Java FullStack Development

Trainer: Shekher Session Time: 11AM – 12:30PM

(Working Professional) Duration: 5 to 6 Months

Experience: 16+ years

Fee: 30k ( Installment1: 15k, Installment2: 15k)

backup videos + class material + mentor support + placement assistance

FullStack Development = Front-end Development + Back-end Development

+ Database

Front-end development: creation of User Interfaces(UI)

A user will sit in front of a web browser and

can see the web pages.

A user can interact with the application, by

providing input, by clicking on the links or

buttons.

To develop the web pages, the front-end

technologies like HTML, CSS, Javascript,

Bootstrap,Angular and React Frameworks are

used.

Back-end development: Back-end application will implement the

business rules of the organization.

It also interacts with the Database and

performs the CRUD operations.

CRUD(Create, Read, Update and Delete)

The Back-end development technologies are

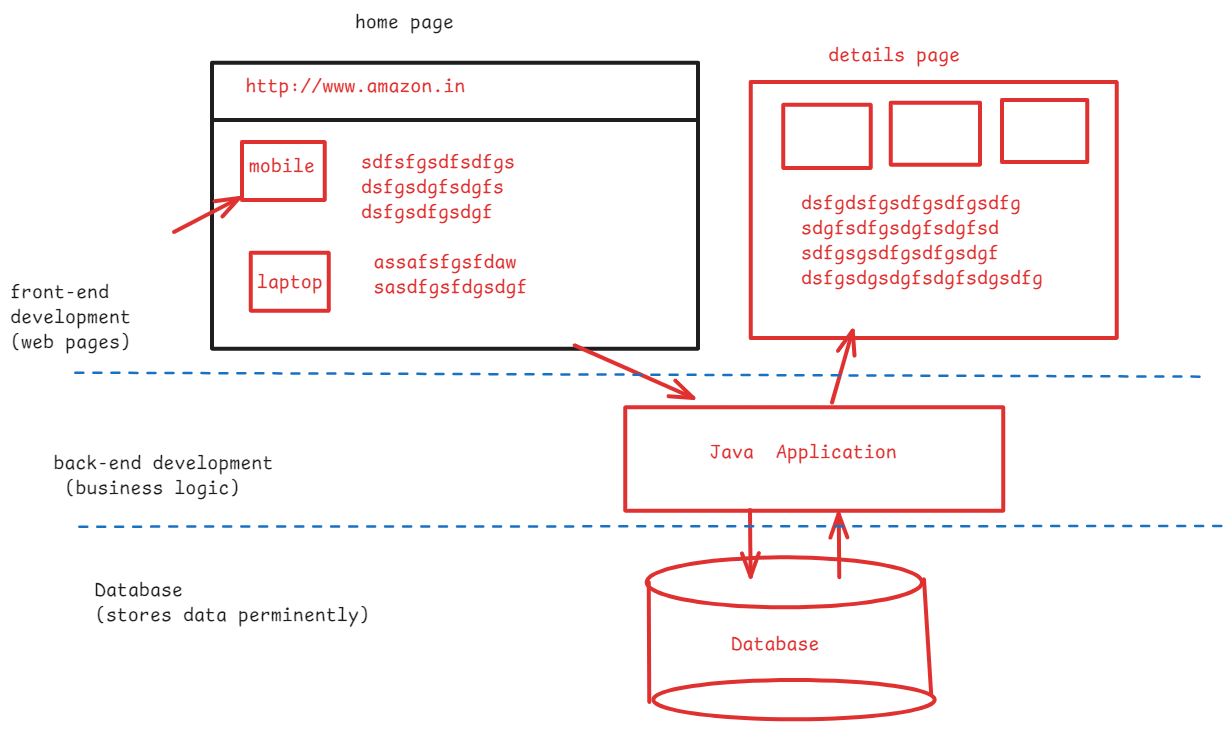
Java, .Net, Python, Ruby,etc..

Database: It will store the business data of the project and

allows the applications to persist or retrieve the

data.

For example: Oracle, MySQL, PostgreSQL, MongoDB, ...



Back-end development:

what are you going to learn?

1. Core Java
2. Advanced Java
3. Frameworks
4. Tools

* In Core Java, the important topic to focus more are,

1. logical programs using loops
2. Arrays
3. String handling
4. OOPS(Object Oriented Programming System)
5. Exception handling
6. Multithreading
7. Collections
8. Java8 features
9. Java11 features
10. Java17 features

* In Advanced Java, you will learn the below technologies.

1. JDBC(Java Database Connectivity)
2. Servlets
3. JSP(Java Server Pages)

* In Frameworks, you will learn

1. Hibernate
2. Spring
3. Spring Boot
4. Microservices

* In Tools, you will learn

1. Maven
2. Junit
3. Log4J
4. Sonar Qube
5. Jenkins, etc..

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

CORE JAVA

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

Programming language?

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

language ? – it is a medium of communication.

* we have natural languages and programming languages.
* For example,

English, Hindi, Telugu, .. are natural languages

Java, C, C++,Python, .. are programming languages

* Programming languages are sub-divided into low-level and high-level languages.
* low-level means, you have to write the programs in 0’s and 1’s. Because computers can understand 0’s and 1’s only.
* But for the developers writing the programs is very difficult.
* That’s why we have high level languages like C, C++, Java, Python, etc..
* In high-level, we write programs like english statements.
* if we write a program in Java then it will be converted into binary language(0’s and 1’s) by the compilers and interpreters(Translators).
* We can’t write the programs in natural languages like hindi/telugu, because we don’t have translators.

How to learn a programming language?

paragraphs programs

sentences statements

(Grammer) (syntax)

words Tokens

* Tokens(words) in any language are 5 types.

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

* statements in any language are 5 types.

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

JAVA INTRODUCTION:

* Java is most popular high-level programming language.
* In 1990, A team of 5 people headed by James Gosling from SUN Microsystems, started developing a programming language with the name was OAK.
* In 1995, OAK was renamed to JAVA and released to the market.
* In 2010, Oracle corporation has acquired/puchased SUN Microsystems. So from 2010, Java software updates and maintainence is done by Oracle.
* Java was released with a philosophy/slogan called “Write Once, Run Anywhere(WORA)”. This made Java language as more popular.
* Wrice Once, Run Anywhere means, you can write the program in one operating system, but you can execute this on Any other operating system.
* Java can be used to develop desktop applications, Web applications, Enterprise applications and Mobile applications(Android).
* Java is mostly used for developing Web applications and Enterprise applications.
* Desktop application/stand-alone means, it is an application, which must be downloaded and installed, then only a user can use the application.
* A desktop application can be used by only one user at a time.
* A web application is a multiuser application. It means, at a time mulitple users can use the application over the internet.
* Enterprise applications are large organization applications like, banking systems.

JAVA MODULES:

* JAVA has been divided into 3 modules, which are also called Editions.

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

* Java SE, is for developing Desktop applications. It is also called Core Java
* Java EE, is for developing web applications and enterprise applications.
* Java ME, is for developing mobile and small devices applications.
* Java SE is the base module to learn other modules.
* As a Java backend developer, you have to learn

Java SE and Java EE module.

installing Java 17 software:

* The software name of Java is JDK(Java Development Kit)
* When we download and install the JDK software, earlier ( before JDK-11), two folders are created.

1. jdk folder
2. jre folder

* jre stands for Java Runtime Environment.
* jdk folder provides tools to compile the programs, to launch the programs and other related tools.
* jre folder provides Java library and JVM(Java Virtual Machine)
* From Java SE-11(JDK-11), when we download and install the Java software, a single folder is created called jdk.
* This jdk folder only contains tools, java library and JVM.
* Java library is also called Java API(Application Programming Interface)
* Java library provides a set of pre-defined classes to develop the programs in Java.
* Every language works in the same way. It means, every language provides a library to develop the programs.
* For example, In C-language, the libarary is a set of pre-defined functions like printf(), scanf(), etc.. to develop the programs in c-language.
* JVM is a component, which will execute the Java programs on an operating system.

Follow the below steps:

1. visit <https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html>
2. Goto Java SE Development Kit 17.0.12 category and click on the link Windows x64 MSI installer.
3. The file jdk-17.0.12\_windows-x64\_bin.msi is downloaded
4. Now Double click on this file, follow the next buttons then close button.
5. The Java software is now installed at,

C:\Program Files\Java\jdk-17

Environment settings:

path setting:

1. In windows search, type environment
2. click on Edit system environment variables
3. click on Environemnt variables button on the window.
4. In the next window, goto system variables, select the path variable and then click on edit button.
5. click on New button, then empty text box is opened.
6. copy the path/location of jdk-17\bin, it means,

C:\Program Files\Java\jdk-17\bin and paste it in the empty box.

1. click on move up button and place this value onto the top.
2. click on ok.

JAVA\_HOME setting:

1. under system variables, click on New button.
2. In the next window, enter

variable name : JAVA\_HOME

variable value : C:\Program Files\Java\jdk-17

click on OK.

After the above settings, open a command prompt and execute the below command.

* java -version

First Java program:

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

1. write the program
2. compile the program
3. execute the program

step-1: open notepad

step-2: write the below program code

class Welcome

{

public static void main(String[] args)

{

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

}

}

step-3: save the program.

File 🡪 save 🡪 double click on D: drive 🡪 change save as type to all files 🡪 enter the filename Welcome.java 🡪 save

step-4: open the command prompt

step-5: compile the program

D:\> javac Welcome.java

step-6: execute the program

D:\> java Welcome

output:

Welcome to AshokIT

Java Features:

1. Platform independent:

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

. Platform independent means, we can compile the program in

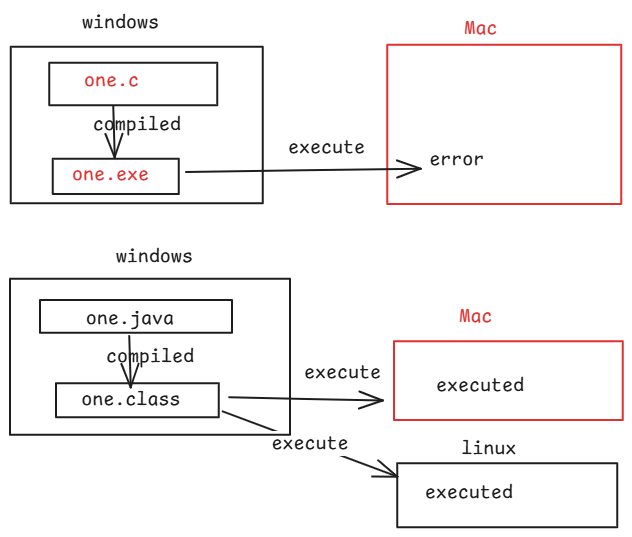
one operating system and we can execute that in any other

operating system.

. Suppose, if you take a C-language program, if you compile on windows, then the machine code is generated to execute on windows. So we can’t execute that machine code on any other operating system. This is called platform dependent.

. Suppose, if you take a Java program, if you compile on windows, then the byte code is generated and this byte code can be executed on any other operating system.

. At the time of execution, byte code is converted to machine code.



. Because of Java’s byte code, Java is called platform independent language.

1. Robust:

------

. Java is Robust language, because of the below 2 reasons.

1. automatic memory management
2. exception handling

. Inside JVM, there is a Garbage Collector, and it will

search for the un-used objects and removes them from the

JVM then collects the memory and given to the JVM. This is

called automatic memory management.

. while executing the program, if any error occurs the it is

called exception. In Java, there is good mechanism to

handle the runtime errors, which is called exception

handling.

3. multithreading:

. multithreading means, executing more than one task at a

time.

. With multithreading, more tasks can be done at a time, so

the performance of an application will be improved.

. For example, In IRCTC application, at a time one can book

the tickets and other can cancel the tickets. This is

multithreading.

4. secured:

. When we compile the java program, byte code is created.

. When we run the java program, this byte code will be

copied/loaded into JVM.

. In JVM, byte code verifier exist and it will check whether

the byte code is original or tampered.

. If it is tampared by some hacker, then the byte code

verifier doesn’t allow to execute the program.

. So, Java is called a secured programming language.

5.Architecture Neutral:

. Suppose, a Java program is written on i3 processor

architecture of windows. Now we can run this program on any

other architectures like i5 or i7 or i8 or silicon etc..

This is called Architecture Neutral.

1. Object oriented:

. Java is an Object Oriented Programming language, because it provides a support to implement the below principles in the software applications development.

1. abstraction
2. encapsulation
3. inheritance
4. polymorphism

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

main method in Java:

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

public static void main(String[] args)

1 2 3 4 5 6

1. public is an access modifier in java.
2. static is a non-access modifier in java.
3. void is a return type of the method
4. main is the name of the method
5. String[](string array) is a type of array
6. args is the name of the array

. Every java program’s execution starts from the main method.

. JVM will execute the Java program. So, JVM calls the main method.

output statement in java:

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

System.out.println();

. To display the output, we have print() and println() method.

. print() method prints the output and also keeps the cursor on the same line.

. println() method prints the output and moves the cursor to the next line.

program:

-------

1. open notepad
2. write the program

class Test

{

public static void main(String[] args)

{

System.out.print("Hello");

System.out.println("Welcome To Ashokit");

System.out.print("Ameerpet");

System.out.println("Hyderabad");

}

}

1. save the program with name Test.java
2. open a command prompt
3. compile the program.

* javac Test.java

1. run the program.

* java Test

output:

HelloWelcome To Ashokit

AmeerpetHyderabad

. If we change the code, to println() methods, and if we compile and execute the program, the output will be,

Hello

Welcome To Ashokit

Ameerpet

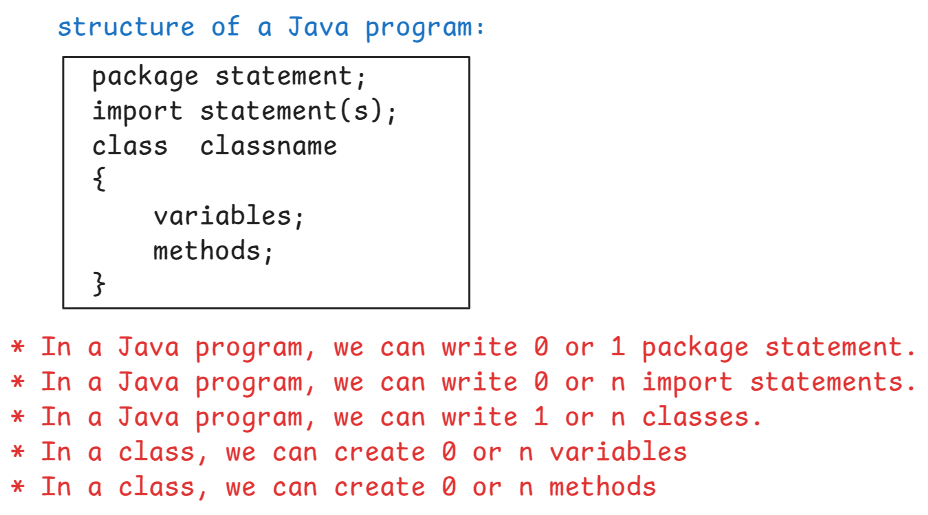
Hyderabad.

Q) I want to execute/run the program 5 times, how many times I have to compile the program?

A) one time

Q) If I do changes to the program, do I need to recompile the program?

A) Yes. If you execute directly without recompiling the program then you will get the previous output.



. To write Java program, at least one class is required.

Q) what is difference between .java file and .class file?

A) .java file contains the source code, but .class file contains the byte code.

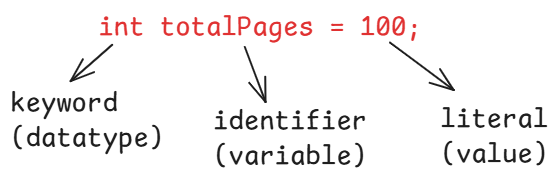
keywords :

* A keyword is also called a reserved word.
* A keyword has a pre-defined meaning in the programming language.
* every languague provides keywords.
* Java has provides more than 55 keywords.

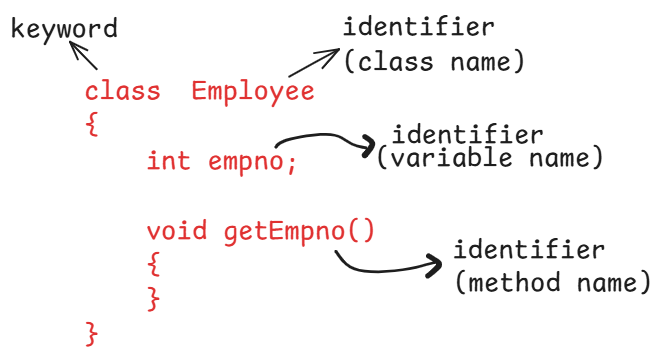
examples: class, if, for, else, break, continue, interface, import, package, int, float, boolean, extends, implements, etc.

identifiers:

* The name given to a variable or a method or a class is called an identifier.
* For example,



* For example,



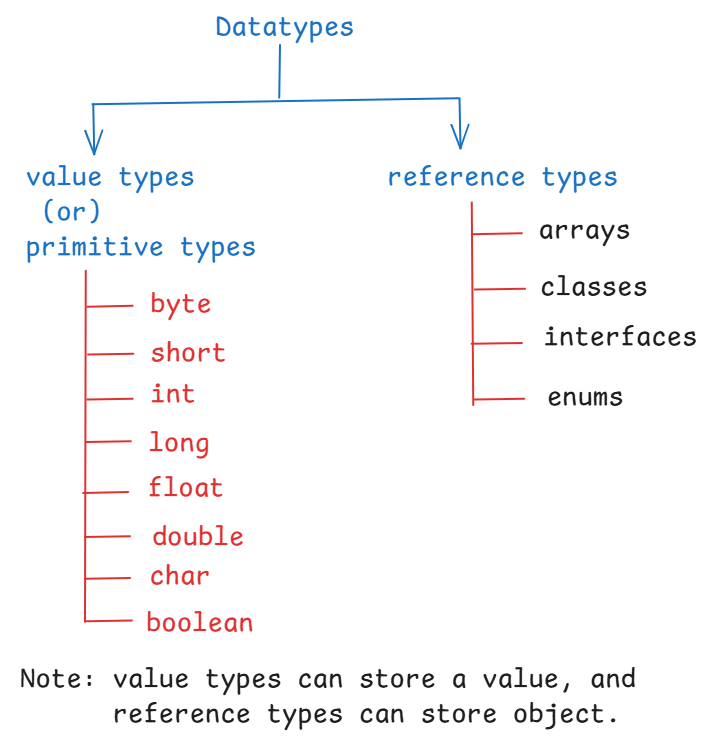
* The rules to define an identifier are,

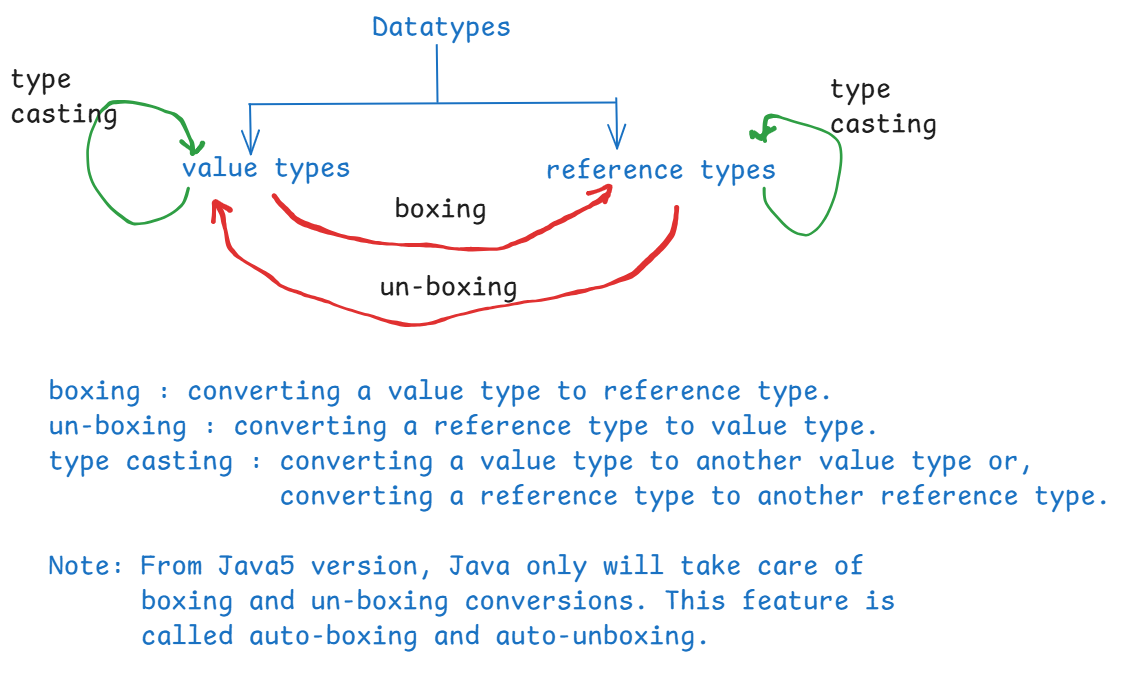
1. don’t use a keyword as an identifier.
2. don’t give a whitespace in the identifier.
3. identifier can have letters, digits, underscore(\_) and dollar($) only.
4. identifier can start with a letter or underscore(\_) or dollar($). But not with a digit.
5. identifiers are case-sensitive.

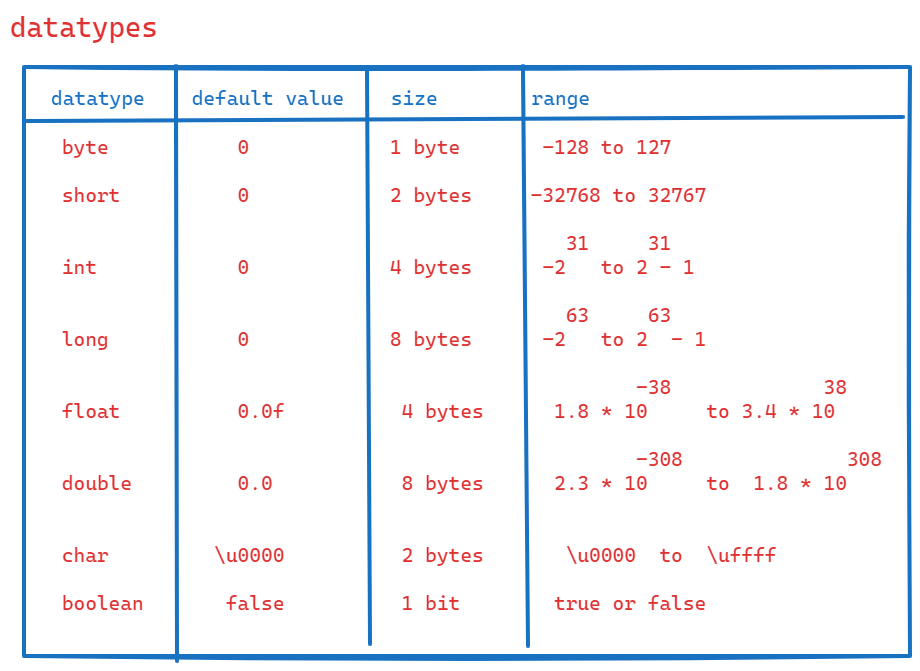
examples:

1. int if = 20; //error
2. int total\_Pages = 100; //valid
3. int my$Var = 50; //valid
4. String if\_success = “good”; // valid
5. String if\_error = “you are b@d guy”; // valid
6. int 9thPlayer\_runs = 5; // error
7. int player 2 runs = 35; // error
8. int player\_2\_runs = 35; // valid
9. int \_\_temp\_\_ = 10; // valid
10. int tot@lPages = 120; // error

Datatypes in Java:







Eclipse IDE:

. IDE – Integrated Development Environment

. We can also develop the Java programs in a notepad. But it takes more time to complete the code.

. If we use IDE software’s like Eclipse or IntelliJ IDEA, or Visual Studio Code then it takes less time to complete the code.

. To develop realtime applications, we also need to integraate our project with many tools. But with notepad, we can’t integrate. So, for developing realtime projects, we must use IDE softwares.  
 . IDE softwares are desktop applications. So, we need to download and then install in our computer.

steps for Eclipse:

1. open your browser like google chrome, and type

<http://www.eclipse.org/downloads>

1. click on download buttons
2. eclipse-inst-jre-win64.exe file is downloaded.
3. double click on the downloaded file.
4. In the window, choose Eclipse IDE for Enterprise Java and Web
5. click on install button.
6. The installation is done at

C:\Users\Windows\eclipse\jee-2024-09\eclipse.

1. A desktop shortcut icon for eclipse is automatically created.

Steps to write a Java program on Eclipse:

1. create a folder to store the projects/applications. This folder is also called as a workspace.
2. For example, create a folder with the name Workspace-11AM in

C:\Users\Windows location.

(The above two steps are to be done for once).

1. Launch eclipse. (double click on eclipse shortcut on desktop)
2. In the Eclipse launcher window, click on Browse button, go to the location C:\Users\Windows and select Workspace-11AM folder.
3. click on launch button
4. click on File menu -🡪 New -🡪 Project… -🡪 java project -🡪 enter the project name: AdditionApp -🡪 execution environment: change from JavaSE-21 to JavaSE-17 🡪 next 🡪 finish.
5. Expand the project AdditionApp 🡪 expand the folder src 🡪 delete module-info.java file
6. Right click on src folder -🡪 New -🡪 class -🡪 enter class name: Addition -🡪 select the main checkbox -🡪 finish
7. write the program code like below.

**public** **class** Addition {

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

**int** a = 10;

**int** b = 20;

**int** c = a + b;

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

}

}

1. Right click on Addition.java -🡪 RunAs -🡪 Java application.

declaraing a variable:

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

* datatype variablename = value;

(or)

datatype variablename;

ex1:

int x = 40;

int y;

y = 50;

ex2:

int x = 40, y = 50; //correct

int x = 40, int y = 50; //error

int x = 40, double z= 3.25; //error

int x = 40; int y = 50; //correct

int x = 50; boolean b = true; //correct

Scanner class:

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

* The Scanner class is provided by Java, to read the input from the user.
* In the program, we have to create the Scanner class object.

Scanner scan = new Scanner(System.in);

1 2 3 4 5

1. classname
2. objectname
3. keyword
4. classname
5. input device(keyboard)

* In Java, System.out represents the monitor(output device) and System.in represents the keyboard(input device)
* Scanner class has provided some methods to read the input values entered by the user.

1.nextInt()

2.nextByte()

3.nextShort()

4.nextLong()

5.nextFloat()

6.nextDouble()

7.nextBoolean()

8.nextLine()

* Scanner is a class provided in the java.util package.
* To use Scanner class in our java program, we have to import the class, using import statement.

ex:

import java.util.Scanner;

//example program

**import** java.util.Scanner;

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

**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();

**int** c = a + b;

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

}

}

Java comments:

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

* The comments are used to describe the classes, or variables or methods of our program.
* The comments are also called metadata in the program. Here metadata means, data about data.
* The comments will help the developers to understand the code easily.
* we can write 3 types of comments in the program.

1. single line comment
2. multiline comment
3. documentation comment

// single line comment

**/\***

**\* line1**

**\* line2**

**\* line3**

**\*/**

/\*\*

\* line1

\* line2

\* line3

\* line4

\*/

/\*

\* This program will display a welcome message

\* to the user, by taking username as input

\*/

**import** java.util.Scanner;

**public** **class** Welcome {

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

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

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

String username = scan.nextLine();

System.***out***.println("Welcome " + username);

}

}

Q) which methods of Scanner class are used to read the string input?

A) next() and nextLine() methods.

Q) what is the difference between next() and nextLine() methods?

A) next() method reads only the first word of the input. But nextLine() methods reads the entire line of the input.

/\*

\* This program converts the given celcius value

\* to forenheit value

\* The formula is f = c \* (9/5) + 32

\*/

**import** java.util.Scanner;

**public** **class** CelciusToForenheit {

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

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

System.***out***.println("Enter the celsius value");

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

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

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

System.***out***.println("Forenheit: " + f );

}

}

/\*

\* This program converts a given forenheit value

\* to the celcius value.

\* The formula is,

\* 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("forenheit : " + f );

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

}

}

/\*

\* write a program to convert the hours input

\* into seconds.

\* For example,

\* hours = 1

\* seconds = 3600

\*/

**import** java.util.Scanner;

**public** **class** HoursToSeconds {

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

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

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

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

**long** s = h \* 60 \* 60;

System.***out***.println("hours = " + h);

System.***out***.println("seconds = " + s);

}

}

/\*

\* This program will calculate the EMI for the

\* given input.

\* The input to this program should be

\* principle amount

\* rate of interest per annum

\* tenure(months)

\* rate of interest should be converted from per annum to per month

\* pm = pa / 12 / 100;

\* To calculate the emi, the formula is,

\* n n

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

\*/

**import** java.util.Scanner;

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

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

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

System.***out***.println("Hey! enter the principle amount");

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

System.***out***.println("Hey! enter the rate of interest per anuum");

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

System.***out***.println("Hey! enter the tenure(no of months)");

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

// convert rate of interest from per annum to per month

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

**double** temp = Math.*pow*(1 + pm, n);

**double** emi = p \* pm \* temp / (temp - 1);

System.***out***.println("Hey! The emi to pay is : " + emi);

scan.close();

}

}

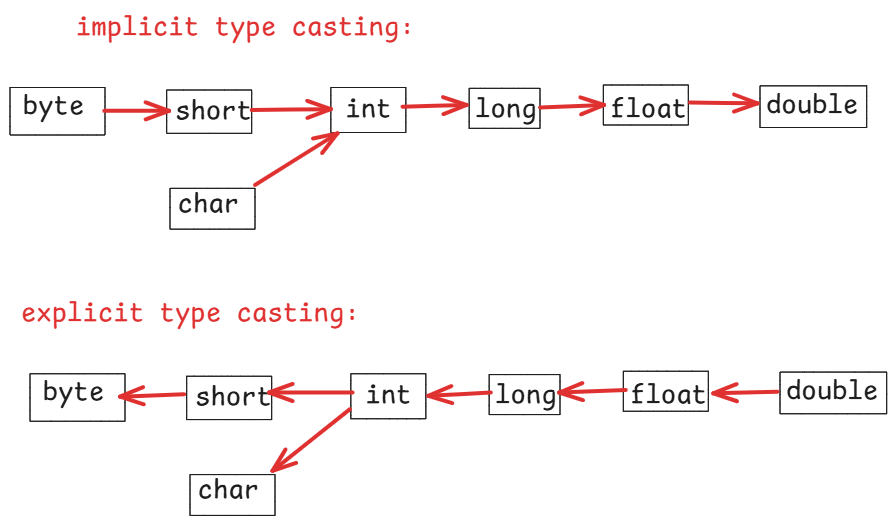
Type casting/conversion:

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

* Type casting means, converting a value/variable from one data type to another data type.

1. implicit type casting/upcasting/widening
2. explicit type casting/downcasting/narrowing

* converting a value from the lower datatype to the higher datatype is called implicit type casting.
* converting a value from the higher datatype to the lower datatype is called explicit type casting.



ex1:

int x = 40;

long y = x; //implicit type casting

ex2:

long y = 90L;

int x = y; // error

int x = (int)y; //explicit type casting.

. In explicit type casting, we have to specify the target data type in parenthesis.

ex3:

boolean flag = true;

int x = flag; //error

. a boolean variable can not be type casted to any other data type.

//example code

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

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

**int** x = 40;

**long** y = x; //implicit type casting

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

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

**long** m = 90L;

**int** n = (**int**)m; //explicit type casting

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

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

**char** ch = 'A';

**int** k = ch; //implicit type casting

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

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

**double** d = 9.67;

**int** t = (**int**)d; //explicit type casting

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

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

}

}

output:

x = 40

y = 40

m = 90

n = 90

ch = A

k = 65

d = 9.67

t = 9

Operators:

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

* An operator is a symbol which performs a well-defined task.
* Every operator works with operands.
* Operators are divided into 3 types.

1. unary operators
2. binary operators
3. ternary operator.

* unary operators works with one operand, binary operators works with two operands and a ternary operator works with 3 parts.

unary operators:

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

1. increment operator( ++ )
2. decrement operator( -- )

operand++ --- post increment

++operand --- pre increment

operand-- --- post decrement

--operand --- pre decrement

post increment = increment the value by 1, after the use.

pre increment = increment the value by 1, before the use.

post decrement = decrement the value by 1, after the use.

pre decrement = decrement the value by 1, before the use.

ex1:

int x = 10;

int y = x++;

S.o.p(“ x = “ + x);

S.o.p(“ y = “ + y);

output:

x = 11

y = 10

ex2:

int x = 10;

int y = ++x;

S.o.p(“x = “ + x);

S.o.p(“y = “ + y);

output:

x = 11

y = 11

ex3:

int a = 50;

int b = a--;

S.o.p( “a = “ + a);

S.o.p(“b = “ + b);

output:

a = 49

b = 50

ex4:

int a = 50;

int b = --a;

S.o.p( “a = “ + a);

S.o.p(“b = “ + b);

output:

a = 49

b = 49

ex5:

int x = 10;

int y = x++ + ++x;

S.o.p(“x = “ + x);

S.o.p(“y = “ + y);

output:

x = 12

y = 22

ex6:

int x = 10;

int y = ++x + ++x;

S.o.p(“x = “ + x);

S.o.p(“y = “ + y);

output:

x = 12

y = 23

ex7:

int x = 10;

int a = 20;

x = x++ - a--;

a = --x + --a;

S.o.p(“x = “ + x);

S.o.p(“a = “ + a);

output:

x = -11

a = 7

ex8:

**double** x = 45.15;

x = x++;

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

output:

x = 45.15

Note: This unary operators can be used with integer types(byte, short, int,long) or decimal types(float, double) or with character.

binary operators:

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

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

Arithmetic operators:

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

+ add

* subtract

\* multiply

/ division

% modulus

* division operator(/) returns quotient and modulus operator(%) returns remainder value.

ex1:

int a = 33;

int b = 11;

S.o.p( a / b );

S.o.p( a % b );

output:

3

0

ex2:

int x = 17;

int y = 4;

S.o.p( x / y );

S.o.p( x % y );

output:

4

1

ex3:

double x = 15;

int y = 4;

S.o.p( x / y );

S.o.p( x % y );

output:

3.75

3.0

ex4:

**double** x = 22;

**int** y = 3;

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

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

output:

7.3333333333

1.0

ex5:

**int** x = 46;

**double** y = 4;

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

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

output:

11.5

2.0

Relational operators:

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

* These operators are used to construct the conditions/prepositions.
* The relational operators returns a boolean value either true or false.

< less than

> greater than

<= less than or equals

>= greater than or equals

!= not equals

== equals

ex1:

i**nt** x = 35;

**int** y = 35;

System.***out***.println(x <= y);

System.***out***.println(x >= y);

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

System.***out***.println(x != y);

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

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

output:

true

true

true

false

false

false

ex2:

**double** x = 39;

**int** y = 35;

System.***out***.println(x <= y);

System.***out***.println(x >= y);

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

System.***out***.println(x != y);

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

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

output:

false

true

false

true

false

true

ex3:

**char** x = 'a';

**int** y = 35;

System.***out***.println(x <= y);

System.***out***.println(x >= y);

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

System.***out***.println(x != y);

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

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

output:

false

true

false

true

false

true

ex4:

**boolean** x = **false**;

**boolean** y = **true**;

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

System.***out***.println(x != y);

output:

false

true

Note: we can compare boolean variables with equals and not equals operators only.

Logical operators:

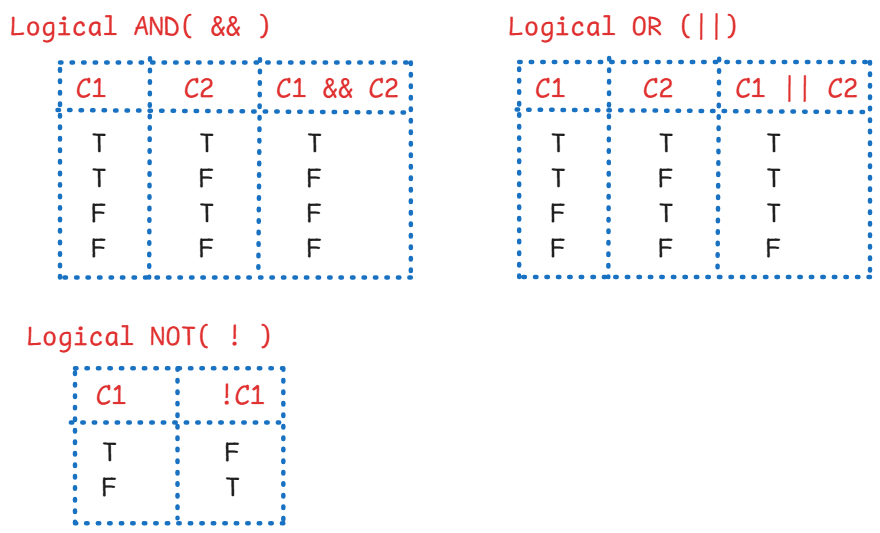
-----------------

* With logical operators, we can combine two conditions into a single condition.

&& Logical AND operator

|| Logical OR operator

! Logical NOT operator



ex1:

**boolean** x = **false**;

**boolean** y = **true**;

System.***out***.println(x == y || x != y);

output:

true

ex2:

**int** x = 10;

System.***out***.println( x++ < 10 || ++x > 12);

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

output:

false

12

ex3:

**int** x = 10;

**boolean** flag = x++ < 10 && ++x >= 12;

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

output:

11

ex4:

**int** x = 10;

**boolean** flag = ++x > 10 || ++x <= 15;

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

output:

11

Note:

In Logical AND, if the first condition is false, then the control doesn’t evaluate the second condition. Because, any how, the result is false.

In Logical OR, if the first condition is true, then the control doesn’t evaluate the second condition. Because, any how the result is true.

ex5:

**int** x = 10;

System.***out***.println(x < 11 || x > 11 || x == 11);

output:

true

Assignment operators:

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

= assign

+= add and assign

-= subtract and assign

\*= multiply and assign

/= divide and and assign

%= modulus and assign

ex1:

int x = 20;

x %= 4; (x = x % 4)

S.o.p(x);

output:

0

ex2:

int a = 39;

a += a; ( a = a + a)

S.o.p(a);

output:

78

ex3:

int count = 10;

count += count++ + ++count; (count = count + count++ + ++count)

S.o.p(count);

output:

32

Bitwise operators:

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

* Bitwise operators works on the bits of the operands.
* Bitwise operators are used in the device driver applications, embedded system applications, cryptography applications, etc..
* In the web applications and enterprise applications development, bitwise operators are not used much.

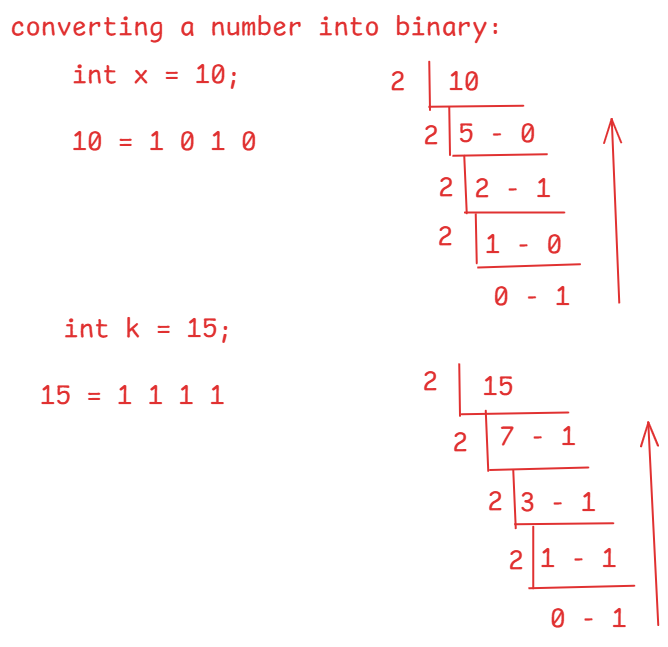
& bitwise AND

| bitwise OR

^ bitwise XOR

<< bitwise left shift

>> bitwise right shift



bitwise AND( & ): It returns a bit 1, if the corresponding bits of the operands is 1. Otherwise, returns 0.

ex:

int x = 10;

int y = 15;

x & y = ?

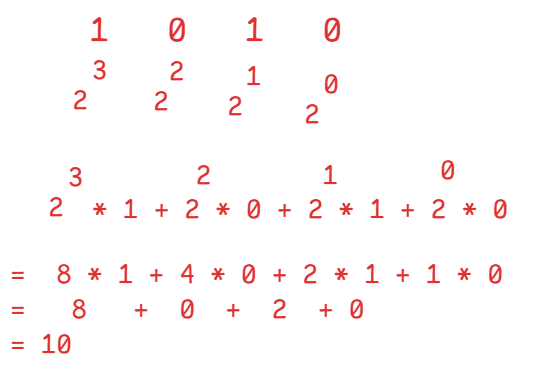
x = 1 0 1 0

y = 1 1 1 1

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

x & y= 1 0 1 0

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



x & y = 10

ex2:

int x = 12;

int y = 17;

x & y = ?

x = 0 1 1 0 0

y = 1 0 0 0 1

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

x & y = 0 0 0 0 0

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

x & y = 0

bitwise OR ( | ): It returns a bit 1 if any of the corresponding bits is 1. Otherwise, returns 0.

ex:

int x = 10;

int y = 20;

x | y = ?

x = 0 1 0 1 0

y = 1 0 1 0 0

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

x | y = 1 1 1 1 0

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

x | y = 30

ex:

int x = 8;

int y = 13;

x | y = ?

x = 1 0 0 0

y = 1 1 0 1

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

x | y = 1 1 0 1

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

x | y = 13

bitwise XOR( ^ ): It returns a bit 1, if corresponding bits are opposite. Otherwise, returns 0.

ex:

int x = 8;

int y = 13;

x ^ y = ?

x = 1 0 0 0

y = 1 1 0 1

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

x ^ y = 0 1 0 1

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

x ^ y = 5

ex:

int a = 12;

int b = 18;

a ^ b = ?

a = 0 1 1 0 0

b = 1 0 0 1 0

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

a ^ b = 1 1 1 1 0

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

a ^ b = 30

left shift( << ): It shifts the bits of the operand to its left by the specified positions. The empty bits created at the right of the operand are filled with 0.

ex:

int x = 10;

x << 2 = ?

x = 1 0 1 0

x << 2 = 1 0 1 0 0 0

x << 2 = 40;

ex:

int a = 19;

a << 2 = ?

a = 1 0 0 1 1

a << 2 = 1 0 0 1 1 0 0

a << 2 = 76

right shift( >> ): It shifts the bits of the operand to its right by the specified positions. The empty bits created at the left of the operand are filled with 0.

ex:

int x = 10;

x >> 2 = ?

x = 1 0 1 0

x >> 2 = 0 0 1 0

x >> 2 = 2

ex:

int x = 20;

x >> 2 = ?

x = 1 0 1 0 0

x >> 2 = 0 0 1 0 1

x >> 2 = 5

Ternary operator( ?: ) :

* Ternary operator has 3 parts.

1. condition
2. value if true
3. value if false

syntax:

(condition) ? value if true : value if false;

ex:

int a = 50;

int b = 30;

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

S.o.p(c);

output: 50

ex:

int x = 16;

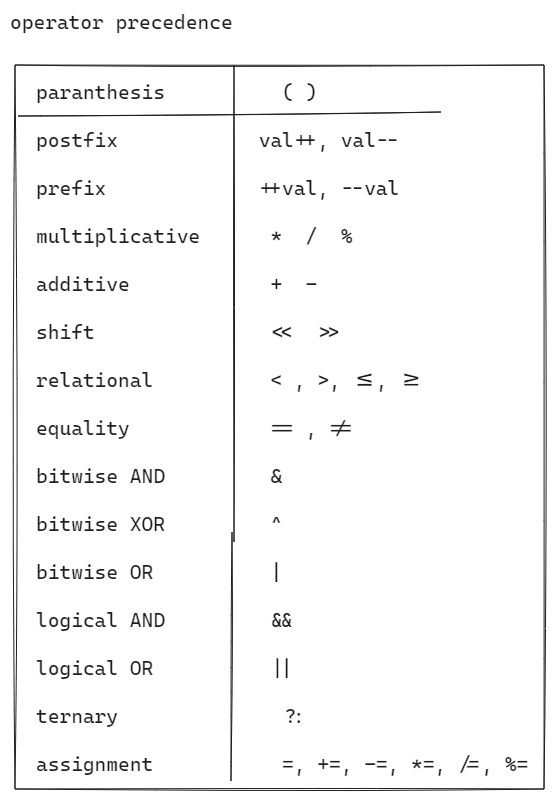
String result = (x % 2 == 0) ? “Even” : “Odd”;

S.o.p(result);

output:

Even

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



ex1:

int x = 3 \* ( 5 – 2) + 5 / 4 -1;

System.out.println(“x = “ + x);

output:

x = 9

ex2:

int x = **6 – 7 \* 8 + 9 / 5 – 8 % 3 – 7 / 2 - 3;**

**System.out.println(“ x = “ + x);**

output:

x = -57

ex3:

int x = **3 \* 5 – 7 + 7 \* 7 – 7 / 7 + 7 % 7;**

**S.o.p(“ x = “ + x);**

**output:**

**x = 56**

**ex4:**

**int x = 10 ^ 20 & 12 | 5;**

**S.o.p(“x = “ + x);**

**output:**

**x = 15**

**ex5:**

**boolean b = false;**

**boolean flag = !b == true || (b = false);**

**S.o.p(“flag = “ + flag);**

**output:**

ex6:

**int i = 2 \* 3 + 5 \* 6 / 7 - 3 / 2 + 7 \* 6 / 3;**

**int j = i - 3 \* 4 - 5 \* 6 + 7 - 3 \* 4 + 7;**

**i = i + j;**

**j = j - i \* 3 / 2;**

**i = i \* 7 / -2 + 5;**

**j = j - 3 \* 4 % 7;**

**S.o.p(i);**

**S.o.p (j);**

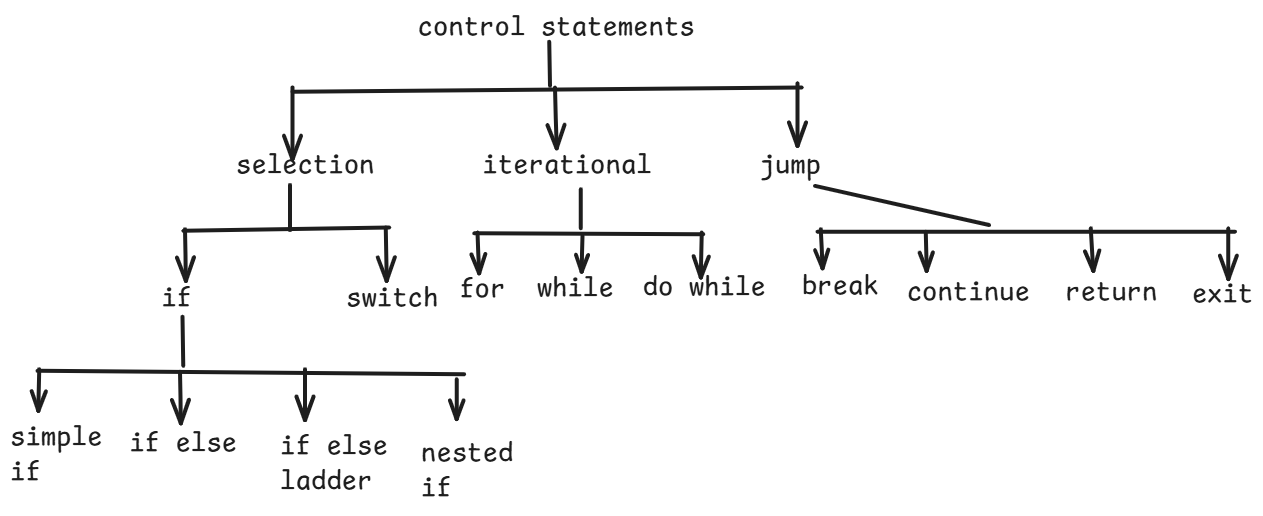
**output:**

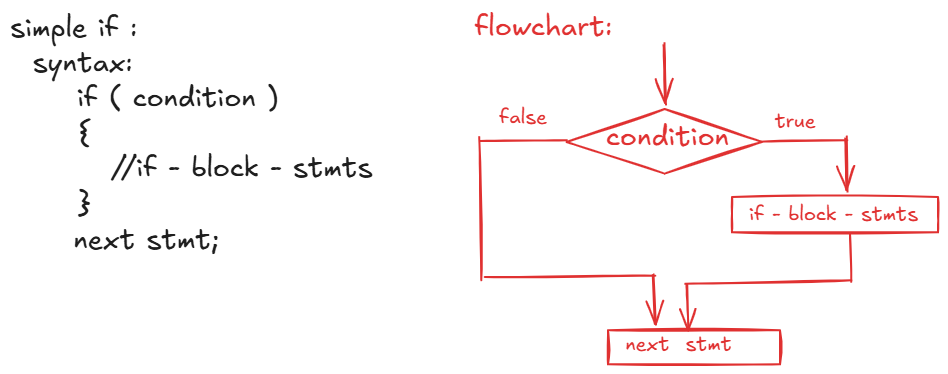
**-16**

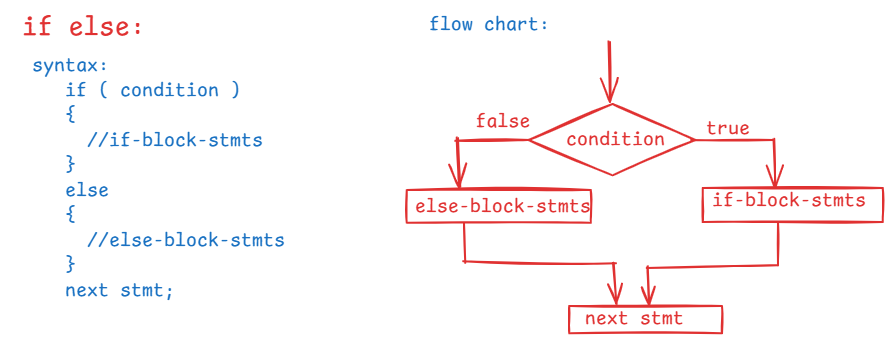
**-31**

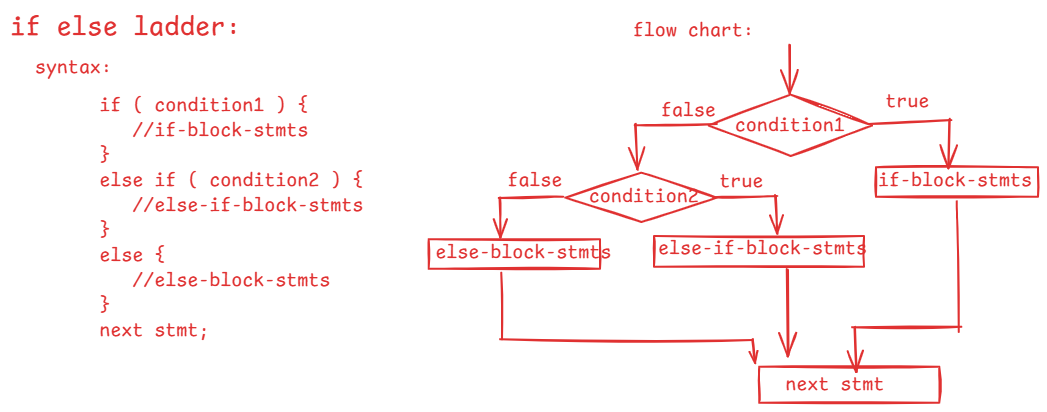
**Control statements**

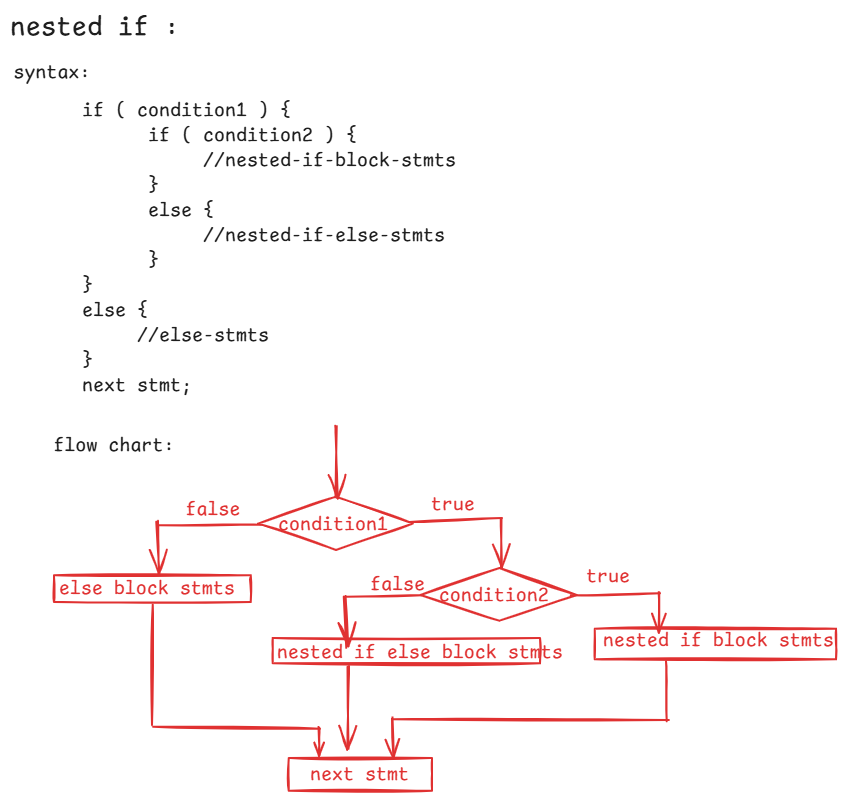
1. **selection control statements**
2. **iterational control statements**
3. **jump control statements**











/\*

\* write a program to read two numbers as input and

\* if they are same display the sum, otherwise display the

\* double of their product.

\*/

**import** java.util.Scanner;

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

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

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

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

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

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

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

**if** ( a == b )

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

**else**

{

**int** k = 2 \* (a \* b);

System.***out***.println("double of their product : " + k);

}

}

}

/\*

\* write a program to read a number as input and

\* do the following.

\* 1. if the number is divisible by 3 then display "Zip"

\* 2. if the number is divisible by 5 then display "Zap"

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

\* 4. Otherwise, display "Rar"

\*/

**import** java.util.Scanner;

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

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

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

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

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

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

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

}

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

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

}

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

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

}

**else** {

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

}

}

}

/\*

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

\* the output based on the below conditions.

\* 1. if 3 sides are equal then display the output "Equilateral triangle"

\* 2. if any 2 sides are equal then display the output "Isosceles triangle"

\* 3. if sides are not equal then display the output "Scalane triangle"

\*/

**import** java.util.Scanner;

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

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

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

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

**int** side1 = scan.nextInt();

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

**int** side2 = scan.nextInt();

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

**int** side3 = scan.nextInt();

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

System.***out***.println("Equilateral triangle");

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

System.***out***.println("Isosceles triangle");

**else**

System.***out***.println("Scalane triangle");

}

}

/\*

\* write a program to find the biggest of 3 numbers

\* with nested if statement.

\*/

**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 == b && b == c) {

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

}

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

**if** ( a > c )

System.***out***.println("first number is big");

**else**

System.***out***.println("third number is big");

}

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

System.***out***.println("second number is big");

**else**

System.***out***.println("third number is big");

}

}

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

/\*

\* write a program to calculate the delivery fee like below.

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

\* 2. For next 3 kms, delivery fee is 8 rupees per km.

\* 3. For remaining kms, delivery fee is 12 rupees per km.

\*/

**import** java.util.Scanner;

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

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

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

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

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

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

**double** deliveryFee = 0;

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

}

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

**double** deliveryFee = (distance - 3) \* 8;

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

}

**else** {

**double** deliveryFee = 3 \* 8 + (distance - 6) \* 12;

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

}

}

}

/\*

\* write a program to check whether a given

\* year is a leap year or not.

\* Leap year conditions:

\* 1. if year is divisible by 4 and not divisible by 100

\* then it is "Leap Year".

\* 2. if year is divisible by 4 and divisible by 100 then

\* it must be divisible by 400 then it is a "Leap Year".

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

\*/

import java.util.Scanner;

public class Solution {

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");

}

}

}

/\*

\* 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** Solution {

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

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

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

String item = scan.nextLine();

System.***out***.println("enter the price");

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

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

{

scan.nextLine();

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

String material = scan.nextLine();

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

{

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

**double** finalPrice = price + gst;

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

}

**else**

{

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

**double** finalPrice = price + gst;

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

}

}

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

{

scan.nextLine();

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

String brand = scan.nextLine();

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

{

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

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

**double** finalPrice = price + gst - discount;

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

}

**else**

{

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

**double** finalPrice = price + gst;

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

}

}

**else**

{

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

}

}

}

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

switch statement:

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

* When you want to define multiple conditions on the same variable and the condition is equality then instead of using if statements, we can use switch statement.

syntax:

switch(variable or expression)

{

case value1: statements;

break;

case value2: statements;

break;

. . .

. . .

case valueN: statements;

break;

default: statements;

}

next statement;

ex1:

int x = 2;

switch( x )

{

case 1: S.o.p(“ONE”);

break;

case 2: S.o.p(“TWO”);

break;

case 3: S.o.p(“THREE”);

break;

default: S.o.p(“INVALID”);

}

output: TWO

ex2:

int x = 1;

switch( x + 1 )

{

case 1: S.o.p(“one”);

break;

case 2: S.o.p(“two”);

case 3: S.o.p(“three”);

break;

default: S.o.p(“invalid”);

}

output:

two

three

* If a case does not have a break statement, then the control continues to execute the statements until break staement occurs.

ex3:

double x = 9.5;

switch(x)

{

case 5.5: S.o.p(“hi”);

break;

case 9.5: S.o.p(“hello”);

break;

default: S.o.p(“bye”);

}

output:

This example has compile-time error. Because, to the switch statement we can pass an integer or a character or a string variable as parameter.

ex4:

**int** x = 5;

**switch**(x)

{

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

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

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

}

output:

Ashokit

Welcome

To

ex5:

**char** ch = 'e';

**switch**(ch)

{

**case** 'a':

**case** 'e':

**case** 'i':

**case** 'o':

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

**break**;

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

}

output:

Vowel

ex6:

String str = "Java";

**switch**(str)

{

**case** "Python":

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

System.***out***.println("is a");

System.***out***.println("Scripting language");

**break**;

**case** "Java":

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

System.***out***.println("is a");

System.***out***.println("Programming language");

**break**;

**default**: System.***out***.println("Invalid language");

}

output:

Java

is a

Programming language.

Iterational statements(loops):

* A loop is nothing but repetative execution of the same group of statements based on a condition.
* The loop statements are,

for,

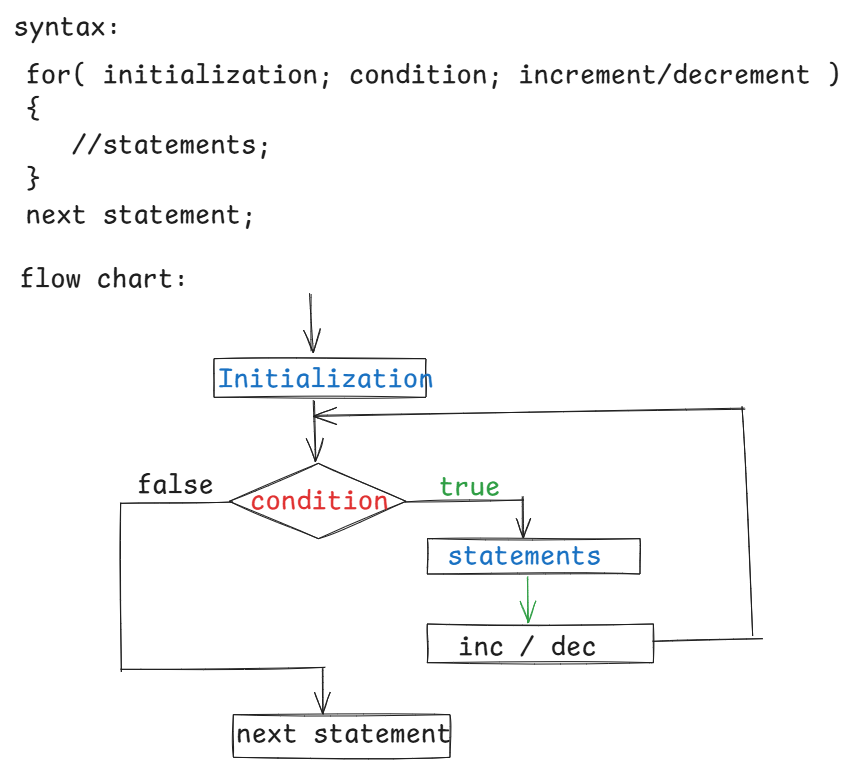
while and

do while.

for loop:

---------

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



ex1:

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

{

S.o.println(“i = “ + i);

}

output:

i = 1

i = 2

i = 3

ex2:

int i = 1;

for ( ; i <= 5; i+=2 )

{

S.o.println( “i = “ + i);

}

output:

i = 1

i = 3

i = 5

ex3:

int i = 1;

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

{

S.o.println(i);

}

output:

error

ex4:

int i = 10;

for ( i = 11; i <= 13; i++ );

S.o.println(“i = “ + i);

output:

i = 14

ex5:

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

{

S.o.println( i + j );

}

output:

6

6

6

ex6:

int i = 1;

for(;;)

{

S.o.println(“hello”);

}

output:

hello is printed infinite times.

/\*

\* write a program to find the sum of N natural

\* numbers using for loop.

\*/

**import** java.util.Scanner;

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

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

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

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

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

*findSum*(number);

}

**private** **static** **void** findSum(**int** n)

{

**int** sum = 0;

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

{

sum = sum + i;

}

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

}

}

output:

Enter a number

15

SUM = 120

/\*

\* (logic-1)

\* write a program to check whether a given

\* number is prime number or not.

\* prime number: A prime number is a number which contains only

\* 2 factors, one and itself.

\* ex:

\* n = 6

\* It is not a prime number, because it has 4 factors(1, 2, 3, 6)

\* n = 5

\* It is a prime number, because it has 2 factors( 1 and 5)

\*/

**import** java.util.Scanner;

**public** **class** PrimeNumberTest {

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

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

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

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

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

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

{

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

}

**else**

{

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

}

}

**private** **static** **boolean** checkPrime(**int** n)

{

**int** count = 0;

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

{

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

{

count++;

}

}

**if** ( count == 2 )

**return** **true**;

**else**

**return** **false**;

}

}

/\*

\* write a program to check whether a given number

\* is a perfect number or not.

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

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

\* a perfect number.

\* example:

\* number = 6

\* The factors excluding 6 are 1, 2, 3 = 1+2+3 =6

\* 6 : is a perfect number

\*

\* number = 28

\* The factors excluding 28 are 1, 2, 4, 7, 14 = 1 + 2+ 4 + 7 + 14 = 28

\* 28 : is a perfect number

\*

\* number = 24

\* The factors excluding 24 are 1, 2, 3, 4, 6, 8, 12 = 36

\* 24 : is not a perfect number

\*/

**import** java.util.Scanner;

**public** **class** PerfectNumberTest {

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

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

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

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

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

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

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

}

**else** {

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

}

}

**private** **static** **boolean** isPerfect(**int** 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**;

}

}

/\*

\* write a program to find the factorial of

\* a given number

\* example:

\* number = 5

\* output: 120

\*/

**import** java.util.Scanner;

**public** **class** PerfectNumberTest {



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



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

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

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

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

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

}

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

**int** fact = 1;

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



fact = fact \* i;

}

**return** fact;

}

}

/\*

\* write a program to print the fibonacci series

\* of n terms.

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

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

\*

\* ex: n = 5

\* output: 0 1 1 2 3

\*/

**import** java.util.Scanner;

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

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

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

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

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

*printFibonacci*(n);

}

**private** **static** **void** printFibonacci(**int** n)

{

**int** firstTerm = 0;

**int** secondTerm = 1;

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

{

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

**int** nextTerm = firstTerm + secondTerm;

firstTerm = secondTerm;

secondTerm = nextTerm;

}

}

}

/\*

\* write a program to display the multiplication table

\* upto 10 rows, for a given number.

\* For example:

\* number = 5

\* output:

\* 5 \* 1 = 5

\* 5 \* 2 =10

\* . . .

\* . . .

\* 5 \* 10=50

\*/

**import** java.util.Scanner;

**public** **class** MultiplicationTable {

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

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

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

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

*printTable*(n);

}

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

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

{

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

}

}

}

Nested for loops:

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

* a for loop inside another for loop is called nested for loop.
* For each iteration of the outer loop, the inner loop is completely executed.

ex1:

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

{

for ( int j = 1; j <= 2; j++ )

{

System.out.println( j );

}

}

output: 1

2

1

2

1

2

ex2:

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

{

for ( int j = 3; j > 0; j-- )

{

if ( i + j > 2 )

S.o.println(j);

}

}

ex3:

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

{

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

{



S.o.println( i+j );

}

}

output:

2

3

4

pattern programs:

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.

/\*

\* write a program to display the right angle

\* triangle for the given rows.

\* for example, if rows = 5,

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

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

\*/

**import** java.util.Scanner;

**public** **class** RightAngleTriangle {

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

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

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

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

*printPattern*(rows);

}

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

//outer loop : rows

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

{

//inner loop: cols

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

{

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

}

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

}

}

}

/\*

\* write a program to display the right angle

\* inverted triangle for the given rows.

\* For example,

\* if rows = 5,

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

\* \* \* \* \*

\* \* \* \*

\* \* \*

\* \*

\*/

**import** java.util.Scanner;

**public** **class** RightAngleInverted {

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

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

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

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

*printPattern*(rows);

}

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

// outer loop : rows

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

// inner loop: cols

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

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

}

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

}

}

}

/\*

\* write a program to display left angle

\* triangle for the given rows.

\* For example,

\* if rows=5,

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

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

\*/

**import** java.util.Scanner;

**public** **class** LeftAngleTriangle {

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

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

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

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

*printPattern*(rows);

}

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

//outer loop: rows

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

//inner loop1: spaces

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

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

}

//inner loop2: stars/cols

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

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

}

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

}

}

}

/\*

\* write a program to display the below pyramid pattern

\* For example,

\* if rows = 5,

\* \*

\* \* \* \*

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

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

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

\*/

**import** java.util.Scanner;

**public** **class** PyramidTriangle {

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

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

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

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

*printPattern*(rows);

}

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

//outer loop: rows/lines

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

//inner loop1: spaces

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

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

}

//inner loop2: stars

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

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

}

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

}

}

}

/\*

\* write a program to display inverted pyramid triangle

\* for the given rows.

\* if rows = 5,

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

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

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

\* \* \* \*

\* \*

\*/

**import** java.util.Scanner;

**public** **class** InvertedTriangle {

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

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

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

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

*printPattern*(rows);

}

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

// outer loop: rows/lines

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

// inner loop1: spaces

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

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

}

// inner loop2: stars

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

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

}

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

}

}

}

/\*

\* write a program to print the complete pyramid for

\* the given rows.

\* if rows = 5,

\* \*

\* \* \* \*

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

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

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

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

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

\* \* \* \*

\* \*

\*/

**import** java.util.Scanner;

**public** **class** CompletePyramid {

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

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

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

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

*printPattern*(rows);

}

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

// outer loop: rows/lines

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

// inner loop1: spaces

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

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

}

// inner loop2: stars

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

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

}

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

}

// outer loop: rows/lines

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

// inner loop1: spaces

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

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

}

// inner loop2: stars

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

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

}

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

}

}

}

/\*

\* write a program to print the Hollow square

\* for the given rows.

\* if rows = 5,

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

\* \* \*

\* \* \*

\* \* \*

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

\*/

import java.util.Scanner;

public class HolloSquareTest {

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

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

int rows = scan.nextInt();

printPattern(rows);

}

private static void printPattern(int rows) {

//outer loop : rows

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

//inner loop: stars

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

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

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

}

else {

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

}

}

System.out.println();

}

}

}

/\*

\* write a program to display the below pattern.

\* if rows = 5,

\* A

\* A B

\* A B C

\* A B C D

\* A B C D E

\*/

**import** java.util.Scanner;

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

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

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

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

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

*printPattern*(rows);

}

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

//outer loop : rows

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

{

//inner loop : characters

**for** ( **int** j = 65; j <= 65 + i - 1; j++)

{

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

}

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

}

}

}

/\*

\* write a program to display the prime numbers within

\* a given range.

\*/

**import** java.util.Scanner;

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

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

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

System.***out***.println("enter low value");

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

System.***out***.println("enter high value");

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

*printPrimeNumbersWithinRange*(x, y);

}

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

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

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

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

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

flag = **false**;

**break**;

}

}

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

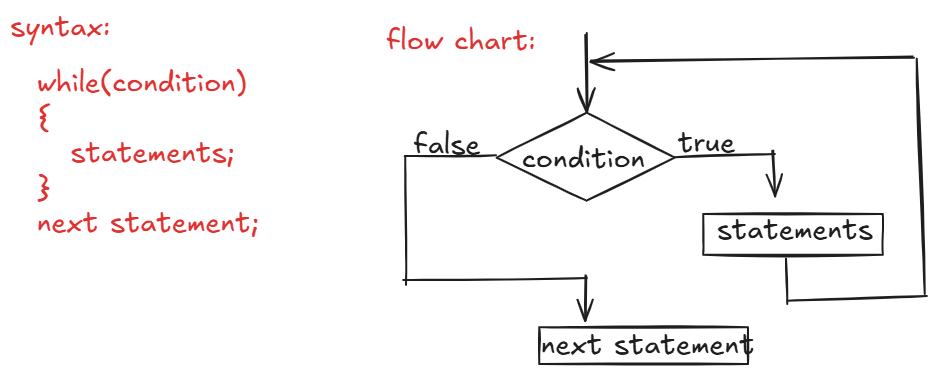
System.***out***.println(i + " : is prime");

}

}

}

while loop:



* when control enters into while statement, it checks for the condition.
* if the condition is true, then the control executes the statements in the while loop.
* again checks the condition is true or not. if yes, the control executes

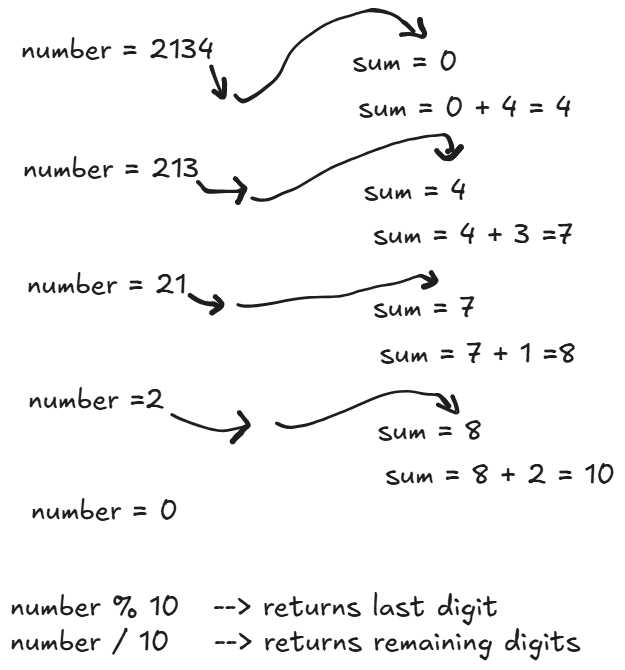
the statements in while loop again.

* the above step repeats until, the given condition is false.
* if the condition is false, then the control goes to the next statement after the loop.

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

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

2. in for loop, you can write initialization, condition and then inc/dec in a single statement. But in while loop, you have to write in separate statements.



/\*

\* write a program to find the sum of digits of a given number

\* example:

\* if number = 2314

\* output: sum = 2+3+1+4 = 10

\*/

**import** java.util.Scanner;

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

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

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

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

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



**int** sum = *findSumOfDigits*(n);

System.***out***.println("You have entered a number : " + n);

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

}

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

**int** sum = 0;

**while**( n > 0 ) {

**int** lastDigit = n % 10;

sum += lastDigit;

n = n / 10;

}

**return** sum;

}

}

/\*

\* write a program to check whether a given number

\* is Armstrong number or not.

\* Armstrong number: If sum of nth power of each digit of a number is

\* equals to the same number then it is an armstrong number.

\* example:

\* number = 371

\* 3 3 3

\* = 3 + 7 + 1 = 371

\* 371 : is armstrong number

\* For example, 1 to 9, 153, 370, 371, 407, 1634, ... are armstrong numbers

\*/

**import** java.util.Scanner;

**public** **class** ArmstrongNumberTest {

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

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

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

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

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

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

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

**else**

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

}

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

//store the given number into another variable

**int** temp = number;

//convert the given number into string

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

//find the length of the string

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

**int** sum = 0;

**while**(number > 0) {

**int** lastDigit = number % 10;

**int** p = (**int**)Math.*pow*(lastDigit, len);

sum = sum + p;

number = number / 10;

}

**if** ( temp == sum )

**return** **true**;

**else**

**return** **false**;

}

}

/\*

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

\* a strong number or not.

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

\* is equals to the same number then it is a strong number.

\* example:

\* number = 145

\* = 1! + 4! + 5!

\* =145

\* 145 : is a strong number

\*/

**import** java.util.Scanner;

**public** **class** StrongNumberTest {

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

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

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

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

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

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

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

**else**

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

}

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

**int** temp = number;

**int** sum = 0;

**while**(number > 0) {

**int** lastDigit = number % 10;

**int** fact = 1;

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

{

fact = fact \* i;

}

sum = sum + fact;

number = number / 10;

}

**if**(sum == temp)

**return** **true**;

**else**

**return** **false**;

}

}

/\*

\* write a program to check whether a given number

\* is a palindrome or not.

\* palindrome: If reverse of a number is equal to the same

\* number, then it is a palindrome.

\* example:

\* number = 121

\* reverse = 121

\* 121 : is a palindrome.

\*/

**import** java.util.Scanner;

**public** **class** PalindromeTest {

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

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

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

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

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

**if** ( f == **true**)

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

**else**

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

}

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

**int** temp = number;

**int** reverse = 0;

**while**(number > 0) {

**int** lastDigit = number % 10;

reverse = reverse \* 10 + lastDigit;

number = number / 10;

}

**if**(reverse == temp)

**return** **true**;

**else**

**return** **false**;

}

}

/\*

\* write a program to display a given number in words.

\* example:

\* number = 1290

\* output: OneTwoNineZero

\*/

**import** java.util.Scanner;

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

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

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

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

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

*printInWords*(number);

}

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

String words="";

**while**(number > 0)

{

**int** lastDigit = number % 10;

**switch**(lastDigit)

{

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

**break**;

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

**break**;

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

**break**;

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

**break**;

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

**break**;

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

**break**;

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

**break**;

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

**break**;

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

**break**;

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

**break**;

}

number = number / 10;

}

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

}

}

/\*

\* write a program to display a pascal triangle for

\* the given rows.

\* example:

\* if rows = 5,

\* 1

\* 1 1

\* 1 2 1

\* 1 3 3 1

\* 1 4 6 4 1

\*/

**import** java.util.Scanner;

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

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

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

System.***out***.println("enter the rows");

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

*printPascalTriangle*(rows);

}

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

//outer loop : rows

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

//inner loop : spaces

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

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

}

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

System.***out***.print(*ncr*(i, k) +" ");

}

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

}

}

**private** **static** **int** ncr(**int** i, **int** k) {

**int** result = *factorial*(i) / (*factorial*(i-k) \* *factorial*(k));

**return** result;

}

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

**int** fact = 1;

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

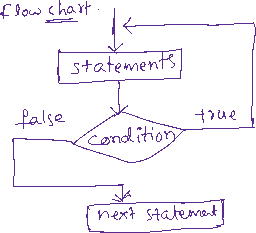
fact = fact \* i;

}

**return** fact;

}

}



/\*

\* This program asks a user to enter a number between 1 to 10

\* until a correct number is entered, using a do while loop.

\* Then a random number is generated and checks both the numbers

\* are matched or not.

\*/

**import** java.util.Random;

**import** java.util.Scanner;

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

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

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

**int** x = 0;

**do**

{

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

x = scan.nextInt();

}

**while**( x < 0 || x > 10);

//create Random class object

Random r = **new** Random();

//generate a random number from 0 to 10

**int** y = r.nextInt(11);

**if**( x == y ) {

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

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

System.***out***.println("Your guess is correct");

}

**else** {

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

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

System.***out***.println("Your guess is wrong");

}

}

}

Jumping statements:

1. break:

-----

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

ex1:

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

if( (i+2) \* 2 > 10 ){

break;

}

S.o.println(i);

}

output: 1

2

3

ex2:

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

2

1

ex3:

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

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

if( i+j-1 > 3)

break outer;

S.o.println(j);

}

}

output: 1

2

3

1

2

1. continue:

---------

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



ex1:

int i=1;

while( i<=5) {

if( i \* i > 5 )

continue;

S.o.p(i);

i++;

}

output: 1

2

ex2:

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

{

inner: for(int j=1; j<=10; j++)

{

if( j > i)

continue outer;

S.o.print(j);

}

S.o.println();

}

output:

1

12

123

1234

12345

1. return:

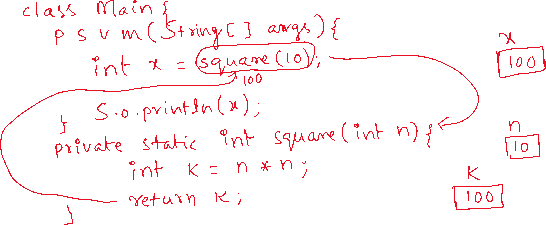
------

* return statement moves the control from a method definition to the method calling point.
* return statement can be used without a value or with a value.

return; //without a value

return value; // with value.

* If a method’s return type is void, then we can used return statement without a value.
* If a method returns a value, then the calling point of that method will be replaced with the returned value.



1. exit:

----

* exit is not a keyword, it is a method in System class.
* whenever you want to terminate the currently running JVM, you can call exit() method.
* To the exit method, you have to pass an integer value as a parameter.
* The value zero indicates normal termination and non-zero indicates abnormal termination.

System.exit(1);

Arrays

* To store a value, we need a variable.
* To store multiple values, we need mulitple variables.
* If multiple variables are created, then we have the below problems.

1. The code becomes complex.
2. The memory for the variables will be allocated at different places. So, every time the JVM has to search for that variable in the entire JVM’s memory. It will decrease the performance of an application.

* So, the solution for the above problems is given as arrays.
* In arrays, we can store mulitple values in a single variable and the memory for the values will be allocated sequentially.
* So, complexity is reduced and the performance will be increased.

Array declaration, creation and initialization:

datatype[] arrayname; //declaration

arrayname = new datatype[size]; //creation

ex:



int[] arr;



arr = new int[5];



arr[0]=8;

arr[1]=3;



arr[2]=9;



arr[3]=1;

arr[4]=5;

* we can combine declaration and creation in a single statement.

datatype[] arrayname = new datatype[size];

ex: int[] arr = new int[10];

* we can also combine declaration, creation and initialization in a single statement.

ex: int[] arr = new int[] {7, 3, 2, 9};



length attribute:

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

* To find the length/size of an array, we have to use length attribute of the array variable.

ex:

int[] arr = new int[5];

S.o.println(arr.length); // output: 5

ex:

int[] arr = new int[] { 2, 6, 3, 8 };

S.o.println(arr.length); //output : 4

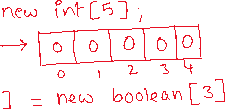
ex:

int[] arr = {4, 7, 2, 9, 0};

//direct initialization of an array. Here, array creation will be

// done automatically.

* If you create an array, based on the datatype of the array, the default values are stored in the array.



length attribute and length() method:

* use length attribute to find the size of an array, and use length() method to find the size of a string.

ex:

String[] str = { “John”, “Jeffery”, “James” };

S.o.println( str.length ); //correct

S.o.println( str.length() ); //error

S.o.println( str[1].length ); //error

S.o.println( str[1].length() ); //correct

ex:

String str = “ashokit”;

S.o.println( str.length ); //error

S.o.println( str.length() ); //correct

limitations of an array:

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

1. once an array is created with a size, then it is fixed. We can’t increase/decrease the size.
2. If an array size is overestimated than required, then the memory is wasted.
3. array can only store the elements of the same data type. It is called homogeneous data.
4. inserting and removing the elements from the middle of an array is inefficient.
5. No built-in methods for common operations like searching, sorting, etc..

/\*

\* write a program to read the size and elements of an array

\* from a user and display the sum of the elements.

\*/

**import** java.util.Scanner;

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

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

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

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

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

**if**( size < 0 ) {

System.***out***.println("You have to enter the size as positive integer only. Try again....");

System.*exit*(1);

}

//create an array

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

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

System.***out***.println("Enter the element for arr[" + i +"]");

arr[i] = scan.nextInt();

}

*findSumOfArrayElements*(arr);

}

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

**int** sum = 0;

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

sum += arr[i]; // sum = sum + arr[i]

}

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

}

}

/\*

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

\* using linear search.

\*/

**import** java.util.Scanner;

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

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

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

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

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

**if**( size < 0 ) {

System.***out***.println("You have to enter the size as positive integer only. Try again....");

System.*exit*(1);

}

//create an array

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

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

System.***out***.println("Enter the element for arr[" + i +"]");

arr[i] = scan.nextInt();

}

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

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

*linearSearch*(arr, searchingElement);

}

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

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

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

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

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

flag = **true**;

**break**;

}

}

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

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

}

}

}

/\*

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

\* using binary search.

\*

\* binary search:

\* 1. array elements must be in ascending order.

\* 2. find the low and high index of the array

\* 3. find the mid index as (low + high) /2

\* 4. if searching element is equal to arr[mid], then element is found, and

\* break the loop.

\* 5. if searching element > arr[mid], then low = mid + 1

\* 6. if searching element < arr[mid], then high = mid - 1

\* 7. repeat the steps 3 to 6, until low <= high

\* 8. if low > high, then element is not found.

\*/

**import** java.util.Arrays;

**import** java.util.Scanner;

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

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

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

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

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

**if**( size < 0 ) {

System.***out***.println("You have to enter the size as positive integer only. Try again....");

System.*exit*(1);

}

//create an array

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

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

System.***out***.println("Enter the element for arr[" + i +"]");

arr[i] = scan.nextInt();

}

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

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

*binarySearch*(arr, searchingElement);

}

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

//sort the array

Arrays.*sort*(arr);

**int** low = 0;

**int** high = arr.length - 1;

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

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

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

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

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

flag = **true**;

**break**;

}

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

high = mid - 1;

**else**

low = mid + 1;

}

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

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

}

}

}

/\*

\* write a program to find max consecutively repeated element

\* in the given array.

\* example:

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

\* output: 1

\*/

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

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

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

*findMaxConsecutiveRepeated*(arr);

}

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

//every element in array repeats for at least once.

// So, we started count with 1

**int** count = 1;

**int** maxi = 0;

**int** element = 0;

**for**(**int** i = 0; i < arr.length - 1; i++) {

**if**(arr[i] == arr[i+1]) {

count++;

}

**else** {

count = 1;

}

**if**(count > maxi) {

maxi = count;

element= arr[i];

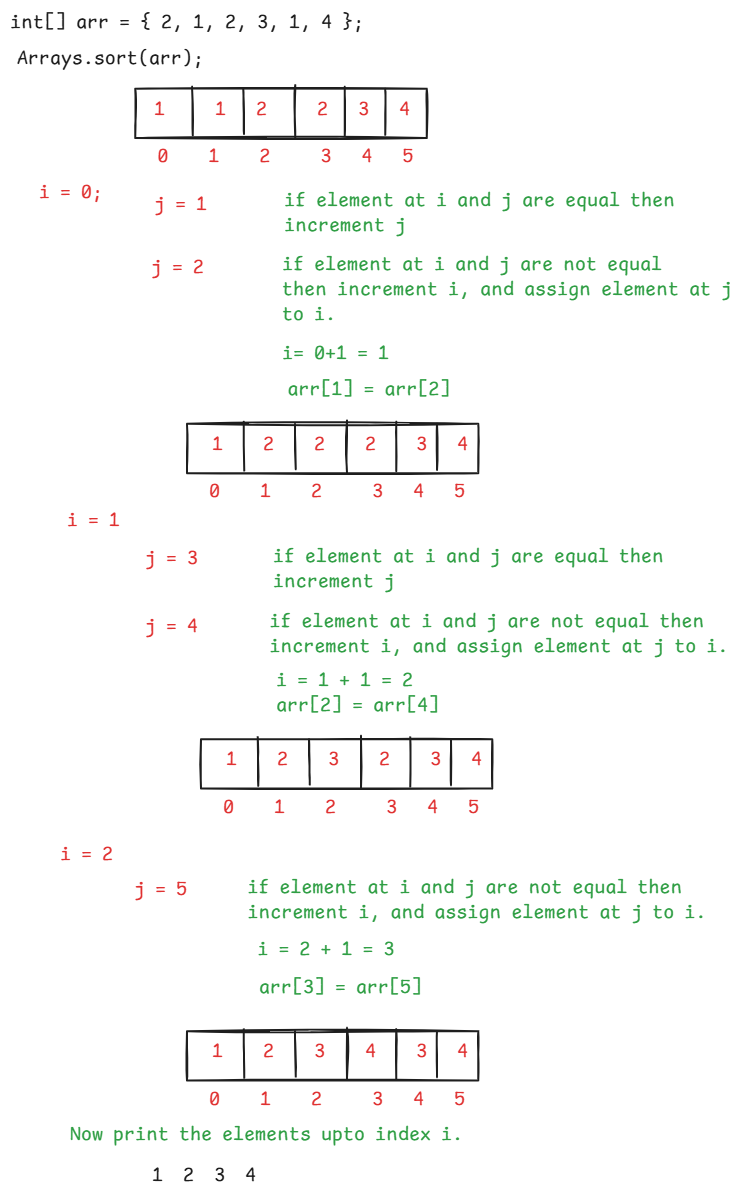
}

}

System.***out***.println("element : " + element + ", is consecutively repeated for : " + maxi + " times");

}

}



/\*

\* write a program to remove the duplicate elements

\* from the given array.

\* logic:

\* 1. sort the array elements.

\* 2. create two variables i and j

\* 3. start i at 0 and j at 1

\* 4. compare the elements arr[i] and arr[j]. if they are equal

\* continue the loop.

\* 5. if not equal then increment i, and copy arr[j] to arr[i]

\* 6. after the loop ends, display the elements from 0 to i.

\*/

**import** java.util.Arrays;

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

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

**int**[] arr = { 4, 9, 6, 4, 9, 7, 9, 9, 8, 6, 5 };

*removeDuplicates*(arr);

}

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

System.***out***.println("original array :");

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

System.***out***.print(arr[p] + " ");

}

Arrays.*sort*(arr);

**int** i = 0;

**for**(**int** j = 1; j < arr.length; j++) {

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

**continue**;

}

**else** {

i++;

arr[i] = arr[j];

}

}

System.***out***.println("\n after removing the duplicates");

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

System.***out***.print(arr[k] + " ");

}

}

}

for each loop:

* In Java, we have two types of for loops.

1. Numerical for loop
2. for each loop

* Numerical for loop is just a normal for loop and we use it for incrementing/decrementing the loop variable.
* for each loop is a special loop, used for iterating the elements of an array or a collection object.
* To write a for each loop, the keyword is “for” only.

syntax:

for( datatype variable : array/collection )

{

statements;

}

ex1:

int[] arr = { 3, 7, 1, 9 };

for(int x : arr) {

S.o.print(x + “ “);

}

output: 3 7 1 9

ex2:

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

int evenSum=0, oddSum=0;

for(int x : arr) {

if(x % 2 == 0)

evenSum += x;

else

oddSum += x;

}

S.o.p(evenSum);

S.o.p(oddSum);

output: 10

9

/\*

\* write a program to find the union of two arrays.

\* for example:

\* int arr1[] = { 9, 3, 5, 6};

\* int arr2[] = { 2, 9, 7, 3};

\* output:

\* 9, 3, 5, 6, 2, 7

\*/

**import** java.util.HashSet;

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

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

**int**[] arr1 = {7, 2, 4, 1, 8, 4};

**int**[] arr2 = {9, 3, 2, 0, 1, 3, 5, 0};

*findUnion*(arr1, arr2);

}

**private** **static** **void** findUnion(**int**[] arr1, **int**[] arr2) {

//create HashSet class object

HashSet<Integer> hs = **new** HashSet<Integer>();

//add the elements of the first array to the HashSet object

**for**(**int** x : arr1) {

hs.add(x);

}

//add the elements of the second array to the HashSet object

**for**(**int** x : arr2) {

hs.add(x);

}

**int**[] arr3 = **new** **int**[hs.size()];

**int** i = 0;

//copy the elements from HashSet to arr3

**for**(**int** x : hs) {

arr3[i] = x;

i++;

}

//display the elements of arr3

**for**(**int** k : arr3) {

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

}

}

}

/\*

\* write a program to find the intersection of two arrays.

\* For example,

\* int[] arr1 = { 7, 3, 9, 1 };

\* int[] arr2 = { 3, 0, 5, 7, 8 };

\* output:

\* 3 7

\*/

**import** java.util.HashSet;

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

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

**int**[] arr1 = { 8, 2, 0, 4, 1};

**int**[] arr2 = { 6, 3, 8, 5, 1};

*findIntersection*(arr1, arr2);

}

**private** **static** **void** findIntersection(**int**[] arr1, **int**[] arr2) {

//create HashSet class object

HashSet<Integer> hs = **new** HashSet<Integer>();

//add the first array elements to the HashSet object

**for** ( **int** x : arr1 ) {

hs.add(x);

}

//read the elements from second array.

**for** ( **int** k : arr2 ) {

**if**( hs.contains(k) ) {

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

}

}

}

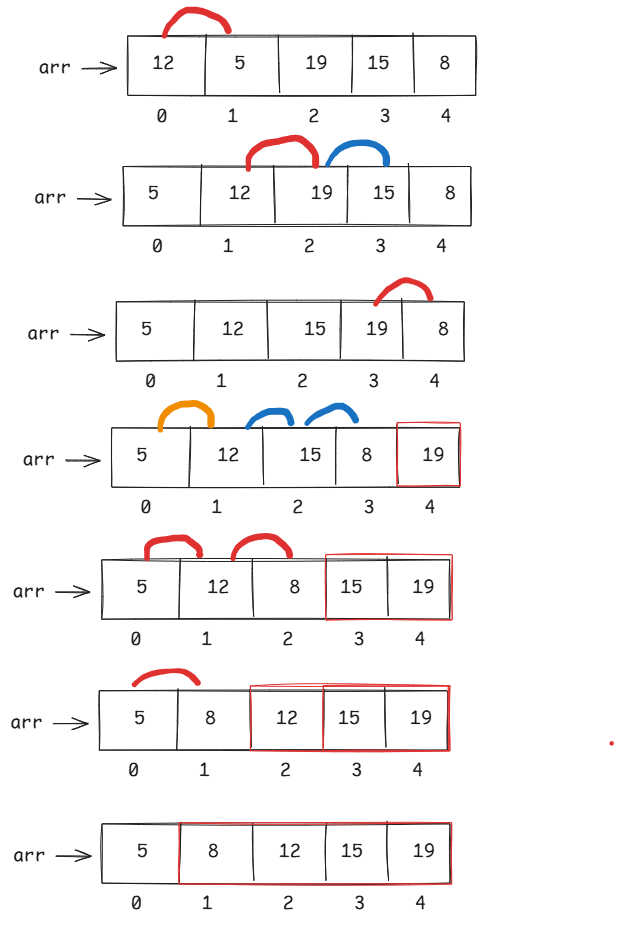
}

Bubble sort:

-----------

* The bubble sort technique works like below.

1. start from the beginning of the array
2. compare first element with second element and if first element is greater then swap them.
3. Move to the next pair, and repeat the same for all the array elements.
4. After one pass, the largest element is moved to the end of the array. It means, the largest element is bubbled.
5. Now again start from the beginning of the array, repeat the same process upto last but one element, so that second largest element is bubbled.
6. Repeat the process until n-1 elements are bubbled.



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

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

**int** arr[] = { 10, 7, 6, 8, 1, 9 };

*bubbleSort*(arr);

}

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

System.***out***.println("Array elements before sorting");

**for**(**int** k : arr) {

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

}

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

**int** n = arr.length;

//outer loop

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

//inner loop

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

**if**( arr[j] > arr[j+1] ) {

**int** temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

} //end inner loop

} //end outer loop

System.***out***.println("Array elements after sorting");

**for**(**int** k : arr) {

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

}

}

}

//finding the factorial with recursion.

**import** java.util.Scanner;

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

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

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

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

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

**int** result = *factorial*(n);

System.***out***.println("factorial = " + result);

}

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

**if**( n == 0 || n == 1 )

**return** 1;

**else**

**return** n \* *factorial*(n-1);

}

}

//fibonacci series with recursion

**import** java.util.Scanner;

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

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

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

System.***out***.println("Enter number of terms to be printed in fibonacci series");

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

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

**int** term = *fibonacci*(i);

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

}

}

**private** **static** **int** fibonacci(**int** k) {

**if**( k == 0 )

**return** 0;

**else** **if**(k == 1)

**return** 1;

**else**

**return** *fibonacci*(k-1) + *fibonacci*(k-2);

}

}

/\*

\* sort the elements of an array with quick sort.

\* quick sort algorithm works like below.

\* 1. choose the last element of the array as pivot.

\* 2. rearrange the elements in such a way that all smaller elements

\* should be at left of the pivot and larger elements should

\* be at the right.

\* 3. Now pivot is in its correct place.

\* 4. apply the same algorithm for left sub array and right sub array

\* 5. Now finally the array is sorted with quick sort.

\*/

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

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

**int**[] arr = {10, 7, 8, 9, 1, 5, 8, 5, 6};

**int** n = arr.length;

*quickSort*(arr, 0, n - 1);

System.***out***.println("Sorted array:");

**for** (**int** num : arr) {

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

}

}

// QuickSort function

**public** **static** **void** quickSort(**int**[] arr, **int** low, **int** high) {

**if** (low < high) {

// Find pivot index

**int** pivotIndex = *partition*(arr, low, high);

// Recursively sort elements before and after pivot

*quickSort*(arr, low, pivotIndex - 1);

*quickSort*(arr, pivotIndex + 1, high);

}

}

// Partition function

**public** **static** **int** partition(**int**[] arr, **int** low, **int** high) {

**int** pivot = arr[high]; // Choose last element as pivot

**int** i = low - 1; // Index of smaller element

**for** (**int** j = low; j < high; j++) {

**if** (arr[j] < pivot) {

i++;

// Swap arr[i] and arr[j]

**int** temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// Swap arr[i+1] and pivot (arr[high])

**int** temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

**return** i + 1; // Return pivot index

}

}

Two dimentional array:

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

* A One dimentional array contains a single row, with multiple columns.
* A Two dimentional array contains multiple rows, with multiple columns. It is also called a matrix.

datatype[][] variable; //declaration

variable = new datatype[rows][cols]; //creation

(or)

datatype[][] variable = new datatype[rows][cols];

(or)

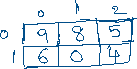
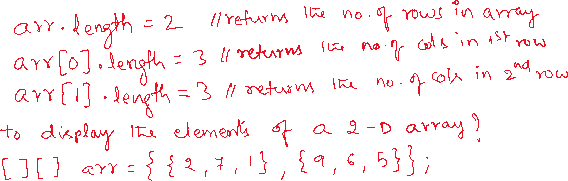
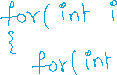
datatype variable[][] = new datatype[rows][cols];

(or)

datatype[] variable[] = new datatype[rows][cols];

ex:

int[][] arr = new int[2][3];



/\*

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

\* separately in the given two dimentional array.

\*/

**import** java.util.Scanner;

**public** **class** TwoDEvenOddSum {

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

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

System.***out***.println("Enter the no of rows");

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

System.***out***.println("Enter the no of cols");

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

//create array

**int**[][] arr = **new** **int**[rows][cols];

//read the input elements

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

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

System.***out***.println("Please enter element for arr[" + i +"][" + j + "]");

arr[i][j] = scan.nextInt();

}

}

*findEvenOddSum*(arr);

}

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

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

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

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

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

evenSum += arr[i][j];

**else**

oddSum += arr[i][j];

}

}

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

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

}

}

/\*

\* write a program to find the sum of left and right

\* diagonal elements of the two dimentional array

\* separately.

\*/

**public** **class** DiagonalSum {

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

**int**[][] arr = { {2, 9, 7}, {1, 4, 6}, {5, 0, 1} };

*findDiagonalSum*(arr);

}

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

**int** leftDiagonalSum = 0, rightDiagonalSum = 0;

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

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

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

leftDiagonalSum += arr[i][j];

}

**if** ( i + j == arr.length - 1) {

rightDiagonalSum += arr[i][j];

}

}

}

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

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

}

}

/\*

\* write a program to find the addition of two matrices.

\* ex:

\* 2 0 9 3 1 8 5 1 17

\* 1 3 1 + 0 2 9 = 1 5 10

\* 6 4 2 4 5 6 10 9 8

\*/

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

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

**int**[][] m1 = { {2, 8, 0}, {1, 4, 7} };

**int**[][] m2 = { {9, 0, 5}, {3, 7, 1} };

//find the rows in matrix1

**int** r1 = m1.length;

//find the cols in matrix1

**int** c1 = m1[0].length;

//find the rows in matrix2

**int** r2 = m2.length;

//find the cols in matrix2

**int** c2 = m2[0].length;

**if**(r1 == r2 && c1 == c2) {

*addMatrices*(m1, m2, r1, c1);

}

**else** {

System.***out***.println("You can't add the matrices");

}

}

**private** **static** **void** addMatrices(**int**[][] m1, **int**[][] m2, **int** r1, **int** c1) {

**int**[][] m3 = **new** **int**[r1][c1];

**for**(**int** i = 0; i < r1; i++ ) {

**for**(**int** j = 0; j < c1; j++) {

m3[i][j] = m1[i][j] + m2[i][j];

}

}

System.***out***.println("matrix1 : ");

**for**(**int** i = 0; i < r1; i++) {

**for**(**int** j = 0; j < c1; j++) {

System.***out***.print( m1[i][j] + " ");

}

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

}

System.***out***.println("matrix2 : ");

**for**(**int** i = 0; i < r1; i++) {

**for**(**int** j = 0; j < c1; j++) {

System.***out***.print( m2[i][j] + " ");

}

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

}

System.***out***.println("matrix3 (matrix1 + matrix2) : ");

**for**(**int** i = 0; i < r1; i++) {

**for**(**int** j = 0; j < c1; j++) {

System.***out***.print( m3[i][j] + " ");

}

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

}

}

}

string handling



* A string is a sequence of characters enclosed within double quotes.
* String is a class provided in java.lang package.
* java.lang package is the default package which is imported into every java program, automatically. So, you no need to import it again.
* String is not a primitive data type, because it is a class.
* There are 2 ways to create a String object in a Java program.

1. as a String literal
2. using new keyword.

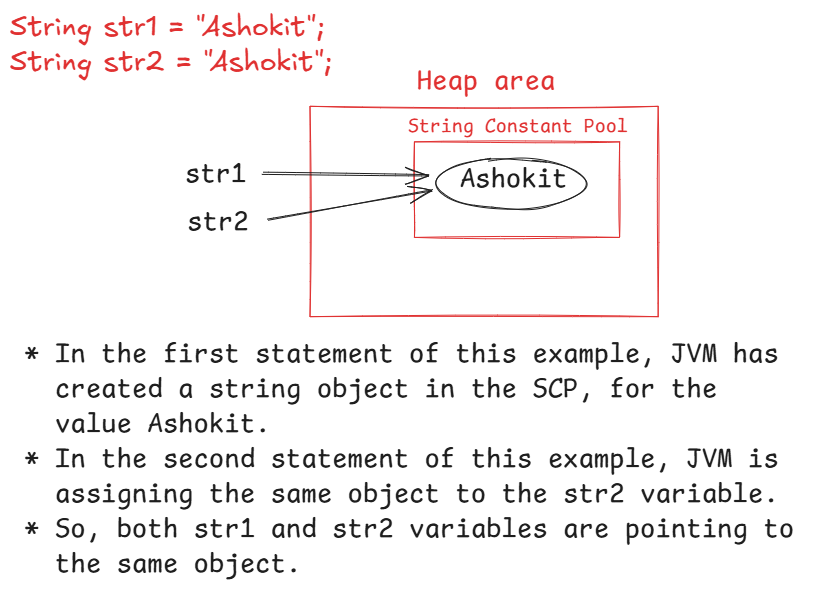
ex:

String str1 = “Ashokit”; //string literal

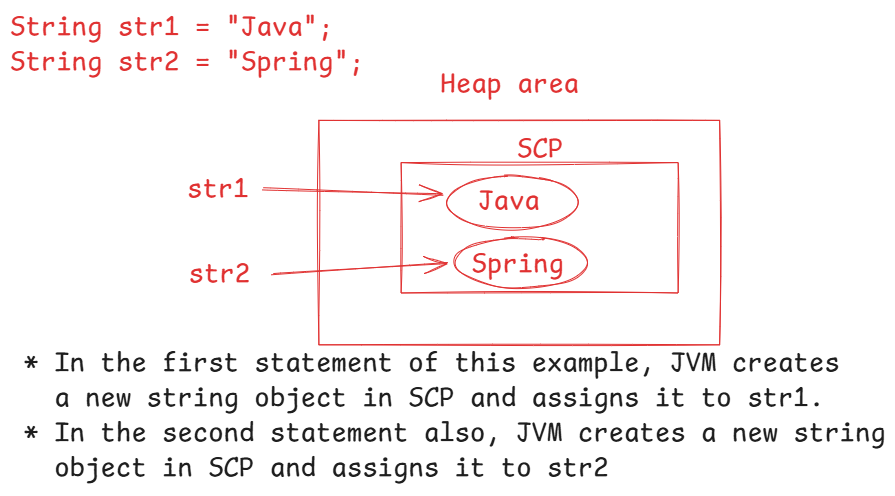
String str2 = new String(“Java”); //using new keyword

* When a string object is created as a literal, the object will be created in the String Constant Pool(SCP).
* This SCP is a special memory area, created in the Heap area of JVM.
* Before creating the object in the SCP, first JVM will verify whether the string object already exists in the SCP or not.
* If exists, then JVM doesn’t create another object again. Because SCP can only store the unique objects.
* If not exists, then JVM creates a new string object in the SCP.
* When you create a string object using new keyword, JVM first creates a string object in SCP for the given literal/value and also a copy of the same object is created in the Heap area.

example1:

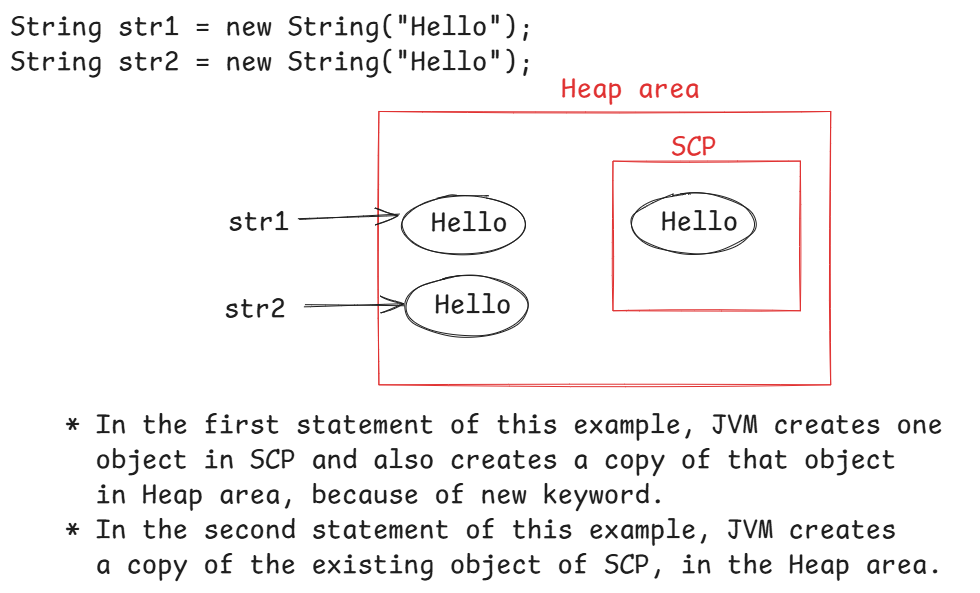


example2:



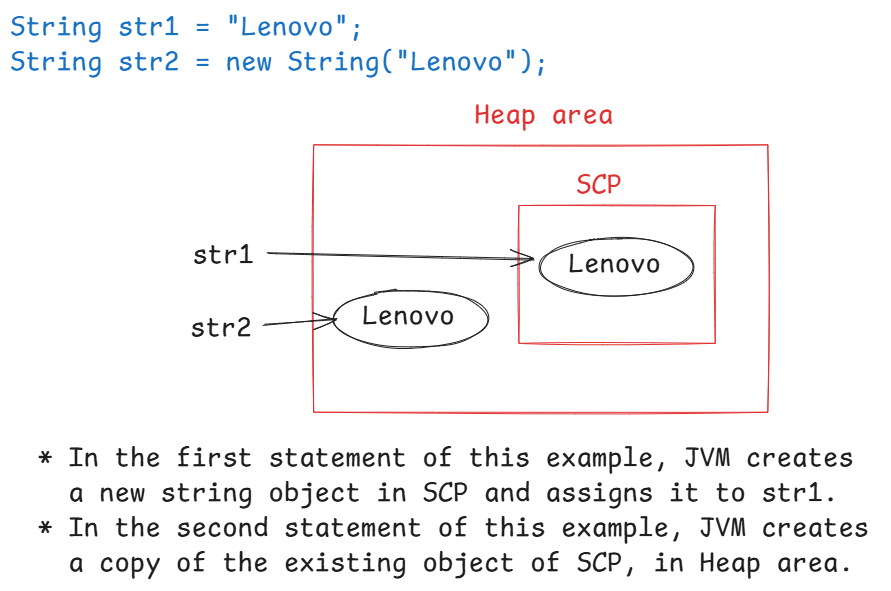
example3:







example4:



example5:

String s1 = new String("Dell");

String s2 = new String("Apple");

\* In this example, JVM creates two objects in SCP,

and also creates their copies in Heap area. So, totally

4 objects are created.

example6:

String str1 = “Ashokit”;

String str2 = new String(“ashokit”);

* In this example, JVM creates a new string object in SCP for

the first statement.

* For the second statement, JVM creates a new string object in SCP and

a copy of that object in Heap area.

* So, totally 3 objects are created.

Q) String str = new String();

In this statement, how many objects are created?

A) Two objects are created. Here, an empty string("") object

is created in SCP and a copy of that object is created

in the heap area. So, totally 2 objects are created.

Q) what is the difference between the below statements?

String str1 = "";

String str2 = " ";

A) The first string is empty string and the second string

is not an empty string. Because, second string has a

white space character.

S.o.println(str1.length()); //output: 0

S.o.println(str2.length()); //output: 1

Q) which of the following statements is correct?

int[] arr = new int[5];

String str = "hello";

1. S.o.println(arr[2].length);

2. S.o.println(arr.length());

3. S.o.println(arr.length);

4. S.o.println(str.length());

5. S.o.println(str.length);

6. S.o.println(str[2].length);

7. S.o.println(str[2].length());

A) 3 and 4

Q) what is the difference between "==" and

equals() ?

A) The equality operator(==) checks whether the two

reference variables are pointing to the same object

or not. If yes, it returns true, otherwise returns false.

equals() method checks the content/value in the two objects.

If they are same then returns true, otherwise returns false.

for ex:

String s1 = "hello";

String s2 = new String("hello");

S.o.println(s1 == s2); //false

S.o.println(s1.equals(s2)); //true

Q) String str1 = new String(“Ashokit”);

String str2 = new String(“ashokit”);

S.o.println(str1 == str2);

S.o.println(str1.equals(str2));

output: false

false

Q) String str1 = new String(“Ashokit”);

String str2 = new String(“Ashokit”);

S.o.println(str1 == str2);

S.o.println(str1.equals(str2));

output: false

true

Q) String str1 = “cat”;

String str2 = “cat”;

S.o.println(str1 == str2);

S.o.println(str1.equals(str2));

output: true

true

strings comparision:

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

1. equality operator(==)
2. equals() method
3. equalsIgnoreCase() method
4. compareTo() method.

* equals() method performs case-sensitive comparision.
* equalsIgnoreCase() method performs case-insensitive comparision.
* compareTo() method performs lexicographical comparision. It means, compares the ASCII values of each character.
* compareTo() method returns either positive or negitive or zero value.
* if string1 is greater than string2 then returns positive value.
* if string1 is less than string2 then returns negitive value.
* if string1 is equal to string2 then returns zero.

ex:

String str1 = “Anil”;

String str2 = “anil”;

S.o.println(str1.equals(str2)); //false

S.o.println(str1.equalsIgnoreCase(str2)); //true

S.o.println(str1.compareTo(str2)); // -32

S.o.println(str1.compareTo(str2) == 0 ); //false

retrieving a portion of a string:

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

* use substring() method to retrieve the portion/part of a given string.
* You can use substring() method in 2 ways.

1. substring(beginIndex) : returns a string from beginIndex to end of the string.
2. substring(beginIndex, endIndex) : returns a string from beginIndex to endIndex-1.

ex1:

String str1 = “The Sky is blue”;

String str2 = str1.substring(4);

String str3 = str1.substring(4, 7);

S.o.println(str2);

S.o.println(str3);

output: Sky is blue

Sky

ex2:

String str = “Cat sat on mat”;

String str2 = str.substring(0, 7);

String str3 = str.substring(4);

S.o.println(str2);

S.o.println(str3);

output: Cat sat

sat on mat

ex3:

String str = “hello”;

String str2 = str.substring(1, 7);

S.o.println(str2);

output: IndexOutOfBoundsException

toLowerCase() and toUpperCase():

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

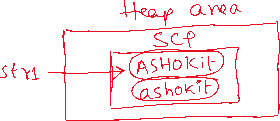
* toLowerCase() converts the letters of a string value to the lower case.
* toUpperCase() converts the letters of a string value to the upper case.
* In Java, String is an immutable object. It means, once the object created, it will not allow to make any changes to its data.
* So, when you perform any operation on a string object, the result will be stored into a new object.

ex1:

String str1 = “ASHOKit”;

str1.toLowerCase();

S.o.println(str1);



output: ASHOKit

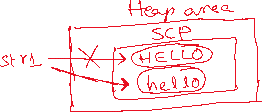
ex2:

String str1 = “HELLO”;

str1 = str1.toLowerCase();

S.o.println(str1);

output: hello



ex3: String str = “java”;

S.o.println(str.toUpperCase());

S.o.println(str);

output: JAVA

java

charAt(index) and indexOf(char):

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

* charAt(index) returns the character a the given index.
* if the given index is more than length-1 then you will get IndexOutOfBoundException.

ex:

String str = “cat sat”;

S.o.println(str.charAt(4)); // s

S.o.println(str.charAt(10)); //IndexOutOfBoundsException.

\* indexOf(char) returns the index of the first occurrence of a given character.

\* if the given character doesn’t exist in the string value then the result will be -1.

ex:

String str = “cat sat”;

S.o.println(str.indexOf(‘a’)); // 1

S.o.println(str.indexOf(‘y’)); // -1

S.o.println(str.indexOf(“sat”)); // 4

lastIndexOf(char): retruns the index of last occurrence of a given character. If not found then returns -1.

ex: String str = “cat sat”;

S.o.println(str.lastIndexOf(‘a’)); //5

indexOf(char, beginIndex): returns the index of occurrence of a character, starting from begin index.

ex: String str = “The cat sat on mat”;

S.o.println(str.indexOf(‘t’ , 7)); // 10

trim() : removes spaces before start of the first non-space character or

after the end of the last non-space character. It doesn’t remove

the spaces in between the characters.

ex1: String username = “ admin”;

if(username.equals(“admin”))

S.o.p(“Welcome Admin”);

else

S.o.p(“Wrong username”);

output: Wrong username

ex2: String username = “ admin”;

if(username.trim().equals(“admin”))

S.o.p(“Welcome Admin”);

else

S.o.p(“Wrong username”);

output: Welcome Admin

ex3:

String str = “ashok it”;

s.o.p(str); //ashok it

s.o.p(str.trim()); //ashok it

replace() method:

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

* It replaces the each substring of a string with a replacement string.

replace(substring, replacement)

ex1: String str1 = “The cat sat on mat”;

str1.replace(“cat”, “dog”);

S.o.p(str1);

output: The cat sat on mat

ex2: String str1 = “The cat sat on mat”;

str1 = str1.replace(“cat”, “dog”);

S.o.p(str1);

output: The dog sat on mat

replaceAll(): replaces each substring of a string that matches the specified regular expression with a given replacement string.

replaceAll(String regex, String replacement)

ex1:

String str = “My contact number : 123-456-7890”;

String result = str.replaceAll(“\\d”, “#”);

S.o.println(result);

output: My contact number : ###-###-####

Note: [\\d](file:///\\d) pattern matches to any digit(0-9)

ex2:

String str = “Spring is a framework”;

String result = str.replaceAll(“\\s”, “\_”);

S.o.println(result);

output: Spring\_is\_a\_framework

Note: [\\s](file:///\\s) pattern matches to a space.

ex3:

String str = “Java is a Programming language”;

String result = str.replaceAll(“\\s+”, “ “);

S.o.println(result);

output: Java is a Programming language

Note: [\\s](file:///\\s)+ pattern matches one or more spaces

ex4:

String str = “Java#@is$great!&”;

String result = str.replaceAll(“[^a-zA-Z0-9]”, “”);

S.o.println(result);

output: Javaisgreat

ex5:

String str = “Ashokit Solutions”;

String result = str.replaceAll(“[aeiouAEIOU]”, “”);

S.o.println(result);

output: shkt Sltns

ex6:

String creditCard=”1234-5678-9012-3456”;

String result = creditCard.replaceAll(“[^-](?=.{4})”, “X”);

S.o.println(result);

output: XXXX-XXXX-XXXX-3456

Note: (?=.{4}) pattern checks if there are 4 characters following the current character or not.

ex7:

String str = “<h1>Hello, <b>Shekher</b>!</h1>”;

String result = str.replaceAll(“<[a-zA-Z0-9/]>”, “”);

S.o.println(result);

output: Hello, Shekher!

split() method:

* It will divide a string into an array of substrings based on a given delimiter.

ex1:

String str1 = “JSP is a technology”;

String[] str2 = str1.split(“ “);

for(String s : str2)

S.o.println(s);

output:

JSP

is

a

technology

ex2:

String str = “apple,banana,orange,grapes”;

String[] fruits = str.split(“,”);

for(String fruit : fruits) {

S.o.println(fruit);

}

output:

apple

banana

orange

grapes

join() method:

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

* It will combine multiple strings together into a single string.
* It is a static method, so we can call it with classname.

ex1:

String[] names = { “Spring”, “Boot”, “Microservices” };

String joinedString = String.join(“ “, names);

S.o.println(joinedString);

output: Spring Boot Microservices

ex2:

String[] technologies = { “JDBC”, “Servlet”, “JSP” };

String joinedString = String.join(“ and “, technologies);

S.o.println(joinedString);

output: JDBC and Servlet and JSP

isEmpty() and isBlank() :

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

* isEmpty() method finds the length of a string and if it is zero then returns true, otherwise returns false.
* isBlank() method finds the length of a string, after trim and if it is zero then returns true, otherwise returns false.

ex1:

String str1 = “”;

S.o.p(str1.isEmpty()); //true

S.o.p(str1.isBlank()); //true

ex2:

String str1 = “ “;

S.o.p(str1.isEmpty()); //false

S.o.p(str1.isBlank()); //true

ex3:

String str1 = “Hello”;

S.o.p(str1.isEmpty()); //false

S.o.p(str1.isBlank()); //false

startsWith() and endsWith():

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

ex: String str1 = “ashokit@gmail.com”;

S.o.println(str1.startsWith(“gmail”)); //false

S.o.println(str1.endsWith(“gmail”)); //false

ex:

String str1 = “ASHOKIT@gmail.com”;

S.o.println(str1.startsWith(“ashok”)); //false

S.o.println(str1.toLowerCase().startsWith(“ashok”)); //true

ex:

String str1 = “ashokit@gmail.com”;

str1.toUpperCase();

S.o.println(str1.endsWith(“GMAIL.COM”)); //false

ex:

String str1 = “ashokit@gmail.com”;

str1 = str1.toUpperCase();

S.o.println(str1.endsWith(“GMAIL.COM”)); //true

valueOf() : It converts a specified value from another data type to the

string type.

It is a static method, so you can call it with classname.

ex1:

String str = “Hello” + String.valueOf(5+5+5);

S.o.println(str); //Hello15

ex2: String str = “Hello” + String.valueOf(12) + String.valueOf(345);

S.o.println(str); //Hello12345

ex3:

String str = “Hello” + String.valueOf(null);

S.o.println(str); // NullPointerException

ex4:

String str = “Hello” + String.valueOf(“null”);

S.o.println(str); //Hellonull

toCharArray() : converts this string to an array of characters.

ex: String str = “hello”;

char[] ch = str.toCharArray();

intern(): moves the cursor from the object in heap area to the object in SCP.

ex:

String str1 = “hello”;

String str2 = new String(“hello”);

S.o.println( str1 == str2 ); //false

S.o.println( str1 == str2.intern() ); //true

Q) what is the difference between the below statements?

username.equals(“admin”)

“admin”.equals(username)

1. if username is null, then the first statement throws NullPointerException. But the second statement returns false.

/\*

\* write a program to count the vowels and consonants in

\* a given string.

\* For example,

\* str = "hello"

\* vowels count : 2

\* consonants count : 3

\*/

**import** java.util.Scanner;

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

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

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

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

String str = scan.nextLine();

*countVowelsAndConsonants*(str);

}

**private** **static** **void** countVowelsAndConsonants(String str) {

**if**(str == **null** || str.trim().length() == 0) {

System.***out***.println("Hey! You must provide a string value");

**return**;

}

**int** vCount = 0, cCount = 0;

//convert the string value to lower case

str = str.toLowerCase();

//convert the string to character array

**char** ch[] = str.toCharArray();

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

**if**(Character.*isLetter*(ch[i])) {

**switch**(ch[i]) {

**case** 'a':

**case** 'e':

**case** 'i':

**case** 'o':

**case** 'u': vCount++; **break**;

**default**: cCount++;

}

}

}

System.***out***.println("vowels count : " + vCount);

System.***out***.println("consonants count : " + cCount);

}

}

/\*

\* write a program to swap the two strings, without using

\* a third variable.

\* For example,

\* str1 = "hello"

\* str2 = "admin"

\* output:

\* str1 = "admin"

\* str2 = "hello"

\*/

**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();

*swap*(str1, str2);

}

**private** **static** **void** swap(String str1, String str2) {

**if**(str1==**null** || str2==**null** || str1.trim().length()==0 ||str2.trim().length()==0) {

System.***out***.println("Hey! you must enter valid input");

**return**;

}

str1 = str1 + str2;

str2 = str1.substring(0, str1.length() - str2.length());

str1 = str1.substring(str2.length());

System.***out***.println("after swapping");

System.***out***.println("first string = " + str1);

System.***out***.println("second string = " + str2);

}

}

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

/\*

\* write a program to check whether the given

\* two strings are Anagrams or not.

\* Anagrams : If we can form another string by rearranging the letters of

\* this string, then they are called Anagrams.

\* ex:

\* str1 = "Listen" str2 = "Silent" -- Anagrams

\* str1 = "State" str2 = "Taste" -- Anagrams

\* str1 = "Hello" str2 = "Admin" -- not Anagrams

\*/

**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("enter first string");

String str1 = scan.nextLine();

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

String str2 = scan.nextLine();

**boolean** flag = *checkAnagrams*(str1, str2);

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

System.***out***.println("Both the strings are anagrams!");

**else**

System.***out***.println("Both the strings are not anagrams!!");

}

**private** **static** **boolean** checkAnagrams(String str1, String str2) {

**if**(str1==**null** || str2==**null** || str1.trim().length()==0 || str2.trim().length()==0) {

System.***out***.println("Hey! you must enter valid input");

System.*exit*(1);

}

//check the lengths

**if**(str1.length() != str2.length())

**return** **false**;

//convert the strings to char arrays

**char**[] ch1 = str1.toLowerCase().toCharArray();

**char**[] ch2 = str2.toLowerCase().toCharArray();

//sort the arrays

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**;

}

}

**return** flag;

}

}

/\*

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

\* palindrome or not, without using built-in reverse method.

\* ex:

\* str = "Liril"

\* output: palindrome

\* str = "admin"

\* output: not a palindrome

\*/

**import** java.util.Scanner;

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

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

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

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

String str = scan.nextLine();

**boolean** status = *checkPalindrome*(str);

**if**(status == **true**) {

System.***out***.println("The given string is palindrome");

}

**else** {

System.***out***.println("The given string is not a palindrome");

}

}

**private** **static** **boolean** checkPalindrome(String str) {

**if**(str == **null** || str.trim().length() == 0) {

System.***out***.println("Hey! You must enter a valid input");

System.*exit*(0);

}

**char** ch[] = str.toLowerCase().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 print the frequency of each character

\* in a given string.

\* frequency ---> the number of times, a character is repeated.

\* For example,

\* str = "missisipi"

\* output:

\* m - 1

\* p - 1

\* s - 3

\* i - 4

\*/

**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 a string");

String str = scan.nextLine();

*findFrequency*(str);

}

**private** **static** **void** findFrequency(String str) {

**if**(str == **null** || str.trim().length() == 0) {

System.***out***.println("Hey! You must enter a valid input");

System.*exit*(0);

}

**char** ch[] = str.toLowerCase().toCharArray();

//sort

Arrays.*sort*(ch);

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

{

**int** count = 0;

**for**(**int** j = i; j < ch.length; j++)

{

**if**(ch[i] == ch[j])

{

count++;

i=j;

}

**else**

{

**break**;

}

} //end of inner loop

System.***out***.println(ch[i] + " - " + count);

} //end of outer loop

} //end of findFrequency()

} //end of class

Q) why strings are immutable?

A) To implement thread-safety and to provide security, string objects are given as immutable objects in Java.

Suppose, In multi-threading, two threads are sharing the same string object and if one thread modifies the string value, then the other thread is not effected, because of string object is immutable. This is called thread-safety.

In an application, sensitive data like usernames, passwords, URLs are strings. Suppose, a hacker has modified the original username/password to steal the information from the database, then the hacker made changes are not stored in the same objects, because strings are immutable objects. So, new objects are created and hacker can’t able to steal the information. This is called security.

StringBuffer class:

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

* StringBuffer is a class of java.lang package.
* StringBuffer is not a child class of String. Just both the classes are from the same package java.lang and both works on the string values.
* StringBuffer objects are mutable and thred-safe objects.
* When a StringBuffer object is created, additional space for 16 characters is allocated and when you modify the StringBuffer object and if the new value crosses the capacity then the capacity will be doubled.

How to create a StringBuffer object?

StringBuffer sb1 = “hello”; //error

StringBuffer sb2 = new StringBuffer(“hello”); //correct

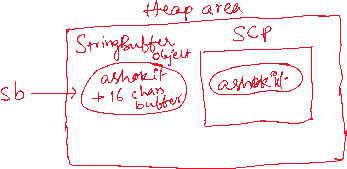
How many objects are created for the below statement?

StringBuffer sb = new StringBuffer(“ashokit”);

Ans) Two objects are created.

For the string literal, object is created in SCP and a new object

is created in Heap area for the StringBuffer.



How many objects are created for the below statements?

StringBuffer sb1 = new StringBuffer(“hello”);

StringBuffer sb2 = new StringBuffer(“hello”);

Ans) Three objects are created.

One object is created in SCP for the string literal and two objects are created in Heap area.

comparing the StringBuffer objects?

1. we can compare using == operator(equality operator)
2. we can compare with equals() method. But it also works like equality operator.

Hint: In StringBuffer, equals() method is not overridden. So, it calls equals() method of Object class and that equals() method again uses equality operator(==) for comparision.

Suppose, if you want to compare the content of StringBuffer objects, first you have to convert the StringBuffer objects to String objects. Then you have to call equals() method of String object.

ex:

StringBuffer sb1 = new StringBuffer(“hello”);

StringBuffer sb2 = new StringBuffer(“hello”);

S.o.println(sb1 == sb2); //false

S.o.println(sb1.equals(sb2)); //false

ex2:

StringBuffer sb1 = new StringBuffer(“hello”);

StringBuffer sb2 = new StringBuffer(“hello”);

S.o.println(sb1 == sb2); //false

S.o.println(sb1.toString().equals(sb2.toString()); //true

key methods of StringBuffer class:

1. append() : appends the new value to the existing value in a StringBuffer object.
2. delete(start, end): deletes the characters from start to end-1 indexes in a StringBuffer object.
3. insert(index, string): inserts the given string value at the given index in a StringBuffer object.
4. reverse() : will reverse the characters in a StringBuffer object.
5. length() : returns the number of characters in a StringBuffer object
6. capacity() : returns the current capacity of the StringBuffer object.

ex1:

//string palindrome code with StringBuffer object.

StringBuffer sb1 = **new** StringBuffer("hello");

String s1 = sb1.toString();

sb1.reverse();

**if**( s1.equals(sb1.toString()))

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

**else**

System.***out***.println("not a palindrome");

ex2:

StringBuffer sb1 = **new** StringBuffer("The is Blue");

sb1.insert(4, "Sky ");

System.***out***.println(sb1);

output: The Sky is Blue

ex3:

StringBuffer sb1 = **new** StringBuffer("The cat sat on mat");

sb1.replace(4, 7, "dog");

System.***out***.println(sb1);

StringBuilder class:

* StringBuilder class is in java.lang package.
* StringBuilder class also works on string values.
* StringBuilder class objects are mutable and not a thread-safe objects.
* The StringBuffer and StringBuilder classes have exactly the same operations(methods).

Q) What is the difference between String and StringBuffer?

A) String objects are immutable and thread-safe objects.

StringBuffer objects are mutable and thread-safe objects.

Q) How StringBuffer is thread-safe object, it is mutable?

A) The methods of StringBuffer class are synchronized methods, so only one thread at a time is allowed to perform the operations. So, it is a thread-safe object.

Q) What is the difference between StringBuffer and StringBuilder?

A) StringBuffer is mutable and thread-safe object.

StringBuilder is mutable and not a thread-safe object.

Q) Why StringBuilder is not thread-safe?

A) The methods of StringBuilder class are not synchronized methods.

Q) In a multi-threading application, which is preferable StringBuffer or StringBuilder?

A) StringBuffer

Q) In a single-thread application, which is preferable StringBuffer or StringBuilder?

A) StringBuilder

Q) Why StringBuilder is provided when StringBuffer already exist?

A) For single thread applications, StringBuilder performance is better than StringBuffer, because StringBuffer methods are synchronized.

/\*

\* write a program to remove the duplicate characters from

\* a given string.

\* Example:

\* str = "Missisipi"

\* output: "misp"

\*/

**import** java.util.Scanner;

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

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

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

System.***out***.println("Enter a string value");

String str = scanner.nextLine();

String newStr = *removeDuplicateChars*(str);

System.***out***.println("The given string : " + str);

System.***out***.println("The string after removing duplicate chars : " + newStr);

}

**private** **static** String removeDuplicateChars(String str) {

**if**(str == **null** || str.trim().length() == 0)

{

System.***out***.println("Hey! You must enter valid input");

System.*exit*(0);

}

//convert the string value to lowercase

str = str.toLowerCase();

//create a boolean array with size 256

**boolean**[] seen = **new** **boolean**[256];

//create a StringBuilder object

StringBuilder builder = **new** StringBuilder();

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

**char** ch = str.charAt(i);

**if**( seen[ch] == **false** )

{

builder.append(ch);

seen[ch] = **true**;

}

}

**return** builder.toString();

}

}

/\*

\* write a program to compress a string value

\* For example,

\* str = "aabccdeez";

\* output: a2b1c2d1e2z1

\*/

**import** java.util.Scanner;

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

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

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

System.***out***.println("Enter a string value");

String str = scanner.nextLine();

String newStr = *compress*(str);

System.***out***.println("The given string : " + str);

System.***out***.println("The string after compression : " + newStr);

}

**private** **static** String compress(String str) {

**if**(str == **null** || str.trim().length() == 0) {

System.***out***.println("Hey! You must enter valid input!");

System.*exit*(0);

}

str = str.toLowerCase();

**int** count = 1;

StringBuilder builder = **new** StringBuilder();

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

{

**if**( i + 1 < str.length() && str.charAt(i) == str.charAt(i+1)) {

count++;

}

**else** {

builder.append(str.charAt(i)).append(count);

count = 1;

}

}

**return** builder.toString();

}

}

Object Oriented Programming System(OOPS)

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

\* To develop a software system, there are 2 programming paradigms

are available.

1. Procedure Oriented Programming model

2. Object Oriented Programming model

\* In Procedure Oriented Programming model(POP), software systems

are developed by creating functions/procedures.

\* For example, The programming languages like C and Pascal are POP

languages.

\* With POP, the disadvantages are,

1. more complexity

2. no modularity

3. no real-world mapping

4. no scalability

\* So, POP model is only suitable for designing and implementing

smaller systems.

\* To overcome the limitations in POP model, in early 1980's OOP model

was introduced.

\* OOP model has defined some principles/practices to design and

implement reusable and maintainable software systems.

\* Those principles are,

1. Encapsulation

2. Inheritance

3. Polymorphism

4. Abstraction

Encapsulation:

\* Encapsulation is a word derived from another word, "capsule".

\* In medical terms, "capsule" is a single unit, where we can keep

some powder and nuts together.

\* Similiarly, In programming, a class is used to implement this

encapsulation principle/practice.

\* By using a class, we can keep the data and related functionality

together with proper access modifiers, we can implement encapsulation.

\* With encapsulation, we can prevent accidentally modifying the data

by the outside world.

Inheritance:

\* Inheritance is a process of acquiring/inheriting the properties and

behaviour from one class to another class.

\* Inheritance provides the advantages like,

1. code reusability

2. improves developers productivity

3. reduces redundency

Polymorphism:

\* Polymorphism is a combination of two Greek words.

\* Poly means many and morphism means forms. So polymorphism represents

"many forms".

\* Polymorphism denotes that a software should provide many ways

to perform an action.

\* For example, a customer can open a bank account by providing

adhaar card or voter id or pan card as a proof. So, the application

has provided many ways to perform an action.

Abstraction:

\* Abstraction is a process of providing the essential information

to the user of the system, by hiding non-essential information.

\* For example, when you download a file from a website/application,

the essential information will be shown to the user like the filname

and the size and also how long time it will take to download the

file. But the other details like physical location of the file,

which programming is used to develop the application, etc.. are

not shown to the user. This is called abstraction.

\* In Java, abstraction can be implemented using abstract classes and

interfaces.

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

class & object:

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

\* A class is a template or blueprint for constructing a group of objects with similar attributes and behavior.

\* The attributes are also called properties or variables and the

behaviour is also called functionality or methods.

\* A class is a collection of variables and methods for a group of objects.

\* A class is a word took from another word classification.

\* An object is an instance of a class, which contains the values

for the variables created in a class.

\* At runtime of an application, the object exist in memory of JVM,

but not the class.

syntax of a class:

<access modifier> class <classname>

{

variables;

methods;

}

ex:

public class Policy

{

int policyId;

String policyType;

String policyHolder;

void calculatePremium()

{

//logic

}

void showPolicyDetails()

{

//logic

}

}

syntax of creating an object:

classname objectname = new classname();

ex:

Policy policy1 = new Policy();

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

/\*

\* This program is a demo for creating a class and object.

\*/

**class** BankAccount

{

**long** accountNumber;

String accountHolderName;

**double** balance;

**void** deposit(**double** amount)

{

balance = balance + amount;

System.***out***.println("amount deposited is : " + amount);

}

**void** withdraw(**double** amount)

{

**if**(amount > balance) {

System.***out***.println("withdraw is not possible");

**return**;

}

balance = balance - amount;

System.***out***.println("amount withdrawn is : " + amount );

}

**void** checkBalance()

{

System.***out***.println("current balance : " + balance);

}

} //end of the BankAccount class

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

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

//creating object

BankAccount account1 = **new** BankAccount();

//assign the data to the variables

account1.accountNumber = 190087111;

account1.accountHolderName = "Miller";

account1.balance = 18000.0;

//invoke the methods

account1.checkBalance();

account1.withdraw(13000.0);

account1.checkBalance();

account1.deposit(3679.0);

account1.checkBalance();

}

}

output:

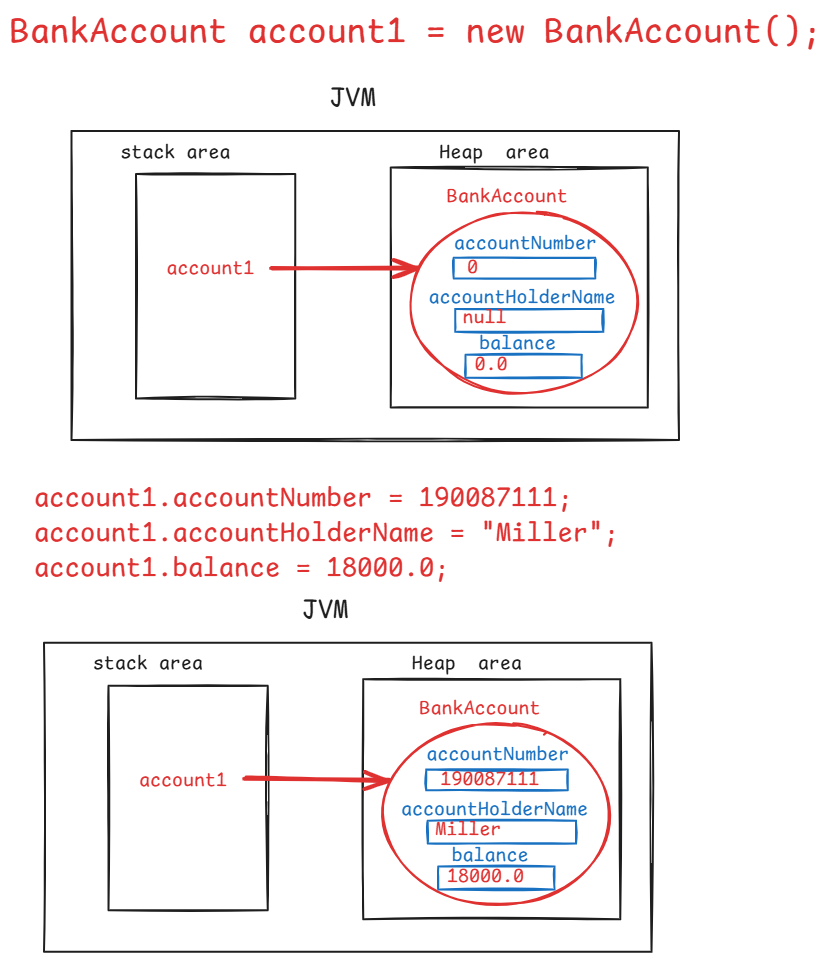
current balance : 18000.0

amount withdrawn is : 13000.0

current balance : 5000.0

amount deposited is : 3679.0

current balance : 8679.0



Q) can we write multiple classes in a Java program?

A) Yes.

Q) can we create multiple objects for a class?

A) Yes.

Q) will the memory is allocated for the methods also, when an object is created?

A) No.

When an object is created, the memory is allocated for variables in the class, but not for methods.

The methods are loaded into another area in JVM called Method Area.

JVM creates links internally between objects in Heap area and the methods in Method area.

access modifiers:

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

* Access modifiers are the keywords in Java, which defines the visibility of an element in a Java Project.
* In Java, we have 4 access modifiers.

1. default
2. private
3. protected
4. public

default:

* If you do not specify any access modifier for a class or a variable or a method then by default it has default access modifier.
* If any member has default access modifier, then it is visible within the same package only.

private:

* You can use this private access modifier for variables or methods or constructors or inner classes, but not for outer classes.
* If any member has private access modifier, then it is visible within the same class only.

protected:

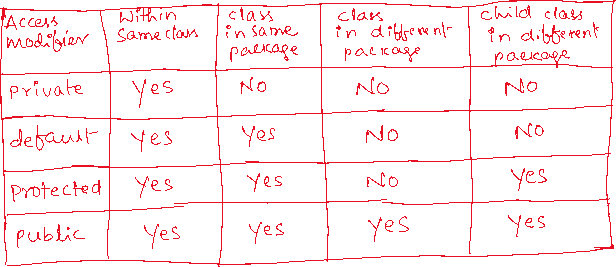
* You can use this protected access modifier for variables or methods or constructors or inner classes, but not for outer classes.
* If any member has protected access modifier, then it is visible within the same package and also in the child classes of another package.

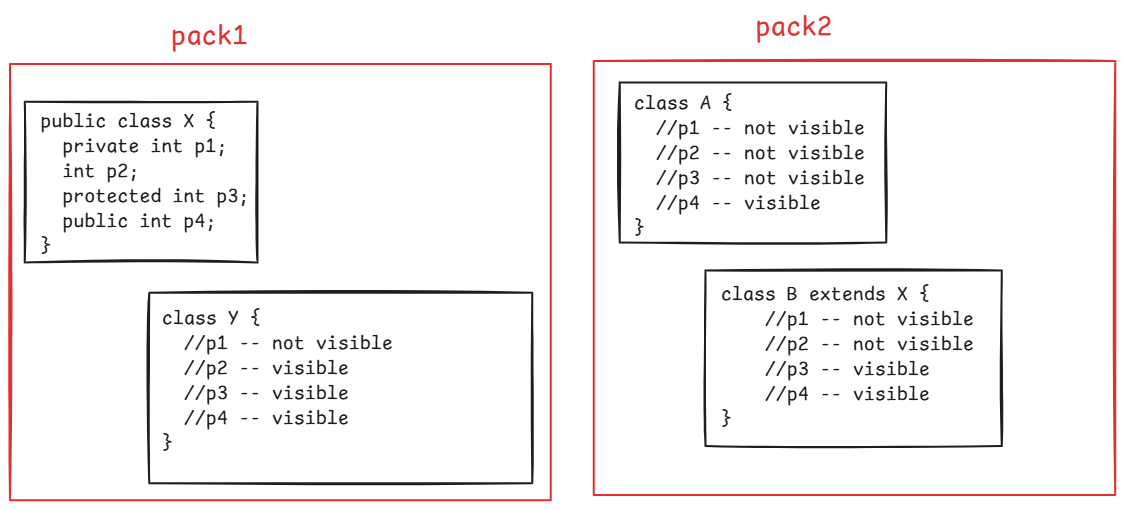
public:

* You can use this public access modifier for variables, methods, classes, constructors and inner classes.
* If any member has public access modifier, then it is visible to the classes in the same package and also to the classes in other packages of the project.

Note: private < default < protected < public

Note: another name of default is package private.





constructor:

* A constructor is a special method in a class and it is used to initialize the objects when they are created.
* Initializing an object means, assigning the values to the fields of an object.
* We can also initialize an object manually by without using a constructor also. But it has some drawbacks.

1. if an object has more fileds, then more lines of code is required to initialize the fields of an object. It will increase complexity.
2. if the fields are private, then you can’t initialize them manually at outside of the class.
3. while initializing the fields manually, invalid data can be assigned.

* As a solution, we can create a constructor inside the class to initialize an object.
* A constructor is a special method in a class, because of the below reasons.

1. The constructor name should match with the classname.
2. The constructor does not have a return type.
3. The constructor is executed automatically as part of the object creation.

Types of constructors:

1. parameter-less constructor / constructor without arguments
2. parameterized constructor / constuctor with arguments.

* In every class of a Java program, at least one constructor is required.
* If a programmer is not created any constructor in a class then Java will create a default constructor.
* This default constructor also belongs to parameter-less constructor category. This default constructor will assign the default values to the fields based on their data type.
* If a programmer has created a constructor in a class then Java does not create a default constructor in that class.
* create a parameter-less constructor, when you want to initialize multiple objects with the same initial state.
* create a parameterized constructor, when you want to initialize each object with a different initial state.
* The two types of constructors are given for flexibility and a programmer can define how the objects should be initialized of a class.
* we can specify the access modifier for a constructor. If not specified then bydefault is default.

**class** RechargePlan {

**private** String planId;

**private** String planType;

**private** String duration;

**private** **double** price;

//parameter-less constructor

**public** RechargePlan() {

planId = "PLAN-199";

planType = "TopUp";

duration = "28 days";

price = 199.0;

}

//parameterized constructor

**public** RechargePlan(String planId, String planType, String duration, **double** price) {

**this**.planId = planId;

**this**.planType = planType;

**this**.duration = duration;

**this**.price = price;

}

**public** **void** showPlanData() {

System.***out***.println("PLAN ID : " + planId);

System.***out***.println("PLAN TYPE : " + planType);

System.***out***.println("DURATION : " + duration);

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

}

} //end of RechargePlan class

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

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

RechargePlan recharge1 = **new** RechargePlan();

RechargePlan recharge2 = **new** RechargePlan("PLAN-599", "REGULAR", "56 days", 599.0);

System.***out***.println("-------- Default plan details ----------");

recharge1.showPlanData();

System.***out***.println("=".repeat(30));

System.***out***.println("-------- Regular plan details ---------------");

recharge2.showPlanData();

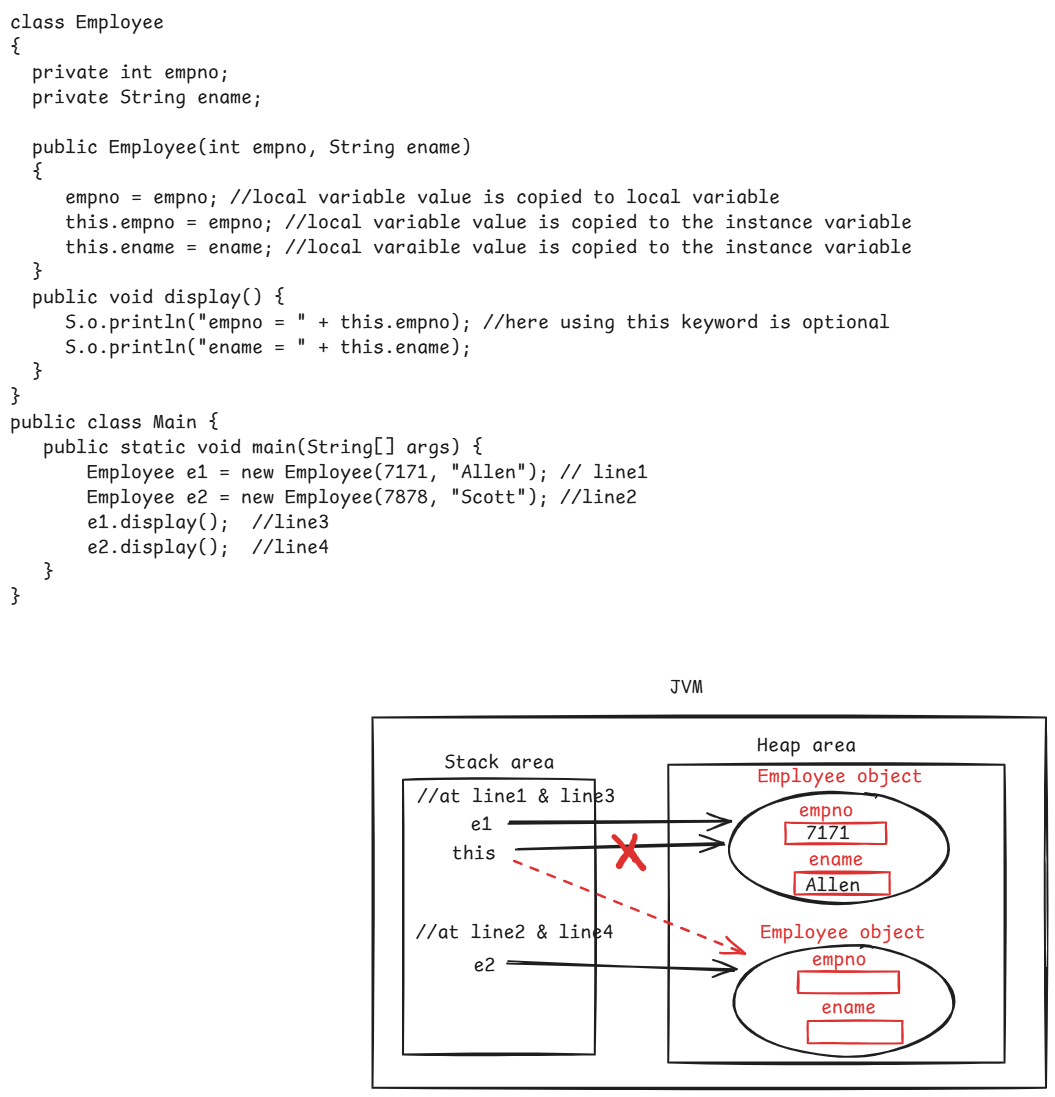
}

}

this keyword:

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

* this is a keyword in Java, and it is used to differentiate an instance variable with a local variable.
* this keyword always refers to the current instance of a class.





Q) what is the output of the below code?

class ClassA {

private int x;

private int y;

public ClassA(int a, int b) {

x = a; //instance variable name is different from local variable

y =b;

}

public void doSomething(int a, int b, int x, int y)

{

x = a + this.x + y;

this.y = this.x + b + a;

this.x = this.x + this.y;

this.y = this.x - this.y;

S.o.println(x);

S.o.println(y);

}

}

class Main {

p s v m(String[] args) {

ClassA classA = new ClassA(20, 35);

classA.doSomething(20, 30, 40, 50);

}

}

output: 90

50

getter and setter methods:

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

* A getter method is created in a class to return the value of an instance variable.
* A setter method is created in a class to update the value of an instance variable.
* If the instance variables doesn’t have private access modifier then creating a getter and setter methods is optional.
* If the instance variables are private then create a getter and setter methods to read or update the value of the variable.
* Make sure that getter and setter methods are public methods.
* Suppose, if you have a private variable int id;
* The getter and setter methods can be created as,

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

* Suppose, we have a private variable String customerName;
* The getter and setter methods can be created as,

public String getCustomerName() {

return customerName;

}

public void setCustomerName(String customerName) {

this.customerName = customerName;

}

* Suppose, we have a private variable, boolean active;
* The getter and setter methods can be created as,

public boolean isActive() {

return active;

}

public void setActive(boolean active) {

this.active = active;

}

static keyword:

* static is a modifier in java that could be for creating variables, methods, blocks and nested classes.
* If any member is declared as static then it belongs to the class itself, not belongs to any specific instance of the class.

static variable:

. when you want to share a data among all instances of the class then you should declare that variable as a static variable.

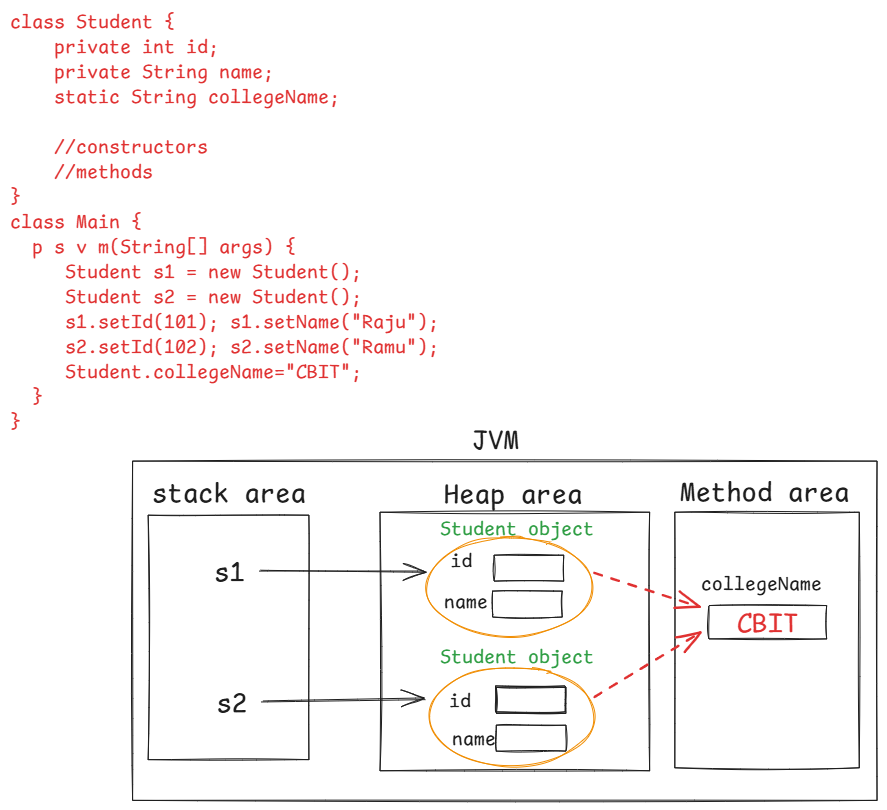
. For example, we have Student class, where the collegeName is common data to be shared across all instances(students). So we can declare the collegeName field as a static variable.

. For a static variable, the memory is allocated for only once in the method area of JVM and it is shared for all objects of the class.

. The memory for a static variable is allocated when the class is loaded into JVM by the Java Class Loader.

. static variables can be accessed directly using the class name.

(ClassName.variableName)



/\*

\* This code counts the number of objects created for a class using

\* static variable.

\*/

**class** ClassA {

**static** **int** *counter*;

**public** ClassA() {

*counter*++;

}

**public** **static** **int** getCounter() {

**return** *counter*;

}

}

**public** **class** CounterExample {

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

ClassA obj = **new** ClassA();

ClassA anotherObj = **new** ClassA();

ClassA someOtherObj = **new** ClassA();

System.***out***.println("The count of objects as of now : " + ClassA.*getCounter*());

ClassA ca1 = **new** ClassA();

ClassA ca2 = **new** ClassA();

ClassA ca3 = **new** ClassA();

ClassA ca4 = **new** ClassA();

System.***out***.println("The count of objects as of now : "+ ClassA.*getCounter*());

}

}

Q) what is the difference between static and instance variables?

A)

Q) what are the different types of variables?

A) 3 types.

1. local variable

2. instance variable

3. static variable

\* The variables which are created inside a method or a block or method parameters are the local variables.

\* The local variables are allocated in the statck area, whenever the method is called. After exection of the method, the local variable is deleted from the stack area.

\* If a local variable is created inside a block, then the variable is deleted from the statck area, when that block execution is completed.

\* The local variables does not contain default value. So, you must initialize the local variable before use.

\* The local variables are created in stack area, instance variables are created in heap area and static variables are created in method area of JVM.

static method:

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

* When you want to define a task in a class and that task should be performed on the given input, but not on the state of an object, then you have to create a static method in a class.
* In a class, we can create both non-static and static methods.
* non-static methods are the instance methods and they perform a task based on the state of an object.
* A static method can be called by using classname. (classname.methodname() )
* If you call a static method with objectname, at runtime JVM will replace objectname with classname only. So, you won’t get any compile-time error.
* static methods will allow to use only static variables of a class. Instance variables of a class are not allowed.
* “this” keyword of Java is also not allowed to use in the static method.

Q) what is the output of the below code?

class A {

static void sayHello() {

S.o.println(“hello”);

}

}

class Main {

p s v m(String[] args) {

A a = null;

a.sayHello();

}

}

A) hello

static block:

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

* if we have a static variable in a class, then we can initialize that static variable either in a constructor or in a static block.
* if we initialize a static variable in the constructor then for each object created for that class, static variable will be re-initialized.
* To avoid this re-initialization, we can create a static block.
* static block is executed for only once when the class is loaded into JVM, by the Java Class Loader. So, static variable will be initialized for only once.
* In a class we can able to create multiple static blocks also.
* If multiple static blocks are created, then they are executed in the order of declaration.

Example:

class ClassA {

private int x;

private static int y;

static {

y = 20;

}

A() {

x = 10;

}

}

Q) which one is first executed, static variable or static block?

A) static variable.

Q)what is the difference between constructor and static block?

A) 1. constructor is executed for each object of a class, but static block is executed for only once, when the class is loaded into the JVM.

2. In a constructor, we can initialize both instance and static variables. But in a static block, we can initialize only static variables.

3. A constructor is executed only when the object is created. But a static block can be executed, by without creating the object also.

Hint: Class.forName(“classname”); statement is used to load a class into

the JVM, without creating an object.

Inner class:

* If you want to create two classes, which are closely related classes.
* For example, you want to create two classes Mobile class and Sim class. Both are closely related classes.
* When two classes are closely related, instead of creating the classes separately, you can create one class as inner class of another.
* For example, you can create Sim class as inner class of Mobile class.
* By creating inner classes, we can make the code more readable and we can get better encapsulation.

class Mobile { //outer class

variables;

methods;

class Sim { //inner class

variables;

methods;

}

}

* inner classes are 4 types.

1. non-static inner class
2. static inner class/nested class
3. local inner class
4. anonymous inner class

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

//sample code(non-static inner class)

**class** ATM //outer class

{

**private** String bankName;

**private** **int** enteredPin;

**public** ATM(String bankName, **int** enteredPin) {

**this**.bankName = bankName;

**this**.enteredPin = enteredPin;

}

**private** **class** PINValidator //non-static inner class

{

**private** **int** actualPin=1234;

**boolean** validate() {

**if**(enteredPin == actualPin)

**return** **true**;

**else**

**return** **false**;

}

} //end of inner class

**public** **void** authenticate() {

PINValidator pinValidator = **new** PINValidator();

**if**(pinValidator.validate()) {

System.***out***.println("Access is granted to : " + bankName);

}

**else** {

System.***out***.println("Access is denied to : " + bankName);

}

}

}

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

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

ATM atm = **new** ATM("Axis Bank", 1234);

atm.authenticate();

}

}

//sample code(static inner class)

**class** OuterClass { //outer class

**static** **int** *x* = 10;

**int** y = 20;

**void** outerMethod() {

System.***out***.println("OuterClass :: outerMethod");

}

**static** **class** InnerClass { //static inner class

**void** innerMethod() {

System.***out***.println("InnerClass :: innerMethod");

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

}

}

}

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

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

OuterClass.InnerClass ic = **new** OuterClass.InnerClass();

ic.innerMethod();

}

}

//sample code(local inner class)

**class** OuterClass { //outer class

**void** outerMethod() {

**class** InnerClass { //local inner class

**void** innerMethod() {

System.***out***.println("InnerClass :: innerMethod");

}

}

InnerClass ic = **new** InnerClass();

ic.innerMethod();

}

}

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

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

OuterClass oc = **new** OuterClass();

oc.outerMethod();

}

}

INHERITANCE:

* Inhertiance is a core principle of OOP, and this practice/principle is widely used in designing and implementing the software.
* When you are designing a group of classes, if any common attributes or functionalities are identified, then you have to move them to another class, and then you have to create the group of classes by inherting from that class.
* This process is called Inheritance.
* By implementing inheritance principle, we can avoid code redundancy(code duplication) and we can get code reusability.
* In inheritance, we can find parent and child classes.
* The class, whose attributes/functionalities are inherited into the other classes, is called parent class or super class or base class.
* The class, who is inherting the attributes/functionalities of another class, is called child class or sub class or derived class.
* Java has provided “extends” keyword, for creating a child class from a parent class.

class ChildClass extends ParentClass

{

}

Types of inheritance:

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

1. single inheritance
2. multi-level inheritance
3. multiple inheritance
4. hierarchical inheritance
5. hybrid inheritance

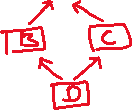
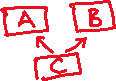
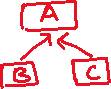
Single: creating a single child class for a Parent class

multi-level: creating a child class for a parent class and again creating a child class for a child class.

multiple: creating a child class from more than one parent class

hierarchical: creating multiple child classes for a parent class.

hybrid: combination hierarchical and multiple.



* In inheritance, child class inherits the functionalities of the parent class and also the child class can add the additional functionalities. So, we create an object for the child class.

example:

**class** ParentClass {

**public** **void** parentMethod() {

System.***out***.println("ParentClass :: parentMethod");

}

}

**class** ChildClass **extends** ParentClass {

**public** **void** childMethod() {

System.***out***.println("ChildClass :: childMethod");

}

}

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

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

ChildClass childObject = **new** ChildClass();

childObject.parentMethod();

childObject.childMethod();

}

}

output:

ParentClass :: parentMethod

ChildClass :: childMethod

example:

* If you create parent class object then you can only call the parent class methods. You can’t call the child class methods.
* **class** ParentClass {
* **public** **void** parentMethod() {
* System.***out***.println("ParentClass :: parentMethod");
* }
* }
* **class** ChildClass **extends** ParentClass {
* **public** **void** childMethod() {
* System.***out***.println("ChildClass :: childMethod");
* }
* }
* **public** **class** Solution {
* **public** **static** **void** main(String[] args) {
* ParentClass parentObject = **new** ParentClass();
* parentObject.parentMethod();
* parentObject.childMethod(); // error
* }
* }

example:

* we can store child class object in the parent class reference variable.
* we can’t store parent class object in the child class reference variable.
* With parent class reference variable, you can call only the parent class methods.
* **class** ParentClass {
* **public** **void** parentMethod() {
* System.***out***.println("ParentClass :: parentMethod");
* }
* }
* **class** ChildClass **extends** ParentClass {
* **public** **void** childMethod() {
* System.***out***.println("ChildClass :: childMethod");
* }
* }
* **public** **class** Solution {
* **public** **static** **void** main(String[] args) {
* ParentClass parentRef = **new** ChildClass();
* ChildClass childRef = **new** ParentClass(); //error
* parentRef.parentMethod();
* parentRef.childMethod(); //error
* }
* }

constructors execution in inheritance:

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

* Every constructor of a class in Java has super() call as a first statement, by default.
* The super() call calls the parameter less constructor of the parent class.
* When you create a child class object, first the control goes to the child class constructor and because super() call is the first statement, the control goes to the parent class constructor and executes it.
* After that the statements of the child class constructor are executed.

example:

**class** ParentClass {



**public** ParentClass() {

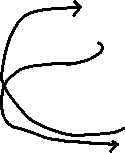


System.***out***.println("ParentClass :: parameter-less constructor");



}

}



**class** ChildClass **extends** ParentClass {



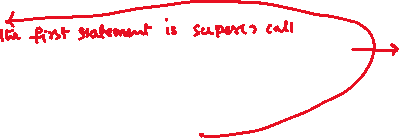
**public** ChildClass() {



System.***out***.println("ChildClass :: parameter-less constructor");



}



}

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

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

ChildClass childObject = **new** ChildClass();

}

}

output:

ParentClass :: parameter-less constructor

ChildClass :: parameter-less constructor

example2:

--------

* If you want to call a parent class constructor explicitly, from the child class constructor then you have to write the super() call statement as a first statement.
* If you want to call parameterized constructor of the parent class then you have to pass parameters to the super() call staement.

**class** ParentClass {

**public** ParentClass() {

System.***out***.println("ParentClass :: parameter-less constructor");

}

**public** ParentClass(**int** x, **int** y) {

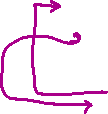


System.***out***.println("ParentClass :: parameterized constructor");



}

}



**class** ChildClass **extends** ParentClass {

**public** ChildClass() {

**super**(10, 20);



System.***out***.println("ChildClass :: parameter-less constructor");

}



}

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

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

ChildClass childObject = **new** ChildClass();

}

}

output:

ParentClass :: parameterized constructor

ChildClass :: paremeter-less constructor

example3:

* In Java, we have this() call and super() call statements.
* this() call calls the same class constructor and super() call calls the parent class constructor.
* We can’t write both this() call and super() call at a time in a constructor.
* this() call or super() call must be the first statement only in a constructor.

**class** ParentClass {

**public** ParentClass(**int** x, **int** y) {

System.***out***.println("ParentClass :: parameterized constructor");



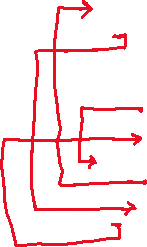
}

}

**class** ChildClass **extends** ParentClass {

**public** ChildClass() {

**this**(10, 20);

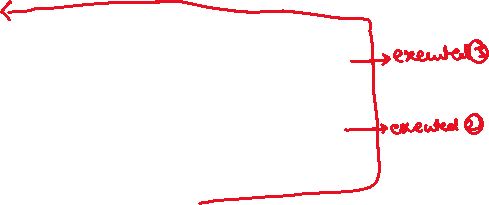


System.***out***.println("ChildClass :: parameter-less constructor");

}

**public** ChildClass(**int** a, **int** b) {

**super**(a, b);



System.***out***.println("ChildClass :: parameterized constructor");

}

}

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

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

ChildClass childObject = **new** ChildClass();

}

}

output:

ParentClass :: parameterized constructor

ChildClass :: parameterized constructor

ChildClass :: parameter-less constructor

Q) what is the difference between super keyword and super() call?

A) 1. super keyword can be used in a constructor or in a method.

super() call can be used only in a constructor.

2. super keyword can be used at any line in a constructor or a method.

super() call can be used as a first line only in a constructor.

3. If child class has same variable/method of parent class then you can call the parent class variable/method with super keyword.

super() call can be used only for calling a constructor, not variables/methods.

Q) When a child class object is created, is parent class object is also created internally?

A) No. But in the child class object only, the memory is also allocated for the parent class variables.

Q) can we call one non-static method in another non-static method of the same class directly?

A) Yes.

Q) can we call a static method in another static method of the same class directly?

A) Yes.

Q) can we call a static method in a non-static method of the same class directly?

A) Yes.

Q) can we call a non-static method in a static method of the same class directly?

A) No. By creating an object only, we can call.

Q) will static methods of a parent class are inherited to the child class?

A) Yes.

Q) can we use super/this keyword in a static method?

A) No.

Q) why protected access modifier is recommended for the variables or methods of parent class?

A) If they are protected, then they are visible to the child classes within the same package and also to the child classes in other packages.

Q) why multiple inheritance is not supported at classes-level in Java?

A) When a child class object is created, the child class constructor should call parent class constructor with super() call. If two parent classes are there, then JVM gets an ambiguity problem that which one should be called.

The same ambiguity problem also occurs for the methods. So, at classes-level, multiple inheritance is not supported.

Note: hybrid inheritance is also not supported at classes-level, because of the same ambiguity problem.

//Parent class

//Payment.java

**import** java.time.LocalDateTime;

**import** java.util.Random;

**public** **class** Payment {

**private** **long** transactionId;

**private** LocalDateTime transactionDate;

**private** **double** amount;

**public** Payment(LocalDateTime transactionDate, **double** amount) {

**super**();

**this**.transactionDate = transactionDate;

**this**.amount = amount;

}

**public** **long** getTransactionId() {

generateTransactionId();

**return** transactionId;

}

**public** LocalDateTime getTransactionDate() {

**return** transactionDate;

}

**public** **double** getAmount() {

**return** amount;

}

**private** **void** generateTransactionId() {

Random random = **new** Random();

transactionId = random.nextLong(28967541987231890L);

}

}

//CardPayment.java

//child class1

**import** java.time.LocalDateTime;

**public** **class** CardPayment **extends** Payment {

**private** String cardType;

**private** **long** cardNumber;

**public** CardPayment(LocalDateTime transactionDate, **double** amount, String cardType, **long** cardNumber) {

**super**(transactionDate, amount);

**this**.cardType = cardType;

**this**.cardNumber = cardNumber;

}

**public** String getCardType() {

**return** cardType;

}

**public** **void** setCardType(String cardType) {

**this**.cardType = cardType;

}

**public** **long** getCardNumber() {

**return** cardNumber;

}

**public** **void** setCardNumber(**long** cardNumber) {

**this**.cardNumber = cardNumber;

}

**public** **void** showCardPaymentDetails() {

System.***out***.println("Transaction id : " + getTransactionId());

System.***out***.println("Transaction date : " + getTransactionDate());

System.***out***.println("Amount : " + getAmount());

System.***out***.println("Card Type : " + getCardType());

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

}

}

//ChequePayment.java

**import** java.time.LocalDateTime;

**public** **class** ChequePayment **extends** Payment {

**private** String chequeType;

**private** **long** chequeNumber;

**public** ChequePayment(LocalDateTime transactionDate, **double** amount, String chequeType, **long** chequeNumber) {

**super**(transactionDate, amount);

**this**.chequeType = chequeType;

**this**.chequeNumber = chequeNumber;

}

**public** String getChequeType() {

**return** chequeType;

}

**public** **void** setChequeType(String chequeType) {

**this**.chequeType = chequeType;

}

**public** **long** getChequeNumber() {

**return** chequeNumber;

}

**public** **void** setChequeNumber(**long** chequeNumber) {

**this**.chequeNumber = chequeNumber;

}

**public** **void** showChequePaymentDetails() {

System.***out***.println("Transaction id : " + getTransactionId());

System.***out***.println("Transaction date : " + getTransactionDate());

System.***out***.println("Amount : " + getAmount());

System.***out***.println("Cheque Type : " + getChequeType());

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

}

}

//Solution.java

**import** java.time.LocalDateTime;

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

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

CardPayment cardPayment = **new** CardPayment(LocalDateTime.*now*(), 14895.0, "Credit Card", 432172192034L);

ChequePayment chequePayment = **new** ChequePayment(LocalDateTime.*now*(), 79245.0, "A/C Pay", 20080018L);

System.***out***.println("Card Payment Data : ");

cardPayment.showCardPaymentDetails();

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

System.***out***.println("Cheque Payment Data : ");

chequePayment.showChequePaymentDetails();

}

}

Association:

-----------

* In OOP, Inheritance and Association referss to the relationship between two or more classes/objects.
* Inheritance refers “IS-A” relationship, and Association refers “HAS-A” relationship.
* In inheritance, we use extends keyword and it represents

“is-a” relationship.

* For example,

CardPayment extends Payment, it means

CardPayment is-a Payment.

* For example,

HealthInsurance extends Insurance, it means

HealthInsurance is-a Insurance

* If you create one class object in another class, then it is called HAS-A relationship.
* Suppose, we have two classes called UserLogin and PasswordEncoder.
* If you are creating PasswordEncoder class object in the UserLogin class, then it becomes HAS-A relationship.
* So, when you create UserLogin class object, it contains PasswordEncoder class object.
* So, we say that UserLogin object is container object and PasswordEncoder object is contained object.
* HAS-A relationship is of two types.

1. Aggregation (weak HAS-A)
2. Composition (Strong HAS-A)

* Aggregation is HAS-A relationship, where the contained object can exist independently without container object.
* For example, a Teacher has many Students, but Students can exist without a Teacher.(Aggregation)
* Composition is HAS-A relationship, where the contained object cannot exist without container object.
* For example, a College has many Departments, but Departments cannot exist with a College. (Composition).

Polymorphism

* Polymorphism is one of the core principles of Object Oriented Programming.
* It allows an object to execute same action for a different type of data and also allows an object to perform an action in a different way for the same input.
* There are 2 types of polymorphism.

1. compile-time polymorphism
2. runtime polymorphism

* compile-time polymorphism means, it allows an object to execute same action for a different data. Method overloading is a way to implement compile-time polymorphism.
* runtime polymorphism means, it allows an object to execute the same action for the same input, but in a different form. Method overriding is a way to implement runtime polymorphism.

Method overloading:

\* It means, creating the same method in a class for more than one time, but with a difference in parameters.

\* The difference should be,

in the number of parameters, or

in the datatype of parameters, or

in the sequence of parameters.

ex1:

class UserLogin {

void signin(String email, String password)

{

//logic

}

boolean signin(long mobile, int otp)

{

//logic

}

}

\* Here, both the methods have the same name, but have a difference in datatype of the parameters. So, it is method overloading.

\* The overloaded methods may have the same or different return types.

ex2:

class RechargeService {

void recharge(long mobile, double amount)

{

//logic

}

void recharge(long mobile, double amount, String promo)

{

//logic

}

}

ex3:

class Demo {

static void m1(int x) {

//logic

}

static void m1(int x, int y) {

//logic

}

}

\* we can overload static methods.

ex4:

class BankOperations {

void verifyAddress(AdhaarCard card) {

//logic

}

void verifyAddress(VoterId id) {

//logic

}

}

Method overriding:

\* Suppose a method in parent class is not matching with the requirements of the child class then child class can change the behavior, by overriding that method.

\* To override a method, inheritance is required.

\* In method overriding, child class method should have the same name, same datatype and same number of parameters of the parent class method. The logic in child class method is different from the logic in the parent class.

\* In method overrding, the overridden method should have the same return type of should have a co-variant type.

\* The access level of the overridden method should be of same level or high level than parent class method.

ex1:

class ClassA {

void m1() {

//logic

}

}

class ClassB extends ClassA {

@Override

public void m1(){ //public is high-level than default, it’s ok

//logic

}

}

ex2:

class ClassA {

public void m1(int x) {

//logic

}

}

class ClassB extends ClassA {

@Override

void m1(int k) { //default is low-level than public, so error.

//logic

}

}

ex3:

class ClassA {

public void m1() {

//logic

}

}

class ClassB extends ClassA {

@Override

public boolean m1() { //return type is different, so error.

//logic

}

}

ex4:

\* Suppose, we have a class Employee with child classes FulltimeEmployee and ParttimeEmployee.

class EmployeeOperations {

public Employee getEmployee(int empid) {

//logic

}

}

class FullTimeEmployeeOperations extends EmployeeOperations {

@Override

public FulltimeEmployee getEmployee(int empid) {

//logic

}

}

\* Here, FulltimeEmployee is a child class of Employee. So, it is a co-variant type of Employee.

\* The overridden method has return type as co-variant type. So, it is allowed.

\* The co-varaint type works for return types, but not for parameter types.

ex:

class EmployeeOperations {

void update(Employee e) {

//logic

}

}

class FulltimeEmployeeOperations extends EmployeeOperations {

@Override

void update(FulltimeEmployee e) { //error

//logic

}

}

Q) what is compile-time polymorphism?

A) The bindings between the method call point to the overloaded method is created at the compile-time. So, it is called compile-time polymorphism.

Q) what is runtime polymorphism?

A) The bindings between the method call point to the overridden method is created at the runtime. So, it is called runtime polymorphism.

Q) what is the difference between overloading and overriding?

A) 1. In overloading, method name should be same but in parameters should have a difference. But in overriding, method name and parameters should be same.

2. In overloading, methods return type can be any thing. But in overriding, methods return type should be same type or co-variant type.

3. Within the same class, we can overload the method. But we can’t override the method.

4. We can overload the constructor, but we can’t override.

5. private and static methods, we can overload. But we can’t override.

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

final keyword in Java:

* final is an important non-access modifier in Java, that can be used with variables, methods and classes, with a distinct purpose.

final variables:

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

* final variables are used to create constants in a program.
* For a final variable, we can assign a value for only once. we can’t reassign the value.
* final variables are 3 types.

1. instance final variables
2. static final variables
3. local final variables

instance final variable:

\* instance final variable is an object-level constant.

\* It means, for an object, the value assigned to the final variable is fixed. We can’t modify the value of that final variable with in that object scope.

\* You must assign the value to a final instance variable either at the variable declaration or in the constructor.

\* For a final variable, you can define a getter method, but you can’t define a setter method.

ex:

**import** java.util.Random;

**class** BankAccount {

**private** **final** **long** accountId;

**private** String accountHolderName;

**public** BankAccount() {

accountId=**new** Random().nextLong(376234198);

accountHolderName="john";

}

**public** **long** getAccountId() {

**return** accountId;

}

**public** **void** setAccountId(**long** accountId) {

**this**.accountId = accountId; //error

}

}

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

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

BankAccount account1 = **new** BankAccount();

BankAccount account2 = **new** BankAccount();

System.***out***.println(account1.getAccountId());

System.***out***.println(account2.getAccountId());

` }

}

static final variable:

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

* static final variable is a class-level constant.
* a static final variable can be assigned with a value either at the variable declaration or in a static block. But not allowed in the constructor.
* A static final variable is a constant for all objects of the class.

ex:

**class** BankAccount {

**private** **final** **long** accountId;

**private** String accountHolderName;

**private** **static** **final** String ***ifsc***;

**static** {

***ifsc*** = "SBIN0001234";

}

**public** BankAccount() {

accountId=**new** Random().nextLong(376234198);

accountHolderName="john";

***ifsc*** = "SBIN0001234"; //error

}

**public** **long** getAccountId() {

**return** accountId;

}

**public** **void** setAccountId(**long** accountId) {

**this**.accountId = accountId; //error

}

}

local final variable:

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

* local final variable can be created with in a method or in a block.
* A local final variable can be assigned with a value for only once and it can’t be reassigned again in that method/block.
* The only modifier that could be used in a method is final.

ex:

**class** Student {

**private** **final** **int** studentId;

**public** Student(**int** studentId) {

**this**.studentId = studentId;

}

**public** **void** calculatePercentage(**int** totalMarks) {

**final** **int** maxMarks = 600;

**double** marksPercentage = ( totalMarks / (**double**) maxMarks) \* 100;

System.***out***.println("percentage = " + marksPercentage + "%");

}

}

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

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

Student student = **new** Student(209981);

student.calculatePercentage(579);

}

}

Note: If you create a class level constant(static final), then according to Java naming conventions, the variable name should be written in upper case.

ex:

public static final int MAX\_USERS = 100;

final method:

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

* If you want to prevent a method from overriding by the child classes then you have to declare that method as a final method.
* a final method in a class indicates that the functionality is fixed/constant and the child classes can’t alter the functionality.
* If a method is private then it is not visible to the child classes. So, there is no use to make a private method as a final method.
* If you make a private method as final then you will not get any compile-time error. But there is no use.

ex:

**class** Student {

**private** **final** **int** studentId;

**public** Student(**int** studentId) {

**this**.studentId = studentId;

}

**public** **final** **void** calculatePercentage(**int** totalMarks) {

**final** **int** maxMarks = 600;

**double** marksPercentage = ( totalMarks / (**double**) maxMarks) \* 100;

System.***out***.println("percentage = " + marksPercentage + "%");

}

}

**class** DayScholar **extends** Student {

**public** DayScholar(**int** studentId) {

**super**(studentId);

}

@Override

**public** **void** calculatePercentage(**int** totalMarks) { //error

//logic

}

}

Q) can we create static final methods?

A) Yes, but anyhave static methods can’t be overridden. So, there is no use of making a static method as final.

Q) can we overload a final method?

A) Yes.

Q) which methods we can’t override?

A) 1. private methods

2. static methods

3. final methods

final class:

-----------

* If a class is declared as final then you can’t create child classes for that class.
* If a class is final, other classes can use this final class by creating an object, but they can’t extend this class.
* final variables can’t be re-assigned.
* final methods can’t be overridden.
* final classes can’t be extended.
* For example, java.lang.String is a final class, so we can’t create a child class for this String class.

Q) can we create a final object?

A) Yes. It means, we can’t re-assign another object.

example:

class A {

...

}

class Main {

public static void main(String[] args) {

final A a = new A(); //correct

a = new A(); //error

}

}

Q) can we create a final constructor?

A) No.

Q) Can we create a static constructor?

A) No.

Note: We can’t use any other modifier except public/private/protected with a constructor.

abstract classes

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

* When you create a parent class and one or more child classes then we call them as a family of classes.
* A parent class will share the common fields and common functionality with the child classes.
* When a parent class wants to share some common functionalities to the child classes and also if it wants to tell the child classes to provide their own implementation for some functionalities then we have to use abstract class.
* For example, we have Shape as a parent class with Circle and Rectangle as child classes. Here, Shape class wants to share the fillColor functionality as common and wants to tell the child classes to provide their own implementation for findArea functionality. So, in this case, we have to make the Shape class as abstract class.
* To tell the child classes that they must provide implementation for a method, we declare that method as abstract method in the parent class.
* In abstract class, we can create concrete methods and abstract methods.
* A concrete method means, which is having header and body in the class.
* An abstract method means, which is having only the header without the body in the class.

ex:

abstract class Shape {

private String color;

public void fillColor() { //concrete method

//logic

}

public abstract void findArea(); // abstract method

}

* If a class has atleast one abstract method, then we must declare that class as abstract class.

class Circle extends Shape {

@Override

public void findArea() {

//logic

}

}

class Rectangle extends Shape {

@Override

public void findArea() {

//logic

}

}

* If a child class is extending the abstract parent class, then that child class has to override all the abstract methods of the parent class. Otherwise, that child class will get a compile-time error.
* If a child class don’t wants to override all the abstract methods of the parent class, then that child class must be declared as abstract class.

ex1:

abstract class A {

void m1() {

}

abstract void m2();

abstract void m3();

}

class B extends A { //compile-time error

@Override

void m2() {

}

}

ex2:

abstract class A {

void m1() {

}

abstract void m2();

abstract void m3();

}

abstract class B extends A { //correct

@Override

void m2() {

}

}

Q) can we create abstract variables?

A) No

Q) can we create abstract constructors?

A) No

Q) can we create abstract blocks?

A) No

Q) can we create abstract methods?

A) Yes.

Q) can we create abstract classes?

A) Yes.

Q) can we create private abstract methods?

A) No.

Q) can we create static abstract methods?

A) No.

Q) can we create final abstract methods?

A) No.

Q) can we create final abstract classes?

A) No

Q) can we create an object for an abstract class?

A) No.

Q) can we create a constructor in the abstract class?

A) Yes. We can initialize the instance variables of the abstract class in the constructor.

Q) can we make a class as abstract, without writing abstract methods?

A) Yes.

Q) When do you make a class as abstract, without writing abstract methods?

A) Reason1: When you don’t want anybody to create an object for your class, you can declare the class as abstract.

Reason2: If the methods in the class are implemented with partial logics, then you can declare the class as abstract.

//InsurancePolicy.java

**public** **abstract** **class** InsurancePolicy {

**private** **double** basePremium;

**public** InsurancePolicy(**double** basePremium) {

**super**();

**this**.basePremium = basePremium;

}

**public** **double** getBasePremium() {

**return** basePremium;

}

**public** **void** setBasePremium(**double** basePremium) {

**this**.basePremium = basePremium;

}

**public** **abstract** **double** calculateRiskFactor();

**public** **double** calculateFinalAmount() {

**double** riskFactor = calculateRiskFactor();

**return** basePremium + riskFactor;

}

}

//HealthPolicy.java

**public** **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;

**if**(age >= 60 && isSmoker == **true**) {

riskFactor = getBasePremium() \* 0.53;

}

**else** **if**( age < 60 && isSmoker == **true**) {

riskFactor = getBasePremium() \* 0.40;

}

**else** {

riskFactor = getBasePremium() \* 0.25;

}

**return** riskFactor;

}

}

//VehiclePolicy.java

**public** **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;

**if**(fuelType.equals("Diesel") && mfgYear <= 2019)

{

riskFactor = getBasePremium() \* 0.49;

}

**else** **if**(fuelType.equals("Diesel") && mfgYear > 2019)

{

riskFactor = getBasePremium() \* 0.27;

}

**else** {

riskFactor = getBasePremium() \* 0.19;

}

**return** riskFactor;

}

}

//Main.java

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

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

HealthPolicy hp = **new** HealthPolicy(32000.0, 62, **true**);

**double** finalAmount = hp.calculateFinalAmount();

System.***out***.println("Final amount for Health Policy : " + finalAmount);

VehiclePolicy vp = **new** VehiclePolicy(52000.0, "Diesel", 2017);

**double** finalAmount2 = vp.calculateFinalAmount();

System.***out***.println("Final amount for Vehicle Policy : " + finalAmount2);

}

}

Interfaces

* In an application, we want to create multiple classes, and we want that those classes must exhibit a similar behavior then we will create an interface in the application.
* By creating an interface, we are defining a contract(set of rules/methods) for the classes.
* The classes must implement the contract, but in their own way.
* To create an interface, the keyword is “interface”.
* With interfaces, we can implement abstraction principle.
* In an interface, we can create public abstract methods and public static final variables.
* If you don’t write public static final modifiers, but also the variables are by default public static final only.
* Similarly, if you don’t write public abstract modifiers, but also the methods are by default public abstract only.

example:

interface CustomerRepository

{

int MAX\_CUSTOMERS = 100;

void saveCustomer();

void fetchCustomer();

}

* A class has to implement an interface, to define the contract in its own way.

example:

class CustomerRepositoryImpl implements CustomerRepository

{

@Override

public void saveCustomer() {

//logic

}

@Override

public void fetchCustomer() {

//logic

}

}

* one class can also implement multiple interfaces. So, we can achieve the mulitple inheritance with interfaces.

example:

interface Crud { interface Sorting {

void create(); void sort();

void read(); }

void update();

void delete();

}

class MyClass implements Crud, Sorting {

@Override

public void create() {

//logic

}

@Override

public void read() {

//logic

}

@Override

public void update() {

//logic

}

@Override

public void delete() {

//logic

}

@Override

public void sort() {

//logic

}

}

* A class can implement an interface and can extend another class at a time. In this case, first we have to use extends keyword and after that we have to use implements keyword.

ex:

class ClassA { interface MyInter {

void m1() { void f1();

//logic }

}

}

class ClassB extends ClassA implements MyInter

@Override

public void f1() {

//logic

}

}

Q) can we create an object for an interface?

A) No. Because an interface is purely incomplete.

Q) can we have constructors in an interface?

A) No. The interfaces does not have the constructors.

Q) Can one interface extend another interface?

A) Yes.

ex:

public interface TestInter1 {

public abstract void m1();

}

public interface TestInter2 extends TestInter1 {

public abstract void m2();

}

Q) Can one interface extend multiple interfaces?

A) Yes.

ex:

public interface ServletContext {

public abstract void getServletContext();

}

public interface JspContext {

public abstract void getJspContext();

}

public interface MyApplicationContext extends ServletContext,

JspContext

{

public abstract void getMyApplicationContext();

}

Types of interfaces:

1. Normal interface
2. Marker interface
3. Functional interface(newly included in Java8)

* A Normal interface means, which contains zero or more public static final variables and more than one public abstract methods.
* A Marker interface means, which contains zero public static final variables and zero public abstract methods.
* A Maker interface is simply an empty interface, which does not contain anything.
* For example, Serializable, Cloneable and Remote are the pre-defined marker interfaces.
* The Marker interfaces are created, when you want to add some special behavior to the objects of a specific type, at runtime.
* To add the special behavior, a class of that objects should implement the marker interface.
* For example, the objects of a class that implements Serializable interface, will get serialization behavior from the JVM. Serialization behavior means, those objects can be transferred over a network.
* A Functional interface means, which contains zero or more public static final variables and a single abstract method.

ex1:

public interface PaymentProcessor {

void processPayment();

void refundPayment();

}

\* PaymentProcessor is a normal interface

ex2:

public interface Renumerator {

double calculateSalary();

}

\* Renumerator is a functional interface.

ex3:

public interface Component {

}

\* Component is a marker interface

Q) Can we create a reference variable for an interface?

A) Yes. We can assign the object of implementation class to that

reference variable.

ex:

interface Service {

void generate();

}

class PdfService implements Service {

@Override

public void generate() {

//logic

}

}

public class Main {

public static void main(String[] args) {

Service service = new PdfService();

service.generate();

}

}

Q) what is the difference between interface reference variable and a class reference variable?

A) With interface reference variable, we can only call the interface methods. But with class reference variable, we can call the interface methods and also the class methods.

//Example

//NotificationService.java

**public** **interface** NotificationService {

**void** sendNotification(String recipient, String message);

}

//EmailNotificationService.java

**public** **class** EmailNotificationService **implements** NotificationService {

@Override

**public** **void** sendNotification(String recipient, String message)

{

System.***out***.println("Email sent to : " + recipient + " with message : " + message);

}

}

//SMSNotificationService.java

**public** **class** SMSNotificationService **implements** NotificationService {

@Override

**public** **void** sendNotification(String recipient, String message)

{

System.***out***.println("SMS sent to : " + recipient + " with message : " + message);

}

}

//NotificationManager.java

**public** **class** NotificationManager {

**public** **static** **void** notifyUser(String recipient, String message, NotificationService service)

{

service.sendNotification(recipient, message);

}

}

//Main.java

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

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

NotificationManager.*notifyUser*("john@gmail.com", "Your order is shipped!", **new** EmailNotificationService());

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

NotificationManager.*notifyUser*("708-1234-123", "Your order is out for delivery!!", **new** SMSNotificationService());

}

}

Q) What is anonymous inner class?

A) Anonymous inner class means, combining class declaration and instantiation at one place, by without creating a class.

We create anonymous inner classes, when we want to implment an interface by without creating a class explicitly.

**interface** RowMapper {

**void** mapRow();

}

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

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

RowMapper mapper = **new** RowMapper() {

@Override

**public** **void** mapRow() {

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

}

};

mapper.mapRow();

}

}

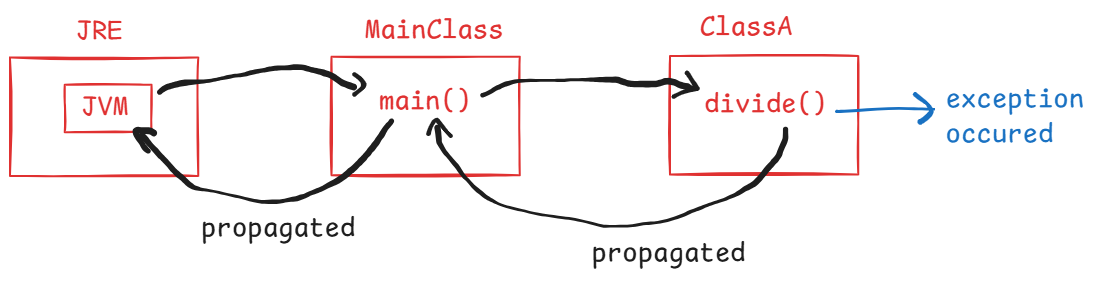
Exception Handling

* A program can have 3 types of errors.

1. compile-time errors
2. runtime errors
3. logical errors

* compile-time errors occurs, due to the mistake did in syntax.
* runtime errors occurs, due to the invalid data given by the user.
* logical errors occurs, due to the mistake did in the logic of the application.
* runtime errors are also called as exceptions.
* JVM can show the runtime errors.
* logical errors can be identified by conducting unit-testing.
* Suppose, if an exception occurs while executing an application, then the default behavior is, suddenly the applicatation gets terminated.
* If application gets suddenly terminated then that application is not user-friendly and also not a reliable application.
* So, while developing an application, we have to add exception handling mechanism to the application. So, that it can prevent sudden termination and can make the application to run smoothly.

The default behaviour of an application:





. Here, exception has occurred in the divide() method.

. The divide() method propagates the exception to the caller

method ( main() method).

. The main() method propagated to the JVM. Now the JVM

terminates the application and displays exception details.

Exception handling keywords:

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

1. try
2. catch
3. finally
4. throws
5. throw

* try block contains a group of statements from which the developer is suspecting that there is a chance of getting an exception.
* catch block contains a group of statements that are used to deal with the exception occurred.
* finally block contains a group of statements that must be executed, irrespective of whether an exception is occurred or not occurred.

//example code

**class** SomeClass {

**void** someMethod(**int** a, **int** b)

{

**int**[] arr = **new** **int**[5];

**try** {

**int** c = a / b;

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

arr[4] = 50;

}

**catch**(Exception ex) {

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

}

**finally** {

System.***out***.println("I am in finally...");

}

}

}

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

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

SomeClass someObject = **new** SomeClass();

someObject.someMethod(15, 0);

}

}

Q) can we write try and catch blocks without finally block?

A) Yes.



Q) can we write try and finally blocks without catch block?

A) Yes.

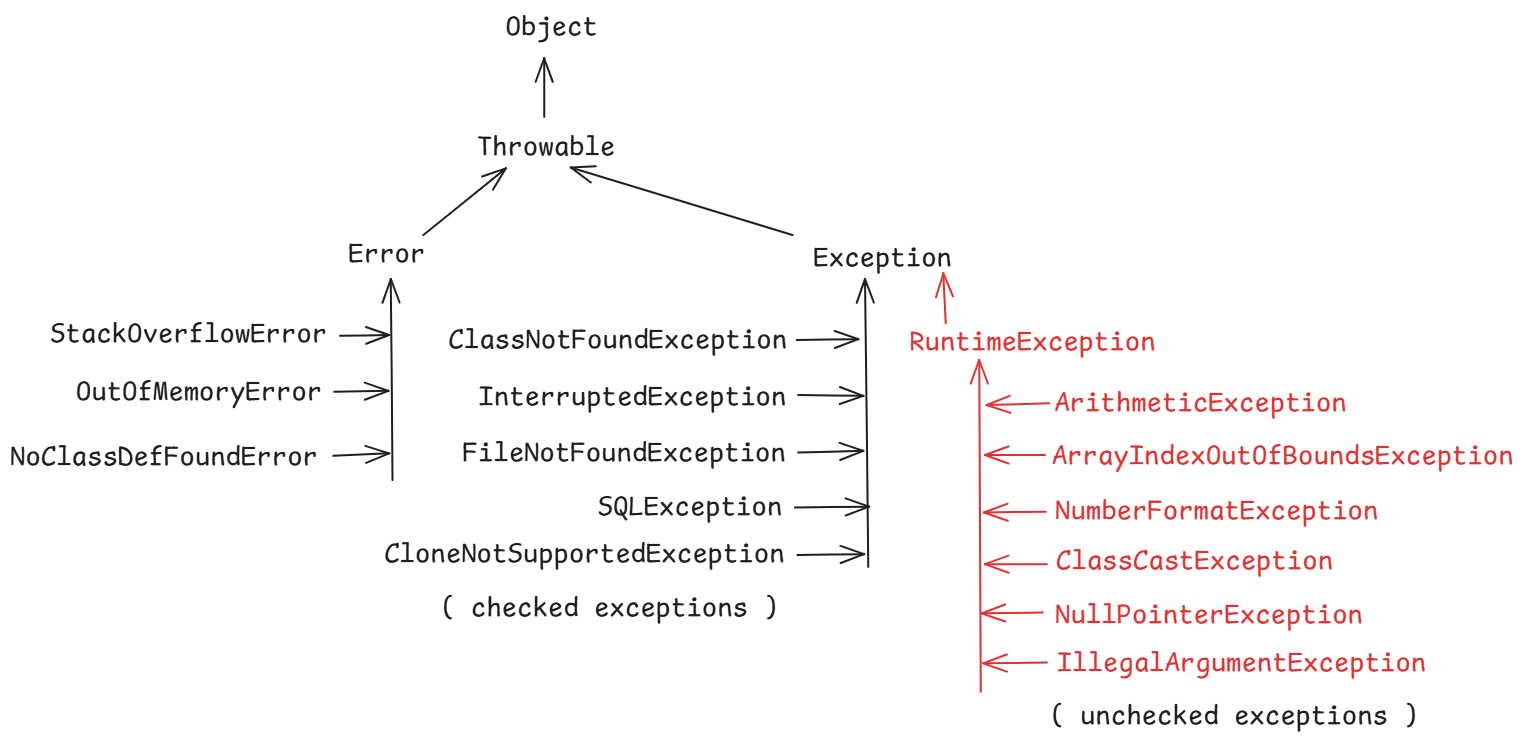


Q) can we write the blocks in try, finally then catch order?

A) No. We can write only in try, catch then finally order.

Exception class hierarchy:

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



Q) what is the difference between Error and Exception?

A) Both are the child classes of a Parent class called Throwable.

Errors represents serious problems in the system/application that

a program should not try to catch.

Exceptions represents the problems in the application that a

program can address that problem.

Q) How many types of exceptions are there in Java?

A) 2 types.

1. checked exceptions

2. unchecked exceptions

Q) what is the difference between checked and unchecked exceptions?

A) checked exceptions are known to the compiler and while compiling the code, it can foresee the exception that it might occur and verifies whether programmer is catching it or atleast propagating it with throws keyword. If not done anything, then generates a compile-time error.

unchecked exceptions are unknown to the compiler and while compiling the code, it doesn’t verify whether the programmer is catching it or atleast propagating it with throws keyword. So, the compiler doesn’t generate any compile-time error.

So, the Java compiler will force the programmer to catch or propagate the checked exception. But doesn’t force for the unchecked exception.

Q) can we write multiple catch blocks for a try block?

A) Yes. Writing catch blocks for specific exceptions followed by catch block for generic exception(Exception class) is valid. But, writing catch block for generic exception(Exception class) followed by catch blocks for specific exceptions is invalid.

example1:

**void** someMethod(**int** a, **int** b)

{

**int**[] arr = **new** **int**[5];

String str = **null**;

**try** {

**int** c = a / b;

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

arr[4] = 50;

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

}

**catch**(Exception ex) { //catch for generic exception

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

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

}

**catch**(ArithmeticException ex) { //compile-time error

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

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

}

**catch**(ArrayIndexOutOfBoundsException ex) { //error

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

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

}

**finally** {

System.***out***.println("I am in finally...");

}

}

* The above code has catch for generic exception followed catch for specific exceptions. So, the code has errors.

example2:

**void** someMethod(**int** a, **int** b)

{

**int**[] arr = **new** **int**[5];

String str = **null**;

**try** {

**int** c = a / b;

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

arr[4] = 50;

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

}

**catch**(ArithmeticException ex) { //specific catch

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

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

}

**catch**(ArrayIndexOutOfBoundsException ex) { //specific catch

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

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

}

**catch**(Exception ex) { //generic catch

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

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

}

**finally** {

System.***out***.println("I am in finally...");

}

}

* The above example has catch blocks for specific exceptions, followed catch for generic exceptions. So, it is valid.

Q) can we write multiple finally blocks for a try block?

A) No.

Q) can we write a try block in another try block?

A) Yes. It is called nested try blocks.

Q) If an exception occurs in the inner try block, who can handle it?

A) inner catch block can handle it. If it is unable to handle it then the control goes to the outer catch block.

Q) If an exception occurs in the outer try block, who can hande it?

A) outer catch block can handle it.

Q) When to write multiple catch blocks for a try block?

A) If you want to handle/deal the different exceptions in different ways, then you have to write multiple catch blocks.

Q) what are the different ways to display the exception details?

A) In 3 ways we can display.

1. by calling getMessage() method

2. print the excepton object

3. by calling printStackTrace()

. getMessage() method returns the exception message string.

. when you print the exception object, it will display

exception classname and exception message string.

. when you call printStackTrace() method, it will display stack trace of an exception. This stack trace contains 4 details.

1. exception classname 2. exception message string

3.methods call stack 4. location of the exception

example:

**void** someMethod(**int** a, **int** b)

{

**int**[] arr = **new** **int**[5];

String str = **null**;

**try** {

**int** c = a / b;

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

arr[4] = 50;

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

}

**catch**(Exception ex) {

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

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

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

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

ex.printStackTrace();

}

}

* If the catch block is executed then it will display the output like below.

/ by zero

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

java.lang.ArithmeticException: / by zero

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

java.lang.ArithmeticException: / by zero

at SomeClass.someMethod(Main.java:7)

at Main.main(Main.java:28)

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

User-defined exceptions:

* we can create an user-defined exception, by extending our class from either Exception or from RuntimeException class.
* If you want to make your exception as a checked exception then extend Exception class.
* If you want to make your exception as a unchecked exception then extend RuntimeException class.
* While creating an user-defined exception class, we don’t create any variables or methods in that class, because we want to use it as an exception class, but not as a business class.
* You can create a constructor with a String parameter in the exception class or you can make your class as an empty class.
* While throwing the exception, suppose you want to throw a message also along with the exception then in that exception class you have to create a constructor with String parameter.
* While throwing the exception, suppose you want to throw the exception without a message, then you can create that exception class as empty.

Example:

public class EmployeeNotFoundException extends RuntimeException

{

public EmployeeNotFoundException(String message)

{

super(message);

}

}

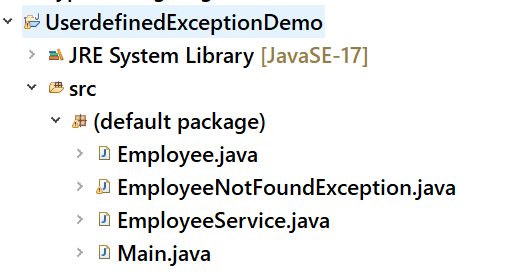
public class DuplicateEmployeeException extends RuntimeException

{

}

* The throw keyword is used to raise an user-defined exception from the try block.
* With throw keyword, you can throw the exception object.
* For pre-defined exceptions, JVM will throw the exception object.

//Example:



//Employee.java

**public** **class** Employee {

**private** **int** empno;

**private** String ename;

**public** Employee(**int** empno, String ename) {

**this**.ename = ename;

**this**.empno = empno;

}

**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;

}

}

//EmployeeNotFoundException.java

**public** **class** EmployeeNotFoundException **extends** RuntimeException {

**public** EmployeeNotFoundException(String message) {

**super**(message);

}

}

//EmployeeService.java

**public** **class** EmployeeService {

Employee[] employees;

**public** EmployeeService() {

employees = **new** Employee[3];

employees[0] = **new** Employee(101, "SMITH");

employees[1] = **new** Employee(102, "JOHN");

employees[2] = **new** Employee(103, "ALLEN");

}

**public** **void** search(**int** empno) {

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

**try** {

**for**(Employee e : employees) {

**if**(e.getEmpno() == empno) {

System.***out***.println("Employee found!");

System.***out***.println("Empno: " + e.getEmpno());

System.***out***.println("Ename: " + e.getEname());

flag = **true**;

**break**;

}

}

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

**throw** **new** EmployeeNotFoundException("Sorry, the given empno : " +empno + " doesn't exist!");

}

**catch**(EmployeeNotFoundException ex) {

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

}

}

}

//Main.java

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

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

EmployeeService service = **new** EmployeeService();

service.search(104);

}

}

throws keyword:

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

* When an exception occurs in a method, there are 2 places where it can be handled.

1. in the same method only by using try and catch blocks.
2. in the called method by using try and catch blocks.

* To delegate/propagate an exception from the current method to its caller method, throws keyword is used.
* If a method is not handling checked exceptions that might occur in that method, then it must propagate those exceptions to the caller method, by using throws keyword. Otherwise, Java compiler will generate an error.
* If a method is not handling unchecked exceptions that might occur in that method, then it is optional to propagate those exceptions to the caller method.
* throws keyword is used in the method signature to propage/delegate the exceptions.
* With throws keyword, we can declare multiple exceptions also.
* If a caller method is not handling the exceptions, then it can also propagate the exceptions to its caller.

//example to understand throws keyword.

**import** java.io.BufferedReader;

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.IOException;

**class** FileOperations

{

**void** readFile() **throws** FileNotFoundException, IOException

{

File file = **new** File("D:/names1.txt");

FileReader fr = **new** FileReader(file);

BufferedReader br = **new** BufferedReader(fr);

String str;

**while**((str = br.readLine()) != **null**)

{

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

}

fr.close();

}

}

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

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

FileOperations op = **new** FileOperations();

**try** {

op.readFile();

}

**catch**(Exception ex) {

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

}

}

}

. In this example, readFile() method is propagating the exceptions to its caller method called main() method. The main() method is handling the exceptions, by using try and catch blocks.

Q) what is the difference between throw and throws ?

A) 1. throw keyword is used inside the method body, but throws keyword is used in the method header.

2. With throw keyword, we can declare only one exception object. But with throws keyword, we can declare multiple exception classes.

3. With throw keyword, we can throw an exception and along with a message also. But with throws keyword, we can propagate only an exception, but not any message.

Q) can we handle multiple exceptions in a single catch block?

A) Yes, by creating multi exception catch block.

ex:

try {

//statements

}

catch(ArithmeticException | ArrayIndexOutOfBoundsException ex)

{

//statements

}

catch(NullPointerException ex)

{

//statements

}

Q) if we write return statement in the try block, will the finally block executes or not?

A) Yes.

Q) if we write exit statement in the try block, will the finally block executes or not?

A) No.

try with resources:

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

* try with resources means, we can pass resources/objects as parameters to the try statement.
* The resources/objects of type Closeable or AutoCloseable are only allowed as parameters.
* In Java, the objects like Database connections, sockets, Input streams and Output streams, etc.. are the AutoCloseable objects.
* Suppose, if you have opened a Database connection in the try block then you have to write a finally block to close the Database connection.
* If you are not closing the resources like connections then the application may get memory leak problem.
* To avoid writing the finally block explicitly for closing such resources, we got try with resources.
* In this try with resources, try statement only will automatically close the resources, after executing the try block, in both the cases when exception occurs or doesn’t occur.
* So, it avoids writing the finally block and also manually closing the resources.

ex:

try( Connection conn = DriverManager.getConnection(url,uname,pwd),

Statement stmt = conn.createStatement() )

{

//statements

}

catch(Exception ex)

{

//statements

}

What are generic classes?

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

* Suppose, I have a requirement to create classes to perform similar operations like add and get, but for different object types like Student, Book and Teacher.
* So, for this requirement I have created 3 classes.

class StudentRecord class BookRecord class TeacherRecord

{ { {

void add(Student s) void add(Book b) void add(Teacher t)

{ { {

} } }

Student get(int id) Book get(int id) Teacher get(int id)

{ { {

} } }

} } }

* If object types are increased then, I have to increase the number of classes.
* So, creating multiple classes for similar operations will increase code redundency.
* A solution to this problem is, creating a generic class.
* A generic class is a class which is created with type parameters.
* The type parameters are specified with in angle brackets(<>).

For example:

class Record<T>

{

void add(T t)

{

}

T get(int id)

{

}

}

* The above generic class can be used for different object types, and we need to specify that object type at the runtime while creating the object of the generic class.

For example,

Record<Student> r1 = new Record<Student>();

Record<Book> r2 = new Record<Book>();

Record<Teacher> r3 = new Record<Teacher>();

* The wild card patterns used in generics are,

<? extends T> : matches to the type T or its subtypes.

<? super T> : matches to the type T or its super types.

<?>: matches to any type.

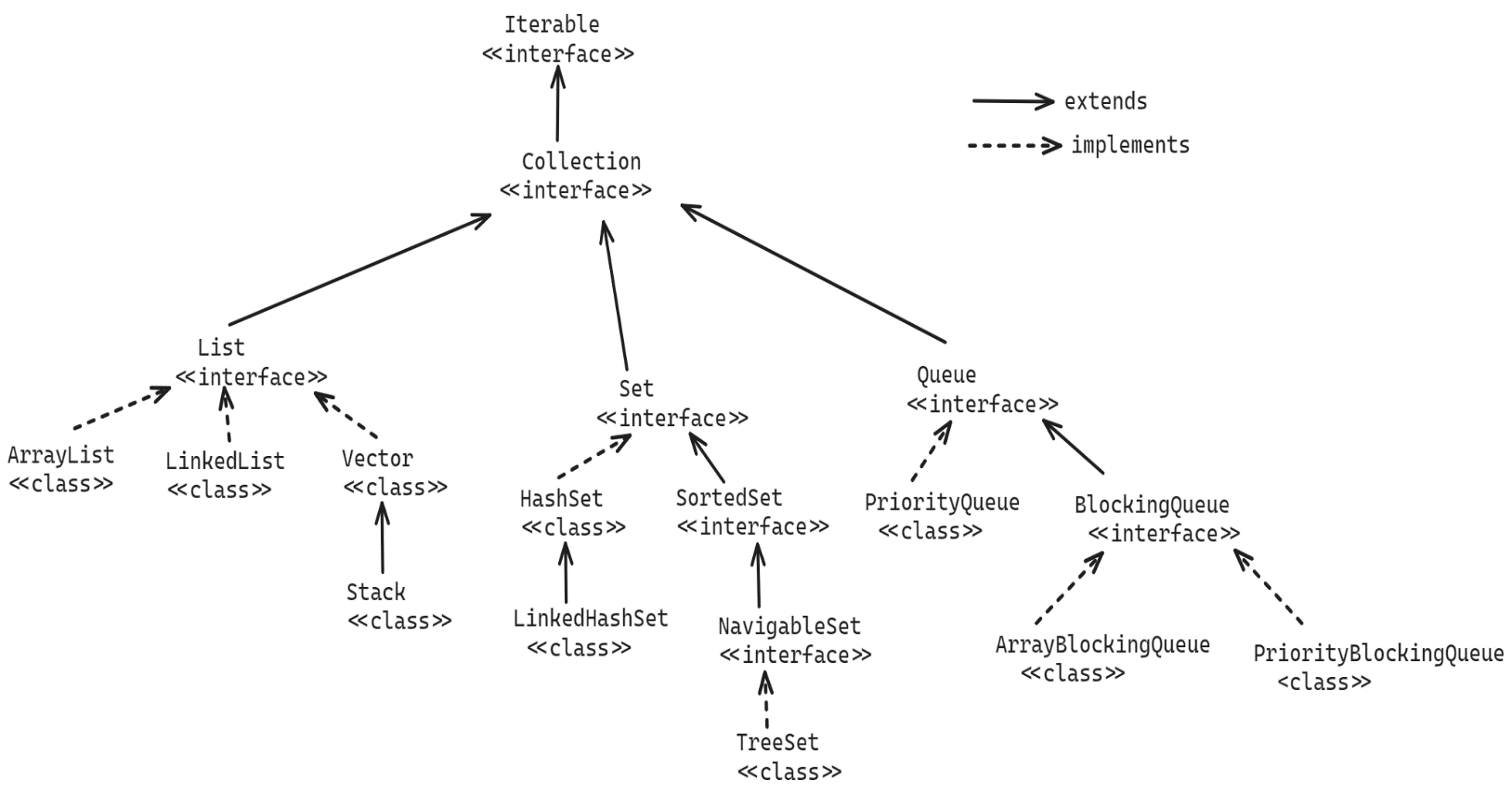
Collections

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

* Data structures is a way to organaize, store and manage the data efficiently, so that it can be used effectively.
* To organize the data efficiently, we have different data structure principles.
* For example,

1. array : fixed set of elements, and can elements can access fastly.
2. linked list : It is a group of elements connected with links, where add/remove of an elements is easy and can grow or shrink automatically.
3. LIFO(stack) : Last In First Out. The elements can be added/removed from the same end.
4. FIFO(queue): First In First Out. The elements can be added from the last and removed from the first.
5. Map : The elements are stored in key-value pairs.
6. tree : The elements are arranged in the hierachical order.
7. graph : The elements are connected with multiple edges.

* Java has provided, collections API, which contains a set of interfaces and classes, which are developed by implementing the data structure principles.
* So, as a programmer we can directly use the pre-defined classes to organize our data efficiently, without implementing those principles manually.
* The main interface of Java collections is the java.util.Collection interface.
* The hieararchy of Collection interface is,



* Collection interface has 3 child interfaces, for different requirements.

1. List : can store the elements in the insertion order, can store duplicate elements and you can access eleements by their index.
2. Set : can store unique elements, do not maintain insertion order, and you can’t access elements with index.
3. Queue: can store the elements in the FIFO, can process elements based on priority, can store duplicate elements.

Methods of Collection interface:

1. add(E e) : Adds an element to the collection. If added successfully then returns true, otherwise returns false.
2. addAll(Collection c): Adds all the elements of the specified collection to this collection.
3. clear(): removes all the elements from this collection.
4. remove(E e) : removes a specified element from this collection. If removed successfully then returns true, otherwise returns false.
5. contains(Object o): checks whether a specified object exists in this collection or not. If exist, returns true. Otherwise, returns false.
6. iterator(): returns an Iterator object over this collection.
7. size(): returns the number of elements in this collection.
8. isEmpty(): returns true, if this collection is empty. Otherwise, returns false.
9. toArray(): returns an array for this collection.
10. removeAll(Collection c): removes the elements from this collection, that are also available in the specified collection.
11. retainAll(Collection c): Retains only the elements in this collection that are contained in the specified collection.

List<E> interface:

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

* When you want to store the elements in a sequence in a collection object then you have to use List object.
* In a List object, each element has position, which is also called index.
* Because the elements are stored in the List object in the order of insertion, we call List as a Ordered collection.
* In a List object, you can store duplicate elements also.
* The implementation classes of List interface are,

ArrayList

LinkedList and

Vector

* The methods of Collection interface are inherited to List interface and also List interface has provided the below key methods.

1. add(index, element): inserts an element at the given position.
2. get(index): returns the element from the given position.
3. indexOf(element): returns the position of the element.
4. lastIndexOf(element): returns the posistion of the last occurrence of the element.
5. remove(index): removes the element from the given position.
6. listIterator() : returns a ListIterator object.
7. set(index, element): replaces an element at the given position.

Traversing a list :

* Traversing a list means, squentially accessing the elements of a list.
* To traverse a list, we have 3 options.

1. Iterator object
2. for each loop
3. for loop

* Iterator is an interface in java.util package and an Iterator object will access the elements of a collection object one at a time.
* In Collection interface, iterator() method is provided and different implementation classes in collections have overridden this iterator() method in their own way.
* iterator() method creates and returns a new Iterator object.
* The Iterator object is created as a wrapper object on top of the exisiting collection object. So, it doesn’t consume more memory.
* Iterator object maintains a cursor, and the cursor is by default placed at before the first element of the collection.
* Iterator interface has provided methods to access the elements.

1. hasNext() : return true, if next element existing after the cursor
2. next() : first moves the cursor to the next element and returns that element.
3. remove(): removes the element at the cursor.

a sample code:

ArrayList<Integer> arrList = new ArrayList<Integer>();

arrList.add(10);

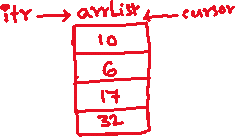
arrList.add(6);

arrList.add(17);

arrList.add(32);



Iterator<Integer> itr =



arrList.iterator();

while(itr.hasNext())

{

Interger i = itr.next();

S.o.println(i);

}

* A for each loop can be used like below.

for(Integer i : arrList)

{

S.o.println(i);

}

* A for loop can be used like below.

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

{

S.o.println(arrList.get(i));

}

ArrayList class:

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

* ArrayList class uses a resizable array(dynamic array) internally to store the elements.
* When we create ArrayList object, internally an array **Object[] elementData** is created with capacity of 10 elements.
* When the elements are execeed the capacity, a new larger array is created with capacity increment of 50%, and elements are copied from previous array to the new array.
* You can specify the initial capacity at the time constructing ArrayList object.

For example,

ArrayList<String> arr1 = new ArrayList<>(); //constructs a list with initial

capacity of ten.

ArrayList<String> arr2 = new ArrayList<>(8); //constructs a list with the

specified initial capacity.

//A sample code to test the ArrayList methods.

**import** java.util.ArrayList;

**import** java.util.Iterator;

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

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

ArrayList<String> arrList = **new** ArrayList<>();

arrList.add("Dell");

arrList.add("Samsung");

arrList.add("Lenono");

arrList.add("Sony");

arrList.add("Apple");

//inserting an element at index 2

arrList.add(2, "Hp");

System.***out***.println("The current size of the list : " + arrList.size());

System.***out***.println("Removing an element from index 4");

arrList.remove(4);

System.***out***.println("The current size of the list : " + arrList.size());

System.***out***.println("Searching for an element 'Sony' in the list : " + arrList.contains("Sony"));

//replace the element at index 3 with element "Asus"

arrList.set(3, "Asus");

//display the list elements

Iterator<String> itr = arrList.iterator();

System.***out***.println("The elements in the ArrayList object : ");

**while**(itr.hasNext()) {

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

}

}

}

* The below example has two Java files.

1. Employee.java
2. Main.java

* In Main.java, we are adding employees to the ArrayList object and then we are performing some searching operations.

Employee.java

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

**public** **class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** String gender;

**private** **double** experience;

**public** Employee(**int** empno, String ename, **double** sal, String gender, **double** experience) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.gender = gender;

**this**.experience = experience;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", gender=" + gender + ", experience="

+ experience + "]";

}

}

Main.java

--------

**import** java.util.ArrayList;

**import** java.util.Iterator;

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

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

ArrayList<Employee> employeesList = **new** ArrayList<>();

employeesList.add(**new** Employee(7807, "Miller", 12799.0, "Male", 4.5));

employeesList.add(**new** Employee(7125, "Killer", 92999.0, "Male", 3.5));

employeesList.add(**new** Employee(7788, "Scott", 32400.0, "Male", 6.2));

employeesList.add(**new** Employee(7878, "Lisa", 22500.0, "Female", 4.9));

employeesList.add(**new** Employee(7296, "King", 56300.0, "Male", 9.5));

employeesList.add(**new** Employee(7705, "Mary", 87800.0, "Female", 11.5));

employeesList.add(**new** Employee(7856, "Pier", 48000.0, "Female", 5.3));

employeesList.add(**new** Employee(8611, "Smith", 19755.0, "Male", 5.2));

System.***out***.println("Show the female employees : ");

**for**(Employee e : employeesList)

{

**if**(e.getGender().equalsIgnoreCase("Female"))

{

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

}

}

System.***out***.println("Show the employees with salary > 30000.0");

**for**(Employee e : employeesList)

{

**if**(e.getSal() > 30000.0)

{

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

}

}

System.***out***.println("Show the employees with experience > 8 years");

Iterator<Employee> itr = employeesList.iterator();

**while**(itr.hasNext()) {

Employee e = itr.next();

**if**(e.getExperience() > 8) {

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

}

}

}

}

Comparable<T> interface:

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

* Comparable<T> is a functional interface, it has a single abstract method called compareTo().

public abstract int compareTo(T t);

* This compareTo() method returns a negitive, zero or positive integer.
* If the current object’s value < other object’s value then returns negitive integer.
* If the current object’s value == other object’s value then returns zero.
* If the current object’s value > other object’s value then returns positive integer.
* When you want to sort the elements of a List object, the elements must be Comparable type.
* Suppose, if you want to sort the employees of a List object, the employees must be Comparable type.
* To make the employees as Comparable type, we have to implement the Employee class from Comparable<T> interface.

//The example code will sort the employees of a list object

//in the salaries ascending order.

Employee.java

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

**public** **class** Employee **implements** Comparable<Employee> {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** String gender;

**private** **double** experience;

**public** Employee(**int** empno, String ename, **double** sal, String gender, **double** experience) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.gender = gender;

**this**.experience = experience;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", gender=" + gender + ", experience="

+ experience + "]";

}

@Override

**public** **int** compareTo(Employee o) {

**return** (**int**) (**this**.getSal() - o.getSal());

}

}

Main.java

---------

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Iterator;

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

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

ArrayList<Employee> employeesList = **new** ArrayList<>();

employeesList.add(**new** Employee(7807, "Miller", 12799.0, "Male", 4.5));

employeesList.add(**new** Employee(7125, "Killer", 92999.0, "Male", 3.5));

employeesList.add(**new** Employee(7788, "Scott", 32400.0, "Male", 6.2));

employeesList.add(**new** Employee(7878, "Lisa", 22500.0, "Female", 4.9));

employeesList.add(**new** Employee(7296, "King", 56300.0, "Male", 9.5));

employeesList.add(**new** Employee(7705, "Mary", 87800.0, "Female", 11.5));

employeesList.add(**new** Employee(7856, "Pier", 48000.0, "Female", 5.3));

employeesList.add(**new** Employee(8611, "Smith", 19755.0, "Male", 5.2));

employeesList.add(**new** Employee(8278, "Sophia", 89555.0, "Female", 6.7));

//sort the employees of the list

Collections.*sort*(employeesList);

System.***out***.println("Display the employees of list:");

**for**(Employee e : employeesList)

{

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

}

}

}

Comparator<T> interface:

-------------==========

* It is a functional interface, which has a single abstrct method called compare().

public abstract int compare(T o1, T o2);

* When multiple sorting strategies are required, then we have to use Comparator interface.
* With Comparable interface, we can sort the elements in a single sorting order.
* For ex, if we want to sort the list of employees in empno’s order and also sal’s order in the application then we can’t do it with Comparable interface.
* To sort the list of elements with Comparator, first you have to create a separate class by implementing Comparator interface.
* If mulitple sorting options are needed, then you have to create multiple comparator classes.

For ex:

public class EmpnoComparator implements Comparator<Employee>

{

@Override

public int compare(Employee o1, Employee o2)

{

//logic

}

}

public class SalComparator implements Comparator<Employee>

{

@Override

public int compare(Employee o1, Employee o2)

{

//logic

}

}

* Finally, to sort the list object, call sort() method of List with Comparator parameter.

ex:

employeesList.sort(new EmpnoComparator());

//The below example code will sort the list of employees

//in empno’s sorting order and also experience’s sorting order.

//Here, we are creating two comparator classes, one for comparing

//the empno’s and the other for comparing the experience’s.

Employee.java

--=========

**public** **class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** String gender;

**private** **double** experience;

**public** Employee(**int** empno, String ename, **double** sal, String gender, **double** experience) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.gender = gender;

**this**.experience = experience;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", gender=" + gender + ", experience="

+ experience + "]";

}

}

EmpnoComparator.java

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

**import** java.util.Comparator;

**public** **class** EmpnoComparator **implements** Comparator<Employee> {

@Override

**public** **int** compare(Employee o1, Employee o2) {

**return** o1.getEmpno() - o2.getEmpno();

}

}

ExpComparator.java

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

**import** java.util.Comparator;

**public** **class** ExpComparator **implements** Comparator<Employee> {

@Override

**public** **int** compare(Employee o1, Employee o2) {

**if**(o1.getExperience() < o2.getExperience())

**return** -1;

**else** **if**(o1.getExperience() > o2.getExperience())

**return** 1;

**else**

**return** 0;

}

}

Main.java

=========

import java.util.ArrayList;

import java.util.Collections;

import java.util.Iterator;

public class Main {

public static void main(String[] args) {

ArrayList<Employee> employeesList = new ArrayList<>();

employeesList.add(new Employee(7807, "Miller", 12799.0, "Male", 4.5));

employeesList.add(new Employee(7125, "Killer", 92999.0, "Male", 3.5));

employeesList.add(new Employee(7788, "Scott", 32400.0, "Male", 6.2));

employeesList.add(new Employee(7878, "Lisa", 22500.0, "Female", 4.9));

employeesList.add(new Employee(7296, "King", 56300.0, "Male", 9.5));

employeesList.add(new Employee(7705, "Mary", 87800.0, "Female", 11.5));

employeesList.add(new Employee(7856, "Pier", 48000.0, "Female", 5.3));

employeesList.add(new Employee(8611, "Smith", 19755.0, "Male", 5.2));

employeesList.add(new Employee(8278, "Sophia", 89555.0, "Female", 6.7));

//sort the list of employees on empno's

employeesList.sort(new EmpnoComparator());

System.out.println("Employees of the list in empno's sorting order:");

for(Employee e : employeesList) {

System.out.println(e);

}

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

//sort the list of employees on experience

employeesList.sort(new ExpComparator());

System.out.println("Employees of the list in experience's sorting order:");

for(Employee e : employeesList) {

System.out.println(e);

}

}

}

Q) what is the difference between Comparable and Comparator?

A) 1. Comparable interface is provided in java.lang package and Comparator interface is provided in java.util package.

2. When you need a single sorting strategy then use Comparable interface. When you need multiple sorting strategies then use Comparator interface.

3. The abstract method of Comparable is compareTo() and the abstract method of Comparator is compare().

Q) ArrayList object is thread safe or not?

A) No, not a thread safe object.

Q) can we make ArrayList object as thread safe or not?

a) Yes. Call synchronizedList() method of Collections class.

ex:

employeesList = Collections.synchronizedList(employeesList);

Q) ArrayList object is mutable object or not?

A) Yes. We can add/remove/replace the elements in ArrayList object.

Q) can we make ArrayList object as immutable or not?

A) Yes. Call unmodifiableList() method of Collections class.

ex:

employeesList = Collections.unmodifiableList(employeesList);

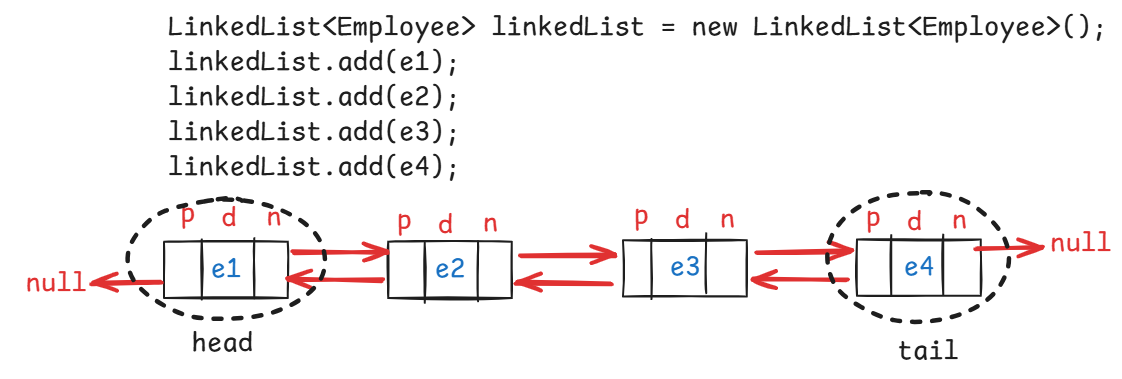
LinkedList class:

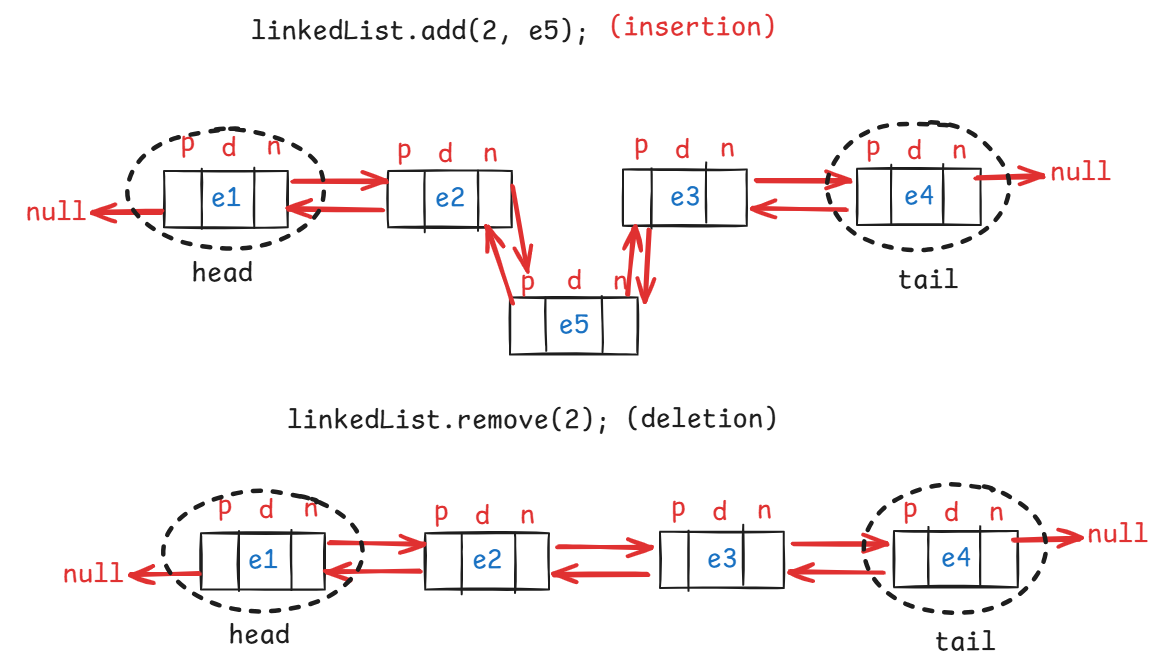
---------------

* LinkedList class is also an implementation of List interface.
* In LinkedList, the elements are stored as nodes.
* So, a LinkedList is a collection of nodes.
* Each node has 3 parts.

1. previous pointer
2. data
3. next pointer

* previous pointer holds the address of the previous node, next pointer holds the address of the next node, and the data part contains the actual element.
* So, a LinkedList object is a double linked list implementation of List interface.
* The first node of the LinkedList is called head/front and the last node of the LinkedList is called tail/rear.





* To a LinkedList object, eleements can be added/removed from head or from tail also.
* The methods are,

addFirst(E e)

addLast(E e)

getFirst()

getLast()

peekFirst()

peekLast()

pollFirst()

pollLast()

removeFirst()

removeLast()

* Difference between peek, poll and remove is,

. peek methods will return the element, but does not remove the element. If the linked list is empty then returns null.

. poll methods will return and removes the element. If the linked list is empty then returns null.

. remove methods will return and removes the element. If the linked list is empty then throws NoSuchElementException.

Q) When to use ArrayList and When to use LinkedList?

A) If you need retrieval more times and insertion/deletion of elements less times then use ArrayList.

If you need insertion/deletion of elements more times and retrieval less times then use LinkedList.

Q) LinkedList object is thread safe or not?

A) No, not a thread safe object.

Q) can we make LinkedList object as thread safe or not?

a) Yes. Call synchronizedList() method of Collections class.

ex:

employeesList = Collections.synchronizedList(employeesList);

Q) LinkedList object is mutable object or not?

A) Yes. We can add/remove/replace the elements in LinkedList object.

Q) can we make LinkedList object as immutable or not?

A) Yes. Call unmodifiableList() method of Collections class.

ex:

employeesList = Collections.unmodifiableList(employeesList);

hashCode() and equals() method:

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

* hashCode() and equals() methods are available in Object class.
* hashCode() method of Object class returns the memory address of the object.
* equals() method of Object class works like equality operator(==).
* Suppose if two different objects of a class have the same data, the hashCode() method of Object class returns the different values. Because their memory addresses are different.
* Similarly, if two differennt objects of a class have the same data, the equals() method of Object class returns false. Because the reference variables are pointing to the different objects.
* But, when we are working with the collections like Set and Map, we have a requirement that, if two different objects of a class have the same data then hashCode() method should return the same value and equals() method should return true.
* So, you have to override the hashCode() and equals() methods in your class.
* In IDE, you no need to override them manually. You can override them with source option.
* Right click in the class, source then Generate hashCode and equals methods.

java.util.Set interface

* Set is an unordered collection of elements without duplicates.
* A Set object doesn’t store the duplicate elements. If a duplicate element is added, then it will skip the element.
* It is unordered collection, because it doesn’t store the elements in the same order internally.
* The implementation classes of Set interface are,

HashSet, LinkedHashSet and TreeSet.

* The set objects internally uses map objects, where map objects uses hash table data structure.
* hash table data structure stores the elements in key-value pairs.
* When you add an element to the set object, that element will be stored as a key and with value as PRESENT.
* Here, PRESENT is a dummy object, created for Object class.

HashSet<E> class:

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

* When you create a HashSet class object, it internally creates HashMap class object, to store the elements.
* HashMap class uses hash table data structure and the table is created with default initial capacity as 16 and load factor 0.75.
* While creating HashSet class object, you can also specify the intial capacity and load factor.

HashSet<String> set = new HashSet<>();

. initial capacity : 16

. load factor : 0.75

HashSet<String> set = new HashSet<>(10);

. initial capacity : 10

. load factor : 0.75

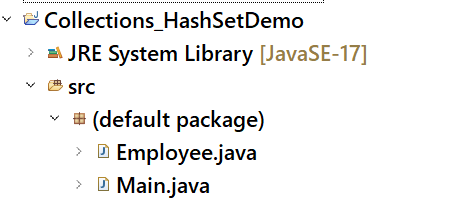
HashSet<String> set = new HashSet<>(10, 0.5f);

. initial capacity : 10

. load factor : 0.5

* The capacity of hash table structure will be doubled based on capacity and load factor.
* if capacity is 16 and load factor is 0.75 then the capacity will be doubled after adding 12(16 \* 0.75) pairs to the hash table.
* if the capacity is 10 and load factor is 0.5 then the capacity will be doubled after adding 5( 10 \* 0.5) pairs to the hash table.
* When you are adding an element to a set object, first hashcode is calculated and then based on the hashcode, bucket index is calculated and then the element will be stored into that bucket as a key with value as PRESENT.
* To access the elements of a set object, you have to use Iterator object or for each loop.

example:



Employee.java

**import** java.util.Objects;

**public** **class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** String gender;

**private** **double** experience;

**public** Employee(**int** empno, String ename, **double** sal, String gender, **double** experience) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.gender = gender;

**this**.experience = experience;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", gender=" + gender + ", experience="

+ experience + "]";

}

@Override

**public** **int** hashCode() {

**return** Objects.*hash*(empno, ename, experience, gender, 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)

&& Double.*doubleToLongBits*(sal) == Double.*doubleToLongBits*(other.sal);

}

}

Main.java

**import** java.util.HashSet;

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

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

HashSet<Employee> employeesSet = **new** HashSet<>();

employeesSet.add(**new** Employee(7807, "Miller", 12799.0, "Male", 4.5));

employeesSet.add(**new** Employee(7807, "Miller", 12799.0, "Male", 4.5));

employeesSet.add(**new** Employee(7125, "Killer", 92999.0, "Male", 3.5));

employeesSet.add(**new** Employee(7788, "Scott", 32400.0, "Male", 6.2));

employeesSet.add(**new** Employee(7878, "Lisa", 22500.0, "Female", 4.9));

employeesSet.add(**new** Employee(7296, "King", 56300.0, "Male", 9.5));

employeesSet.add(**new** Employee(7705, "Mary", 87800.0, "Female", 11.5));

employeesSet.add(**new** Employee(7856, "Pier", 48000.0, "Female", 5.3));

employeesSet.add(**new** Employee(8611, "Smith", 19755.0, "Male", 5.2));

employeesSet.add(**new** Employee(8278, "Sophia", 89555.0, "Female", 6.7));

System.***out***.println("Displaying the elements of the HashSet object: ");

**for**(Employee e : employeesSet) {

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

}

}

}

Q) HashSet object is thread safe or not?

A) No, not a thread safe object.

Q) can we make HashSet object as thread safe or not?

a) Yes. Call synchronizedSet() method of Collections class.

ex:

employeesSet = Collections.synchronizedSet(employeesSet);

Q) HashSet object is mutable object or not?

A) Yes. We can add/remove/replace the elements in HashSet object.

Q) can we make HashSet object as immutable or not?

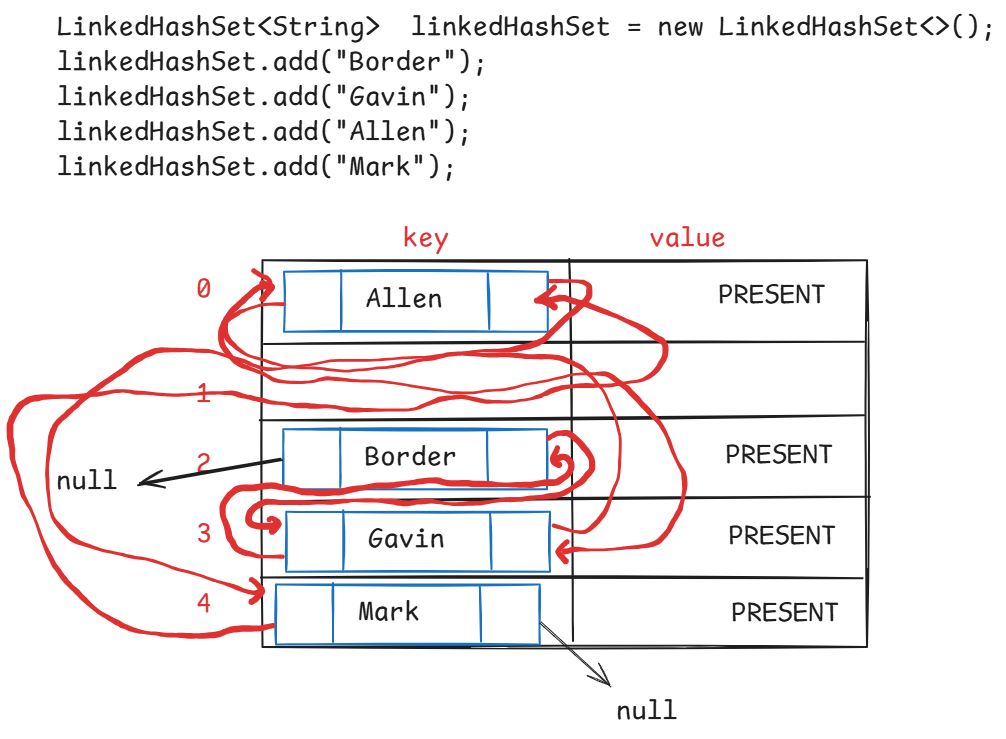
A) Yes. Call unmodifiableSet() method of Collections class.

ex:

employeesSet = Collections.unmodifiableSet (employeesSet);

LinkedHashSet<E> class:

* Hash table and linked list implementation of the Set interface, with predictable iteration order.
* This implementation differs from HashSet in this it maintains a doubly-linked list running through all of its entries.
* This linked list defines the iteration ordering, which is the order in which elements were inserted into the set (*insertion-order*).



Q) LinkedHashSet object is thread safe or not?

A) No, not a thread safe object.

Q) can we make LinkedHashSet object as thread safe or not?

a) Yes. Call synchronizedSet() method of Collections class.

ex:

employeesSet = Collections.synchronizedSet(employeesSet);

Q) LinkedHashSet object is mutable object or not?

A) Yes. We can add/remove/replace the elements in LinkedHashSet object.

Q) can we make LinkedHashSet object as immutable or not?

A) Yes. Call unmodifiableSet() method of Collections class.

ex:

employeesSet = Collections.unmodifiableSet (employeesSet);

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:

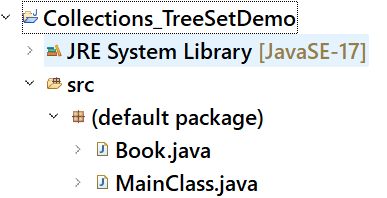
--------------------------

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.
* When you want a collection object, which should not store duplicates and also it should maintain sorting order then you have to use TreeSet class.
* A TreeSet(TreeMap) stores elements in sorted order based on the keys, but NOT in bucket index order.
* 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.



Book.java

**import** java.util.Objects;

**public** **class** Book **implements** Comparable<Book>

{

**private** **int** bookId;

**private** String bookName;

**private** **double** price;

**public** Book(**int** bookId, String bookName, **double** price) {

**super**();

**this**.bookId = bookId;

**this**.bookName = bookName;

**this**.price = price;

}

**public** **int** getBookId() {

**return** bookId;

}

**public** **void** setBookId(**int** bookId) {

**this**.bookId = bookId;

}

**public** String getBookName() {

**return** bookName;

}

**public** **void** setBookName(String bookName) {

**this**.bookName = bookName;

}

**public** **double** getPrice() {

**return** price;

}

**public** **void** setPrice(**double** price) {

**this**.price = price;

}

@Override

**public** String toString() {

**return** "Book [bookId=" + bookId + ", bookName=" + bookName + ", price=" + price + "]";

}

@Override

**public** **int** hashCode() {

**return** Objects.*hash*(bookId, bookName, price);

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

Book other = (Book) obj;

**return** bookId == other.bookId && Objects.*equals*(bookName, other.bookName)

&& Double.*doubleToLongBits*(price) == Double.*doubleToLongBits*(other.price);

}

@Override

**public** **int** compareTo(Book o) {

**if** (**this**.bookId < o.getBookId())

**return** -1;

**else** **if**(**this**.bookId > o.getBookId())

**return** 1;

**else**

**return** 0;

}

}

MainClass.java

**import** java.util.TreeSet;

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

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

TreeSet<Book> ts = **new** TreeSet<>();

ts.add(**new** Book(15529, "Java", 599.0));

ts.add(**new** Book(11756, "Oracle", 399.0));

ts.add(**new** Book(43098, "Python", 499.0));

ts.add(**new** Book(27707, "Spring", 899.0));

ts.add(**new** Book(32189, "React", 499.0));

ts.add(**new** Book(48891, "Angular", 499.0));

System.***out***.println("Displaying the elements of TreeSet collection:");

**for**(Book book : ts)

{

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

}

}

}

Queue<E> interface:

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

* Queue orders elements in First In First Out(FIFO) manner, but not necessarily.
* A Queue represents a collection designed for holding the elements prior to processing.
* UseCase1: In Operating Systems, processes are added to the Queue as they arrive. The CPU picks a process from the front of the queue and executes it.
* UseCase2: In a Customer Service call center, incoming calls are placed in a queue. The first call is attended first.
* UseCase3: Employees submit print jobs, they are stored in a queue. The printer processes jobs one by one from the front of the queue.
* A queue maintains two pointers called head/fornt and tail/rear.
* The elements are inserted from the tail pointer and removed from the head pointer.
* The methods for inserting the element into the Queue are,

1. add(E e) : adds an element and returns true, if successful. Otherwise, throws IllegalStateException.
2. offer(E e): adds an element and returns true, if successful. Otherwise, returns false.

* The methods for removing element from the Queue are,

1. poll() : retrieves and removes the element from the head pointer. Returns null if the queue is empty.
2. remove(): retrieves and removes the element from the head pointer. Throws NoSuchElementException, if the queue is empty.

* The methods for retrieving the element from the Queue are,

1. element() : retrieves but does not remove the element from the head pointer. Throws NoSuchElementException, if the queue is empty.
2. peek() : retrieves but does not remove the element from the head pointer. Returns null, if the queue is empty.

Deque<E> interface:

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

* Deque<E> interface extends Queue<E> interface, and it is called double ended queue.
* It means, a Deque object allows addition/deletion/retrieval of elements from both head and tail pointers.
* The methods are,

1. addFirst(e)
2. addLast(e)
3. offerFirst(e)
4. offerLast(e)
5. getFirst()
6. getLast()
7. removeFirst()
8. removeLast()
9. pollFirst()
10. pollLast()
11. peekFirst()
12. peekLast()

* PriorityQueue class implements Queue interface, and LinkedList class implements Deque interface.

PriorityQueue<E> class:

* PriorityQueue class does not maintain the elements in the FIFO order. It maintains the elements in the natural sorting order or the order defined by the Comparator object.
* The default capacity of the PriorityQueue object is 11.
* The PriorityQueue object is an unbounded queue, which means, there is no size restrictions. We can add any number of elements.
* The elements must be Comparable elements, for sorting in natural sorting order.
* While constructing PriorityQueue object, if we pass Comparator object as a parameter then the elements may not be Comparable elements.
* we can not add null value to PriorityQueue object. If added, we will get NullPointerException.

Example code:

**import** java.util.Iterator;

**import** java.util.PriorityQueue;

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

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

PriorityQueue<Integer> pq = **new** PriorityQueue<>();

pq.add(5);

pq.add(3);

pq.add(10);

pq.add(1);

pq.add(8);

System.***out***.println("displaying the elements of PriorityQueue with Iterator");

Iterator<Integer> itr = pq.iterator();

**while**(itr.hasNext())

{

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

}

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

System.***out***.println("diplaying the elements of PriorityQueue with for each loop");

**for**(Integer i : pq) {

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

}

System.***out***.println("displaying the elements of PriorityQueue with Iterator");

Iterator<Integer> itr2 = pq.iterator();

**while**(itr2.hasNext())

{

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

}

}

}

output:

displaying the elements of PriorityQueue with Iterator

1

3

10

5

8

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

diplaying the elements of PriorityQueue with for each loop

1

3

10

5

8

displaying the elements of PriorityQueue with Iterator

1

3

5

8

10

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, PriorityBlockingQueue 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.

Example code on ArrayBlockingQueue<E> class:

**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 removed : " + 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();

}

}

output:

Inside : Thread-0

element put is : 1

element put is : 2

element put is : 3

element put is : 4

element put is : 5

Element removed : 1

element put is : 6

finished : Thread-0

Q) what is the difference between array and ArrayList?

A) 1. array has a fixed size. But ArrayList has dynamic size.

2. array can hold homogeneous elements. But ArrayList can hold heterogeneous elements.

3. arrays use less memeory. But ArrayList use more memory

4. You have to use loops to perform operations on arrays. But ArrayList has methods like add(), remove(), size(), etc…

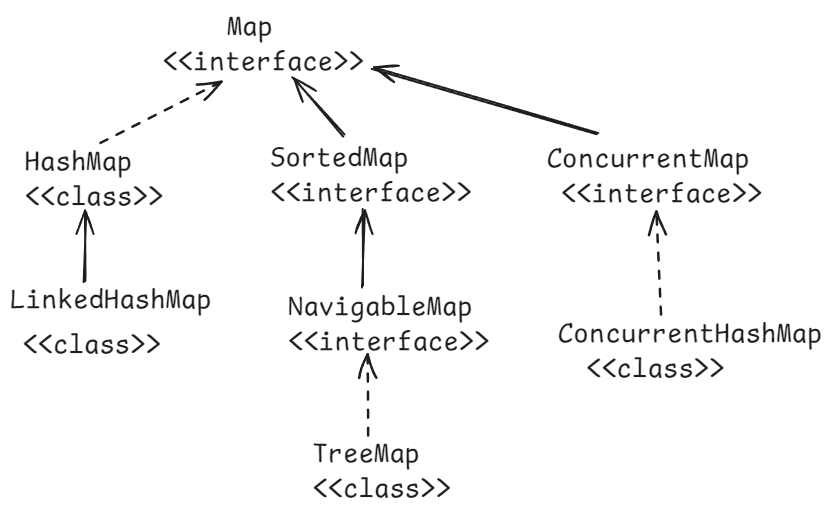
Q) what is the difference between ArrayList and Vector classes?

A) 1. ArrayList is not a thread-safe object. But Vector is a thread-safe object.

2. ArrayList performance is faster. But Vector performance is slower.

3. ArrayList increases size by 50% when capacity is exceeded. But Vector doubles the size when capacity is exceeded.

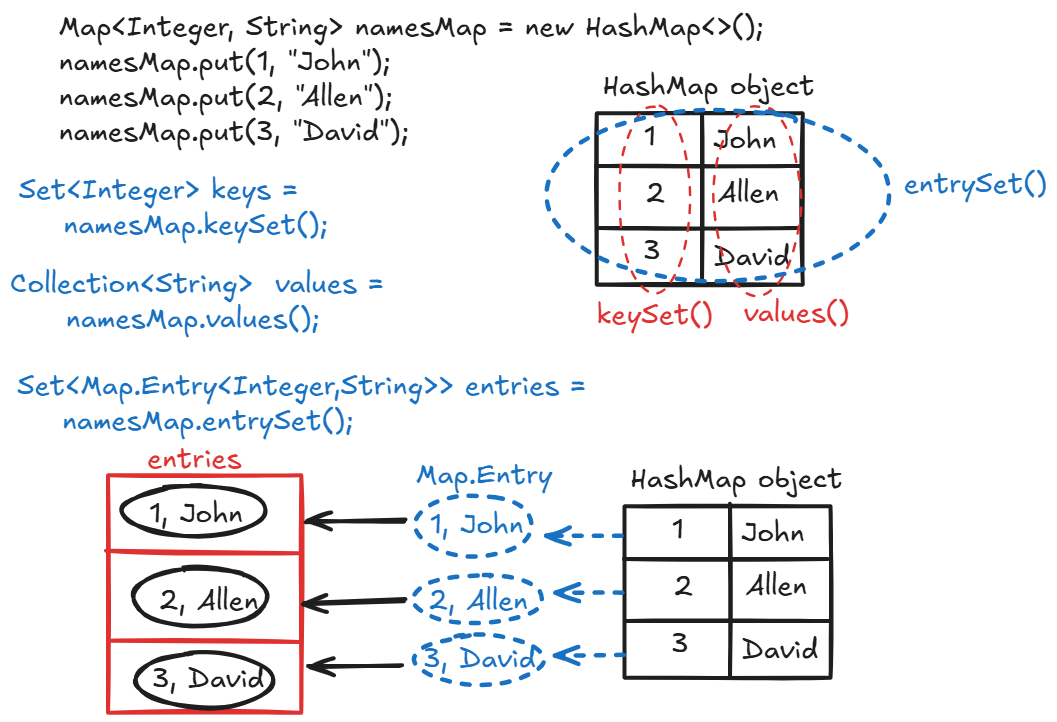
Map<K,V> interface:



* When you want to a collection to store the elements in a key&value pairs, then you have to use Map object.
* A Map object internally uses hash table data structure, which can store the data in the key & value pairs.
* A Map object can not store duplicate keys. It means, keys are unique. But it can store duplicate values.
* A Map object can store atmost one null key, but multiple null values.
* If you put a duplicate key, then a map object will replace the previous value with the current value for that key.

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.



Q) When to use HashMap object?

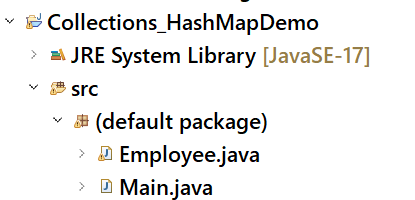
A) When you want to store the data as key and value pairs in a collection and if you do not want to maintain the data in insertion order then you can use HashMap object.

Q) When to use LinkedHashMap object?

A) When you want to store the data as key and value pairs in a collection and if you want to maintain the data in insertion order of the keys, then you can use LinkedHashMap object.

Q) When to use a TreeMap object?

A) When you want to store the data as key and value pairs in a collection and if you want to maintain the data in sorting order of the keys, they you can use TreeMap object.



Employee.java

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

**import** java.util.Objects;

**public** **class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** String gender;

**private** **double** experience;

**public** Employee(**int** empno, String ename, **double** sal, String gender, **double** experience) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.gender = gender;

**this**.experience = experience;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", gender=" + gender + ", experience="

+ experience + "]";

}

}

Main.java

--------

//This example creates HashMap object, to store the

// empno as a key and the Employee object as a value.

import java.util.HashMap;

import java.util.Map;

import java.util.Map.Entry;

import java.util.Set;

public class Main {

public static void main(String[] args) {

HashMap<Integer, Employee> employeesMap = new HashMap<>();

Employee e1 = new Employee(7807, "Miller", 12799.0, "Male", 4.5);

Employee e2 = new Employee(7788, "Scott", 32400.0, "Male", 6.2);

Employee e3 = new Employee(7705, "Mary", 87800.0, "Female", 11.5);

Employee e4 = new Employee(8278, "Sophia", 89555.0, "Female", 6.7);

Employee e5 = new Employee(7296, "King", 56300.0, "Male", 9.5);

employeesMap.put(e1.getEmpno(), e1);

employeesMap.put(e2.getEmpno(), e2);

employeesMap.put(e3.getEmpno(), e3);

employeesMap.put(e4.getEmpno(), e4);

employeesMap.put(e5.getEmpno(), e5);

Set<Map.Entry<Integer, Employee>> entries = employeesMap.entrySet();

for(Entry<Integer, Employee> entry : entries)

{

System.out.println("key : " + entry.getKey());

System.out.println("value : " + entry.getValue());

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

}

}

}

Q) HashMap object is thread safe or not?

A) No, not a thread safe object.

Q) can we make HashMap object as thread safe or not?

a) Yes. Call synchronizedMap() method of Collections class.

ex:

employeesMap = Collections.synchronizedMap(employeesMap);

Q) HashMap object is mutable object or not?

A) Yes. We can add/remove/replace the elements in HashMap object.



Q) can we make HashMap object as immutable or not?

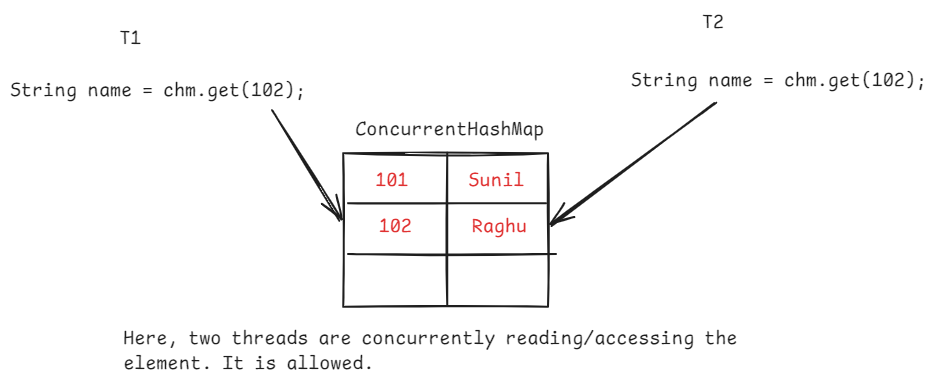
A) Yes. Call unmodifiableMap() method of Collections class.

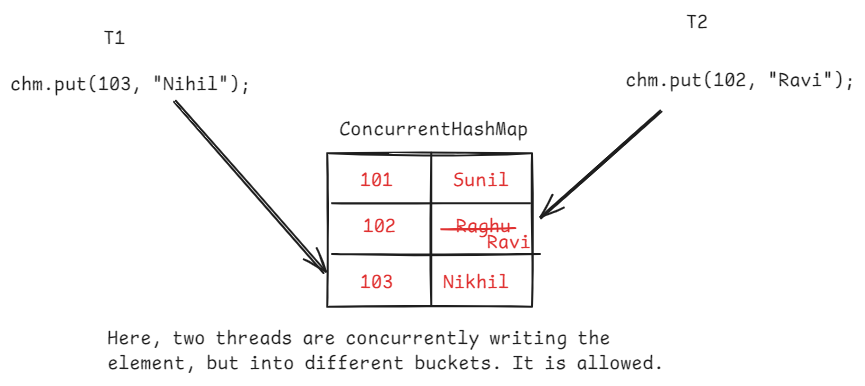
ex:

employeesMap = Collections.unmodifiableMap(employeesMap);

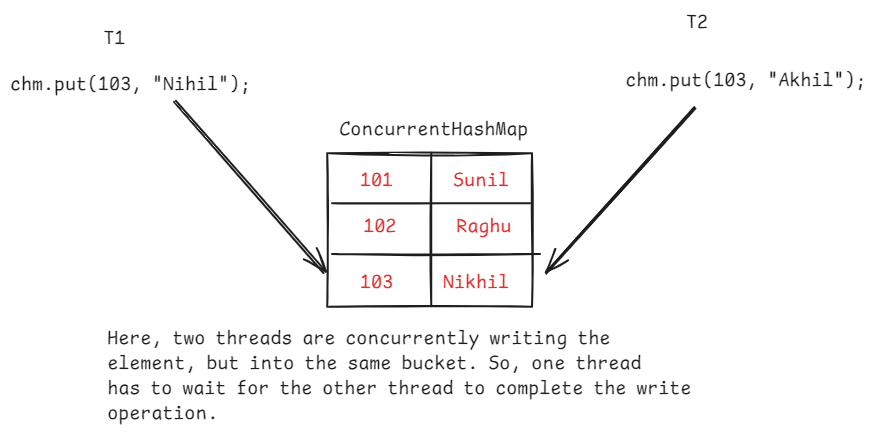
ConcurrentHashMap :

* 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.

Q) what is the difference between Hashtable and HashMap objects?

A) 1. Hashtable is a legacy class introduced in Java 1.0

HashMap is a collection framework class introducted in Java 1.2

2. Hashtable is a synchronized class(by default, thread-safe).

HashMap is not a synchronized class(by default, not thread safe).

3. Hashtable does not allow null key or null value.

HashMap allows one null key, multiple null values.

Fail-fast and Fail-safe Iterator:

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

* Fail-fast and Fail-safe are the behaviours of an iterator.
* Fail-fast and Fail-safe indicates how an iterator behaves when any structural changes are made on the collection object, while iterator is iterating that object.
* Fail-fast means, while iterating a collection object, if any changes are made to that object by adding/removing an element then the iterator throws ConcurrentModificationException.
* Fail-safe means, while iterating a collection object, if any changes are made to that object by adding/removing an element then the iterator doesn’t throw any exception.

Fail-fast code:

ArrayList<Integer> arrList = new ArrayList<>();

arrList.add(12);

arrList.add(32);

arrList.add(29);

arrList.add(90);

arrList.add(57);

Iterator<Integer> itr = arrList.iterator();

while(itr.hasNext())

{

int k = itr.next();

System.out.println(k);

arrList.remove(3);

}

Fail-safe code:

ConcurrentHashMap<Integer, Integer> hm = **new** ConcurrentHashMap<>();

hm.put(1, 100);

hm.put(2, 200);

hm.put(3, 300);

Set<Entry<Integer,Integer>> entries = hm.entrySet();

Iterator<Entry<Integer, Integer>> itr = entries.iterator();

**while**(itr.hasNext()) {

Entry<Integer,Integer> entry = itr.next();

System.***out***.println(entry.getKey() + " -----> " +

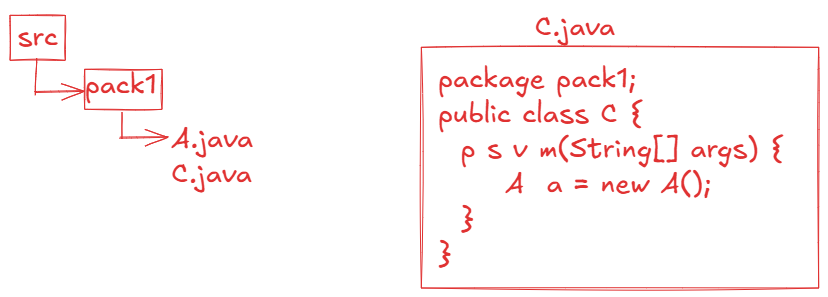
entry.getValue());

hm.put(4, 400);

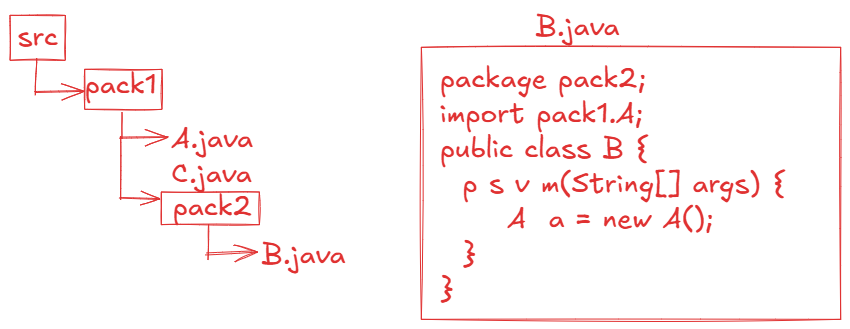
}

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.

Java 8 features

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

* Java8 is one of the major versions of Java, provided multiple important features to the realtime application development.

1. default 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() { //static method

//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 any number of static methods and public static final variables.

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 declared 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.
* Some pre-defined functional interfaces are,

Comparable<T> -------- int compareTo(T t)

Comparator<T> -------- int compare(T t1, T t2)

Runnable -------- void run()

Consumer<T> -------- void accept(T t)

Function<T,R> -------- R apply(T t)

Supplier<T> -------- T get()

Predicate<T> -------- boolean test(T t)

Lambda expression:

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

* Suppose, if you have a functional interface and you want to implement it then you have to create a class and in that class you have to override the single abstract method of that interface.
* Suppose, if you have a requirement to implement a functional interface multiple times, then you have to create multiple classes, and in each class you have to override the single abstract method of that interface.
* For example, Comparator<T> is a functional interface and You want to implement it to sort the list of employees in empno’s ascending order, then you have to create a class to override compare() method.
* Suppose, if you want to sort the list of employees in mulitple orders(empno’s/ename’s/sal’s/department’s/etc.. ascending or descending) then you have to create mulitple classes to override compare() method.
* So, the number of classes will increase in a project.
* In Java8, a solution is given as lambda expressions.
* lambda expressions are only used for providing the implementation for functional interfaces, not for normal interfaces.

syntax of lambda expression:

(parameters) -> 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.

ex: A sample lambda expression to implement Comparator<Employee> interface is,

(e1, e2) -> e1.getEmpno() – e2.getEmpno();

//sample code

**package** com.pack;

**import** java.util.ArrayList;

**import** java.util.Comparator;

**import** java.util.function.Consumer;

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

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

ArrayList<Employee> employeesList = **new** ArrayList<>();

employeesList.add(**new** Employee(7807, "Miller", 12799.0, "Male", 4.5));

employeesList.add(**new** Employee(7125, "Killer", 92999.0, "Male", 3.5));

employeesList.add(**new** Employee(7788, "Scott", 32400.0, "Male", 6.2));

employeesList.add(**new** Employee(7878, "Lisa", 22500.0, "Female", 4.9));

employeesList.add(**new** Employee(7296, "King", 56300.0, "Male", 9.5));

employeesList.add(**new** Employee(7705, "Mary", 87800.0, "Female", 11.5));

employeesList.add(**new** Employee(7856, "Pier", 48000.0, "Female", 5.3));

employeesList.add(**new** Employee(8611, "Smith", 19755.0, "Male", 5.2));

employeesList.add(**new** Employee(8278, "Sophia", 89555.0, "Female", 6.7));

//sort the list of employees in empno's ascending order

//sort() method has Comparator<T> as parameter and it is

//a functional interface. So, you can pass lambda expression.

employeesList.sort((e1, e2) -> e1.getEmpno() - e2.getEmpno());

/\*

for(Employee e : employeesList)

{

System.out.println(e);

}

\*/

//instead of using for each loop for iterating a collection

//we can use forEach() method also.

//forEach() method has Consumer<T> as parameter and it is a

//functional interface. So, you can pass lambda expression

//as parameter.

employeesList.forEach(e -> System.***out***.println(e));

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

//sort the list of employees in exp's ascending order

Comparator<Employee> c = (e1, e2) -> {

**if**(e1.getExperience() < e2.getExperience())

**return** -1;

**else** **if**(e1.getExperience() > e2.getExperience())

**return** 1;

**else**

**return** 0;

};

employeesList.sort(c);

/\*

for(Employee e : employeesList)

{

System.out.println(e);

}

\*/

employeesList.forEach(e -> System.***out***.println(e));

}

}

Optional<T> class:

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

* If an object has null value, and if you are calling method of that object then NullPointerException will be thrown.
* In a project, we have more objects, so the most frequently occrurred exception is NullPointerException.
* To avoid the NullPointerException, you have add the null checks.

ex:

public class Service {

public Employee getEmpById(int empid){

//reads an employee from the database

//if does not exist, then returns null

}

}

public class Controller {

public void show() {

Service s = new Service();

Employee e = s.getEmpById(101);

S.o.p(e.getEname());

S.o.p(e.getSal());

}

}

* In the above example, if 101 employee does not exist then

getEmpById() method returns null.

* In show() method, the variable e stores null and when you call e.getEname(), NullPointerException will be thrown.
* So, we have to modify the Controller class code like below.

public class Controller {

public void show() {

Service s = new Service();

Employee e = s.getEmpById(101);

if(e != null) { // null check

S.o.p(e.getEname());

S.o.p(e.getSal());

}

}

}

* If more null checks are added in the code, then the code becomes complicated.
* To reduce the null checks and also to avoid NullPointerException, Java 8 version has provided Optional<T> class.
* If a method has to return an object, then first has to store that object into Optional class object and then the method should return the Optional class object.
* Optional class object is a container object, it can store another object inside it.

creating Optional<T> class object:

1. Optional<Employee> opt = Optional.empty();

. creates empty Optional object.

2. Optional<Employee> opt = Optional.of(e1);

. If e1 is not null, then Optional object is created and

inside it e1 object will be stored.

. If e1 is null then NullPointerException will be thrown.

3. Optional<Employee> opt = Optional.ofNullable(e1);

. If e1 is not null, then Optional object is created and

inside it e1 object will be stored.

. If e1 is null then empty Optional object is created.

Methods of Optional<T> class:

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

1. isPresent() : If value is present in the Optional object then returns true. Otherwise, returns false.

2. get() : Returns the value present in the Optional object.

ex:

Optional<String> opt = Optional.ofNullable(“Hello World”);

if(opt.isPresent()) {

String str = opt.get();

System.out.println(str);

}

3. ifPresent(Consumer<T> consumer) : Executes the given action, if the value is present in the Optional object. Otherwise, do’s nothing.

. This method accepts Consumer object as parameter.

. Consumer is a functional interface, so we can pass lambda expression.

ex:

Optional<String> opt = Optional.ofNullable(“Hello World”);

opt.ifPresent( str -> System.out.println(str) );

4. orElse(T another): if the Optional contains value, it returns that value. If it is empty then returns the another value provided as parameter.

ex1:

Optional<String> opt = Optional.ofNullable(“Hello world”);

String result = opt.orElse(“Bye world”);

System.out.println(result); //output: Hello world

ex2:

Optional<String> opt = Optional.empty();

String result = opt.orElse(“Bye world”);

System.out.println(result); //output: Bye world

5. orElseThrow(): If the Optional contains a value, then returns that value. If it is empty, then throws NoSuchElementException.

ex1:

Optional<String> opt = Optional.ofNullable(“Hello world”);

String result = opt.orElseThrow();

System.out.println(result); //output: Hello world

ex2:

Optional<String> opt = Optional.empty();

String result = opt.orElseThrow(); // throws NoSuchElementException

Java8 streams

* To systamatically organize a group of elements and to perform operations like, searching or sorting or add or remove etc.., then we use a Collection object.
* To process the elements of a Collection object, we need to iterate the collection with Iterator or we can use for each loop.
* If a Collection object has millions of elements, then processing the elements will take longer time.
* So, to solve this problem, Java8 has provided Stream interface.
* Collection and Stream are used for different purpose. Collection object is used for storing the elements and Stream object is used for processing the elements.

creating a Stream object:

1. we can a Stream for a Collection object.

Stream<Employee> stream = employeesList.stream();

2. we can create a Stream for an array.

Stream<Integer> stream = Arrays.stream(arr);

3. we can create a Stream object for raw values.

Stream<String> stream = Stream.of(“Tom”, “john”, “Jack”);

4. we can create an empty stream object.

Stream<Void> stream = Stream.empty();

* A stream operations are two types.

1.intermediate operations

2.terminal operations.

\* intermediate operation means, it transform a stream to another stream.

\* terminal operation means, which produces the result.

example:

Stream<Employee> stream1 = employeesList.stream();

Stream<Employee> stream2 = stream1.filter( e -> e.getSal() > 1000);

long count = stream2.count();

. Here, filter() is an intermediate operation and count() is a terminal operation.

. filter(), map(), sorted(), skip(), limit(), flatMap(), etc.. are intermediate operations.

. count(), max(), min(), collect(), reduce(), forEach(), etc.. are terminal operations.

ex1: print the employees of a list with salary > 25000

ArrayList<Employee> employeesList = **new** ArrayList<>();

employeesList.add(**new** Employee(7807, "Miller", 12799.0, "Male", 4.5));

employeesList.add(**new** Employee(7125, "Killer", 92999.0, "Male", 3.5));

employeesList.add(**new** Employee(7788, "Scott", 32400.0, "Male", 6.2));

employeesList.add(**new** Employee(7878, "Lisa", 22500.0, "Female", 4.9));

employeesList.add(**new** Employee(7296, "King", 56300.0, "Male", 9.5));

employeesList.add(**new** Employee(7705, "Mary", 87800.0, "Female", 11.5));

employeesList.add(**new** Employee(7856, "Pier", 48000.0, "Female", 5.3));

employeesList.add(**new** Employee(8611, "Smith", 19755.0, "Male", 5.2));

employeesList.add(**new** Employee(8278, "Sophia", 89555.0, "Female", 6.7));

//employeesList.stream().filter(e -> e.getSal() > 25000).forEach(e -> System.out.println(e));

Stream<Employee> stream = employeesList.stream();

Stream<Employee> stream2 = stream.filter(e -> e.getSal() > 25000);

stream2.forEach(e -> System.***out***.println(e));

ex2: print the highest paid employee of the list.

Optional<Employee> opt = employeesList

.stream()

.sorted((e1, e2) -> {

**if**(e1.getSal() < e2.getSal())

**return** 1;

**else** **if**(e1.getSal() > e2.getSal())

**return** -1;

**else**

**return** 0;

})

.findFirst();

opt.ifPresent(e -> System.***out***.println(e));

ex3: print the second highest paid employee of the list.

Optional<Employee> opt = employeesList

.stream()

.sorted((e1, e2) -> {

**if**(e1.getSal() < e2.getSal())

**return** 1;

**else** **if**(e1.getSal() > e2.getSal())

**return** -1;

**else**

**return** 0;

})

.skip(1)

.findFirst();

opt.ifPresent(e -> System.***out***.println(e));

ex4: print the employees of list by increasing the salary by 2000

employeesList

.stream()

.map( e -> {

e.setSal(e.getSal() + 2000);

**return** e;

})

.forEach(e -> System.***out***.println(e));

//Stream<Employee> stream = employeesList.stream();

//Stream<Employee> stream2 = stream.map( e -> {

// e.setSal(e.getSal() + 2000);

// return e;

// });

//stream2.forEach(e -> System.out.println(e));

ex5: print first 5 high salary employees.

employeesList.stream()

.sorted((e1, e2) -> {

**if**(e1.getSal() < e2.getSal())

**return** 1;

**else** **if**(e1.getSal() > e2.getSal())

**return** -1;

**else**

**return** 0;

})

.limit(5)

.forEach(System.out::println);

ex6: collect the names from a list to another list, if length > 3

List<String> lst1 = Arrays.*asList*("Tom", "Jeffry", "Jack", "Doe", "John", "Sophia", "Ema");

List<String> lst2 = lst1.stream()

.filter(str -> str.length() > 3)

.collect(Collectors.*toList*());

lst2.forEach(System.***out***::println);

ex7: print the sum of the numbers of a list.

List<Integer> lst = Arrays.*asList*(12, 45, 23, 74);

**int** total = lst.stream().collect(Collectors.*summingInt*(i -> i.intValue()));

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

parallelStream():



* If the source of the data (may be a collection object or an array) has huge elements, like 10 million elements, then a stream can process them, but it takes more time.
* So, we can use parallel stream for processing faster.
* parallel stream will divide the elements into data chunks and executes the operations on each chunk at different cpu core.
* After processing, it will join the results from each cpu core and returns the final output.
* So, parallel stream can utilize the full potential of the cpu.
* Just, we need to call parallelStream() method in place of stream() method.

ex:

List<Employee> lst2 = lst1.parallelStream()

.filter(e -> e.getSal() > 50000)

.collect(Collectors.toList());

Date/Time API

* Before Java8 version, we have java.util.Date and java.util.Calendar classes to work with data and time values.
* The problem with these classes is, they are mutable objects. So, in mulithreaded applications, if one thread changes the value, then it effects on other thread. This is a bug.
* The other issue is, the year starts from 1900, not from 0.
* For example, if I create,
* Date d = new Date(120, 0, 21);
* then, it represents the year 2020, not 120.
* So, Java 8 has provided, new classes to work with Date and Time values in java.time package.
* LocalDate class
* LocalTime class
* LocalDateTime class
* Period class

For example,

LocalDate today = LocalDate.now(); //The LocalDate object is

created with current system date.

LocalDate expDate = LocalDate.of(2025, 11, 30);

* LocalDate, LocalTime, and LocalDateTime classes objects are immutable objects. So, if you make any changes then new objects are created with results.

ex:

LocalDate date1 = LocalDate.of(2025, 1, 31);

LocalDate date2 = date1.plusDays(3);

S.o.p(date1); //output: 2025-01-31

S.o.p(date2); //output: 2025-02-3

Note: you have other methods like,

plusMonths(), plusYears(), plusWeeks(), minusDays(), minusMonths(), minusYears(), minusWeeks().

LocalTime currentTime = LocalTime.now(); //system time

LocalTime endTime = LocalTime.of(18, 30, 45);

* Suppose, if you want to find the difference between two dates in terms of days then,
* LocalDate d1 = LocalDate.*of*(2024, 8, 15);
* LocalDate d2 = LocalDate.*of*(2025, 2, 18);
* **long** days = ChronoUnit.***DAYS***.between(d1, d2);
* System.***out***.println("difference in days : " + days);
* suppose, if you want to find the difference between two times, interm of minutes then,

LocalTime t1 = LocalTime.*of*(9, 35, 40);

LocalTime t2 = LocalTime.*of*(19, 15, 39);

**long** minutes = ChronoUnit.***MINUTES***.between(t1, t2);

System.***out***.println("difference in minutes : " + minutes);

* Period class compares the given two dates based on the values of dates, months and years. It means, it will not provide the as the difference in total no of days or total months or total years.

LocalDate d1 = LocalDate.of(2024,02, 20);

LocalDate d2 = LocalDate.now();

Period p = Period.between(d1, d2);

S.o.println(p.getDays()); //output: 0

S.o.println(p.getMonths()); //output: 0

S.o.println(p.getYears()); //output: 1

Base64 Encoding and Decoding:

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

* Base64 class is provided in Java8, under java.util package for encoding and decoding the data in Base64 format.
* Base64 is an encoding scheme that converts binary data(bytes) into encoded string(ASCII string).
* Base64 class has provided two static inner classes called Encoder and Decoder, where Encoder class converts binary data to encoded string and Decoder class will do the reverse.
* Encoding and Decoding are concepts are used for data storage, security and transmission.

**package** com.pack;

**import** java.util.Base64;

**import** java.util.Base64.Decoder;

**import** java.util.Base64.Encoder;

**public** **class** Base64Example {

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

Encoder encoder = Base64.*getEncoder*();

String str = "hello@world";

String encoded = encoder.encodeToString(str.getBytes());

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

System.***out***.println("encoded string : " + encoded);

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

Decoder decoder = Base64.*getDecoder*();

**byte**[] byts = decoder.decode(encoded);

String original = **new** String(byts);

System.***out***.println("encoded string : " + encoded);

System.***out***.println("original : " + original);

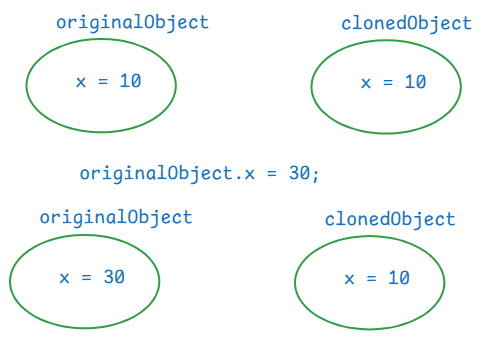
}

}

Cloning

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

* cloning is a process of creating a duplicate object from an existing object.
* Suppose, object creation for a class is a complex task, like the constructor of the class has a huge code and the objects to be created are the complex objects, then creating multiple such complex objects will be a time taken process.
* So, Java has provided a solution called Cloning.
* First we create one object for a class, then we clone it to create multiple duplicate objects.
* Once an object is cloned, the original object and the cloned object acts as two different objects.
* So, if you make any changes to original object, it doesn’t refect on cloned 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 your 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:

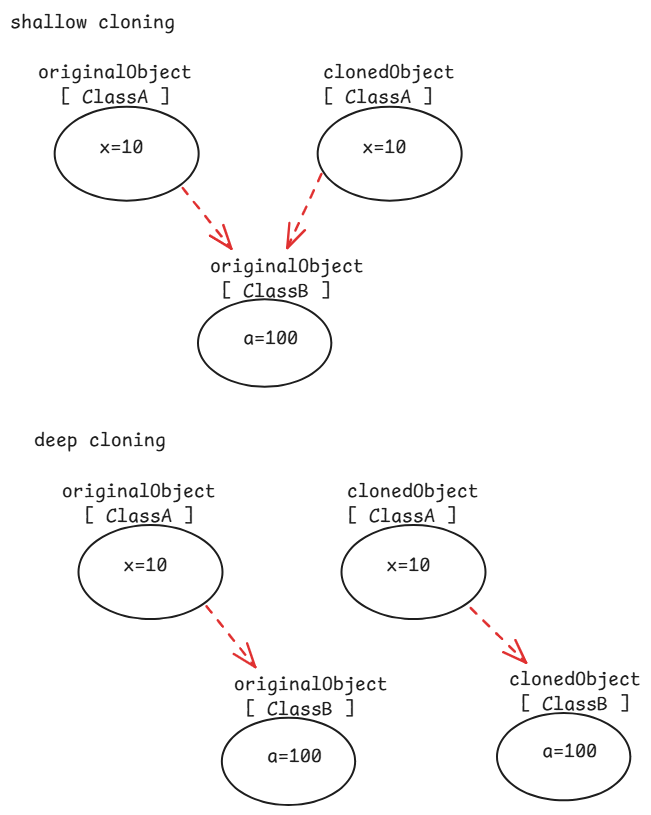
1. shallow cloning
2. deep cloning

. 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]]

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

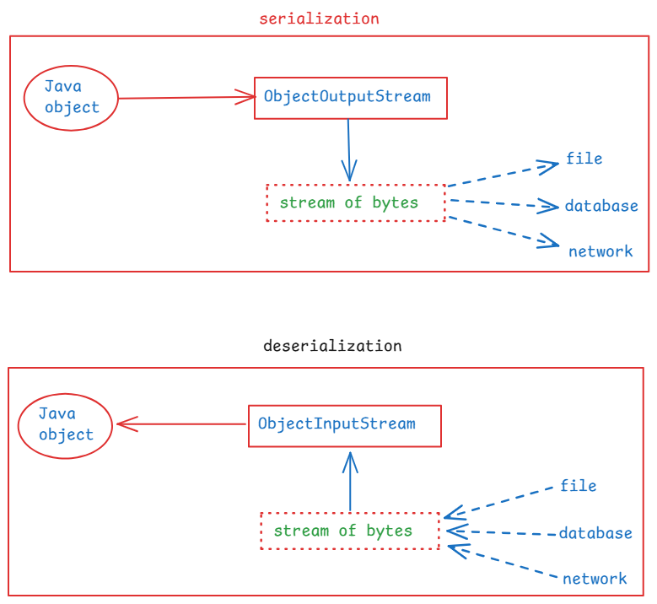
After making changes to the dependency object, through original dependent object

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

}