPassport(**boolean** hasPassport) {

**this**.hasPassport = hasPassport;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", experience=" + experience

+ ", hasPassport=" + hasPassport + ", gender=" + gender + "]";

}

}

MainClass.java

**package** com.pack;

**import** java.util.ArrayList;

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

**private** **static** ArrayList<Employee> getEmployees() {

ArrayList<Employee> empList = **new** ArrayList<>();

empList.add(**new** Employee(7878, "Scott", 3000.0, 5.7, **false**, "Male"));

empList.add(**new** Employee(7201, "King", 4000.0, 6.5, **true**, "Male"));

empList.add(**new** Employee(7311, "Miller", 2000.0, 4.0, **false**, "Male"));

empList.add(**new** Employee(7532, "Lisa", 7000.0, 6.5, **true**, "Female"));

empList.add(**new** Employee(7809, "Clark", 5000.0, 5.9, **true**, "Male"));

empList.add(**new** Employee(7991, "Emma", 6000.0, 6.7, **false**, "Female"));

empList.add(**new** Employee(7549, "Sophia", 15000.0, 8.7, **true**, "Female"));

empList.add(**new** Employee(7188, "Blake", 7000.0, 5.9, **false**, "Male"));

empList.add(**new** Employee(7741, "Charlet", 9000.0, 7.5, **true**, "Female"));

empList.add(**new** Employee(7090, "David", 8000.0, 7.2, **false**, "Male"));

empList.add(**new** Employee(7762, "Rose", 9000.0, 8.9, **true**, "Female"));

**return** empList;

}

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

ArrayList<Employee> empList = *getEmployees*();

System.***out***.println("Printing Female employees of the ArrayList");

**for** ( Employee e : empList ) {

**if** ( e.getGender().equalsIgnoreCase("Female") ) {

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

}

}

System.***out***.println(":=".repeat(40));

System.***out***.println("Retrieving the employees who have a passport");

Iterator<Employee> iterator = empList.iterator();

**while**( iterator.hasNext() ) {

Employee emp = iterator.next();

**if** ( emp.isHasPassport()) {

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

}

}

System.***out***.println(":=".repeat(40));

System.***out***.println("Retrieving the employees with sal>5000, hava a passport and name should contain letter 's'");

**for** ( Employee e : empList ) {

**if** ( e.getSal() > 5000 && e.isHasPassport() && e.getEname().toLowerCase().contains("s") )

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

}

}

}

sorting the list:

* we can sort the elements of a list, in two ways.

1. using Comparable<T> interface
2. using Comparator<T> interface.

* Both Comparable and Comparator interfaces are the functional interfaces.
* Comparable interface has abstract method compareTo(T o)

and Comparator interface has abstract method

compare(T o1, T o2)

* To sort the elements of a list using Comparable, the class of the elements should implement Comparable interface and has to override compareTo().
* compareTo() method returns an integer value and it could be negitive, zero or positive.
* If negitive/zero is returned then order of the elements is not changed. But if positive is returned then order is changed.
* To sort the list, we have to call a static method sort() of java.util.Collections class.
* This sort() method can be called with either a single parameter or with 2 parameters.

sort(List<T> list)

sort(List<T> list, Comparator<T> comparator)

* To call the first method, the elements of the list must be Comparable elements.

//Employee.java

**public** **class** Employee **implements** Comparable<Employee> {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** **double** experience;

**private** **boolean** hasPassport;

**private** String gender;

**public** Employee(**int** empno, String ename, **double** sal, **double** experience, **boolean** hasPassport, String gender) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.experience = experience;

**this**.hasPassport = hasPassport;

**this**.gender = gender;

}

**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** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

**public** **boolean** isHasPassport() {

**return** hasPassport;

}

**public** **void** setHasPassport(**boolean** hasPassport) {

**this**.hasPassport = hasPassport;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", experience=" + experience

+ ", hasPassport=" + hasPassport + ", gender=" + gender + "]";

}

@Override

**public** **int** compareTo(Employee o) {

**return** (**int**) (**this**.getSal() - o.getSal());

}

}

MainClass.java

**package** com.pack;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Iterator;

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

**private** **static** ArrayList<Employee> getEmployees() {

ArrayList<Employee> empList = **new** ArrayList<>();

empList.add(**new** Employee(7878, "Scott", 3000.0, 5.7, **false**, "Male"));

empList.add(**new** Employee(7201, "King", 4000.0, 6.5, **true**, "Male"));

empList.add(**new** Employee(7311, "Miller", 2000.0, 4.0, **false**, "Male"));

empList.add(**new** Employee(7532, "Lisa", 7000.0, 6.5, **true**, "Female"));

empList.add(**new** Employee(7809, "Clark", 5000.0, 5.9, **true**, "Male"));

empList.add(**new** Employee(7991, "Emma", 6000.0, 6.7, **false**, "Female"));

empList.add(**new** Employee(7549, "Sophia", 15000.0, 8.7, **true**, "Female"));

empList.add(**new** Employee(7188, "Blake", 7000.0, 5.9, **false**, "Male"));

empList.add(**new** Employee(7741, "Charlet", 9000.0, 7.5, **true**, "Female"));

empList.add(**new** Employee(7090, "David", 8000.0, 7.2, **false**, "Male"));

empList.add(**new** Employee(7762, "Rose", 9000.0, 8.9, **true**, "Female"));

**return** empList;

}

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

ArrayList<Employee> empList = *getEmployees*();

Collections.*sort*(empList);

System.***out***.println("Employees after sorting based on salary");

**for** ( Employee e : empList ) {

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

}

}

}

* With Comparable, we can able to sort the elements of a list, always on a single property at a time.

Comparator interface:

* To sort the elements of a list with Comparator, first we have to create a class by implementing Comparator interface and we have to override compare() method.

For example:

**public** **class** EmpnoAscComparator **implements** Comparator<Employee> {

@Override

**public** **int** compare(Employee o1, Employee o2) {

**return** o1.getEmpno() - o2.getEmpno();

}

}

* By creating mulitple comparator classes, we can sort the elements of a list on multiple properties with in a single application.

For example:

**package** com.pack;

**import** java.util.Comparator;

**public** **class** EmpnoDescComparator **implements** Comparator<Employee> {

@Override

**public** **int** compare(Employee o1, Employee o2) {

**return** o2.getEmpno() - o1.getEmpno();

}

}

* In main class, I can modify the code like below.

ArrayList<Employee> empList = *getEmployees*();

Collections.*sort*(empList, **new** EmpnoAscComparator());

System.***out***.println("Employees after sorting based on empno ascending order");

**for** (Employee e : empList) {

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

}

System.***out***.println(":=".repeat(50));

Collections.*sort*(empList, **new** EmpnoDescComparator());

System.***out***.println("Employees after sorting based on empno descending order");

**for** (Employee e : empList) {

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

}

Q) what are the differences between Comparable and Comparator?

A)

LinkedList<E> class:

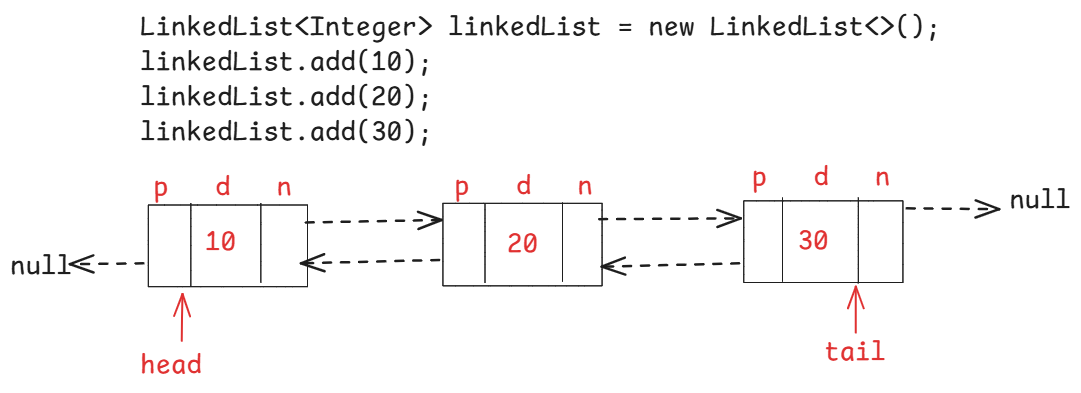
----------------------

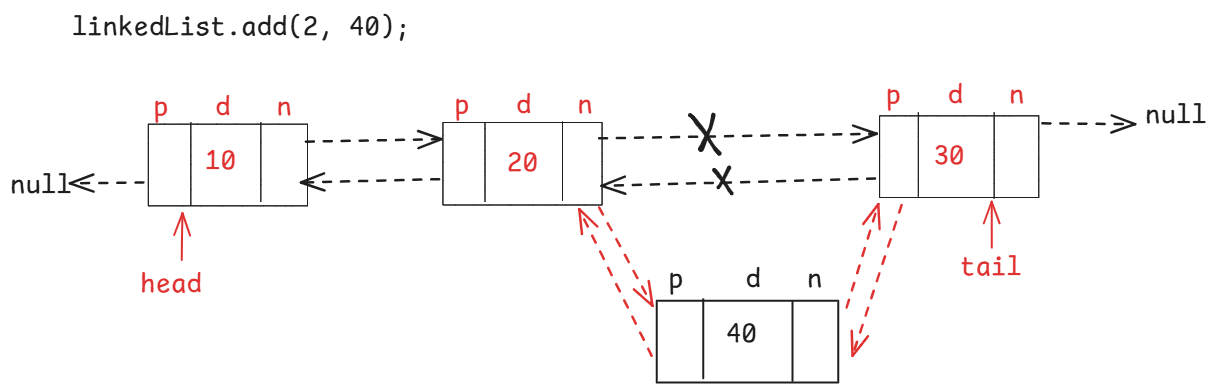
* LinkedList is a collection of Nodes.
* For each element, a Node object is created and this object

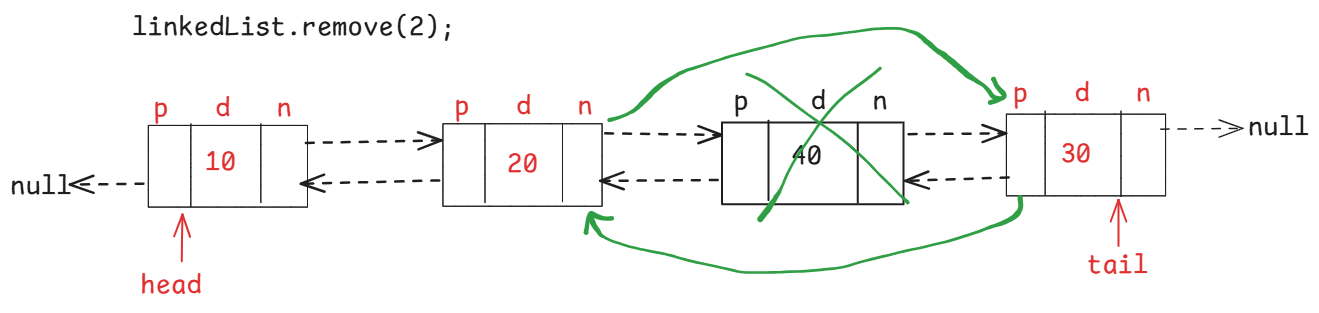
consists 3 parts.

1. data
2. next
3. previous

* next and previous are the two pointers where, next pointer points to the next node of the current node and previous pointer points to the previous node of the current node.
* LinkedList class is an implementation of double linked list data structure.
* A LinkedList also maintains two pointers called head and tail.
* head points the first node and tail points to the last node of the list.







* In a linked list, we can add/remove/fetch the elements from the head or from the tail directly.

For adding:

addFirst(E e)

addLast(E e)

For removing:

removeFirst()

removeLast()

pollFirst()

pollLast()

For fetching:

peekFirst()

peekLast()

getFirst()

getLast()

Q) what is the difference between removeFirst() & pollFirst()?

A) Both the methods retrieves and removes the first element. But, if the list is empty then removeFirst() throws NoSuchElementException, and pollFirst() returns null.

Q) what is the difference between peekFirst() & getFirst()?

A) Both the methods retrieves, but does not remove the first element. But, if the list is empty then peekFirst() returns null, and getFirst() throws NoSuchElementException.

**import** java.util.LinkedList;

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

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

LinkedList<Integer> linkedList = **new** LinkedList<>();

linkedList.add(10);

linkedList.add(20);

linkedList.add(30);

//inserting element at index 2

linkedList.add(2, 40);

linkedList.add(50);

linkedList.add(60);

System.***out***.println("The current head of the linked list : " + linkedList.peekFirst());

System.***out***.println("The current tail of the linked list : " + linkedList.peekLast());

System.***out***.println("The size of linked list before removal : " + linkedList.size());

//remove an element from index 2

linkedList.remove(2);

System.***out***.println("The size of linked list after removal : " + linkedList.size());

linkedList.addFirst(90);

linkedList.addLast(140);

System.***out***.println("The current head of the linked list : " + linkedList.peekFirst());

System.***out***.println("The current tail of the linked list : " + linkedList.peekLast());

//remove the first and last elements of the linked list

linkedList.pollFirst();

linkedList.pollLast();

System.***out***.println("The current head of the linked list : " + linkedList.peekFirst());

System.***out***.println("The current tail of the linked list : " + linkedList.peekLast());

System.***out***.println("The 3rd element of the linked list : " + linkedList.get(2));

}

}

Q) what is the difference between ArrayList and LinkedList?

A) When more times retrieval of elements is required in the application and occasionally adding/removing is required then use ArrayList object.

When more times adding/removing of elements is required in the application and occasionally retrieval is required then use LinkedList object.

Q) what is the difference between arrayList.get(index) and linkedList.get(index)?

A) In case of ArrayList, the cursor will directly jumps to the given index randomly and returns the element at the index.

In case of LinkedList, the cursor will move from the head pointer to the given index, one by one node then returns the element at the index.

ArrayList class implements RandomAccess, it is a marker interface, but LinkedList class doesn’t implement RandomAcess.

Q) is ArrayList object a thread-safe?

A) Not a thread-safe

Q) can we make ArrayList object as a thread-safe object?

A) Yes, by calling synchronizedList() of Collections class.

List<Employee> newLst = Collections.synchronizedList(empList);

Q) is ArrayList object a mutable object?

A) Yes.

Q) can we make ArrayList object as immutable object?

A) Yes. By calling unmodifiableList() method of Collections class.

List<Employee> newLst = Collections.unmodifiableList(empList);

Q) is LinkedList object a thread-safe?

A) Not a thread-safe

Q) can we make LinkedList object as a thread-safe object?

A) Yes, by calling synchronizedList() of Collections class.

List<Employee> newLst = Collections.synchronizedList(empList);

Q) is LinkedList object a mutable object?

A) Yes.

Q) can we make LinkedList object as immutable object?

A) Yes. By calling unmodifiableList() method of Collections class.

List<Employee> newLst = Collections.unmodifiableList(empList);

toString(), hashCode() and equals() methods:

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

* when you want to display an object’s data as a string then you have to override toString() method in that class.
* If you don’t override the toString() method and if you directly display the object, then Object class toString() method is called and it returns a string of format, classname@hexadecimalcode.
* So, you need to override the toString() method.

ex:

**class** ClassA {

**private** **int** a;

**private** **int** b;

ClassA(**int** a, **int** b) {

**this**.a = a;

**this**.b = b;

}

**public** **int** getA() {

**return** a;

}

**public** **void** setA(**int** a) {

**this**.a = a;

}

**public** **int** getB() {

**return** b;

}

**public** **void** setB(**int** b) {

**this**.b = b;

}

@Override

**public** String toString() {

**return** "ClassA [a=" + a + ", b=" + b + "]";

}

}

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

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

ClassA ca = **new** ClassA(10, 20);

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

}

}

* The requirement in the application is, each object should be given a unique code.
* If two objects are having the same content, then they must contain same unique code.
* To generate a unique code for an object, in Object class we have a method called hashCode().
* But this hashCode() method of the Object class will return different codes, even though they have the same content.
* Because, the hashCode() method of Object class, converts the memory address of the object to the integer value and returns it.
* So, we have to override the hashCode() method in our class, in such a way that if two objects are equal then they should have same hash code.

example:

**import** java.util.Objects;

**class** ClassA {

**private** **int** a;

**private** **int** b;

ClassA(**int** a, **int** b) {

**this**.a = a;

**this**.b = b;

}

**public** **int** getA() {

**return** a;

}

**public** **void** setA(**int** a) {

**this**.a = a;

}

**public** **int** getB() {

**return** b;

}

**public** **void** setB(**int** b) {

**this**.b = b;

}

@Override

**public** **int** hashCode() {

**return** Objects.*hash*(a, b);

}

}

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

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

ClassA ca1 = **new** ClassA(10, 20);

System.***out***.println(ca1.hashCode());

ClassA ca2 = **new** ClassA(10, 20);

System.***out***.println(ca2.hashCode());

ClassA ca3 = **new** ClassA(30, 40);

System.***out***.println(ca3.hashCode());

}

}

* The requirement in the application is, if two objects of a class have the same content/data, then we want to consider them as equal objects.
* When equals() method is called for comparision, the equals() method belongs to Object class is called. This equals() method internally uses the operator ==, for comparision and returns false, even they have same content.
* So, we have to override equals() method in the class, to compare the content/data in the two objects.

example:

**import** java.util.Objects;

**class** ClassA {

**private** **int** a;

**private** **int** b;

ClassA(**int** a, **int** b) {

**this**.a = a;

**this**.b = b;

}

**public** **int** getA() {

**return** a;

}

**public** **void** setA(**int** a) {

**this**.a = a;

}

**public** **int** getB() {

**return** b;

}

**public** **void** setB(**int** b) {

**this**.b = b;

}

@Override

**public** **int** hashCode() {

**return** Objects.*hash*(a, b);

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (**this**.getClass() != obj.getClass())

**return** **false**;

ClassA other = (ClassA) obj;

**return** **this**.a == other.a && **this**.b == other.b;

}

}

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

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

ClassA ca1 = **new** ClassA(10, 20);

ClassA ca2 = **new** ClassA(10, 20);

System.***out***.println(ca1 == ca2);

System.***out***.println(ca1.equals(ca2));

}

}

Q) what is the difference between java.util.List and java.util.Set?

A) List is ordered collection and allows duplicate elements.

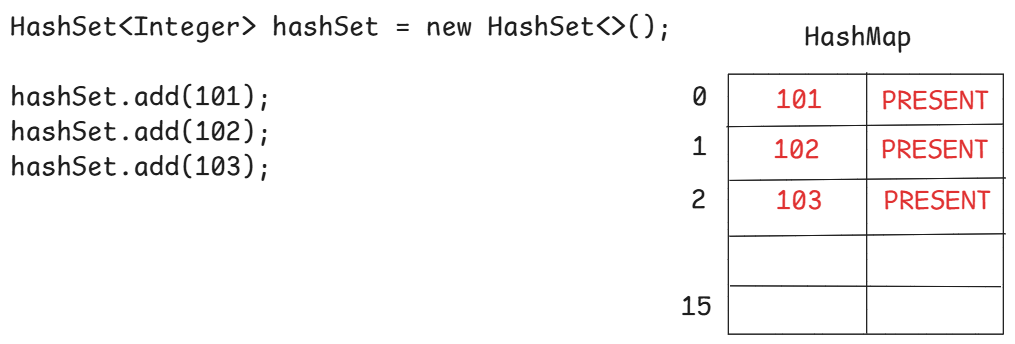
Set is un-ordered collection and doesn’t allow duplicate elements.

From a List, you can access an elements by using its index, but in a Set, you can not use indexes.

**HashSet class:**

**-------------**

* When we create a HashSet class object, it internally creates HashMap class object, with default initial capacity 16 and with load factor 0.75.
* A Map object, is a Hash table implementation, and it can store the data in key-value pairs.
* When we add an element to the HashSet object, internally this element is stored as a key and with value PRESENT into the HashMap object.
* This PRESENT is a dummy object, created for the Object class.



* The internal HashMap object, has 16 buckets with indexes 0 to 15.
* When 12 buckets are filled ( capacity \* loadfactor), then the internal map object capacity will be doubled.
* 16 \* 0.75 = 12
* The load factor is float value and it must be specified with in the range of 0.0 to 1.0.

Example:

HashSet<Integer> hashSet = new HashSet<>(10, 0.5f);

. Here, HashSet object is created with initial capacity 10 and the load factor 0.5.

* when we add an element to the HashSet object, internally, first hash code of the element is calculated, then bucket index is calculated with a formula hashCode(key) & (n-1), then the element is inserted into that bucket.

what happens, if we add a duplicate element?

* First hash code of the added element is calculated, then bucket index is calculated, and if that bucket is not empty then with equals() method, the added element is compared with the existing element in the bucket.
* if they are equal, then the added element will be discarded. But no error or an exception will be thrown.

//example code with HashSet object.

**import** java.util.HashSet;

**import** java.util.Iterator;

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

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

HashSet<String> hashSet = **new** HashSet<>();

hashSet.add("banana");

hashSet.add("pomogranite");

hashSet.add("apple");

hashSet.add("strawberry");

hashSet.add("pineapple");

hashSet.add("custerdapple");

hashSet.add("banana"); //duplicate element

System.***out***.println("The elements in HashSet object ");

Iterator<String> iterator = hashSet.iterator();

**while**( iterator.hasNext() ) {

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

}

}

}

//example code with Employee objects

//Employee.java

**import** java.util.Objects;

**public** **class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** **double** experience;

**private** **boolean** hasPassport;

**private** String gender;

**public** Employee(**int** empno, String ename, **double** sal, **double** experience, **boolean** hasPassport, String gender) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.experience = experience;

**this**.hasPassport = hasPassport;

**this**.gender = gender;

}

**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** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

**public** **boolean** isHasPassport() {

**return** hasPassport;

}

**public** **void** setHasPassport(**boolean** hasPassport) {

**this**.hasPassport = hasPassport;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", experience=" + experience

+ ", hasPassport=" + hasPassport + ", gender=" + gender + "]";

}

@Override

**public** **int** hashCode() {

**return** Objects.*hash*(empno, ename, experience, gender, hasPassport, 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*(experience) == Double.*doubleToLongBits*(other.experience)

&& Objects.*equals*(gender, other.gender) && hasPassport == other.hasPassport

&& Double.*doubleToLongBits*(sal) == Double.*doubleToLongBits*(other.sal);

}

}

//Solution.java

**import** java.util.HashSet;

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

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

HashSet<Employee> hashSet = **new** HashSet<>();

hashSet.add(**new** Employee(7878, "Scott", 3000.0, 5.7, **false**, "Male"));

hashSet.add(**new** Employee(7201, "King", 4000.0, 6.5, **true**, "Male"));

hashSet.add(**new** Employee(7311, "Miller", 2000.0, 4.0, **false**, "Male"));

hashSet.add(**new** Employee(7532, "Lisa", 7000.0, 6.5, **true**, "Female"));

hashSet.add(**new** Employee(7809, "Clark", 5000.0, 5.9, **true**, "Male"));

hashSet.add(**new** Employee(7991, "Emma", 6000.0, 6.7, **false**, "Female"));

hashSet.add(**new** Employee(7532, "Lisa", 7000.0, 6.5, **true**, "Female")); //duplicate element

**for** ( Employee e : hashSet ) {

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

}

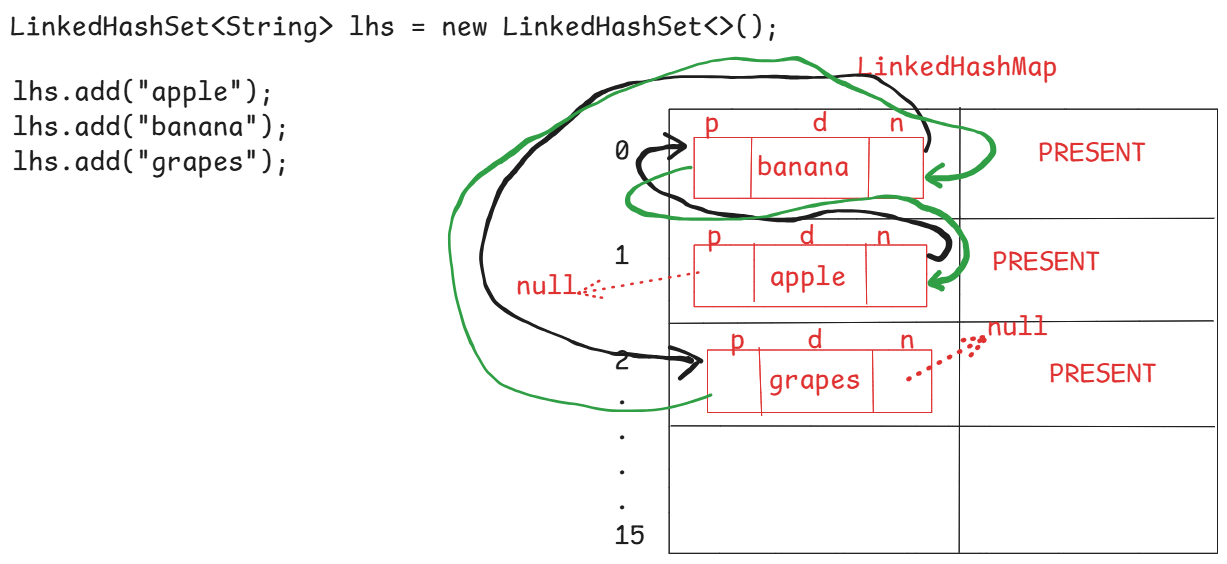
}

}

LinkedHashSet<E> class:

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

* LinkedHashSet class is a double linked list data structure implementation of Set interface.
* LinkedHashSet is a child class of HashSet class.
* HashSet class uses HashMap object internally, where as LinkedHashSet class uses LinkedHashMap object internally.
* HashSet object does not maintain insertion order of the elements, but LinkedHashSet object maintains the insertion order of the elements.



//Demo example

**import** java.util.LinkedHashSet;

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

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

LinkedHashSet<String> lhs = **new** LinkedHashSet<>();

lhs.add("apple");

lhs.add("banana");

lhs.add("strawberry");

lhs.add("grapes");

System.***out***.println("LinkedHashSet elements ");

**for** (String str : lhs) {

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

}

lhs.remove("banana");

System.***out***.println("LinkedHashSet elements after removing 'banana' ");

**for** (String str : lhs) {

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

}

}

}

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.

//demo program

**import** java.util.TreeSet;

**class** ClassA **implements** Comparable<ClassA> {

**private** **int** x;

**public** ClassA(**int** x) {

**this**.x = x;

}

@Override

**public** String toString() {

**return** "ClassA [x=" + x + "]";

}

@Override

**public** **int** compareTo(ClassA o) {

**return** **this**.x - o.x;

}

}

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

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

TreeSet<ClassA> ts = **new** TreeSet<>();

ts.add( **new** ClassA(20) );

ts.add( **new** ClassA(12) );

ts.add( **new** ClassA(15) );

ts.add(**null**);

**for** ( ClassA ca : ts) {

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

}

}

}

//demo on subset

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

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

TreeSet<String> fruits = **new** TreeSet<>();

fruits.add("banana");

fruits.add("grapes");

fruits.add("apple");

fruits.add("strawberry");

fruits.add("pineapple");

fruits.add("mango");

fruits.add("watermelon");

fruits.add("orange");

System.***out***.println("The elements of TreeSet");

**for** ( String str : fruits ) {

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

}

System.***out***.println("The subset elements of TreeSet");

SortedSet<String> fruitsSubSet = fruits.subSet("banana", "pineapple");

**for** ( String str : fruitsSubSet ) {

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

}

}

}

//demo on TreeSet with Comparator.

**import** java.util.Comparator;

**import** java.util.TreeSet;

**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** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + "]";

}

}

**class** SalComparator **implements** Comparator<Employee> {

@Override

**public** **int** compare(Employee o1, Employee o2) {

**return** (**int**) ( o1.getSal() - o2.getSal() );

}

}

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

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

TreeSet<Employee> employeesTreeSet = **new** TreeSet<>( **new** SalComparator() );

employeesTreeSet.add(**new** Employee(7676, "Scott", 7000.0));

employeesTreeSet.add(**new** Employee(7299, "King", 5000.0));

employeesTreeSet.add(**new** Employee(7877, "Miller", 9000.0));

employeesTreeSet.add(**new** Employee(7044, "Sophia", 6000.0));

employeesTreeSet.add(**new** Employee(7132, "Emma", 4000.0));

System.***out***.println("The elements of the TreeSet object");

**for** ( Employee e : employeesTreeSet ) {

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

}

}

}

Q) can we add null to the HashSet object?

A) Yes.

Q) can we add null to the LinkedHashSet object?

A) Yes.

Q) can we add null to the TreeSet object?

A) No, At runtime, NullPointerException will be thrown.

Queue<E> interface:

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

* Queue orders elements in First In First Out(FIFO) manner, but not necessarily.
* A queue maintain two pointers called head and tail.
* 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.

//Demo – PriorityQueue

**import** java.util.Comparator;

**import** java.util.Iterator;

**import** java.util.PriorityQueue;

**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** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + "]";

}

}

**class** EmpnoComparator **implements** Comparator<Employee> {

@Override

**public** **int** compare(Employee o1, Employee o2) {

**return** o1.getEmpno() - o2.getEmpno();

}

}

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

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

PriorityQueue<Employee> priorityQueue = **new** PriorityQueue<Employee>(**new** EmpnoComparator());

priorityQueue.add(**new** Employee(7676, "Scott", 7000.0));

priorityQueue.add(**new** Employee(7299, "King", 5000.0));

priorityQueue.add(**new** Employee(7877, "Miller", 9000.0));

priorityQueue.add(**new** Employee(7044, "Sophia", 6000.0));

priorityQueue.add(**new** Employee(7132, "Emma", 4000.0));

Iterator<Employee> it = priorityQueue.iterator();

**while**( it.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.

//Demo on ArrayBlockingQueue

**import** java.util.concurrent.ArrayBlockingQueue;

**class** Thread1 **extends** Thread {

ArrayBlockingQueue<Integer> abq;

**public** Thread1(ArrayBlockingQueue<Integer> abq) {

**this**.abq = abq;

}

@Override

**public** **void** run() {

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

**for** (**int** i = 1; i <= 6; i++) {

**try** {

**if**( i==6 ) {

System.***out***.println("element taken : " + abq.take());

}

abq.put(i);

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

} **catch** (Exception ex) {

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

}

}

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

}

}

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

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

ArrayBlockingQueue<Integer> abq = **new** ArrayBlockingQueue<Integer>(5);

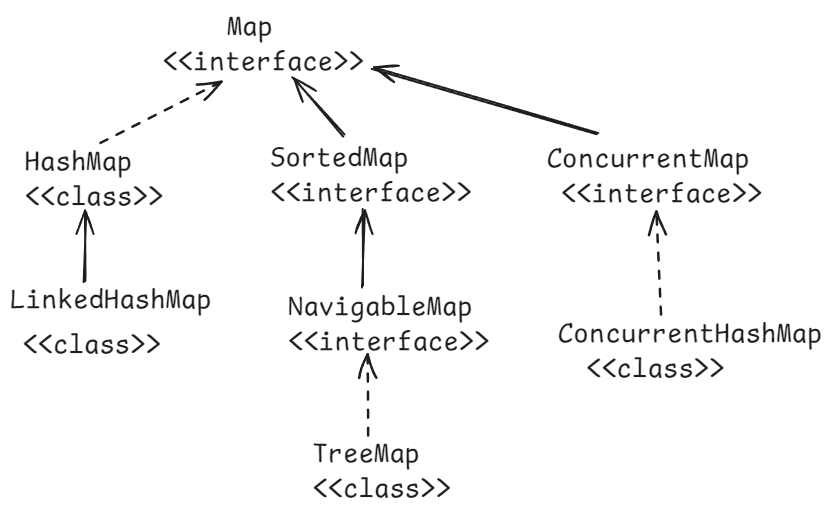
Thread1 t1 = **new** Thread1(abq);

t1.start();

}

}

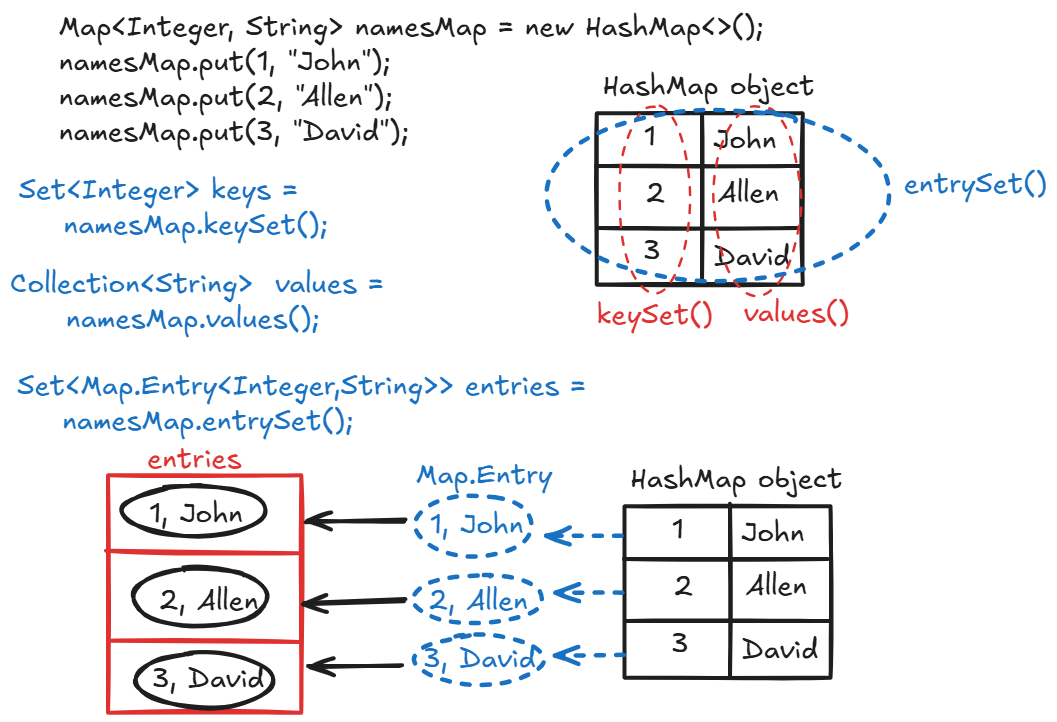
Map<E> interface:



* 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 object.

**import** java.util.Collection;

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Set;

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

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

HashMap<Integer, String> namesMap = **new** HashMap<>();

namesMap.put(1, "John");

namesMap.put(2, "Allen");

namesMap.put(3, "David");

namesMap.put(4, "Emma");

namesMap.put(5, "Juvan");

namesMap.put(3, "Lisa");

namesMap.putIfAbsent(5, "Sophia");

namesMap.put(**null**, **null**);

System.***out***.println("keys of the map are :");

Set<Integer> keys = namesMap.keySet();

**for** ( Integer k : keys ) {

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

}

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

System.***out***.println("values of the map are : ");

Collection<String> values = namesMap.values();

**for**(String str : values ) {

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

}

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

Set<Map.Entry<Integer, String>> entries = namesMap.entrySet();

System.***out***.println("entries of the map are : ");

**for** ( Map.Entry<Integer, String> entry : entries ) {

System.***out***.println(entry.getKey() + " -- " + entry.getValue() );

}

}

}

Q) If I want to store empno as key and that Employee object as a value then how do you create a HashMap object?

A) HashMap<Integer, Employee> hm = new HashMap<>();

Q) is HashMap object a thread-safe?

A) Not a thread-safe

Q) can we make HashMap object as a thread-safe object?

A) Yes, by calling synchronizedMap() of Collections class.

Map<Integer, Employee> newMap = Collections.synchronizedMap(empMap);

Q) is HashMap object a mutable object?

A) Yes.

Q) can we make HashMap object as immutable object?

A) Yes. By calling unmodifiableMap() method of Collections class.

Map<Integer, Employee> newMap = Collections.unmodifiableMap(empMap);

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

Q) what is the difference between Hashtable and HashMap classes?

A) Hashtable is a legacy class and HashMap is a collection framework class.

Hashtable is a synchronized object, so it is thread-safe by default. But

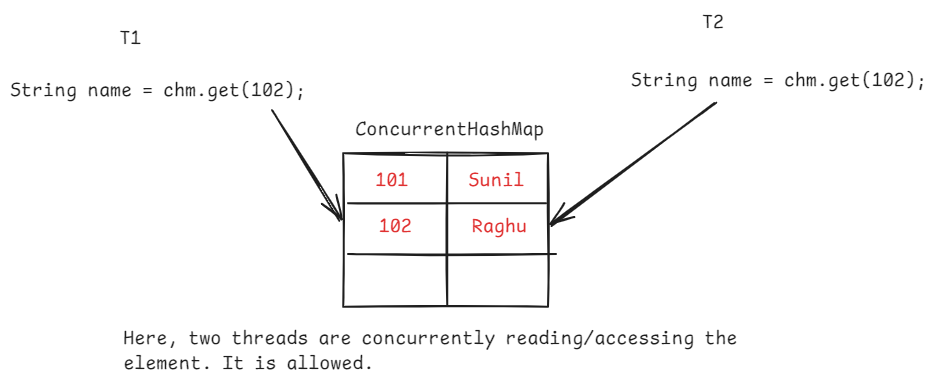
HashMap is not a synchronied object, so it is not a thread-safe by

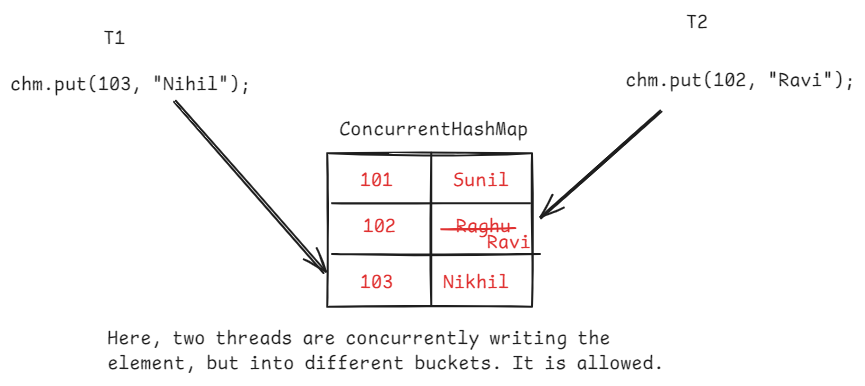
default.

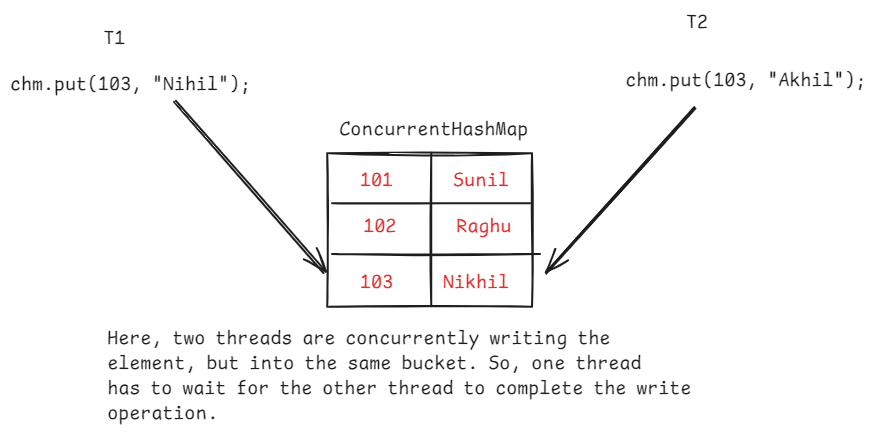
Hashtable does not allow null keys and null values. But HashMapp allows.

ConcurrentHashMap :

* ConcurrentHashMap class implements ConcurrentMap interface and ConcurrentMap interface extends Map interface.
* ConcurrentHashMap class is a thread-safe variant of HashMap class.
* ConcurrentHashMap allows multiple threads at a time to access its elements.
* ConcurrentHashMap provides better performance in multi-threading applications, because of bucket-level locking.
* ConcurrentHashMap also has default initial capacity 16 with load factor 0.75.







Q) what is difference between HashMap and ConcurrentHashMap objects?

A) 1. HashMap object is not a thread-safe by default. But ConcurrentHashMap object is thread-safe by default.

2. If HashMap object is converted to thead-safe, then entire HashMap object is locked. So, one thread has to wait, for the completion of another thread, even though the two threads wants to modify the different buckets.

But in ConcurrentHashMap object, the threads are no need to wait, while working with different buckets. So, ConcurrentHashMap object provides better performance.

3.HashMap object allows to store null key and null value. But ConcurrentHashMap object can not store null key or null value.

4. Iterator on HashMap object is fail-fast and Iterator on ConcurrentHashMap object is fail-safe.

Fail-fast and Fail-safe iteration:

* Fail-fast and Fail-safe terms represents, how an iterator behaves if the underlying collection object’s structure is modified, while iterating over it.
* Fail-fast iterator means, while iterating over a collection object, suppose if that collection object’s structure is modified by adding/removing an element, then iterator throws ConcurrentModificationException.
* Fail-safe iterator means, while iterating over a collection object, suppose if that collection object’s structure is modifies by adding/removing an element, then iterator does not throw ConcurrentModificationException.
* Iterator on collection objects like ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, HashMap, LinkedHashMap, TreeMap and Queue is Fail-fast. But Iterator on ConcurrentHashMap and CopyOnWriteArrayList objects is Fail-safe.

//fail-fast example code:

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

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

HashMap<Integer, String> namesMap = **new** HashMap<>();

namesMap.put(1, "John");

namesMap.put(2, "Allen");

namesMap.put(3, "David");

namesMap.put(4, "Emma");

namesMap.put(5, "Juvan");

namesMap.put(3, "Lisa");

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

Set<Map.Entry<Integer, String>> entries = namesMap.entrySet();

System.***out***.println("entries of the map are : ");

Iterator<Map.Entry<Integer, String>> iterator = entries.iterator();

**while**(iterator.hasNext()) {

Map.Entry<Integer, String> entry = iterator.next();

System.***out***.println(entry.getKey() + " -- " + entry.getValue());

namesMap.put(6, "Dolly");

}

}

}

//Fail-safe example code:

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

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

ConcurrentHashMap<Integer, String> namesMap = **new** ConcurrentHashMap<>();

namesMap.put(1, "John");

namesMap.put(2, "Allen");

namesMap.put(3, "David");

namesMap.put(4, "Emma");

namesMap.put(5, "Juvan");

namesMap.put(3, "Lisa");

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

Set<Map.Entry<Integer, String>> entries = namesMap.entrySet();

System.***out***.println("entries of the map are : ");

Iterator<Map.Entry<Integer, String>> iterator = entries.iterator();

**while**(iterator.hasNext()) {

Map.Entry<Integer, String> entry = iterator.next();

System.***out***.println(entry.getKey() + " -- " + entry.getValue());

namesMap.put(6, "Dolly");

}

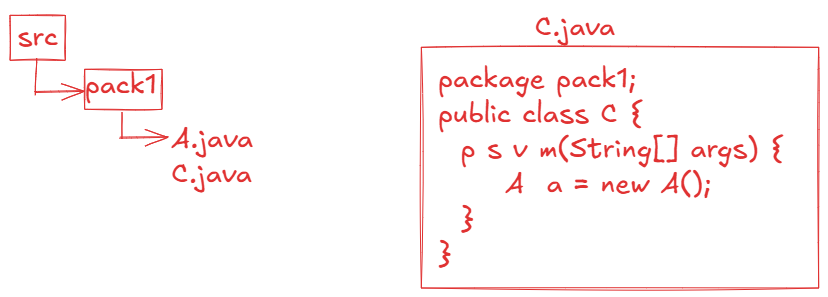
}

}

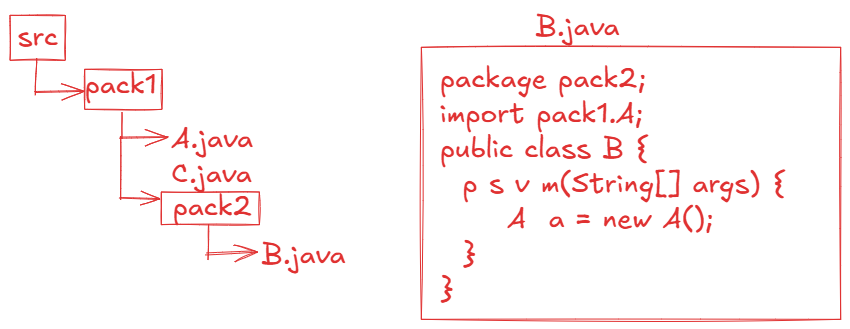
Packages:

--------

* a package is a group of classes, which exihibit similar functionalities.
* Suppose, the classes which exihibit request handling functionality will be grouped into one package and the classes which exhibit business logic functionality will be grouped into another package.
* Java API has provided multiple packages, where each package consists a group of classes and they exhibit similar functionalities.
* For example, the classes in java.lang package will proivde basic java language functionality and the classes in java.util package will provide utility functionality.
* A package consists classes, interfaces and may be sub-packages.
* A package can be created with a keyword called package.
* Every package name is a folder in the project.
* A package name and a sub-packge name are separated with a dot(.)
* With packages, we can also avoid the name collisions.
* Suppose, if I want to create two classes with the same name, then I can store them in two different packages, to avoid the name collision.
* Suppose, if two classes are with in the same package then we don’t require import statement to use one class in another class.



* Suppose, if a class is in parent package and another class is in sub-package and if we want to use parent package class in child package class or vice-versa then import statement is required.



Note: If class A in parent package has default or protected variables then they are not directly visisble to the class B in sub-packge and vice-versa.

Java8 features

* Java8 is one of the major versions of Java, provided multiple important features to the realtime application development.

1. defualt and static methods in interface

2. functional interfaces

3. lambda expressions

4. Optional class

5. method reference operator

6. stream api

7. date/time api

8. Collectors class

9. Base64Encoder

10. CompletableFuture class

default and static methods :

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

* Before Java8 version, we can write public static final variables and public abstract methods in interface.
* Suppose, in a project development, an interface is created with hundreds of implementation classes already and it is running smoothly.
* Now, suddenly, a change is made to a requirement and to implement this change into the project, you need to add an abstract method to the interface.
* If you add a new abstract method to the interface then all the implmentation classes must override the new abstract method.
* When hundreds of implementation classes exist, then it is a time consuming process.
* So, In Java8, default methods concept is introduced.
* Instead of creating a new abstract method, you can create a default method with default implementation in the interface.
* So, the classes can use the default implementation or if they want then they can override it. So, with the default methods, the implementation classes are no need to override the new method.

ex:

interface Remunerator {

double calculateSalary(); //abstract method

default void deductFoodFee() { //default method

//default implementation

}

}

* Suppose, If a requirement has changed and all the implmentation classes has to implement this requirement in exactly the same way, then instead of creating an abstract method and overriding it in all the implementation classes, you can create a static method in the interface.

example:

public interface Remunerator {

double calculateSalary(); //abstract method

default double deductFoodFee() { //default method

//default implementation

}

static double deductInsurancePremium() {

//logic

}

}

Q) what is the difference between default methods and static methods in interface?

A) default methods can be overridden by the implementation classes, but static methods can’t be overridden.

Q) what is the interface definition from Java8?

A) Interface is a collection of public static final variables, public abstract methods, default methods and static methods.

Functional interfaces:

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

* a functional interface is an interface with a single abstract method(SAM).
* a functional interface can contain any number of default methods and static methods and constants.

ex:

interface I1 {

void m1(); //abstract method

default void m2() { //default method

//default logic

}

static void m3() { //static method

//logic

}

}

* The above interface has only one abstract method. So, it is called as a functional interface.

ex:

interface I1 {

void m1(); //abstract method

void m2(); //abstract method

default void m3() { //default method

//default logic

}

}

* The above interface has 2 abstract methods, so it is a normal interface.

@FunctionalInterface:

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

* This annotation is used to specify that an interface is a functional interface.
* If more than one abstract method is created in the interface, then we will get the compile-time error.

ex1:

@FunctionalInteface

public interface MyInterface {

void m1();

void m2();

default void m3() {

S.o.p(“default method”);

}

}

* This example gets compile-time error, because it has 2 abstract methods.

ex2:

public interface MyInterface {

void m1();

void m2();

default void m3() {

S.o.p(“default method”);

}

}

* This example does not get any error. It is going to become normal interface.

ex3:

@FunctionalInterface

public interface MyInterface {

void m1();

}

* This example is functional interface.

ex4:

@FunctionalInterface

**public** **interface** MyInterface {

**public** **abstract** **void** m1();

**public** **abstract** **boolean** equals(Object o);

**public** **abstract** **int** hashCode();

}

* Eventhough this example has 3 abstract methods in the interface, but still it is a functional interface only.
* The methods equals() and hashCode() are added as abstract methods, but they are matching with the methods of Object class.
* So, if we add any Object class methods as abstract methods in an interface, then they are not considered as abstract methods.
* If a class implements this interface then that class must override m1() method, but overrriding equals() and hashCode() methods is optional.

Functional interface ----- abstract method

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

Runnable ----- void run()

Callable ----- V call()

Comparator<E> ----- int compare(E o1, E o2)

Comparable<E> ----- int compareTo(E o)

Consumer<T> ----- void accept(T t)

Predicate<T> ----- boolean test(T t)

Function<T> ----- R apply(T t)

Supplier<T> ----- T get()

Lambda expression:

requirement:

. I have a list of employees, I would like to sort the list in ascending order of their salaries. What to do?

Ans: 1. create a class by implementing Comparator interface and override compare() method.

2.call sort() method of Collections class with arguments, list and Comparator object.

requirement:

. I have a list of employees, I would like to sort the list in descending order of their salaries. What to do?

Ans: 1. create a class by implementing Comparator interface and override compare() method.

2.call sort() method of Collections class with arguments, list and Comparator object.

requirement:

. I have a list of employees, I would like to sort the list in ascending order of emp numbers. what to do?

Ans:

1. create a class by implementing Comparator interface and override compare() method.

2.call sort() method of Collections class with arguments, list and Comparator object.

. In a project, if mulitple requirements are there like this, then you have to create mulitple classes to implement this functional interface.

. If number of classes are increased, then complexity is also increased in the project.

. So, to implement functional interfaces, by without creating the classes, we have got lambda expressions in Java 8.

syntax of lambda expression:

(arguments) -> body;

* The number of arguments and type of the arguments should match with the parameters of the abstract method in the functional interface.
* The body of the lambda expression should return a value that must match with the return type of the abstract method.
* If there are zero or more than one arguments then paranthesis is mandatory. If there is a single argument then paranthesis is optional.
* If the body has mulitple statements then curly braces are mandatory, and for a single statement, the curly braces are optional.
* The datatype of the arguments is also optional. But don’t specify the datatype for one argument and not for another argument.
* We can store a lambda expression in interface reference variable.

example:

Comparator<Employee> comparator = (e1, e2) -> {

return e1.getEmpno() – e2.getEmpno();

};

(or)

Comparator<Employee> comparator = (Employee e1, Employee e2) -> {

return e1.getEmpno() – e2.getEmpno();

};

sort(Comparator<T> c) & forEach(Consumer<T> c) :

* From Java8 version, sort() method is included in the List interface as a default method and forEach() method is included in the Iterable interface as a default method.
* So, From Java8 version, suppose, if you want to sort the list elements then you no need to use Collections.sort() method. Similarly, to traverse the elements of a collection, you no need to use Iterator object or for each loop.
* If a method has an object of type Functional interface as parameter, then we can pass lambda expression as a parameter while calling that method.

/\*

\* write a program to sort and then display

\* list of employees with the default methods

\* sort() and forEach()

\*/

**package** com.pack;

**import** java.util.Arrays;

**import** java.util.List;

**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;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + "]";

}

}

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

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

List<Employee> lstEmp = Arrays.*asList*(

**new** Employee(7542, "David"),

**new** Employee(7367, "Allen"),

**new** Employee(7824, "King"),

**new** Employee(7719, "Miller")

);

/\*

//sort the list

Comparator<Employee> comparator = (e1, e2) -> {

return e1.getEmpno() - e2.getEmpno();

};

lstEmp.sort(comparator);

//display the list

Consumer<Employee> consumer = (e) -> System.out.println(e);

lstEmp.forEach(consumer);

\*/

//sort the list

lstEmp.sort( (e1, e2) -> e1.getEmpno() - e2.getEmpno() );

//display the list

lstEmp.forEach( e -> System.***out***.println(e));

}

}

Optional<T> class:

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

* In a realtime project, we have all objects in the code.
* we interact with the objects, by calling its operations/methods.
* If an object has null value and if you call any method on it, then NullPointerException will be thrown.
* The most frequently occurred exception in Java project is NullPointerException.
* To avoid NullPointerExcepton’s, in the code we have to perform null checks.

For example:

Employee e = service.getEmployeeById(7671);

if ( e != null )

{

S.o.println(“empno = “ + e.getEmpno());

S.o.println(“Sal = “ + e.getSal() );

}

* If you add more null checks, then the code becomes more complicated.

For example:

if ( student != null ) {

Address address = student.getAddress();

if ( address != null ) {

State state = address.getState();

if ( state != null ) {

City city = state.getCity();

if ( city != null ) {

String str = city.getName();

S.o.println(str);

}

}

}

}

* To reduce the null checks and also to avoid NullPointerException’s in a Java code, Java8 has provided Optional<T> class.
* If a method has to return an object, then it has to return Optional object, by storing the value in the Optional object.
* Optional class object is a container object, which may or may not contain the non-null value.

Creating Optional<T> object:

* 3 ways.
  + 1. empty()
    2. of(T value)
    3. ofNullable(T value)

Optional<String> opt = Optional.empty(); //creates empty Optional instance

Optional<String> opt = Optional.of(“ashokit”); //creates Optional object, contains “ashokit” string.

Optional<String> opt = Optional.of(null); //throws NPE

Optional<String> opt = Optional.ofNullable(“ashokit”);

Optional<String> opt = Optional.ofNullable(null); //creates empty Optional instance. But doesn’t throw NPE.

* some key methods of Optional class are,

isPresent() : returns true, if the value is present. Otherwise returns false.

get() : returns the value present in the Optional object. If value is not present then NoSuchElementException will be thrown.

ex:

Optional<String> opt = Optional.ofNullable(“ashokit”);

if ( opt.isPresent() ) {

String str = opt.get();

S.o.p(str);

}

ifPresent(Consumer<T> consumer): executes the given consumer, if the value is present in the Optional object. Otherwise, do nothing.

ex:

Optional<Employee> opt = Optional.ofNullable(e);

opt.ifPresent( e -> System.out.println(e) );

orElse(T another): Returns the value if present, otherwise returns the another.

example:

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

**private** **static** Employee getEmployee() {

**return** **new** Employee(2877, "Miller");

//return null;

}

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

Employee emp = *getEmployee*();

Optional<Employee> opt = Optional.*ofNullable*(emp);

Employee e = opt.orElse(**new** Employee(3232, "John"));

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

}

}

output: Employee[empno=2877, ename=Miller]

orElseThrow(Supplier<T> supplier): Returns the value if present in the Optional object. Otherwise, throws the exception object passed as a supplier.

Employee e = opt.orElseThrow( () -> new NoSuchElementException() );

S.o.p(e);

stream api

* If you want to store a group of objects, we use a collection.
* By storing a group of elements, we can perform operations on the elements of a collection.
* To perform the operations, we need to iterate the collection using Iterator or for each loop.
* If the collection has more elements, then processing will take more time.
* This problem is identified and in Java8, a solution is provided as java.util.stream.Stream interface.
* a collection and a stream are used for different purpose. A collection is used for storing the data and a stream is used for processing the data.

creating a Stream object:

. 4 ways.

* + 1. we can create a Stream for a collection.

Stream<Employee> s = empList.stream();

* + 1. we can create a Stream for an array.

Stream<Integer> s = Arrays.stream(arr);

* + 1. we can create a Stream for a raw values.

Stream<Integer> s = Stream.of(1,2,3,4,5);

* + 1. we can create an empty stream.

Stream<Void> s = Stream.empty();

* To perform a computation, stream [operations](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#StreamOps) are composed into a stream pipeline. A stream pipeline consists of a source (which might be an array, a collection, a generator function, an I/O channel, etc), zero or more intermediate operations (which transform a stream into another stream, such as [filter(Predicate)](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#filter-java.util.function.Predicate-)), and a terminal operation (which produces a result or side-effect, such as [count()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#count--) or [forEach(Consumer)](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#forEach-java.util.function.Consumer-)).
* To process the elements of a Stream, we have two types of operations.

1. intermediate operations
2. terminal operations.

For example:

Stream<Employee> stream = empList.stream();

Stream<Employee> stream2 = stream.filter( e -> e.getSalary() > 5000);

. Here, filter() is an intermediate operation. Because it is transforming one stream to another .

long count = stream2.count();

. Here, count() is a terminal opertion. Because it is producing a result not another stream.

* The stream operations like filter(), map(), sorted(), iterate(), peek(), skip(), limit(), flatMap() etc.. are intermediate operations. Because they produces/returns another stream.
* The stream operations like count(), collect(), findFirst(), max(), min(), reduce(), forEach(), etc.. are terminal operations.

ex1:

print the employees of a list with salary >= 5000

empList.stream().filter( e -> e.getSal() >= 5000 )

.forEach(e -> System.***out***.println(e));

ex2:

count the employees of a list with salary >= 5000

long count = empList.stream()

.filter(e -> e.getSal() >= 5000)

.count();

ex3:

find the employee of a list with highest salary

Optional<Employee> opt = empList.stream()

.sorted( (e1,e2) -> (**int**) (e2.getSal() - e1.getSal()))

.findFirst();

//opt.ifPresent(e -> System.out.println(e));

**if**(opt.isPresent()) {

Employee e = opt.get();

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

}

ex3:

find the employees of a list with salary > 4000 and sort them in ascending order and then display.

empList.stream().filter( e -> e.getSal() > 4000 )

.sorted( (e1, e2) -> (int)(e1.getSal() – e2.getSal()))

.forEach( e -> System.out.println(e));

ex4:

collect the employees names of a list of employees and store them in another list.

/\*

Stream<Employee> stream = empList.stream();

Stream<String> stream2 = stream.map(Employee::getEname);

List<String> lst = stream2.collect(Collectors.toList());

lst.forEach(System.out::println);

\*/

List<String> lst = empList.stream()

.map(Employee::getEname)

.collect(Collectors.*toList*());

lst.forEach(System.***out***::println);

ex5:

find the second highest paid employee from a list of employees.

empList.stream()

.sorted( (e1,e2) -> (**int**) (e2.getSal() - e1.getSal()))

.skip(1)

.findFirst()

.ifPresent(System.out::println);

ex6:

checking for an employee with sal > 8000 in the list of employees.

boolean flag = empList.stream()

.anyMatch(Employee::getSal > 8000);

ex7:

checking for all the employees with sal > 3000 in the list of employees.

boolean flag = empList.stream()

.allMatch(Employee::getSal > 3000);

ex8:

create a new list for a list of employees, after incrementing their salary by 5000

List<Employee> newList = empList.stream()

.map(e ->

{

e.setSal(e.getSal() + 5000);

**return** e;

}

)

.collect(Collectors.*toList*());

newList.forEach(System.***out***::println);

ex9:

print only first 5 highest paid employees

empList.stream()

.sorted( (e1, e2) -> (**int**) (e2.getSal() - e1.getSal()))

.limit(5)

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

ex9:

// Collect names into a List, whose length > 3

List<String> names = Arrays.*asList*("John", "Jane", "Tom", "Doe", "Jill", "Jeffry", "Jackson");

List<String> collectedNames = names.stream()

.filter(name -> name.length() > 3)

.collect(Collectors.*toList*());

collectedNames.forEach(System.***out***::println);

ex10:

// Join names with a comma separator

List<String> names = Arrays.*asList*("John", "Jane", "Tom", "Doe", "Jeffry");

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

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

ex11:

List<Integer> numbers = Arrays.*asList*(1, 2, 3, 4, 5);

// Sum all numbers

**int** sum = numbers.stream()

.collect(Collectors.*summingInt*(Integer::intValue));

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

ex12:

Group the employees of a list by their gender value.

Map<String, List<Employee>> map = empList.stream()

.collect(Collectors.*groupingBy*(Employee::getGender));

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

parallelStream():

. If the source of data to process is huge, like a list of 10 million employees and if we use a stream then it takes more time to process the elements.

. Because, a normal stream can process the elements sequentially.

. To improve the performance, we have to use parallel stream.

. parallel stream divides the source of data into data chunks and

executes the stream operations on each chunk on a separate core of the

cpu.

. after processing, it will join the results obtained from each core and

returns the final result.

. So, parallel stream utilizes the full potential of the cpu, to perform

the operations.

For example:

List<Employee> lst = empList.parallelStream()

.filter(e -> e.getSal() > 5000)

.collect(Collectors.toList());

1. Spliterator interface
2. The Spliterator is a special-purpose iterator designed for parallel processing of elements in streams and collections.
3. It can split a data source (such as a Collection or Stream) into multiple parts for more efficient parallel execution, hence the name **Spl**it-**iterator**.
4. This is especially useful when processing large collections using the Stream API.
5. The key methods of Spliterator interface are,

tryAdvance() and trySplit()

1. tryAdvance() method will perform the given action on the next element of the spliterator, if exists and returns true. If next elements does not exist then returns false.
2. trySplit() method will split the Spliterator object into two parts.
3. When you are processing large collections, to divide the collection into parts and to perform parallel processing, we use trySplit() method.
4. tryAdvance() method is like a combination of hasNext() method and next() method of Iterator.

ex:

List<String> lst = Arrays.*asList*("John", "Jack", "Tom", "Jeffry", "Jill", "Miller", "Allen");

Spliterator<String> split1 = lst.spliterator();

**while**( split1.tryAdvance(System.***out***::println));

ex:

List<String> lst = Arrays.*asList*("John", "Jack", "Tom", "Jeffry", "Jill", "Miller", "Allen");

Spliterator<String> split1 = lst.spliterator();

Spliterator<String> split2 = split1.trySplit();

//Here, split2 processes first half of the collection and

// split1 processes second half of the collection.

System.***out***.println("printing first half");

split2.forEachRemaining(System.***out***::println);

System.***out***.println("printing second half");

split1.forEachRemaining(System.***out***::println);

ex:

List<String> lst = Arrays.*asList*("John", "Jack", "Tom", "Jeffry", "Jill", "Miller", "Allen");

Spliterator<String> split1 = lst.spliterator();

Spliterator<String> split2 = split1.trySplit();

Spliterator<String> split3 = split2.trySplit();

Spliterator<String> split4 = split3.trySplit();

**if**(split4 != **null**)

split4.forEachRemaining(System.***out***::println);

System.***out***.println("printing first half");

split3.forEachRemaining(System.***out***::println);

System.***out***.println("printing second half");

split2.forEachRemaining(System.***out***::println);

System.***out***.println("printing third half");

split1.forEachRemaining(System.***out***::println);

1. Date/Time API
2. LocalDate class
3. LocalTime class
4. LocalDateTime class
5. ChronoUnit(enum)
6. Period class

LocalDate class and LocalTime class and LocalDateTime class have private constructor in the class. So, we can not create object for these classes with new keyword.

. These classes have static factory methods like now() and of() for constructing the objects.

For example:

LocalDate date1 = LocalDate.now();

. LocalDate object is created with current system date.

LocalDate date2 = LocalDate.of(2024, 11, 30);

. LocalDate object is created with given date

. LocalDate object stores the date in yyyy-MM-dd format.

. The objects of LocalDate/LocalTime/LocalDateTime are immutable objects. It means, if we make any changes the result will stored in a new object.

For example:

LocalDate date2 = LocalDate.*of*(2024, 11, 30);

System.***out***.println(date2);

LocalDate date3 = date2.plusWeeks(2);

System.***out***.println(date3);

output:

2024-11-30

2024-12-14

. On a LocalDate object, we can call factory methods like,

plusDays(), plusMonths(), plusWeeks(), plusYears(),

minusDays(), minusMonths(), minusWeeks() and minusYears().

finding the difference between two dates:

LocalDate date1 = LocalDate.*of*(2023, 10, 19);

LocalDate date2 = LocalDate.*now*();

System.***out***.println("Difference in days : " + ChronoUnit.***DAYS***.between(date1, date2)); //366

System.***out***.println("Difference in months : " + ChronoUnit.***MONTHS***.between(date1, date2)); //12

System.***out***.println("Difference in years : " + ChronoUnit.***YEARS***.between(date1, date2)); // 1

finding the difference between two times:

LocalTime time1 = LocalTime.*of*(9, 35, 55);

LocalTime time2 = LocalTime.*now*();

System.***out***.println("Difference in hours : "+ ChronoUnit.***HOURS***.between(time1, time2));

System.***out***.println("Difference in minutes : "+ ChronoUnit.***MINUTES***.between(time1, time2));

System.***out***.println("Difference in seconds : "+ ChronoUnit.***SECONDS***.between(time1, time2));

. Period class compares the two dates on days, months and years wise. It means, it will not return the differnce total days, or total months.

ex:

LocalDate date1 = LocalDate.*of*(2023, 10, 19);

LocalDate date2 = LocalDate.*now*();

Period p = Period.*between*(date1, date2);

System.***out***.println(p.getDays()); // 0

System.***out***.println(p.getMonths()); // 0

System.***out***.println(p.getYears()); // 1

converting a string to LocalDate object:

* if a string has value in yyyy-MM-dd format then it can be converted directly to a LocalDate object by calling parse() method.

ex:

String str = "2024-10-12";

LocalDate date = LocalDate.*parse*(str);

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

ex2:

String str = "12, December 2024";

DateTimeFormatter formatter =

DateTimeFormatter.*ofPattern*("dd, MMMM yyyy");

LocalDate date = LocalDate.*parse*(str, formatter);

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