Siddhi Institute

Java & SQL Foundations

*by*

Srikanth Pittala Sr. Automation Engineer

Java

INDEX

1. data types
2. Type conversation
3. Polymorphism
4. Abstract class
5. Interface
6. Inheritance
7. Encapsulation
8. Conditional statements
9. Loop statements
10. Keywords
11. Access modifiers
12. Package
13. Exception
14. Difference between throw & throws
15. This & super keywords
16. Athematic & logical operation
17. Garbage in java
18. Global variables & local variables
19. Static , non Static & return type methods
20. difference between arguments & parameters
21. List & ArrayList
22. Set
23. HashSet, HashMap, HashTable
24. Enums
25. Advantages & disadvantages of Java
26. Jar

# Introduction to Java & Language Basics

1. Introduction to Java -

James Gosling with few others initiated the Java language project in June 1991. Initially called Oak, later by Green and finally Java.

JDK 1.0 (January 21, 1996)

...

Java SE 6 (December 11, 2006)

Java SE 7 (July 28, 2011)

Java SE 8 (March 18, 2014)

A computer programming language that is intended to let application developers "write once, run anywhere" (WORA) meaning that a compiled Java program can run on all platforms that support Java. Java applications are typically compiled to "byte code" that can run on any "Java Virtual Machine" regardless of computer architecture.

1. Automatic Memory Management -

Java uses an automatic garbage collector; it is responsible for recovering the memory when once objects are no longer in use (unreachable objects). Java doesn't support C/C++ style pointer arithmetic.

1. Java Software Installation -
   1. Download and install JDK (JavaSE8) software
   2. Download Eclipse IDE (Search for Eclipse in Google)
2. Your First Java Program -
   1. Manual Approach
      1. Open Notepad
      2. Type the below program

public class First {

public static void main(String [] args) { System.out.println("Hello World");

}

}

* + 1. Save the above file as First.java in some folder e.g. c:\examples
    2. Locate java.exe and javac.exe in your machine

e.g. C:\Program Files\Java\jdk1.8.0\_91\bin

* + 1. Open cmd
    2. cd \example
    3. Set PATH, it enables OS to locate javac and java.

set PATH="C:\Program Files\Java\jdk1.8.0\_91\bin";%PATH%

* + 1. Compile the program

javac First.java

* + 1. Output will be a .class file for each class in the program. e.g.

First.class

[use "dir" command to list the directory]

* + 1. Set CLASSPATH to let Java runtime identify where the class files are located.

e.g. set CLASSPATH=c:\examples

* + 1. Submitting the class file for execution

e.g. java First

Here "First" is the name of the class which contains main method.

* 1. Using Eclipse
     1. Download and Install Eclipse
     2. Open Eclipse
     3. It will prompt you for workspace; i.e. a folder to place the projects.
     4. Open File Menu => Other => Select "Java Project" => Click "Next"
     5. Project Name => Type "HelloWorld" => Click "Finish"
     6. Expand "HelloWorld" from "PackageExplorer" => Choose "src" folder => Right click => Select "New" => "Class"
     7. Name = "First"; Select "public static void main(String[] args)"
     8. Select Finish
     9. Open First.java; complete the program as specified below.

public class First {

public static void main(String[] args) { System.out.println("Hello World!");

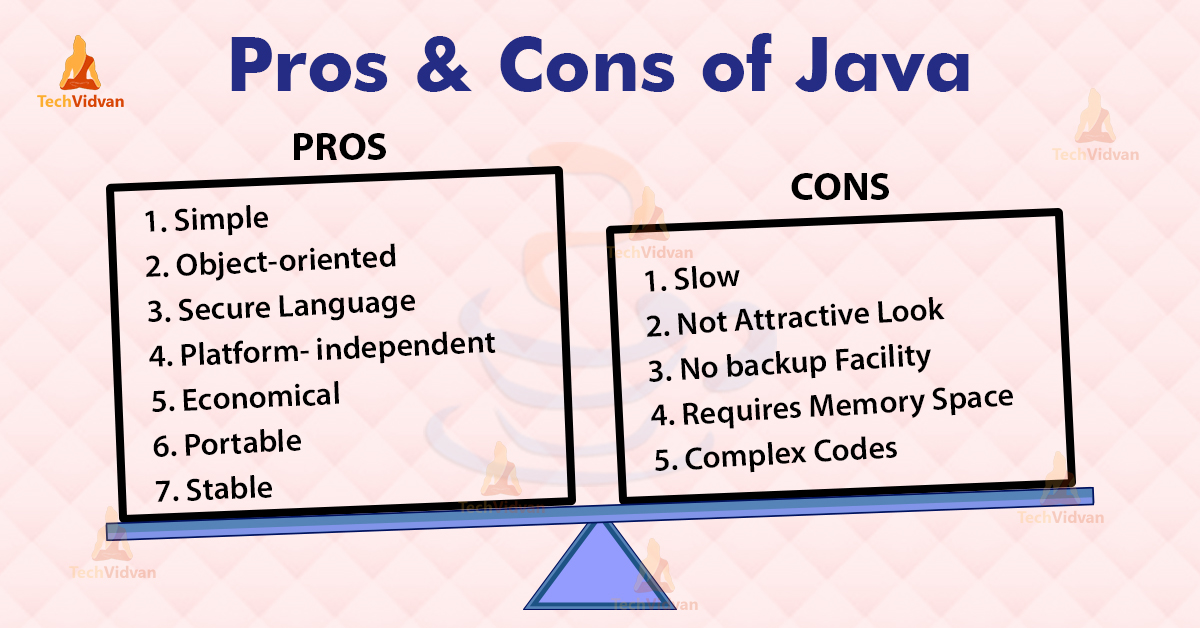
}

}

* + 1. Select First.java => Right click => Run As => "Java Application"
    2. Find the below output in Console view; (if not found; view can be selected through "Window Menu" => "Show View" => "Console").

Hello World!

**Advantages and disadvantages of java.**



Variables:

* Variable is name of memory location to store values.
* We use variable to store data and use the data once it is required.

Types of variables :

* There are three types of variables in Java:

1. local variable
2. instance variable
3. static variable

1) Local Variable

A variable declared inside the body of the method, constructor or block is called local variable.

They are created once method, constructor or block is invoked.

You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

It is mandatory to initialize the local variable before using otherwise we will not be able to compile the code.

Note : A local variable cannot be defined with "static" keyword.

2) Instance Variable / Global Variable

A variable declared inside the class but outside the body of the method, constructor or block is called an instance variable. It is not declared as static.

If you don't provide any values then java will assign default values for them.

It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

Example:

**public** **class** A

{

**static** **int** *m*=100;//static variable

**void** method()

{

**int** n=90;//local variable

}

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

{

**int** data=50;//instance variable

}

}

Datatypes -

Java is strictly a typed programming language, where in java applications before representing data first we have to confirm which type of data we are representing. In this context, to represent type of data we have to use "data types".

EX: i = 20; > invalid, no data type representation.

int i=20;--> Valid, type is represented then data is represented.

In java applications , data types are able to provide the following advantages.

1.We are able to identify memory sizes to store data.

EX: int i=10;--> int will provide 4 bytes of memory to store 10 value.

2.We are able to identify range values to the variable to assign.

EX: byte b=130; > Invalid

byte b=125; > Valid

Reason: 'byte' data type is providing a particular range for its variables like -128 to 127, in this range only we have to assign values to byte variables.

To prepare java applications, JAVA has provided the following data types.

1.Primitive Data Types / Primary Data types

Numeric Data Types

Integral data types/ Integer Data types:

byte ------> 1 bytes > 0

short------> 2 bytes > 0

int--------> 4 bytes > 0

long-------> 8 bytes > 0

Non-Integral Data Types:

float------> 4 bytes > 0.0f

double-----> 8 bytes > 0.0

Non-Numeric Data types:

char ---------> 2 bytes > ' ' [single space]

boolean-------> 1 bit > false

2.User defined data types / Secondary Data types

All classes, all abstract classes, all interfaces, all arrays, string,......

No fixed memory allocation for User defined data types

To identify "min value" and "max value" for each and every data type, JAVA has provided the following two constant variables from all the wrapper classes.

MIN\_VALUE and MAX\_VALUE

Note: Classes representation of primitive data types are called as Wrapper Classes

Primitive Data Types Wrapper Classes

byte java.lang.Byte

short java.lang.Short

int java.lang.Integer

long java.lang.Long

float java.lang.Float

double java.lang.Double

char java.lang.Character

boolean java.lang.Boolean

EX:

class Test{

public static void main(String[] args)

{

System.out.println(Byte.MIN\_VALUE+" >"+Byte.MAX\_VALUE);

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

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

System.out.println(Long.MIN\_VALUE+" >"+Long.MAX\_VALUE);

System.out.println(Float.MIN\_VALUE+" >"+Float.MAX\_VALUE);

System.out.println(Double.MIN\_VALUE+" >"+Double.MAX\_VALUE);

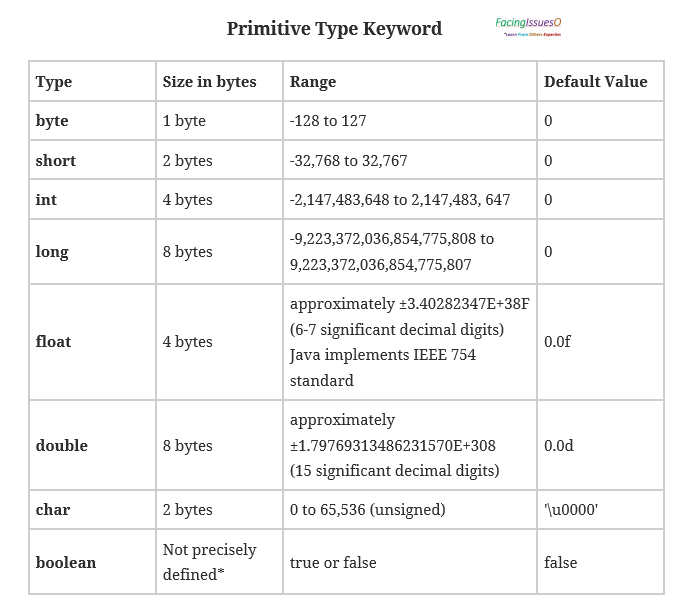
System.out.println(Character.MIN\_VALUE+" >"+Character.MAX\_VALUE);

//System.out.println(Boolean.MIN\_VALUE+"----->"+Boolean.MAX\_VALUE); -

--> Error

}

}



1. Local variable and Global variable

* Variable which are declared inside the block of method are called as local variables

Note : • Local variable contain garbage values.

* Variable both static & non-static which are declared outside the method block & inside the class block are called global variables

Note: Global variable contains null value/default values.

1. Literals -

Literal is a constant assigned to the variables .

EX:

int a=10;

int > data types

a > variables/ identifier

= > Operator

10 > constant[Literal].

; > Special symbol.

To prepare java programs, JAVA has provided the following set of literals.

1.Integer / Integral Literals:

byte, short, int, long ----> 10, 20, 30,....

char -----> 'A','B',.....

2.Floating Point Literals:

float ----> 10.22f, 23.345f,.....

double----> 11.123, 456.345,....

3.Boolean Literals:

boolean > true, false

4.String Literals:

String ---> "abc", "def",......

Note: JAVA7 has given a flexibility like to include '\_' symbols in the middle of the literals in order to improve readability.

EX:

float f=12345678.2345f; float f=1\_23\_45\_678.2345f;

If we provide '\_' symbols in the literals then compiler will remove all '\_' symbols which we provided, compiler will reformate that number as original number and compiler will process that number as original number.

6.1) Keywords

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

If any predefined word having both word recognization and internal functionality then that predefined word is called as Keyword.

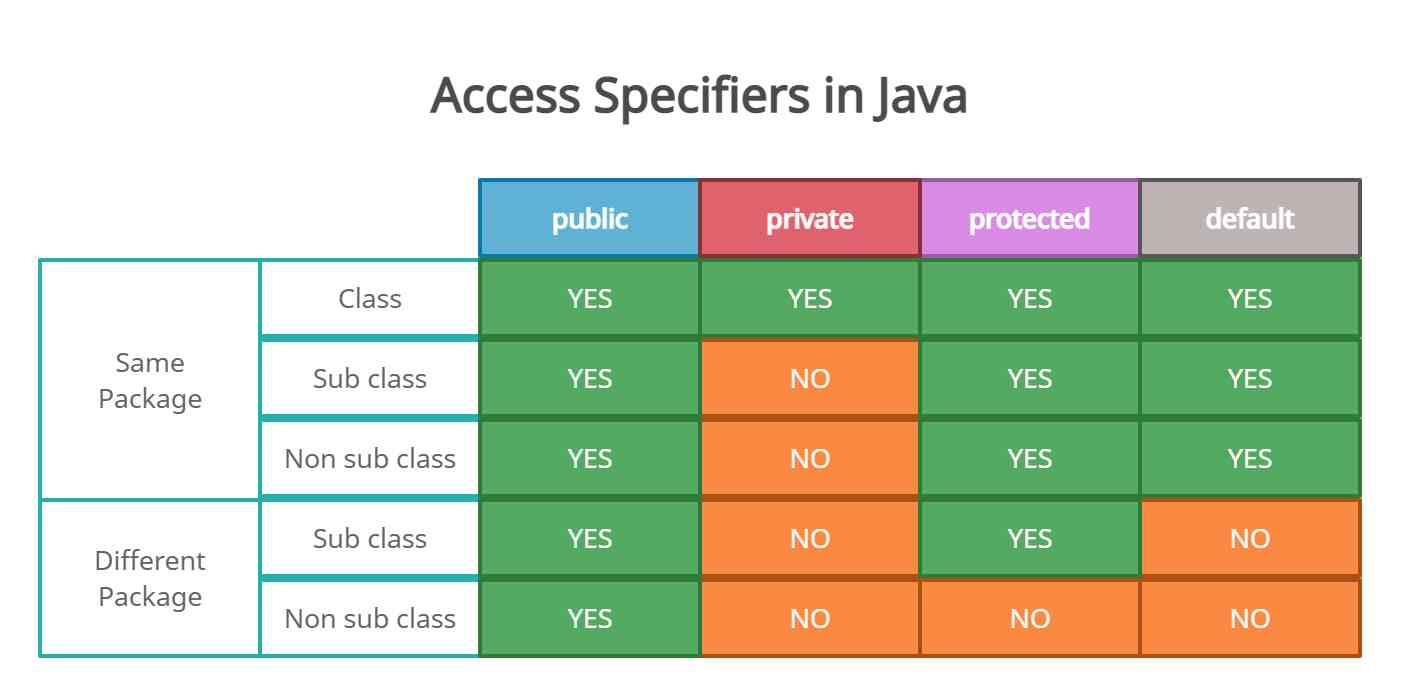
To prepare java applications, Java has provided the following list of keywords.

Data types and Return types:

bytes, short, int, long, float, double, char, boolean, void.

Access Modifiers:

public, protected, private, static, final, abstract, native, volatile, transient, synchronized, strictfp,.....



Protected

* The protected access modifier is accessible within package and outside the package but through inheritance only.
* The protected access modifier can be applied on the data member, method, and constructor. It can't be applied on the class.
* It provides more accessibility than the default modifier.

Default keyword:

A Java default keyword is an access modifier. **If you didn't assign any access modifier to variables, methods, constructors and, classes, by default, it is considered as default access modifier.**

**Points to remember**:

* The default access modifier is accessible within the package only.
* Unlike private and protected, we can create a default outer class by not assigning any access modifier to it. In such a case, it not restricted to take class name similar to a program name.
* If you are overriding any method, overridden method (i.e., declared in the subclass) must not be more restrictive. So, the default method or variable can't be allowed to use in private access modifier.

Practice below:

1. example to determine whether the default variable is accessible or not outside the package.
2. example to determine whether the default variable is accessible or not outside the class within the package.
3. example to determine whether the default method is accessible or not outside the package.
4. example to determine whether the default method is accessible or not outside the package using inheritance.
5. example to determine whether we use default outer class.
6. example to determine whether we create the instance of default constructor from outside the class.
7. example to determine whether the default method is overridden to sub-class using default access modifier.
8. example to determine whether the default method is overridden to sub-class using private access modifier.
9. example to determine whether the default method is overridden to sub-class using default access modifier.
10. example to determine whether the default method is overridden to sub-class using public access modifier.

Note Ref: <https://www.javatpoint.com/default-keyword-in-java>

Final Keyword:

* The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

* The final keyword can be applied with the variables, a final variable that have no value it is called **blank final variable or uninitialized final variable**. It can be initialized in the **constructor** only. The blank final variable can be **static** also which will be initialized in the static block only.

Practice below:

1. If you make any variable as final, you cannot change the value of final variable(It will be constant).
2. If you make any method as final, you cannot override it.
3. If you make any class as final, you cannot extend it.
4. Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it

1. Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable. It can be initialized only in constructor.

1. A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block
2. If you declare any parameter as final, you cannot change the value of it.
3. Q) Can we declare a constructor final?

No, because constructor is never inherited.

Note Ref: <https://www.javatpoint.com/final-keyword>

Flow Controllers:

if, else, switch, case, default, for, while, do, break, continue, return,....

Class/ Object Related:

class, enum, extends, interface, implements, package, import, new, this, super,....

Exception Handling Related Keywords:

throw, throws, try, catch, finally

6.3) JAVA Naming Conventions

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

Java is a case sensitive programming language, where in java applications, there is a separate recognization for lower case letters and for upper case letters.

To use lower case letters and Upper-case letters separately in java applications, JAVA has provided the following conventions.

All class names , abstract class names, interface names and Enum names must be started with upper case letter and the sub sequent symbols must also be upper case letters.

EX:

String StringBuffer

InputStreamReader

All java variables must be started with lower case letters, but the subsequent symbols must be upper case letters.

EX:

in, out, err

pageContext, bodyContent

All java methods must start with lower case letter , but the sub sequent symbols must be upper case letters

EX:

concat(--)

forName(--) getInputStream()

All java Constant variables must be provided in Upper Case letters.

EX:

MIN\_PRIORITY NORM\_PRIORITY MAX\_PRIOITY

All java package names must be provided in lower case letters

EX:

java.util java.lang.reflect javax.servlet.jsp.tagext

Note: All the above conventions are mandatory for predefined library, they are optional form User defined library, but suggestible.

EX:

String str=new String("abc"); > Valid

string str=new string("abc"); > Invalid

class Employee{ > Valid and Suggestible}

class student{ > valid, but, not suggestible}

1. Operators -

Arithmetic Operators :-

+ - / \* %

NOTE : % can be applied to floating types in java

NOTE: If numerator is less than denominator then % will give output as the numerator only.

Technical Notes -

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

* 1. Result sign during % operation; sign of the numerator is the sign of the result. 11 % 3 => 2

-11 % 3 => -2

11 % -3 => 2

-11 % -3 => -2

* 1. % Is also applicable for floating point operation 6.25 % 2.5 => 1.25

divides up to integral multiple i.e. 2 and the remainder is the result.

Operator Precedence and associativity -

Priority order -

1) / , \* , %

2) + , -

Evaluation of operator depends on its priority. If more than one operator with equal priority contends then whichever occurs first from the left will be evaluated(if it is left to right associativity).

Example 1 -

x = a \* b / c \* d + e % f; 3 1 1 1 2 1

first.

All 1s here are contending; hence whichever occurs first from the left will be evaluated

* 1. R1 <= a \* b R2 <= R1 / c R3 <= R2 \* d R4 <= e % f

R5 <= R3 + R4 R6 <= x = R5

Example 2 -

x = a \* b / (c \* d) + e % f; 4 2 2 1 2 3 2

Parenthesis has higher priority e.g.

R1 <= c \* d R2 <= a \* b R3 <= R2 / R1 R4 <= e % f

R5 <= R3 + R4 R6 <= x = R5

Sometimes associativity could be right to left for some operators e.g. = (assignment) a = b = c = 3;

1 1 1

Here associativity is right to left for assignment ie.

R1 <= c = 3 R2 <= b = R1 R3 <= a = R2

Note - Assignment operator returns the value which is assigned.

increment and decrement operators

++ , --

post increment :- var++ pre increment :- ++var

Note: It must be applied to variables and not to constants i.e. 5++ is an error.

++a (or) a++ means increment a by 1.

b = a++;

* + 1. assign the value of a in the expression
    2. increment a by 1.

b = ++a;

1. increment a by 1
2. assign the value of a in the expression

Example -

public class Main {

public static void main(String[] args) { int a, b, c;

a = 5; a++;

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

a = 5;

b = a++; // b is 5; a is 6 System.out.println(

"a is " + a + " b is " + b);

a = 5;

c = ++a; // a is 6; c is 6 System.out.println(

"a is " + a + " c is " + c);

}

}

Similar is the case with "--" decrement operator.

Relational Operators :-

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

expression.

These operators return boolean value which is either true or false based on the

e.g.

int a = 10 , b = 5; boolean c = a < b;

System.out.println( c ); // false.

Logical Operators :-

&& , || , !

They operate on the logical values of the given expression. e.g.

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

c = (a > 9) && (b > 6);

// Evaluates to false as b > 6 is false.

Technical Notes -

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

**& Vs &&**

In the case of Logical-AND(&&) operator, if the first operand value is false then it is not required to check second operand value, directly, we can predict the result of overall expression is false.

In the case of '&' operator, even first operand value is false , still, JVM evaluates second operand value then only JVM will get the result of overall expression is false, here evaluating second operand value is unnecessary, it will increase execution time and it will reduce application performance.

In the case of '&&' operator, if the first operand value is false then JVM will get the overall expression result is false without evaluating second operand value, here JVM is not evaluating second operand expression unnecessarily, it will reduce execution time and it will improve application performance.

Note: If the first operand value is true then it is mandatory for JVM to evaluate second operand value in order to get overall expression result.

EX:

class Test

{

public static void main(String[] args)

{

int a=10;

int b=10;

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

{

}

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

int c=10;

int d=10;

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

{

}

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

}

}

1. && operator doesn't evaluate second expression if first expression evaluates to false.
2. || operator doesn't evaluate second expression if first expression evaluates to true.

e.g. int a = 10, b = 5; boolean c;

c = (a++ > 10) && (b++ > 5);

What are the values of a, b and c? a = 11

b = 5

[Since second expr is not evaluated] c = false

1. Be extra cautious while using ! operator as it is associated with the immediate expression. e.g.

int a = 10 , b = 20 ; boolean c ;

c = ! a > b;

// **Error** invalid expression as ! is associated with variable `a` and not `a > b`. correction :-

c = ! (a > b) ;

Ternary or Conditional Operator -

It uses three operands hence the name Ternary operator. It can be thought of as an if-then-else statement.

Usage -

conditionalExpression ? expr1 : expr2

If conditionalExpression evaluates to true then expr1 will be evaluated, otherwise expr2.

Example -

public class Main {

public static void main(String[] args) { int a = 10 , b = 20 , c;

c = (a < b) ? a : b ; System.out.println("value of c is " + c);

}

}

Value of c is 10 as (a < b) is true.

1. Type Conversions -

The process of converting data from one data type to another data type is called as "Type Casting".

There are two types of type castings are existed in java.

1.Primitive data Types Type Casting

2.User defined Data Types Type Casting

Note: To perform User defined data types of type casting we need either "extends" relation or "implements" relation between user defined data types.

1. Primitive data Types Type Casting:

The process of converting data from one primitive data type to another primitive data type is called as Primitive data types type casting.

There are two types of primitive data types type castings.

1. Implicit Type Casting
2. Explicit Type Casting

**Implicit Type Casting:**

The process of converting data from lower data type to higher data type is called as Implicit Type Casting.

To cover all the possibilities of implicit type casting JAVA has provided the following chart.

Chart

Description automatically generated with medium confidence

If we want to perform implicit type casting in java applications then we have to assign lower data type variables to higher data type variables.

EX:

byte b=10; int i = b;

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

Status: No Compilation Error

OUTPUT: 10 10

If we compile the above code, when compiler encounter the above assignment statement then compiler will check whether right side variable data type is compatible with left side variable data type or not, if not, compiler will rise an error like "possible loss of precision". If right side variable data type is compatible with left side variable data type then compiler will not rise any error and compiler will not perform any type casting.

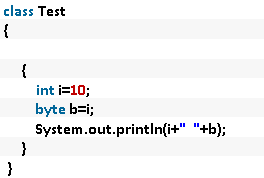
When we execute the above code, when JVM encounter the above assignment statement then JVM will perform the following two actions.

1.JVM will convert right side variable data type to left side variable data type implicitly[Implicit Type Casting]

2.JVM will copy the value from right side variable to left side variable.

Note: Type Checking is the responsibility of compiler and Type Casting is the responsibility of JVM.

EX2:



Status: Compilation Error, Possible loss of precision.

EX3:

Graphical user interface, text, application, email

Description automatically generated

Status: Compilation Error

EX4:

Graphical user interface, text, application, email, Teams

Description automatically generated

Status: Compilation Error, Possible loss of precision.

Reason: byte and short internal data representations are not compatible to convert into char.

EX5:

Graphical user interface, text, application, Teams

Description automatically generated

Status: No Compilation Error

OUTPUT: A 65

Reason: Char internal data representation is compatible with int.

EX6:

Graphical user interface, text, application

Description automatically generated

Status: Compilation Error, possible loss of precision.

Reason: When we assign a value to a variable of data type, if the value is greater the max limit of the left side variable data type then that value is treated as of the next higher data type value.

Note: For both byte and short next higher data type is int only.

EX7:

Graphical user interface, text, application

Description automatically generated

Status: Compilation Error, possible loss of precision.

EX8:

Graphical user interface, text, application

Description automatically generated

Status: Compilation Error, Possible loss of precision.

Reason: X,Y and Z are three primitive data types.

X+Y=Z

1. If X and Y belongs to {byte, short, int} then Z should be int.

2. If either X or Y or both X and Y belongs to {long, float, double} then Z should be higher(X,Y).

byte+byte=int

byte+short=int

short+int=int

byte+long=long

long+float=float

float+double=double

EX9:

Graphical user interface, text, application, Teams

Description automatically generated

Status: No Compilation Error

OUTPUT: 10 10.0

EX10:

Graphical user interface, text, application

Description automatically generated

Status: Compilation Error, possible loss of precision.

Reason:

two class rooms

Class Room A: 25[Size] banches---> 3 members per bench---> 75

Class Room B: 50[Size] banches---> 1 member per bench > 50

long---> 8 bytes--> less data as per its internal data arrangement. float--> 4 bytes--> more data as per its internal data arrangement.

Due to the above reason, float data type is higher when compared with long data type so that, we are able to assign long variable4 to float variable directly, but, we are unable to assign float variable to long variable directly.

**Explicit Type Casting:**

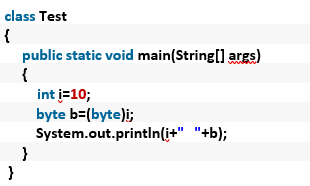
The process of converting data from higher data type to lower data type is called as Explicit Type Casting.

To perform explicit type casting we have to use the following pattern. P a = (Q) b;

(Q)  Cast operator

Where P and Q are two primitive data types, where Q must be either same as P or lower than P as per implicit type casting chart.

EX1:



Status: No Compilation Error

OUTPUT: 10 10

When we compile the above code, when compiler encounter the above assignment statement, compiler will check whether cast operator provided data type is compatible with left side variable data type or not, if not, compiler will rise an error like "Possible loss of precision". If cast operator provided data type is compatible with left side variable data type then compiler will not rise any error and compiler will not perform type casting.

When we execute the above program, when JVM encounter the above assignment statement then JVM will perform two actions.

1.JVM will convert right side variable data type to cast operator provided data type. 2.JVM will copy value from right side variable to left side variable.

EX2:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Status: No Compilation Error

OUTPUT: 10 10

EX3:

Graphical user interface, text, application

Description automatically generated

Status: No Compilation Error

OUTPUT: 65 A

EX4:

Graphical user interface, text, application, Teams

Description automatically generated

Status: No Compilation Error

OUTPUT: A 65

Note: In Implicit type casting, conversions are not possible between char and byte, short and char, but, in explicit type casting conversions are possible in between char and byte, char and short, why because, explicit type casting is forcible type casting.

EX5:

Graphical user interface, text, application, email

Description automatically generated

Status: Compilation Error.

EX6:

Graphical user interface, text, application

Description automatically generated

Status: Compilation Error

EX7:

Graphical user interface, text, application

Description automatically generated

Status: No Compilation Error

OUTPUT: 60

EX8:

Graphical user interface, text, application, email

Description automatically generated

Status: No Compilation Error

OUTPUT: 22

EX9:

Graphical user interface, text, application

Description automatically generated

Status: No Compilation Error

OUTPUT: -126

1. Control flow statements -

Control flow statements break up the flow of execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code.

Various control flow statements supported by Java

* + 1. If
    2. Switch
    3. While
    4. do while
    5. for

1. if

Enables you to execute a block of code only if a particular test expression evaluates to true. It can be used in combination with `else` to execute a block which should be evaluated in case if it is false.

Example -

* 1. if (dayOfWeek == SUNDAY)

{

Enjoy the holiday !!!

}

* 1. if (dayOfWeek != SUNDAY)

{

Be energetic at work !!

}

else

{

Enjoy the holiday !!!

}

Example -

import java.util.Scanner; public class Main {

public static void main(String[] args) { int a, b;

Scanner scan = new Scanner(System.in); System.out.println("Enter a");

a = scan.nextInt();

System.out.println("Enter b"); b = scan.nextInt();

if (a > b)

{

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

}

else

{

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

}

}

}

switch -

Switch statement can have several execution paths. Deciding whether to use if-then- else statements or a switch statement is based on readability and the conditional expression. The break statements are necessary because without them, statements in switch blocks fall through: All statements after the matching case label are executed in sequence, regardless of the expression of subsequent case labels, until a break statement is encountered or till the end of the switch.

Example -

* 1. int digit = ;

switch(digit) {

case 1 : System.out.println("one"); break;

case 2 : System.out.println("two"); break;

...

}

* 1. int digit = ;

switch(digit) {

case 1 : System.out.println("one"); case 2 : System.out.println("two"); case 3 : System.out.println("three");

break;

case 4 : System.out.println("four");

default: System.out.println("Not in 1,2,3,4"); break;

}

Output when digit = 1

one two three

Output when digit = 2

two three

Output when digit = 4

four

Not in 1,2,3,4

1. while -

The while statement continually executes a block of statements while the given conditional expression evaluates to true.

Usage -

while (expression) statement

Example -

public class Main {

public static void main(String[] args) { int n = 1; System.out.println("Before while"); while (n <= 5) {

System.out.println(" Hello "); n++;

}

System.out.println("Done");

}

}

Eclipse Tip (Tracing the above program in debug) -

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

1. Put a break point after the declaration of n. i.e. click on the left vertical bar of the code window. You should see a circle.
2. Right click on the program and choose Debug As menu and select Java application.
3. It asks for debug perspective choose yes
4. Use F6 (Step Over) to trace.

d) do..while -

The do...while statement continually executes a block of statements while the given conditional expression evaluates to true. Difference between while and do..while is that in case of do..while the block is executed atleast once.

Usage -

do

block;

while (conditionalExpression);

Example -

int n = 1; do {

System.out.println("Value of n is " + n); n++;

} while (n <= 5);

f) for -

Executes a block of statements while the given conditional expression evaluates to true.

Usage -

for (initialization; conditionalExpression; increment ) block

Example -

for (int n=1; n <= 5; n++) { System.out.println("Value of n is " + n);

}

1. Branching Statements -
2. break statement -

Its purpose is to terminate the execution of the statement with in which it is used. It statement has two forms labeled and unlabeled. It could be used with in switch, while, do..while, for statements. If used with label it will terminate the execution of the corresponding statement identified with the given label.

Example -

* 1. Unlabeled usage -

public class Main {

public static void main(String[] args) { int n = 1;

while (n <= 10) { System.out.println(n); n++;

if (n > 5) {

break;

}

}

System.out.println("Done");

}

}

Prints the values from 1 .. 5 Done

* 1. Labeled usage -

public class Main {

public static void main(String[] args) { int i=1, j;

abc:

while (i <= 3) { j = 1;

while (j <= 4) {

System.out.println(

" i = " + i + " j = " + j);

j++;

if (i == 3 && j == 2) { break abc;

}

} i++;

}

}

}

1. continue statement -

The continue statement skips the current iteration of a for, while , or do-while loop.

Like `break` it can be used with or with out label.

Example -

* 1. Unlabeled usage -

public class Main {

public static void main(String[] args) { int n;

for (n = 1; n <= 10; n++) { if (n % 2 == 0) {

continue;

}

System.out.println(n);

}

}

}

Prints the values from 1 3 5 7 9

* 1. Labeled usage -

Similar to break.

1. return statement -

The return statement exits from the current method, and control flow returns to where the method was invoked. It has two forms with value and with out value. With out value should be used with functions whose return type is void.

Examples -

* 1. int mul(int a, int b) { return a\*b;

}

* 1. void test(int x) {

if (x == 5) { return;

}

// do something.

}

1. Arrays -

An array is a container for a collection of homogeneous elements. The length of the array is established when the array is created.

Single Dimensional Arrays in Java -

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

Usage -

Datatype [] identifier;

Example -

int [] arr;

Here 'arr' is a reference variable to an array of integers.

Memory Allocation -

Memory allocation can be done using `new` operator.

Example -

arr = new int[10];

Allocates an array of 10 integer variables and returns the reference to it.

Initialization -

Elements created using new will contain zero in case of primitive types and

`null` in case of reference types. It could be done as illustrated in the below examples.

Example -

int [] arr = {10,20,30};

OR

int [] arr;

arr = new int[] {10,20,30};

Accessing elements of an Array -

An Element of an array could be accessed using its index. Index starts with 0 (zero). i.e. Using the above example

System.out.println( arr[0] ); // 10 System.out.println( arr[1] ); // 20 System.out.println( arr[2] ); // 30 System.out.println( arr[3] );

// Leads to "ArrayIndexOutOfBoundsException"

length property -

The length of an array could be determined using its `length` property. It returns the size of the array. i.e. Using the above example

int n = arr.length; // Returns 3

Example -

import java.util.Scanner; public class Main {

public static void main(String[] args) { int a[] = {10, 20, 30, 40, 50, 10, 20};

int ele;

Scanner scan = new Scanner(System.in);

System.out.println("Enter element to search"); ele = scan.nextInt();

for(int i = 0; i < a.length; i++) { if (a[i] == ele) {

System.out.println( "found at ­ " + i);

}

}

}

}

Technical Notes -

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

* 1. Consider the below declarations,

int a[], b;

Here `a` a reference variable to an integer array,

`b` is simply an integer variable.

int [] a, b;

Here both `a` and `b` are reference variables to an integer array.

\*2) Enhanced for loop for Arrays -

Simplifies looping through arrays or collections. Below example illustrates iterating through an array of integer elements.

e.g.

int a[] = {10, 20, 30, 40};

// You can read it as "for each element in a". for (int ele : a) {

System.out.println( ele );

}

Example -

import java.util.Scanner; public class Main {

public static void main(String[] args) { int a[];

int n;

Scanner scan = new Scanner(System.in);

System.out.println("Enter the size of the array"); n = scan.nextInt();

// Allocate the memory to hold the values. a = new int[n];

// Read the values into the array. for(int i=0; i < n; i++) {

System.out.println(

"Enter the value for a[" + i + "] :"); a[i] = scan.nextInt();

}

// Print the values of the array.

System.out.println(

"Values in the array are ");

// You can read it as for each integer value in a. for (int value : a) {

System.out.println(value);

}

// int big = Integer.MIN\_VALUE; int big = a[0];

for (int i=0; i < a.length; i++) { if (a[i] > big) {

big = a[i];

}

}

System.out.println("Biggest of the array : "+ big);

}

}

Passing arrays to functions -

In Java you are not passing address of any variable, instead if the argument is a reference variable then you need to remember that you are passing the reference hence any changes made to the original object using the reference will affect the calling function. As in the below case.

public class Main {

public static void main(String[] args) { int arr[] = {10, 20, 30}; change(arr);

System.out.println("Values in the array"); for (int value : arr) {

System.out.println(value); // 15 25 35

}

}

static void change(int [] a) {

for (int i=0; i < a.length; i++) { a[i] = a[i] + 5;

}

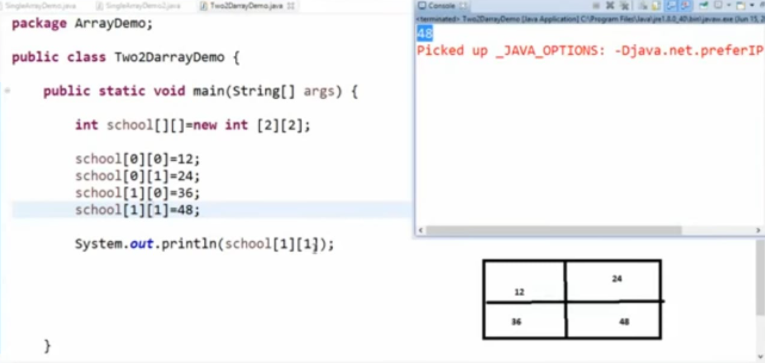
}

}

2 Dimensional Arrays in Java -

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

Example:



# Object Oriented Programming

**Main Class Section:**

Main Class is a java class ,it includes main() method.

The main intention of main() method is,

1. To manage application logic which we want to execute by JVM directly we have to use main() method.

2. To define starting point and ending point to the application execution we have to use main() method.

Syntax:

public static void main(String[] args){

----instructions----

}

Note: main() method is a conventional method with fixed prototype and with user defined implementation part.

Classes and Objects -

A class is a template for creating an object. In general it could be considered as a datatype which combines data along with the operations. This process of combining the data along with operations is referred to as Encapsulation. Primary advantage of encapsulation is to restrict direct access of data which is known as Data Hiding i.e. in simple words data is hidden from direct access.

An object which is also known as instance or occurrence holds a copy of attributes which represent the current state.

State of an object - All its attribute values combined.

Example -

class Bike {

// Attributes or Fields or Data boolean power = false;

int currentGear = 0; int currentSpeed = 0;

// Operations or Methods or Behavior void start() {

power = true; currentGear = 0;

currentSpeed = 0;

}

void stop() {

power = false; currentGear = 0;

currentSpeed = 0;

}

void changeGear(int gearNo) { currentGear = gearNo;

}

void accelerate(int newSpeed) { currentSpeed = newSpeed;

}

void print() {

System.out.println("Bike ­ "); System.out.println("\t Power ­ " + power); System.out.println("\t Gear ­ " + currentGear); System.out.println("\t Speed ­ " + currentSpeed);

}

}

public class Main {

public static void main(String[] args) {

// LHS ­ Reference variable

// RHS ­ Creates Object using new. Bike b1 = new Bike();

Bike b2 = new Bike(); b1.start();

b2.start(); b2.changeGear(2); b2.accelerate(90);

}

}

Example 2 -

b1.print();

b2.print();

class Amount

{

// It cann't be accessed directly from outside of class.

private int value;

// Allowed to access

public void setValue(int n) {

if (n >= 0) { // Constraint, value should be +ve. value = n;

}

}

public int getValue() { return value;

}

}

public class Main {

public static void main(String[] args) { Amount a1 = new Amount(); a1.setValue(100); System.out.println(a1.getValue());

/\*

* If we didn't restrict direct access to
* value then there is no point in setValue.

\*

* If value field is not private below
* assignment is possible.

\*/

// a1.value = ­100;

a1.setValue(­100); // will not set since it is < 0 System.out.println(a1.getValue());

}

}

Naming Convention -

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

The notation that we use is referred to as Camel Notation. i.e. Capital letter for each new word in the identifier.

Type Identifiers - Start with Capital

e.g. Bike, Amount etc.

Variables - Start with lower case, after the first word we should capitalize.

e.g. value, totalCount etc.

Constants - Use capital case for complete identifier. Word separation could be done using underscore (\_).

e.g. MAX\_ACTIVE\_INTERVAL, PI etc.

Note - It’s a good practice to not to use acronyms and give some meaningful, readable names. getHistoricalData - Do

getHD - Don't

Method - Start with lower case and each new word should be written in upper case. getActiveConnectionCount() etc.

**method in Java?**

A method is a block of code or collection of statements, or a set of code grouped together to perform a certain task or operation. It is used to achieve the reusability of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the easy modification and readability of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.



**Static Method or Static Method with return type:**

A method that has static keyword is known as static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method. We can also create a static method by using the keyword static before the method name.

**Note**: In static method we can’t access instant variables and instant method directly, we need to create an object then we can able to access by it’s object.

The main advantage of a static method is that we can call it without creating an object. It can access static data members and also change the value of it. It is used to create an instance method. It is invoked by using the class name. The best example of a static method is the main() method.

**Instance Method or Instance method with return type:**

The method of the class is known as an instance method. It is a non-static method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class.

**Abstract Method or abstract method with return type:**

The method that does not has method body is known as abstract method. In other words, without an implementation is known as abstract method. It always declares in the abstract class. It means the class itself must be abstract if it has abstract method. To create an abstract method, we use the keyword abstract.

Access Modifiers (Part 1)

private - Any member of a class if it is specified as private, it is accessible only from within the members of the same class.

public - If a member is declared as public it is accessible from outside the class.

**Encapsulation in Java**

Encapsulation in Java is a process of wrapping code and data together into a single unit, for example, a capsule which is mixed of several medicines.

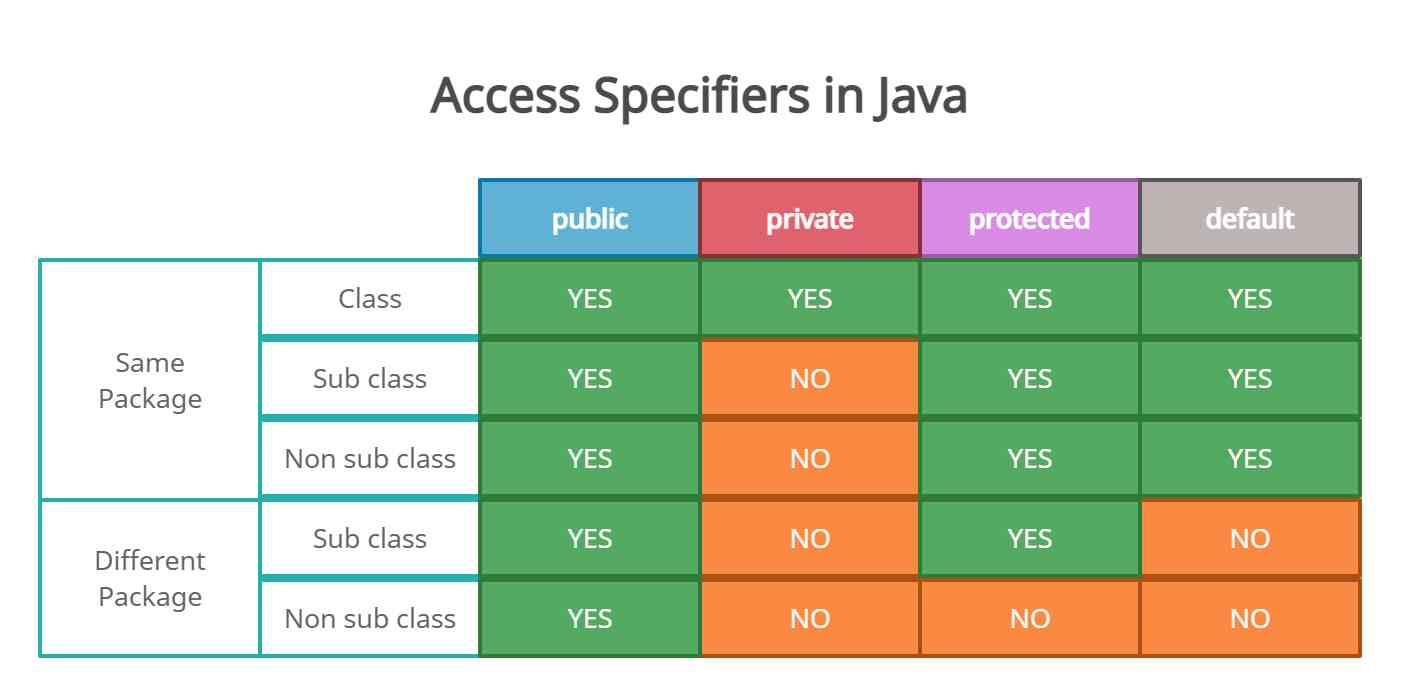
We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

**Advantage of Encapsulation in Java:**

By providing only a setter or getter method, you can make the class read-only or write-only.

It provides you the control over the data. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve data hiding in Java because other class will not be able to access the data through the private data members.



**Difference between Argument and Parameter in Java**

The difference between parameter and argument is sometimes confusing in Java because many programmers often, incorrectly, use them interchangeably but they both have different meanings.

1. A parameter is a variable in the definition of a method whereas an argument is an actual value of this variable that is passed to the method’s parameter.
2. A parameter is also called formal parameter whereas an argument is also called actual parameter.
3. During the time of call, each argument is always assigned to the parameter in the method definition whereas parameters are local variables that are assigned by value of the arguments when method is called.

**Call by Value and Call by Reference in Java:**

There is only call by value in java, **not** call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

|  |  |
| --- | --- |
| Call By Value | Call By Reference |
| While calling a function, we pass values of variables to it. Such functions are known as “Call by Values”. | While calling a function, instead of passing the values of variables, we pass address of variables(location of variables) to the function known as “Call by References. |
| In this method, the value of each variable in calling function is copied into corresponding dummy variables of the called function. | In this method, the address of actual variables in the calling function are copied into the dummy variables of the called function. |
| With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function. | With this method, using addresses we would have access to the actual variables and hence we would be able to manipulate them. |
| Thus actual values of a and b remain unchanged even after exchanging the values of x and y. | Thus actual values of a and b get changed after exchanging values of x and y. |
| In call by values we cannot alter the values of actual variables through function calls. | In call by reference we can alter the values of variables through function calls. |

Example :

**package** PackageOne;

**public** **class** Cbr

{

**int** value;

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

{

**int** x = 5;

Cbr cb = **new** Cbr();

cb.value = 8;

*callbyvalue*(x);

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

*callbyref*(cb);

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

}

**public** **static** **void** callbyvalue(**int** x)

{

x = 10;

}

**public** **static** **void** callbyref(Cbr cbr)

{

cbr.value = 15;

}

}

**Polymorphisms**:

Diagram

Description automatically generated

**Overloading** -

A method or a function is said to be overloaded when there are multiple definitions available for that function with different signatures.

It is part of compile time polymorphism, or we can say dynamic binding.

Example -

class Sample {

public void print(int a) { System.out.println("int ­ " + a);

}

public void print(double a) { System.out.println("double ­ " + a);

}

public void print(String a) { System.out.println("String ­ " + a);

}

}

public class Main {

public static void main(String[] args) { Sample s = new Sample(); s.print(10); // print(int) s.print(10.2); // print(double) s.print("abc"); // print(String)

}

}

Technical Notes -

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

1. You cannot overload with same arguments.

void print(int a) {

}

// Invalid.

void print(int a) {

}

1. You cannot overload by simply changing return type.

void print(int a) {

}

// Invalid.

int print(int a) {

}

Function definitions are determined based on arguments and not based on return types.

Static Members / Class members -

Any member defined in the class could be either static member (also called as class member) or non-static member (also class as instance member).

class Sample {

public static int a; // static member or class member public int b; // non­static member or instance member.

....

}

Here 'a' is shared by the entire class i.e. only single copy of 'a' is available. Whereas each instance of a class Sample will get its own copy of 'b'.

Since static member belong to class, you don't need an object to reference it.

i.e. Sample.a => valid.

Since non-static member belong to object/instance, you need an instance to reference it.

i.e. Sample.b [Not valid outside the class definition]

Sample obj = new Sample(); obj.b => valid.

Example 2 -

class Sample {

// static member

public static void f() {}

// non­static member public void g() {}

}

What are valid from outside the class?

* 1. Sample.f();
  2. Sample.g();
  3. Sample obj = new Sample(); obj.f();
  4. Sample obj = new Sample(); obj.g();
  5. valid; since f() is a static member
  6. invalid; since g() is non-static hence needs object.
  7. valid; since objects too share static members. But you don't need object any way.
  8. valid, since g() is a non-static member and it is referrenced using object.

*Note - A static member function can only refer to other static members of the same class directly.*

*i.e. it cannot refer to non-static members directly.*

Technical Note -

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

1. Utility methods are generally written as static.

e.g. Math.abs(n), Math.sin(n) .....

1. Constants are ideally declared as static final. e.g.

static final double PI = 3.1416;

Example -

import java.text.SimpleDateFormat; import java.util.Date;

class MyDateUtils {

private static final String DATE\_PATTERN = "dd/MM/yyyy";

public static String getFormattedDate(Date d) { SimpleDateFormat sdf =

new SimpleDateFormat(DATE\_PATTERN); return sdf.format(d);

}

}

public class Main {

public static void main(String[] args) { Date now = new Date(); System.out.println("Today : " + now);

System.out.println( "Formatted : " +

MyDateUtils.getFormattedDate(now));

}

}

1. Initializers and Constructors Basics -

Initializers -

There are two types of initializers blocks available for a Java class

* 1. static initializer :-

A code block that is executed automatically when the class is loaded into memory. We can use it to design the code that must be executed only once per the class. Use this initializer to execute something before the class is being used.

is created.

* 1. non-static initializer :-

A code block which is executed automatically every time when a new object

Example -

class Sample {

static {

System.out.println("static initializer");

}

{

System.out.println( "non­static initializer");

}

}

public class Main {

public static void main(String[] args) { Sample s1 = new Sample();

Sample s2 = new Sample(); Sample s3 = new Sample();

}

}

Output -

static initializer

non-static initializer non-static initializer non-static initializer

Technical Note -

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

We can define any number of initializers in a class they are executed in the sequence in which they are defined.

Constructors -

Constructor is responsible for initializing the object. Its name should be same as that of class name but with **no** return type.

**points to remember:**

1. Constructor has same name as class name
2. Constructor does not have any return type
3. Constructor get invoked automatically when you create object
4. Constructor in java cannot be static. final. abstract and Synchronized

**Note1:** We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

**Note2**: If there is no constructor in a class, compiler automatically creates a default constructor. And these default constructors will initialize default values to the variable’s that are declared in the class.

**Note3**: If there is no constructor in a class, compiler automatically creates a default constructor.

**Note4**: if two classes having parent -child relation, then if we create object for child class then, java will call parent class default constructor automatically.

It is also continued for multi-level inheritance

**Type of constructor**

1- Default constructor (No-argument constructor)

2- Parameterized constructor (Constructor with some arguments)

Example -

class Sample {

private int value;

public Sample() { // Constructor value = 5;

}

public void setValue(int n) { value = n;

}

public int getValue() { return value;

}

}

public class Main {

public static void main(String[] args) { Sample s = new Sample(); System.out.println(s.getValue()); // 5

}

}

Techinical Note -

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

In the above class Sample, setValue method is called as mutator and getValue method is called as accessor.

Constructor Overloading -

Like methods we can also overload constructors; which provides user with options to initialize objects.

**Signatures consist of**

1-Number of arguments

2-Type of argument

3-Onder of argument

Example -

class Sample {

// Fields private int a;

// Constructors public Sample() {

a = 5;

}

public Sample(int n) { a = n;

}

// Methods

public void setValue(int n) { a = n;

}

public int getValue() { return a;

}

}

public class Main {

public static void main(String[] args) { Sample s1 = new Sample();

Sample s2 = new Sample(10);

System.out.println(s1.getValue()); // 5

System.out.println(s2.getValue()); // 10

}

}

Practice below:

1. purpose of a default constructor
2. Parameterized Constructor
3. Constructor Overloading
4. Copy Constructor
5. Copying values without constructor
6. Q) Does constructor return any value?

Yes, it is the current class instance (You cannot use return type yet it returns a value).

1. Q) Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

Technical Notes -

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

* + 1. Object creation must match with one of the accessible constructors.

class Sample {

// One arg constructor public Sample(int n) {

}

// Three arg constructor

public Sample(int x, int y, int z) {

}

}

public class Main {

public static void main(String[] args) {

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Sample | | | s1 | = | new | Sample(); | // INVALID |
| } | } | Sample Sample Sample | s2 s3 s4 | =  =  = | new new new | Sample(10); // VALID Sample(10,20); // INVALID Sample(10,20,30); // VALID | |
| this keyword - |  |  |  |  |  |  | |

This is one of the reserved keyword in Java, this keyword is also known as reference variable that refers to current class object.

this keyword can be used with variables and constructor.

Uses of this,

1. Resolves ambiguities between local and instance variable.
2. with variable: (We can use this keyword to refer current class instance variable)
3. with constructor (Using this () to invoke current class constructor)

Example -

class Sample {

int value; // instance variable

// local variable public void setValue(int value) {

this.value = value;

}

public int getValue() { return value;

}

}

public class Main {

public static void main(String[] args) { Sample s1 = new Sample(); s1.setValue(10); // this ­> s1 System.out.println(s1.getValue());

Sample s2 = new Sample(); s2.setValue(20); // this ­> s2 System.out.println(s2.getValue());

}

}

1. To make a call to the constructor of the same class from another constructor of the same class; provided it should be the **first statement** in it.
2. Note: we **can not** call more then one constructor in any constructor

Example -

class Sample {

int value;

// no arg constructor public Sample() {

this(10); // call to arg constructor

}

// arg consructor public Sample(int n) {

value = n;

}

public void setValue(int value) { this.value = value;

}

public int getValue() { return value;

}

}

public class Main {

public static void main(String[] args) { Sample s1 = new Sample(); System.out.println(s1.getValue()); Sample s2 = new Sample(100); System.out.println(s2.getValue());

}

}

**Super Keyword**:

* The super keyword in Java is a reference variable which is used to refer immediate parent class object.
* Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.
* super keyword can be used with variables, methods, and constructor.
* super keyword will be used when we use inheritance concept.

**with variable:**

Scenario- When parent class and child class have the same variable with same name and if child class want to access the parent class variable, in that case we will use super keyword

Ex: super. keyword

**with method:**

Scenario- When parent class and child class have the same method name and if we want to call parent class method, in that case we will use super keyword

Ex: super. method ()

super. method (a, b)

**with constructor:**

Scenario- Whenever we need to call base class constructor from child class, we can use super()

Ex: super ()

super (a, b)

**points to remember for super:**

1. super () should be first statement in derived class constructor
2. If base class does not have zero argument constructor, then java throws compile time exception.
3. If we do not call base class constructor, then by default java internally calls base class constructor with zero argument
4. when child class calls constructor of parent class and again if parent class calls constructor of grandparent class, then this process is known as constructor chaining.

**Usage of Java super Keyword:**

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super () can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

* We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

class Animal

{

String color="white”;

}

class Dog extends Animal

{

String color="black";

void printColor()

{

System.out.println(color);//prints color of Dog class

System.out.println(super.color);//prints color of Animal class

}

}

class TestSuper1

{

public static void main(String args[])

{

Dog d=new Dog();

d.printColor();

}

}

2) super can be used to invoke parent class method

* The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

class Animal

{

void eat()

{

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

}

}

class Dog extends Animal

{

void eat(){System.out.println("eating bread...");}

void bark(){System.out.println("barking...");}

void work()

{

super.eat();

bark();

}

}

class TestSuper2

{

public static void main(String args[])

{

Dog d=new Dog();

d.work();

}

}

3) super is used to invoke parent class constructor.

* The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

class Animal

{

Animal(){System.out.println("animal is created");}

}

class Dog extends Animal

{

Dog()

{

super();

System.out.println("dog is created");

}

}

class TestSuper3

{

public static void main(String args[])

{

Dog d=new Dog();

}

}

**Final Keyword In Java:**

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called **blank final variable** or **uninitialized final variable**. It can be initialized in the constructor & initializer (static or non-static). The blank final variable can be **static** also which will be initialized in the static block only.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable (It will be constant).

2) Java final method

If you make any method as final, you cannot override it.

3) Java final class

If you make any class as final, you cannot extend it.

1. Inheritance and Overriding -

Inheritance -

Allows us to extend an existing class to re-use its functionality. Use "extends" keyword to inherit the properties of an existing class.

Using inheritance, you can use parent class variables and method using child class object.

**Why Inheritance:**

1. Code Reusability
2. No code duplication
3. For method overriding

**Scenarios:**

1. Child class reference and child class object- This will allow to access all the methods of base class and child class
2. Base class reference and Base class object- This will allow to access all the methods of base class only.
3. Base class reference and child class object- This will allow to access all the variable & methods of base class and common method’s will be override to child class method.
4. Child class reference and Base class object- This will be invalid scenario. Child cannot hold parent.

Example -

class BasicCalculator {

public double add(double a, double b) { return a + b;

}

public double sub(double a, double b) { return a ­ b;

}

}

class ScientificCalculator extends BasicCalculator { public double sin(double deg) {

double radians = deg \* 3.1417 / 180; return Math.sin(radians);

}

}

public class Main {

public static void main(String [] args) { BasicCalculator calc1 =

new BasicCalculator(); System.out.println(calc1.add(10, 20));

System.out.println(calc1.sub(10, 20));

ScientificCalculator calc2 =

new ScientificCalculator(); System.out.println(calc2.add(100, 200));

System.out.println(calc2.sub(100, 200)); System.out.println(calc2.sin(90));

}

}

\*\*\*\*Without Inheritance -

Instead of extending BasicCalculator; ScientificCalculator could aggregate object of BasicCalculator and delegate the functionality to it when required.

class ScientificCalculator {

BasicCalculator basicCalc = new BasicCalculator(); public double add(double a, double b) {

return basicCalc.add(a, b);

}

public double sub(double a, double b) { return basicCalc.sub(a, b);

}

public double sin(double deg) {

double radians = deg \* 3.1417 / 180; return Math.sin(radians);

}

}

Overriding -

Redefining an existing method of the base class using the same method signature. This is done in order to modify its definition.

It is part of run time polymorphism, or we can say dynamic binding.

**Scenario:**

Parent p=new Parent (); - Parent class method will be called

Child c=new Child (); - Child class method will be called

Parent p=new Child (); - Child class method will be called

Child c=new Parent (); - Invalid (Child class reference variable cannot hold parent class Object)

Example -

class Vehicle {

public void print () { System.out.println("Print vehicle info.");

}

}

class Car extends Vehicle {

// Overridden base class (Vehicle) method print. public void print() {

System.out.println("Print car info.");

}

}

public class Main {

public static void main(String [] args) { Vehicle v1 = new Vehicle(); v1.print(); // print in Vehicle

Car c1 = new Car(); c1.print(); // print in Car Vehicle v2 = new Car(); v2.print(); // print in Car

}

}

Can we call print ()? Determined by type. Which print ()? Determined by object.

@Override - It’s a good practice to use this annotation for the overridden methods to ensure that proper rules while overriding are followed.

Method Hiding:

* Both parent and child class methods should be static

Example -

class Vehicle {

public static void print () { System.out.println("Print vehicle info.");

}

}

class Car extends Vehicle {

// Hiding

Public static void print() {

System.out.println("Print car info.");

}

}

public class Main {

public static void main(String [] args) { Vehicle v1 = new Vehicle(); v1.print(); // print in Vehicle

Car c1 = new Car(); c1.print(); // print in Car Vehicle v2 = new Car(); v2.print(); // print in Vehicle

}

}

final modifier -

* 1. Identifier declared as final is a constant. i.e. it cannot be modified after initialization. Example 1 -

final int MAX\_VALUE = 10;

MAX\_VALUE = 20; // ERROR; you cannot modify.

Example 2 -

In case of local variables first assignment is considered as initialization; once initialized you cannot modify.

{

final int MAX\_VALUE;

MAX\_VALUE = 10; // VALID; Initialization MAX\_VALUE = 20; // ERROR; Cannot modify.

}

* 1. If a method is declared as final it cannot overridden.

class Base {

public final void print() {

....

}

}

class Derived extends Base { @Override

public void print() {}

// INVALID;

// Cannot override final method of class Base.

}

* 1. If a class is declared as final it can't be extended.

final class Sample {}

class Demo extends Sample {}

// INVALID; You cannot extend final class.

Abstract Classes and Interfaces -

Abstract Classes -

* Abstract methods are methods with no implementation and without a method body. They do not contain any method statement.
* The declaration of an abstract method must end with a semicolon ;,start with (**public** abstract) keyword.
* The child classes which inherit the abstract class must provide the implementation of these inherited abstract methods.
* Abstraction is a process of showing the essentials features and hiding the background details.
* Any class which abstract keyword will be known as Abstract Class.
* If in class even if a single method is abstract method, then class will become Abstract class.
* An abstract class can contain abstract method and non-abstract method as well.
* We cannot create object of Abstract class.
* Abstract refers to something which is not complete, you can think of it as a statement without including the actual implementation details. Similarly, an abstract class is something which will serve as base class with some abstract declarations and provide some common functionality for a group of related sub classes.
* If an interface is implemented in abstract class, then interface abstract method implementation is an optional (if implemented then no need to implement in extended class of abstract class)

Example 1:

**package** java Tutorials;

**public** **interface** Emp

{

**public** **int** salary();

**public** **void** goodies();

**public** **void** work();

}

**package** java Tutorials;

**public** **abstract** **class** ITDepart **implements** Emp

{

**public** **abstract** **int** salary();

**public** **void** goodies()

{

System.out.println("Laptop");

}

**public** **void** work()

{

System.out.println("Automation");

}

}

**package** java Tutorials;

**public** **class** HRTeam **extends** ITDepart

{

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

{

HRTeam hr=**new** HRTeam();

hr.goodies();

hr.work();

hr.salary();

}

**public** **int** salary()

{

System.***out***.println("5LPA");

**return** 500000;

}

}

e.g. 2 .

abstract class Graphic { abstract void paint();

// paint is only declared but not defined.

// hence it is declared as abstract.

}

When a class consists of atleast one abstract method then it should be declared as abstract as well. Abstract class could not be instantiated i.e. You can create a reference but not object.

Graphic g;

// Allowed since it is only a reference variable.

g = new Graphic();

// Not allowed; you cannot instantiate Graphic

// since it is abstract.

Concrete class could be instantiated i.e. you can create object.

e.g. class Line, class Rectangle

g = new Line(); // Allowed since it is concrete.

Example 1 -

abstract class Graphic {

// Protected members are accessible through

// derived classes. protected int x1,y1, x2, y2;

public void setStart(int x1, int y1) { this.x1 = x1;

this.y1 = y1;

}

public void setEnd(int x2, int y2) { this.x2 = x2;

this.y2 = y2;

}

public abstract void paint();

}

class Line extends Graphic {

@Override

public void paint() { System.out.printf(

"draw line from (%d,%d) to (%d,%d)", x1, y1, x2, y2);

}

}

class Rectangle extends Graphic {

@Override

public void paint() { System.out.printf(

"draw rectangle from (%d,%d) to (%d,%d)", x1, y1, x2, y2);

}

}

public class Main {

public static void main(String [] args) { Graphic g; // VALID

// g = new Graphic(); // INVALID

g = new Line(); g.setStart(10,10); g.setEnd(20,20); g.paint(); // draw line

}

}

Interfaces -g = new Rectangle(); g.setStart(5, 5);

g.setEnd(10,10);

g.paint(); // draw rectangle

* An interface is a collection of **abstract methods, static methods**, and constants (i.e., static, and final fields). It is used to achieve complete abstraction.
* Every interface in java is abstract by default. So, it is not compulsory to write abstract keyword with an interface.
* Interface fields are **public and final** by default, and the methods are public and abstract.
* Interface is 100% abstract class
* Interface will contain the list of abstract method declarations and static constants (if any). Primary purpose of the interface is to list out operations (services) available or needed. You cannot instantiate interface, but it acts as a type, hence reference variable creation is allowed. A class can implement an interface using "implements" keyword.

Diagram

Description automatically generated

**Scenarios:**

1. Child class reference and child class object- This will allow to access all the methods of base class and child class
2. Base class reference and Base class object- This will allow to access all the methods of base class only.
3. Base class reference and child class object- This will allow to access all the methods of base class only.
4. Child class reference and Base class object- This will be invalid scenario. Can not create object of interface

Example -

interface AudioPlayer { void play();

void stop();

}

class AudioPlayerImpl implements AudioPlayer { @Override

public void play() { System.out.println("Play definition");

}

@Override

public void stop() { System.out.println("Stop definition");

}

}

public class Main {

public static void main(String [] args) { AudioPlayer player = new AudioPlayerImpl(); player.play();

player.stop();

}

}

Points to Note

AudioPlayer player;

// player is a ref. var to any object of type

// AudioPlayer.

player = new AudioPlayer();

// ERROR; you cannot instantiate AudioPlayer

// as it is an interface.

player = new AudioPlayerImpl();

// VALID; is AudioPlayerImpl is a

// concrete class and it implements

// AudioPlayer interface.

A class can implement multiple interfaces. Reminder; a class cannot extend more than one class, but it can implement multiple interfaces.

Example -

interface AudioPlayer { void playAudio();

}

interface VideoPlayer { void playVideo();

}

class MP3PlayerImpl implements AudioPlayer, VideoPlayer { @Override

public void playVideo() { System.out.println("playVideo with Audio");

}

@Override

public void playAudio() { System.out.println("playAudio");

}

}

public class Main {

public static void main(String [] args) { AudioPlayer audioPlayer = new MP3PlayerImpl(); audioPlayer.playAudio();

VideoPlayer videoPlayer = new MP3PlayerImpl(); videoPlayer.playVideo();

MP3PlayerImpl mp3Player = new MP3PlayerImpl(); mp3Player.playAudio();

mp3Player.playVideo();

}

}

Method of the interface as "public abstract" by default even though it is not explicitly declared. Data fields will be "public static final" by default.

interface Math {

double PI = 3.1417;

double add(double a, double b);

}

It is equivalent to

interface Math {

public static final double PI = 3.1417;

public abstract double add(double a, double b);

}

Interface doesn't hold data member, it only holds static constants, so that they can be referred using interface itself.

e.g. Math.PI

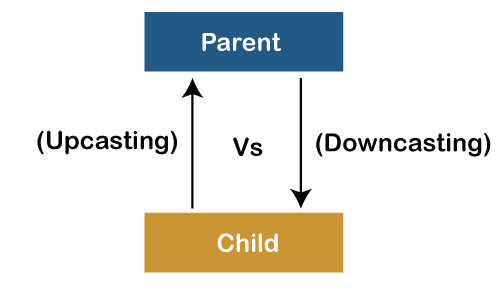
**Difference between abstract class and interface:**

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:** | **Example:** |
| public abstract class Shape{ | public interface Drawable{ |
| public abstract void draw(); | void draw(); |
| } | } |

**Upcasting and Downcasting:**

* Upcasting and Downcasting is the type of object typecasting. In Java, the object can also be typecasted like the datatypes
* In Upcasting, we typecast a child object to a parent object
* Downcasting, we typecast a parent object to a child object simultaneously

**Note**: We can perform Upcasting implicitly or explicitly, but downcasting cannot be implicitly possible.



**Upcasting:**

* **Upcasting** is a type of object typecasting in which a **child object** is typecasted to a **parent class object**. By using the Upcasting, we can easily access the variables and methods of the parent class to the child class.

**Example:**

**class** Parent

{

**void** PrintData()

{

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

    }

}

**class** Child **extends** Parent

{

   @overide

**void** PrintData()

{

       System.out.println("method of child class");

     }

}

**class** UpcastingExample

{

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

{

Parent p;

Child c =  **new** Child();

       p = (Parent) c;  // explicit upcasting

       Parent p = **new** Child();  or

p = c  //implicit upcasting

       p.PrintData();

    }

}

OUTPUT :

method of child class

**Downcasting:**

* **Upcasting is another type of object typecasting. In Upcasting, we assign a parent class reference object to the child class. In Java, we cannot assign a parent class reference object to the child class, but if we perform downcasting, we will not get any compile-time error. However, when we run it, it throws the "ClassCastException". Now the point is if downcasting is not possible in Java, then why is it allowed by the compiler? In Java, some scenarios allow us to perform downcasting. Here, the subclass object is referred by the parent class.**

**Example:**

//Parent class

**class** Parent {

    String name;

    // A method which prints the data of the parent class

**void** showMessage()

   {

       System.out.println("Parent method is called");

    }

}

// Child class

**class** Child **extends** Parent {

**int** age;

    // Performing overriding

   @Override

**void** showMessage()

    {

        System.out.println("Child method is called");

    }

}

**public** **class** Downcasting{

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

    {

        Parent p = **new** Child();

        p.name = "Shubham";

Child c = **new** Child();

        // Performing Downcasting Implicitly

        //Child c = new Parent();  or c = p // it gives compile-time error

        // Performing Downcasting Explicitly

        c = (Child)p;

        c.age = 18;

        System.out.println(c.name);

        System.out.println(c.age);

        c.showMessage();

    }

}

OUTPUT:

Shubham

18

Child method is called

**Shallow Copy Vs. Deep Copy:**

**Shallow Copy**

* In Shallow copy, a copy of the original object is stored and only the reference address is finally copied.
* Changes in one entity are reflected in the other entities as well, then we can say we have done a shallow copy.
* In shallow copy, new memory allocation never happens for the other entities, and the only reference is copied to the other entities.

**Example**:

class ABC

{

// instance variable of the class ABC

int x = 30;

}

public class ShallowCopyExample

{

// main method

public static void main(String argvs[])

{

// creating an object of the class ABC

ABC obj1 = new ABC();

// it will copy the reference, not value

ABC obj2 = obj1;

// updating the value to 6

// using the reference variable obj2

obj2.x = 6;

// printing the value of x using reference variable obj1

System.out.println("The value of x is: " + obj1.x);

}

}

OUTPUT: The value of x is: 6

**Deep Copy**

* In Deep copy, the copy of the original object and the repetitive copies both are stored.
* Changes in one entity are not reflected in the other entities, then we can say we have done a deep copy.
* In the deep copy, a new memory allocation happens for the other entities, and reference is not copied to the other entities. Each entity has its own independent reference.

Example:

class ABC

{

// instance variable of the class ABC

int x = 30;

}

public class DeepCopyExample

{

// main method

public static void main(String argvs[])

{

// creating an object of the class ABC

ABC obj1 = new ABC();

// it will copy the reference, not value

ABC obj2 = new ABC();

// updating the value to 6

// using the reference variable obj2

obj2.x = 6;

// printing the value of x using reference variable obj1

System.out.println("The value of x is: " + obj1.x);

}

}

OUTPUT : The value of x is: 30

**Mutable and Immutable:**

**Mutable Objects:**

* The mutable objects are objects whose value can be changed after initialization. We can change the object's values, such as field and states, after the object is created.

**Immutable Objects:**

* The immutable objects are objects whose value cannot be changed after initialization. We cannot change anything once the object is created. For example, primitive objects such as int, long, float, double, all legacy classes, Wrapper class, String class, etc.

**How to Create a Mutable Class:**

* The following two things are essential for creating a mutable class:

1. Methods for modifying the field values
2. Getters and Setters of the objects

**How to Create an Immutable Class:**

* The following things are essential for creating an immutable class:

1. Final class, which is declared as final so that it can't be extended.
2. All fields should be private so that direct access to the fields is blocked.
3. No Setters
4. All mutable fields should be as final so that they can not be iterated once initialized.

**Packages** -

A grouping of classes or interfaces etc. with a name. The package statement is the first statement in the java source file which indicates package to which the types declared in the source file belong to.

Creating Packages -

1. Right click on src => new => package => give some package name

e.g. com.packagedemo.service

1. Create few sample classes in it.

Right click on the package and select new => class, specify the class name.

AccountService.java -

package com.packagedemo.service;

public class AccountService { public void openAccount() {

System.out.println("open account");

}

}

CardService.java -

package com.packagedemo.service;

public class CardService { public void newCard() {

System.out.println("new card");

}

}

1. Referring to classes which are part of external package

For the below scenarios create a class Main with main method selected and leave package field as empty.

* 1. *Referring to classes with absolute name i.e. qualified name.*

class Main {

public static void main(String [] args) { com.packagedemo.service.AccountService s1;

s1 = new com.packagedemo.service.AccountService(); s1.openAccount();

com.packagedemo.service.CardService c1;

c1 = new com.packagedemo.service.CardService(); c1.newCard();

}

}

* 1. *Importing entire package (wildcard import)* - use wildcard \*; it imports all the classes of the package; hence they need not be qualified with package name. It is applicable for types from within that package and not sub-packages i.e. if service package has a sub-package utils (say) then the import is not applicable for utils; it needs a separate import.

import com.packagedemo.service.\*; class Main {

public static void main(String [] args) { AccountService s1;

s1 = new AccountService(); s1.openAccount(); CardService c1;

c1 = new CardService(); c1.newCard();

}

}

* 1. *Importing specific class* - Import is applicable only for that class hence any other class from that package requires qualified name or import.

import com.packagedemo.service.AccountService; class Main {

public static void main(String [] args) { AccountService s1;

s1 = new AccountService(); s1.openAccount(); com.packagedemo.service.CardService c1;

c1 = new com.packagedemo.service.CardService(); c1.newCard();

}

}

(or) to refer to CardService directly import it.

import com.packagedemo.service.AccountService; import com.packagedemo.service.CardService;

import - Allows us to refer to the classes directly without qualifying them with package name. PREFERRED APPROACH - Import specific class.

Access modifiers -

public, private, protected, no modifier(default)

class Sample {

private void f() {

// Visible to the class.

}

void g() {

// Visible to the class + members of same package.

}

protected void h() {

// Visible to the class + members of the same package

// + Visible to derived(sub) classes outside package.

}

public void i() {

// ALL

}

}

Access modifiers before class -

private(only for nested classes), public, default.

A.java :-

package com.packagedemo; class Sample {

}

public class A {

}

No modifier specified to class Sample hence it is default and accessible with in the package

i.e. through members of "com.packagedemo".

Since class A is public it is accessible outside package. Also note that public classes should be saved with the file name that is same as that of class name hence A.java.

***Note*** *- Hence no two public classes can be saved in the same file.*

**Strings**

1. String class -

String could be considered as primitive type in java since any literal within double quotes is considered as String i.e. "ABC". It is immutable i.e. once the object is created and initialized it cannot be modified.

i.e.

String s = "abc"; s = s + "xyz";

Here object being referenced is not modified instead ref held in 's' is changed to a newly

created object as illustrated in the below diagram.

Immutability gives flexibility to share the object. In the below scenario Java doesn’t create two objects instead it will share the same object.

String s1 = "abc"; String s2 = "abc";

Here both s1 and s2 point to same object since object for "abc" is not created twice instead it

is created once and added to the string pool and will be obtained from it whenever there is a reference to "abc" exists in the program.

if (s1 == s2) {

System.out.println("equal");

// ANS since s1 and s2 refer to same object.

} else {

System.out.println("not equal");

}

String functions -

length() :-

returns the length of the string

indexOf( String stringToFind ) :-

searches for the given string from beginning. return value same as above.

indexOf(String stringToFind, int searchPos ) :- searches from the given position.

substring( int startIndex ) :-

Returns a substring from startIndex to the end of the string.

substring( int startIndex , int endIndex ) :-

Returns a substring from startIndex to endIndex - 1.

trim() :-

Returns a string after removing the leading and trailing spaces from the given string.

split( delimeter ) :- [its arg is regular expression instead]

Example -

String s = "1;2;3";

String parts[] = s.split(";");

parts array will be "1","2","3"

Example -

String s1 = "1,2,,3"

String parts[] = s1.split(",");

returned array is {"1","2","","3"} getBytes() :- Returns bytes associated with the string.

Example -

String s = "hello";

byte arr[] = s.getBytes();

Constructing String from bytes :-

Example -

Example -

byte arr[] = { 65 , 66 , 67 }; String s = new String( arr );

i.e. "ABC"

public class Main {

public static void main(String [] args) { String str = " sample string ";

//0123456789ABCDEF

System.out.println( str.length() ); // 16 System.out.println( str.indexOf('s')); // 2 System.out.println( str.indexOf('s', 5));

// 9 (search from 5) System.out.println( str.indexOf("str")); // 9 System.out.println( str.indexOf("str", 10)); // ­1 System.out.println( str.substring(4));

// "mple string " System.out.println( str.substring(4, 10));

// "mple s" System.out.println( str.trim()); // "sample string"

String s = "10;20;30;40";

String arr[] = s.split(";");

// { "10", "20", "30", "40" }

System.out.println("Resultant array"); for (String ele : arr) {

System.out.println(ele);

}

}

}

String comparison -

== operator :-

Performs reference comparison when used between reference variables. Example -

String s1 = new String("abc"); String s2 = new String("abc"); if( s1 == s2 ) .. false.

equals() :-

Performs case sensitive content comparison. Example -

if( s1.equals(s2) ) ..true

equalsIgnoreCase() :-

For case insensitive comparison. Example -

String s1 = "abc"; String s2 = "ABC";

if( s1.equalsIgnoreCase(s2) ) ... true

compareTo() :-

Compares and returns an integer value based on comparison(case sensitive). Example -

int n = s1.compareTo(s2);

n < 0 if s1 < s2 n > 0 if s1 > s2

n == 0 if s1 equal to s2.

Example -

public class Main {

public static void main(String [] args) { String s1 = new String("abc"); String s2 = new String("abc"); String s3 = new String("ABC");

System.out.println(s1 == s2); // false System.out.println(s1.equals(s2)); // true System.out.println(s1.equals(s3)); // false System.out.println(s1.equalsIgnoreCase(s3));

// true System.out.println(s1.compareTo(s2)); // 0 System.out.println(s1.compareTo(s3)); // +ve System.out.println(s3.compareTo(s1)); // ­ve System.out.println(s3.compareToIgnoreCase(s1));

// 0

}

}

1. StringBuilder -

Use StringBuilder instead of String if you are looking to modify a string i.e. you might append to or replace content of an existing object.

StringBuilder sb = new StringBuilder("abc"); sb.append("xyz");

Default capacity for a StringBuilder instance is 16, you can specify as per your need. Example -

StringBuilder sb1 = new StringBuilder(); // Capacity 16. StringBuilder sb2 =

new StringBuilder("abc"); // Capacity ­ 3 + 16 StringBuilder sb3 = new StringBuilder(60); // Capacity ­ 60

append(...) -

Appends the String value of the given argument to the content of the StringBuilder. Example -

StringBuilder sb = new StringBuilder("sample"); sb.append("content"); // sb : samplecontent.

delete(int start, int end) -

Deletes the subsequence from start to end-1 in the StringBuilder's char sequence. Example -

StringBuilder sb = new StringBuilder("samplecontent"); sb.delete(3, 6);

// sb ­ samcontent i.e.

// Deletes the char sequence from

// index position 3 to 5.

insert(int offset, String value) -

Inserts the String value of the second argument into the string builder. Example -

StringBuilder sb = new StringBuilder("sample"); sb.insert(3, "test"); // sb ­ samtestple

replace(int start, int end, String s) -

Replaces the sequence of characters from start to end-1 with the given String. Example -

StringBuilder sb = new StringBuilder("sample"); sb.replace(3, 5, "test");

// "pl" is repaced with "test" ­ samteste

reverse() -

Reverses the char sequence. Example -

StringBuilder sb = new StringBuilder("sample"); sb.reverse(); // sb ­ elpmas

Technical Note :-

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

1) equals method is not overridden in StringBuilder hence when invoked equals in the Object class will handle the invocation.

i.e. StringBuilder sb1 = new StringBuilder("abc"); StringBuilder sb2 = new StringBuilder("abc"); if( sb1.equals(sb2) )

System.out.println("equal"); else

System.out.println("not equal");//ans

Solution :-

Convert the given StringBuilder objects to String objects and then compare. i.e.

String s1 = sb1.toString(); String s2 = sb2.toString(); if( s1.equals(s2) )

System.out.println("equal"); // ans else

System.out.println("not equal");

3) StringBuffer -

StringBuffer is a threadsafe version of StringBuilder. Hence it should be used when threadsafety is required.

Exception Handling

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

Ref : <https://www.javatpoint.com/exception-handling-in-java>

Refer all topic under above link

1. Exceptions -

An exception represents a failure condition while performing certain operation.

Overview -

Consider the below example import java.util.Scanner; public class Main {

private static int div(int a, int b) { return a / b;

}

public static void main(String [] args) { int num1, num2;

Scanner scan = new Scanner(System.in);

System.out.println("Enter first number"); num1 = scan.nextInt(); System.out.println("Enter second number"); num2 = scan.nextInt();

System.out.println("Result ­ " + div(num1, num2));

}

}

Input -

Enter first number 12

Enter second number

0

Output -

Exception Reason

Exception in thread "main" java.lang.ArithmeticException: / by zero at Main.div(Main.java:6) [Stack Trace]

at Main.main(Main.java:20)

i.e. failed in div; line no 6; it is called from main; line no 20.

Q) How to handle the above exception ?

Ans) One way of handling the exception is to use try...catch block. e.g.

try

{

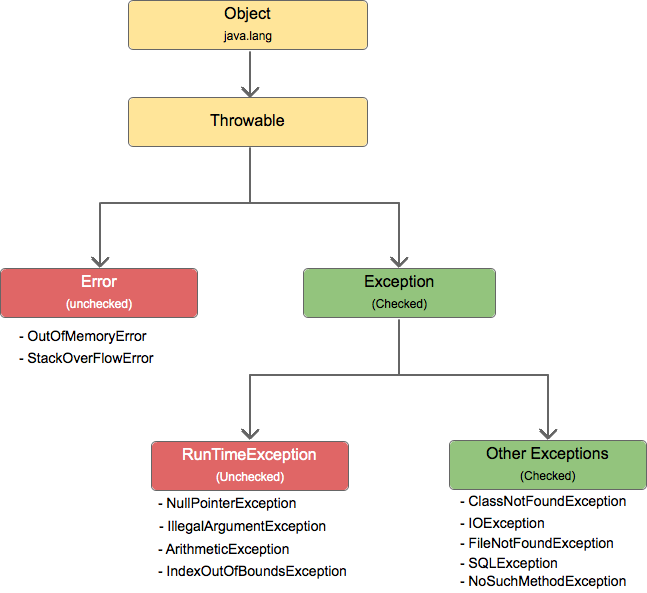
System.out.println("Result - " + div(num1, num2));

} catch (ArithmeticException e) {

System.out.println("Could not perform div operation");

}

Exception Class Hierachy -



Throwable is the base class for all exception classes. Its classified into Error and Exception. Error represents something that failed at VM level i.e. NoSuchMethodError, StackOverflowError or OutOfMemoryError etc, these are not supposed to be handled in the program. Failures that occur within the application are Exceptions; they occur due to some failure as part of the operation e.g. ArithmeticException, NullPointerException, IOException, SQLException etc..

ArithmeticException, NullPointerException etc which are subclasses for RuntimeException even these are not supposed to be handled in the program, because they too represent some critical failure. Others i.e. which are not Error types and RuntimeException types could be handled.

Error and RuntimeException types are critical and are not supposed to be handled, hence java compiler doesn't look for any handler for them. Hence they are called Unchecked Exceptions and the rest are class Checked exceptions i.e. when they are expected to occur java compiler looks for a handler.

try...catch block :-

A try block surrounds the code which might throw an exception and it is completed with catch or finally or both. Catch block acts as the handler for the corresponding exception.

i.e.

try {

// code which may throw exception

} catch(ExceptionType1 expVar) {

// handler for ExceptionType1

} catch(ExceptionType2 expVar) {

// handler for ExceptionType2

} finally {

// a block that executes irrespective of exceptions.

}

Example - *Note - Below example has used deprecated functionality.*

import java.io.DataInputStream; import java.io.IOException; public class Main {

public static void main(String [] args) { int a, b, c;

String str;

DataInputStream din =

new DataInputStream(System.in);

try {

System.out.println("Enter first number"); str = din.readLine();

a = Integer.parseInt(str); System.out.println("Enter second number"); str = din.readLine();

b = Integer.parseInt(str); c = a / b;

System.out.println("Result ­ " + c);

} catch(IOException e) { System.out.println(

"Failure to read the source ");

} catch(NumberFormatException e) { System.out.println("Invalid number");

} catch(ArithmeticException e) { System.out.println(

"Denominator cannot be zero");

}

}

}

Input/Output -

1)

Enter first number 10

Enter second number jghjhg

Invalid number

2)

Enter first number 10

Enter second number 0

Denominator cannot be zero

3)

Enter first number 10

Enter second number 2

Result - 5

1. Custom Exceptions-

You can define your own exceptions by extending the Throwable class or any Exception

class.

throw statement -

throw statement is used to throw an exception. Any object which is thrown should be of Throwable type.

Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.

throws clause -

throws clause at the end of the function lists the exceptions which might be thrown from the function.

Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.

Example -

class OrderException extends Exception { public OrderException(String reason) {

super(reason);

}

}

class OrderService {

public void placeOrder(

String product, int quantity,

String shipToAddress) throws OrderException {

if (quantity <= 0 ) {

throw new OrderException("Invalid quantity");

}

if (! isShippingAvailableToLocation(shipToAddress)) { throw new OrderException(

"Cannot ship to the specified location");

}

System.out.println("Placed order for " + product);

}

private boolean isShippingAvailableToLocation(

String shipToAddress) { return shipToAddress.contains("India");

}

}

public class Main {

public static void main(String [] args) { OrderService service = new OrderService();

String product = "monitor"; int quantity = 1;

String shipToAddress = "US";

try { service.placeOrder(

product, quantity, shipToAddress); System.out.println("Order placed");

} catch(OrderException e) { System.out.println(

"Failed to place order; " + e.toString());

}

}

}

1. finally block -

A block which is executed irrespective of exceptions or return or break statements once the control enters try block it will execute finally before it can return. Ideal for closing the resources.

Consider the below sample

try {

open the file read/write close the file

} catch(IOException e) {

// handle failure

}

In the above example file will not be closed if there is any IO failure while read/write operation. We can better it by adding close in catch block.

try {

open the file read/write close the file

} catch(IOException e) {

// handle failure if (file is open)

close the file.

}

Still it’s not a solution, what if some other exception other than IOException occurs or there is a return statement? Instead of closing the resource at all possible cases, we can do it in finally block. i.e.

try {

open the file read/write

} catch(IOException e) {

// handle failure

} finally {

close the file.

}

Example -

import java.util.Scanner; class Exp1 extends Exception {} public class Main {

public static void main(String [] args) { int n;

Scanner scan = new Scanner(System.in);

try {

System.out.println("Enter some value"); n = scan.nextInt();

if (n == 1) {

return;

} else if (n == 2) { throw new Exp1();

}

System.out.println("value of n is " + n);

} catch(Exception e) { System.out.println("Exception caught");

} finally {

System.out.println("Inside finally"); scan.close();

}

System.out.println("Some other code");

}

}

Java Inner Classes

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

1. Introduction -

# Collections

Collection stands for a collection of objects. Util package in java provides several data structures that support operations over collections. You can divide them into three types.

* 1. List - Allows duplicates and also provide positional access.
  2. Set - Doesn't allow duplicates
  3. Map - Key and value pairs.

Collection is the common interface for these implementations. List, Set and Map are sub- interfaces of Collection.

Diagram

Description automatically generated

import java.util.\*; public class Main {

public static void main(String[] args) {

// Allows duplicates. List l = new LinkedList(); l.add(10);

l.add(20);

l.add(30);

l.add(10);

System.out.println("Content of the list"); System.out.println(l); // 10 20 30 10

// Set doesnot allow duplicates. Set s = new HashSet(); s.add(10);

s.add(20);

s.add(30);

s.add(10); // will not add

System.out.println("Content of the set"); System.out.println(s); // 10 20 30

// Is a key and value pair. e.g. dictionary

// with word and its meaning OR index. Map m = new HashMap();

m.put("rice", 50);

m.put("sugar", 50);

System.out.println("Content of the map"); System.out.println(m); // [rice = 50, sugar = 50]

}

}

1. List -

* Allows duplicates
* Provides positional access such as

get(0), get(1), remove(1) etc.

* Implementations
* LinkedList

Preferred when modifications(add/remove) are expected in the middle of the list.

* ArrayList

Preferred when modifications are expected at the end of the list. Slightly faster than LinkedList.

* Vector (old and not used anymore) Example -

import java.util.List; import java.util.ArrayList;

public class Main {

public static void main(String[] args) { List lst = new ArrayList(); lst.add(10);

lst.add(20);

lst.add(30); // 10 20 30

// 0 1 2 (position)

System.out.println(lst); System.out.println("Element at 1 : " + lst.get(1));

lst.add(1, 25); // 10 25 20 30 System.out.println(lst);

Object value = lst.remove(2); // 20 System.out.println("Removed value: " + value);

}

}

Tip -

toString() - It is part of Object class prints class name followed by @ and hashCode(). If you

want to print your own output just override.

class Sample {

public String toString() { return "Hello";

}

}

public class Main {

public static void main(String[] args) { Sample s = new Sample(); System.out.println(s.toString()); System.out.println(s);

// equivalent to s.toString().

}

}

1. Set -

* Doesn't allow duplicates.
* No positional access like List.
* Implementations
* HashSet
  + Insertion order depends on hashCode.
* LinkedHashSet
  + Preserves the insertion order.

HashSet -

Tip -

\*\*\*\*

Example -

* Uses hashCode to store elements.
* Uses hashCode and equals for verifying the presence of the element.

boolean add(Object o) =>

return boolean i.e. true if element is added otherwise false.

import java.util.HashSet; import java.util.Set;

// Remove the duplicates from the given array. public class Main {

public static void main(String[] args) {

int arr[] = {10, 20, 10, 20, 30, 20, 40 };

Set s = new HashSet(); for (int value : arr) {

if (! s.add(value)) { // i.e."if not added".

System.out.println(

"Duplicate found " + value);

}

}

System.out.println(s);

}

}

equals and hashCode -

Q) How HashSet determines if an element exists? Ans) compares hashCode

-> if not matched with any other element's hashCode it is considered unique.

-> if hashcode matches then it does equality check. If not equal it will add otherwise ignores the element.

Example -

import java.util.HashSet; import java.util.Set;

class Fruit {

private String name;

public Fruit(String name) { this.name = name;

}

// TIP ­ You don't need to write this code;

// simply right click

// select source menu, generate equals and hashCode.

@Override

public int hashCode() { return name.hashCode();

}

@Override

public boolean equals(Object other) {

return this.name.equals(((Fruit)other).name);

}

public String toString() { return name;

}

}

public class Main {

public static void main(String[] args) { Fruit f1 = new Fruit("Apple"); Fruit f2 = new Fruit("Mango"); Fruit f3 = new Fruit("Apple");

Set s = new HashSet(); s.add(f1);

s.add(f2);

s.add(f3);

// Added if you don't override hashCode. Because

// hashCode method in Object class always returns

// a unique one.

// i.e. f1.hashCode() != f3.hashCode()

// Since hashCode is not equal it is treated as

// unique and added to the set.

System.out.println(s);

}

}

LinkedHashSet -

import java.util.HashSet; import java.util.LinkedHashSet; import java.util.Set;

public class Main {

public static void main(String[] args) {

// HashSet => insertion order not preserved. Set s1 = new HashSet();

s1.add("abc");

s1.add("xyz");

s1.add("pqr");

System.out.println(s1); // [pqr, abc, xyz]

// LinkedHashSet => preserves insertion order. Set s2 = new LinkedHashSet();

s2.add("abc");

s2.add("xyz");

s2.add("pqr");

System.out.println(s2); // [abc, xyz, pqr]

}

}

TreeSet -

TreeSet is a SortedSet. It arranges the elements during the insertion itself. Objects added to the tree set should be Comparable or we should provide a Comparator.

Example -

import java.util.TreeSet; public class Main {

public static void main(String[] args) { TreeSet s = new TreeSet(); s.add(10);

s.add(2);

s.add(20);

System.out.println(s); // [2, 10, 20]

}

}

Iterator -

Allows to iterate through the elements of the collection. i.e. it decouples us from the underlying data structure.

import java.util.Collection; import java.util.HashSet; import java.util.Iterator; import java.util.LinkedList; import java.util.Set;

import java.util.TreeSet;

public class Main {

public static void main(String[] args) { LinkedList list = new LinkedList(); list.add(10);

list.add(20); display(list);

Set s = new HashSet(); s.add(10);

s.add(20); display(s);

TreeSet ts = new TreeSet(); ts.add(10);

ts.add(20); display(ts);

}

static void display(Collection col) { Iterator i = col.iterator(); while (i.hasNext()) {

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

}

}

// From Java1.5

static void displayUsingFor(Collection col) {

// for each object obj in the collection col. for (Object obj : col) {

System.out.println(obj);

}

}

}

/\* Iterator also supports remove(); it removes the object

* previously returned by next().

\*

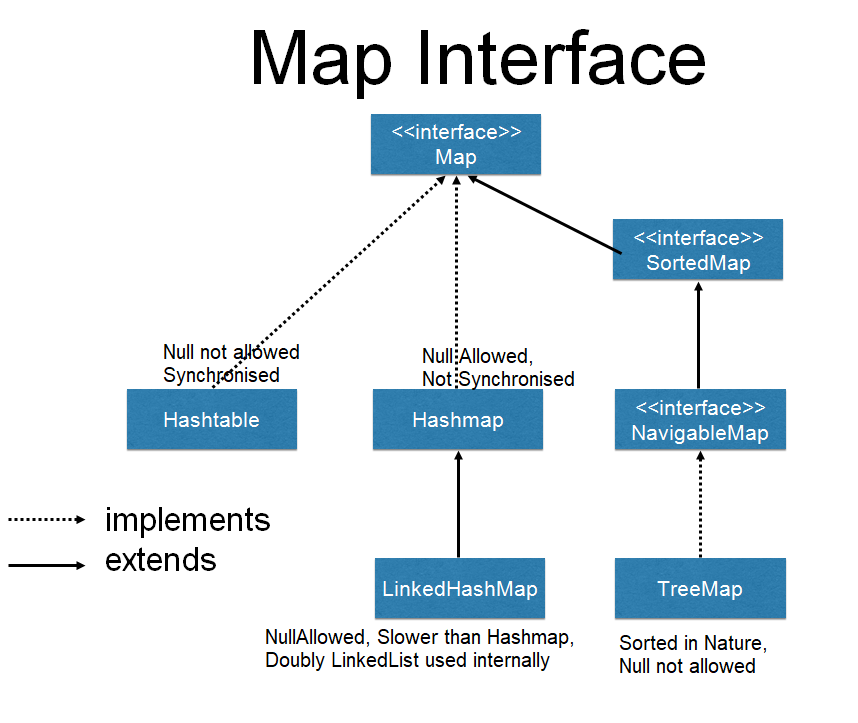
* Iterator i = ;
* while (i.hasNext()) {
* Object obj = i.next();
* if (toBeDeleted(obj)) {
* i.remove();

\* }

\* }

\*/

1. Map -



A map is a collection of entries. Each entry is a key and value pair.

Basic operations

Object oldValue = map.put(key, value) Object value = map.get(key)

Example -

import java.util.HashMap; import java.util.Map;

public class Main1 {

public static void main(String[] args) {

Map m = new HashMap(); m.put("rice", 50);

m.put("wheat", 50);

Integer ricePrice = (Integer)m.get("rice"); Integer wheatPrice = (Integer)m.get("wheat");

System.out.println(ricePrice); System.out.println(wheatPrice);

}

}

Example - Display the word count

import java.util.HashMap; import java.util.Map;

public class Main {

public static void main(String[] args) { String sentence =

"This is some sample sentence This is to test map"; String words[] = sentence.split(" ");

Map wordMap = new HashMap(); for (String word : words) {

Integer value = (Integer)wordMap.get(word); if (value == null) {

value = 1;

} else {

value = value + 1;

}

wordMap.put(word, value);

}

System.out.println(wordMap);

}

}

Example - Iterating through the Map; one approach is to get the key set (set since no duplicate keys) and iterate through the keyset and get the values.

import java.util.HashMap; import java.util.Map; import java.util.Set;

public class Main {

public static void main(String[] args) {

Map m = new HashMap(); m.put("sym", "together");

m.put("par", "equal");

Set keySet = m.keySet(); for (Object key : keySet) {

System.out.println(

" Key :" + key +

" Value : " + m.get(key));

}

}

}

Table

Description automatically generated

**Enums:**

* The Enum in Java is a data type which contains a fixed set of constants.
* It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.
* Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly.
* Enum can have fields, constructors, and methods
* Enum may implement many interfaces but cannot extend any class because it internally extends Enum class

**Example:**

**class** EnumExample1

{

//defining the enum inside the class

**public** **enum** Season { ***WINTER***, ***SPRING***, ***SUMMER***, ***FALL*** }

//main method

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

{

//traversing the enum

Season sea = Season. ***SPRING***;

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

**for** (Season s : Season.*values*())

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

}

}

#### **Difference between Enums and Classes**

An enum can, just like a class, have attributes and methods. The only difference is that enum constants are public, static and final (unchangeable - cannot be overridden).

An enum cannot be used to create objects, and it cannot extend other classes (but it can implement interfaces).

Crate Jar file and how to reuse it?

SQL

# Practise Set 1 - Basics

## Create -

CREATE TABLE [dbo].Employee( employeeID INT,

name VARCHAR(100),

salary NUMERIC(20,2)

);

**Insert -** Inserts record(s) into the table.

INSERT INTO [dbo].Employee VALUES (1,'a',1000);

*Note - You can also insert values for specific fields (columns). Default values if any will be considered for the other columns if there is no default value then it will be NULL.*

INSERT INTO [dbo].Employee (employeeID, name)

VALUES (2,'b');

**Select** - Displays all records.

SELECT \* FROM [dbo].Employee;

**Delete -** Deletes all records.

DELETE FROM [dbo].Employee;

## Drop table -

DROP TABLE [dbo].Employee;

# Practise Set 2 – Constraints

## Create -

CREATE TABLE [dbo].Employee( employeeID INT PRIMARY KEY, name VARCHAR(100) NOT NULL,

salary NUMERIC(20,2) NOT NULL Check(salary > 0)

);

## Insert -

INSERT INTO [dbo].Employee VALUES(1, 'a', 1000 );

INSERT INTO [dbo].Employee VALUES(1, 'b', 2000 );

­­ FAILS; Violates Primary Key

INSERT INTO [dbo].Employee VALUES(2, 'b', 2000 ); ­­ SUCCESS

INSERT INTO [dbo].Employee VALUES(3, NULL, 3000 );

­­ FAILS; Violates Not Null constraint for name column.

INSERT INTO [dbo].Employee VALUES(3, 'c', 3000 ); ­­ SUCCESS

INSERT INTO [dbo].Employee VALUES(4, 'd', 0 );

­­ FAILS; Violates Check constraint for

­­ Salary column.

INSERT INTO [dbo].Employee VALUES(4, 'd', 4000 ); ­­ SUCCESS;

## Drop Table -

DROP TABLE [dbo].Employee;

# Practise Set 3 – DML

## Create -

CREATE TABLE [dbo].Employee( employeeID INT PRIMARY KEY, name VARCHAR(100) NOT NULL,

salary NUMERIC(20,2) NOT NULL Check(salary > 0)

);

## Insert -

INSERT INTO [dbo].Employee VALUES(1, 'a', 1000 );

INSERT INTO [dbo].Employee VALUES(2, 'b', 2000 );

INSERT INTO [dbo].Employee VALUES(3, 'c', 3000 );

INSERT INTO [dbo].Employee VALUES(4, 'd', 4000 );

INSERT INTO [dbo].Employee VALUES(5, 'e', 4000 );

INSERT INTO [dbo].Employee VALUES(6, 'be', 2000 );

## Select -

-- Project ALL Columns

SELECT \* FROM [dbo].Employee;

-- Project only name and salary columns

SELECT name, salary FROM [dbo].Employee;

-- Project only salary

SELECT salary FROM [dbo].Employee;

-- Project distinct salary

SELECT DISTINCT salary FROM [dbo].Employee;

-- Display employees whose salary is greater than 2000.

SELECT \* FROM [dbo].Employee WHERE salary > 2000;

-- Display employee whose name is 'd'

SELECT \* FROM [dbo].Employee WHERE name = 'd';

-- Display all employees whose name starts with 'b'

SELECT \* FROM [dbo].Employee WHERE name like 'b%';

-- Display all employees whose salary is between 2000 and 3000.

SELECT \* FROM [dbo].Employee

WHERE salary >= 2000 AND salary <= 3000;

-- OR

SELECT \* FROM [dbo].Employee

WHERE salary between 2000 AND 3000;

-- Display all employees whose salary is 2000 or 3000.

SELECT \* FROM [dbo].Employee

WHERE salary = 2000 OR salary = 3000;

-- Display the names of all employees whose salary is less than 4000.

SELECT name FROM [dbo].Employee WHERE salary < 4000;

-- Display all employees whose ids are 1, 3 and 5.

SELECT \* FROM [dbo].Employee WHERE

employeeID = 1

OR employeeID = 3 OR employeeID = 5;

-- OR

SELECT \* FROM [dbo].Employee WHERE employeeID IN (1, 3, 5);

-- Display all employees except employee ids 1 and 3.

SELECT \* FROM [dbo].Employee

WHERE employeeID <> 1 AND employeeID <> 3;

-- OR

SELECT \* FROM [dbo].Employee WHERE employeeID NOT IN (1, 3);

-- Display Employee information in the acending order of the Salary.

SELECT \* FROM [dbo].Employee ORDER BY Salary;

-- Display Employee information in the acending order of the Salary.

SELECT \* FROM [dbo].Employee ORDER BY Salary DESC;

**Delete** - With Where clause

-- Delete the employee records whose salary is greater then 3000.

DELETE FROM [dbo].Employee WHERE salary > 3000;

## Update

-- Update Name of EmployeeID 1 to 'ABC'

UPDATE [dbo].Employee SET Name = 'ABC'

WHERE EmployeeID = 1;

-- Add 5% bonus to all Employees

## Drop -

UPDATE [dbo].Employee

SET Salary = Salary \* 1.05;

DROP TABLE [dbo].Employee;

# Practise Set 4 – Group By, Having

## Create -

CREATE TABLE [dbo].Student(

StudentID INT IDENTITY(1,1) PRIMARY KEY, Name VARCHAR(50) NOT NULL,

Course VARCHAR(10) NOT NULL, Score NUMERIC(3,2) NOT NULL

);

## Alter (demo) -

ALTER TABLE [dbo].Student

ALTER COLUMN Score NUMERIC(5,2) NOT NULL;

## Insert -

INSERT INTO [dbo].Student(Name,Course,Score) VALUES('A','CS',90);

INSERT INTO [dbo].Student(Name,Course,Score) VALUES('B','CS',80.45);

INSERT INTO [dbo].Student(Name,Course,Score) VALUES('C','CS',85.23);

INSERT INTO [dbo].Student(Name,Course,Score) VALUES('D','IT', 87);

INSERT INTO [dbo].Student(Name,Course,Score) VALUES('E','IT',89);

## Select -

SELECT \* FROM [dbo].Student;

-- Display Average score of Students in each Course.

SELECT Course, AVG(Score) FROM [dbo].Student GROUP BY Course;

-- Display No of Students, Minimum Score, Maximum Score from each Course.

SELECT

Course,

Count(\*) AS NumOfStudents, MIN(Score) AS MinimumScore, MAX(Score) AS MaximumScore

FROM [dbo].Student GROUP BY Course;

-- Display courses whose average score is greater than 86.

SELECT Course FROM [dbo].Student GROUP BY Course

HAVING AVG(Score) > 86;

## Drop -

DROP TABLE [dbo].Student;

# Practise Set 5 – Foreign Key, Join & Sub Queries

## Create -

CREATE TABLE [dbo].Dept(

DeptID INT IDENTITY(1,1) PRIMARY KEY, Name VARCHAR(50) NOT NULL,

Location VARCHAR(50) NOT NULL

);

CREATE TABLE [dbo].Employee(

EmpID INT IDENTITY(1,1) PRIMARY KEY, FirstName VARCHAR(50) NOT NULL, LastName VARCHAR(50) NOT NULL,

DeptID INT REFERENCES [dbo].Dept NOT NULL, Salary NUMERIC(10,2) NOT NULL CHECK (Salary > 0)

);

## Insert -

INSERT INTO [dbo].Dept(Name,Location) VALUES('Inventory', 'Loc1');

INSERT INTO [dbo].Dept(Name,Location) VALUES('Sales', 'Loc2');

INSERT INTO [dbo].Dept(Name,Location) VALUES('HR', 'Loc2');

­­ \*Ideal as we don’t know the ID of the department

­­ Inventory, we need to query and set it for the

­­ insertion.

INSERT INTO [dbo].Employee (FirstName, LastName, DeptID, Salary)

VALUES('A','A', (SELECT [DeptID] FROM [dbo].Dept WHERE

Name='Inventory'), 11000);

INSERT INTO [dbo].Employee (FirstName, LastName, DeptID, Salary) VALUES('B','B', 2, 12000);

INSERT INTO [dbo].Employee (FirstName, LastName, DeptID, Salary) VALUES('C','C', 3, 21000);

INSERT INTO [dbo].Employee (FirstName, LastName, DeptID, Salary) VALUES('D','D', 3, 22000);

## Select -

SELECT \* FROM [dbo].Dept; SELECT \* FROM [dbo].Employee;

-- Display employees who work for HR department

SELECT \* FROM [dbo].Employee WHERE DeptID in (

SELECT DeptID FROM [dbo].Dept WHERE Name = 'HR' );

-- Display employee and their corresponding department information

SELECT \* FROM [dbo].Employee AS E

INNER JOIN [dbo].Dept AS D ON (E.DeptID = D.DeptID);

-- OR

SELECT \* FROM [dbo].Employee AS E, [dbo].Dept AS D WHERE (E.DeptID = D.DeptID);

-- OR With Meaningful aliases and projections.

SELECT

E.EmpID,

E.FirstName + ' ' + E.LastName AS EmpName, E.Salary,

D.Name AS DeptName, D.Location AS DeptLocation

FROM [dbo].Employee AS E

INNER JOIN [dbo].Dept AS D ON (E.DeptID = D.DeptID);

-- Display employees whose department location is 'Loc2'

SELECT \* FROM [dbo].Employee WHERE DeptID in (

SELECT DeptID FROM [dbo].Dept WHERE Location = 'Loc2' );

-- OR Using Joins

SELECT E.\* FROM [dbo].Employee AS E, [dbo].Dept AS D WHERE ((E.DeptID = D.DeptID) AND (D.Location = 'Loc2'));

## Drop -

DROP TABLE [dbo].Employee; DROP TABLE [dbo].Dept;

# Practise Set 6 – Many to Many with Junction Table

## Create -

CREATE TABLE [dbo].Course(

CourseID INT IDENTITY(1,1) PRIMARY KEY, Name VARCHAR(50) NOT NULL,

Fee NUMERIC(10, 2) NOT NULL

);

CREATE TABLE [dbo].Student(

StudentID INT IDENTITY(1,1) PRIMARY KEY, FirstName VARCHAR(50) NOT NULL, LastName VARCHAR(50) NOT NULL,

);

CREATE TABLE [dbo].CourseEnrollments( CourseID INT REFERENCES [dbo].Course, StudentID INT REFERENCES [dbo].Student

);

## Insert -

INSERT INTO [dbo].Course(Name,Fee) VALUES('SQL',1000);

INSERT INTO [dbo].Course(Name,Fee) VALUES('DS',1500);

INSERT INTO [dbo].Course(Name,Fee) VALUES('C++',1500);

INSERT INTO [dbo].Student(FirstName,LastName) VALUES('a','a');

INSERT INTO [dbo].Student(FirstName,LastName) VALUES('b','b');

INSERT INTO [dbo].Student(FirstName,LastName) VALUES('c','c');

INSERT INTO [dbo].Student(FirstName,LastName) VALUES('d','d');

INSERT INTO [dbo].CourseEnrollments VALUES(1,1);

INSERT INTO [dbo].CourseEnrollments VALUES(1,2);

INSERT INTO [dbo].CourseEnrollments VALUES(2,3);

## Select -

SELECT \* FROM [dbo].Student; SELECT \* FROM [dbo].Course;

SELECT \* FROM [dbo].CourseEnrollments;

­­ Find all students who joined DS SELECT \* FROM [dbo].Student

WHERE StudentID IN (

SELECT StudentID FROM [dbo].CourseEnrollments WHERE CourseID =

(SELECT CourseID FROM [dbo].Course WHERE Name='DS'));

-- Display student information along with Course info

SELECT s.\*, c.\* FROM [dbo].Student AS s INNER JOIN [dbo].CourseEnrollments AS ce

ON (s.StudentID = ce.StudentID) INNER JOIN [dbo].Course as c

ON (ce.CourseID = c.CourseID);

-- Display student information along with Course info. irrespective of whether they

-- joined any course or not.

SELECT s.\*, c.\* FROM [dbo].Student AS s

LEFT OUTER JOIN [dbo].CourseEnrollments AS ce

ON (s.StudentID = ce.StudentID) LEFT OUTER JOIN [dbo].Course as c

ON (ce.CourseID = c.CourseID);

## Drop -

DROP TABLE [dbo].CourseEnrollments; DROP TABLE [dbo].Course;

DROP TABLE [dbo].Student;

# Practise Set 7 – Stored Procedure, Function & Triggers

## Create -

CREATE TABLE [dbo].Student(

StudentID INT IDENTITY(1,1) PRIMARY KEY, FirstName VARCHAR(50) NOT NULL, LastName VARCHAR(50) NOT NULL,

);

CREATE TABLE [dbo].Hist\_Student(

Hist\_StudentID INT IDENTITY(1,1) PRIMARY KEY,

StudentID INT,

FirstName VARCHAR(50) NOT NULL, LastName VARCHAR(50) NOT NULL,

ModifiedDate DateTime NOT NULL

);

## Insert -

INSERT INTO [dbo].Student(FirstName, LastName) VALUES('A','A');

INSERT INTO [dbo].Student(FirstName, LastName) VALUES('B','B');

## Procedure -

CREATE PROCEDURE MyProcedure(@id int) AS

BEGIN

SELECT \* FROM [dbo].Student WHERE StudentID = @id;

END GO

-- Executing Procedure

EXEC MyProcedure @id = 1;

## Function -

CREATE Function MyFunction(@id int) RETURNS VARCHAR(50)

AS BEGIN

DECLARE @name VARCHAR(50);

SELECT @name = (FirstName + ' ' + LastName ) FROM [dbo].Student

WHERE StudentID = @id; RETURN @name;

END GO

-- Using Function

SELECT

StudentID, [dbo].MyFunction(StudentID) AS Name

FROM [dbo].Student;

## Trigger -

CREATE TRIGGER CaptureStudentUpdate ON [dbo].Student AFTER UPDATE

AS BEGIN

INSERT INTO [dbo].Hist\_Student

(StudentID, FirstName, LastName, ModifiedDate) SELECT

deleted.StudentID, deleted.FirstName, deleted.LastName, GETDATE()

FROM deleted;

END GO

-- Test

SELECT \* FROM [dbo].Student; SELECT \* FROM [dbo].Hist\_Student;

UPDATE [dbo].Student SET LastName = 'P' WHERE StudentID = 1;

SELECT \* FROM [dbo].Student; SELECT \* FROM [dbo].Hist\_Student;

## Drop -

DROP TRIGGER [dbo].CaptureStudentUpdate; DROP FUNCTION [dbo].MyFunction;

DROP PROCEDURE [dbo].MyProcedure; DROP TABLE [dbo].Hist\_Student; DROP TABLE [dbo].Student;

*Further Reading :* [*http://www.w3schools.com/sql/default.asp*](http://www.w3schools.com/sql/default.asp)