**Core Java Notes**

**Jdk 1.8.0 download & install**

**Drive change:** C:/user/evvu(system name)>cd /d E:\java(folder name)

Go to my computer properties click on advance then click environment and path set **C:/program Files/java/jdk1.8.0\_144/bin** and click ok, ok, ok

**Cmd: E:\java>javac filename.java**

**E:\java>java class name**

**\*Type Conversion in case of any Expressions:**

short

byte int long float double

char

Any arithmetic expression these can be different types of values. Before the operator numeric type of value always is converted into another type implicitly:

**Float:** (long) 5 + 5;

res = 5 + 5 + 5.5 + 5.5

res = 10 + 5.5 + 5.5

res = 15.5 + 5.5

res = 20

**Double:**

**\*Variable\***

1. Variable are the name of memory location variable are also known as the container of the values.
2. Variable always be declared of any specific type.
3. Read and write are two operations for preformed on to the variable.
4. Variable are also the identifiers so, we have to follow the values of the naming the variables.

**Example: int x, y;**

**X= 5;**

**Y= 6;**

**\*Keyword\***

Keywords are the reserved words that always have their special meaning in java, these are **51 keywords** **and** **2 extra reserve words** that not a keyword in the current version of java.

|  |  |
| --- | --- |
| 1. **Data Type:**    1. byte    2. short    3. int    4. long    5. float    6. double    7. boolean | 1. **Control Statement:**     1. if    2. else    3. for    4. while    5. do    6. switch    7. case    8. default    9. continue |
| 1. **Access Specific:**     1. public    2. protected    3. private    4. static | 1. **Class / Object / Method:**    1. class    2. interface    3. enum    4. abstract    5. super    6. this    7. new    8. extends    9. implements    10. instanceof    11. void    12. return |
| 1. **Exception Handling:**    1. try    2. catch    3. finally    4. throw    5. throws | 1. **Package:**    1. package    2. import |
| 1. **Threading & i/o:**    1. synchronized    2. volatile | 1. **Debugging:**    1. Assest |
| 1. **reserved but value keyword:**    1. goto    2. count | 1. **reserved words for (constant):**    1. true    2. false    3. null |
|  |  |

**Note:-**

1. Keywords always be written in the small letter.
2. Keywords can never be word as the identifier.

**\*Operators\***

Operators are word to perform the operations depending upon the number of operands there are three types of operators:-

1. **Unary operators:** Always works on the single operand.
2. **Binary operators:** Always works on the two operands.
3. **Ternary Operators:**  Always works on the three operands.

**Type of operators based on type of operands-**

1. **Arithmetic Operators:** These are the binary operators and works upon the numeric type:

**‘ + ’ , ‘ - ’ , ‘ \* ’ , ‘ / ’ , ‘ % ’**

Each and every operation always have their priority and associational.

**Priority Operations Associational:**

|  |  |  |
| --- | --- | --- |
| **High** | \* , / , % | Left to Right |
| **Low** | + , - | Left to Right |

1. **Relational Operations:**  These are the binary operators and always used numeric values as operands (some operation can use a Boolean type also as the operand).

Relational operator always is Boolean type:

**‘ > ’ , ‘ < ’ , ‘ >= ’ , ‘ <= ’ , ‘ == ’ , ‘ != ’**

**Example:** Boolean b;

b = 5 >3;

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

1. **Logical Operation:** Operands of logical operator always be the Boolean type:

**Binary:** 1. && (AND)

2. (OR)

**Unary:** ! (NOT)

|  |  |  |
| --- | --- | --- |
| **Operand 1** | **Operand 2** | **Result** |
|  |  | && OR |
| T | T | T T |
| T | F | F T |
| F | T | F T |
| F | F | F F |

**Example:** Boolean b;

b = 5 > 6 && 3 < 5;

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

1. **Increment / Decrement Operator:**

**Pre :** ++x , --x;

**Post:** x++ , x--;

int x , y;

x= 5;

y= x++;

System.out.println ( x ); // 6

System.out.println ( y ); // 6

**\*JVM (Java Virtual Machine)\***

JVM is an abstract computing machine that enables a computer to run a java program.

**There are three nations of JVM:**

1. Specification.
2. Implementation.
3. Instance.
4. **Specification:** A specification is a document that formally describes what is requirement of JVM implementation. Having a single specification ensure all implementation are interoperable.
5. **Implementation:** A JVM implementation is a computer program that meets the requirement of the JVM specification.
6. **Instance:** An instance of a JVM implementation is an implementation running in a process that executed computer program complied into byte code.

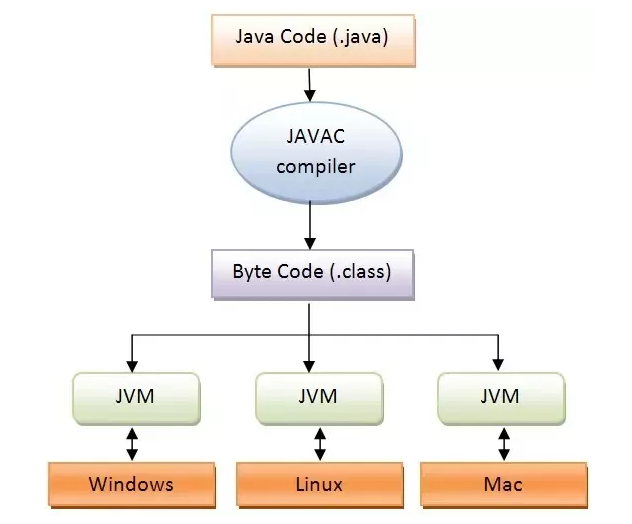
**Byte Code**

**App / Software**

**JVM**

**Window**

**Hard**



JVM is an implementation running in a process that executes a computer program compiled into java byte code.

It is a specification that provide runtime environment in which java byte code can be executed.

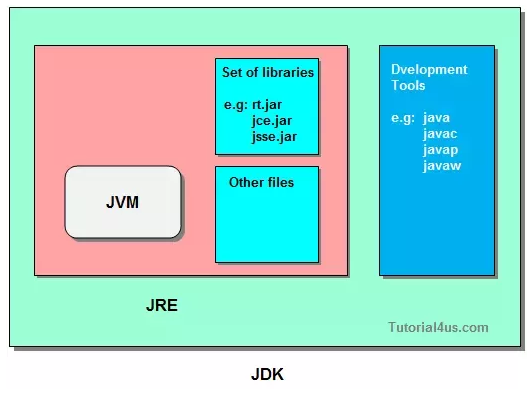
**\*JRE (Java Runtime Environment):**

1) Implementation of JVM (hotspot virtual machine).

2) Library.

**\*JDK (JSE Development Kit):**

The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets. JDK includes JRE and Development Tools.

****

**1) Development Tools:** It used for create and build applications

2) JRE32

**Note :**  JSE (Java Standard Edition)

1. int x, y;

x= 5;

y= x++ **+** ++x **+** x++;

System.out.println( x ); //8

System.out.println( y ); //19

1. int x, y, z;

x= 5;

y= 6;

boolean b= ++x < y **&&** y++ == x;

System.out.println( x ); // 6

System.out.println( y ); // 6

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

**\*JIT (Just In Time):**

Byte code converts to OS.

JIT is 3rd type language:

1. A part from compiler & interpreter.
2. It is the combination of compiler & interpreter.
3. The working of JIT as follows:

**Whenever any method is invoked 1st time JIT converts its instruction into the native code (assembly of OS) from the byte code and stored into the temporary cache area within the JVM in the main memory (RAM).**

Whenever the same method is invoked again then JIT picks up their converted instruction from the cache and provided to the OS for the execution.

**JVM Diagram:** Byte code

**Main()**

M1()

-----------

1.1 1.6 1.5

M1()

-----------

1.2

1.3

**cache**

1.4 1.7

**OS**

* 1. 1st time m1 method invoked.
  2. Conversion of the instruction one by one and stored into the cache and provide to the OS for execution.
  3. 1.2
  4. 1.2
  5. Conversion and execution completed and control returned into the main method.
  6. M1 method called again.
  7. Converted instructions picked up cache & provide to the OS for execution.

**This is the working JIT.**

**Note:** In java native code (executable code) can never be visible. Because it always be produce runtime.

**\* OOP’s (Object Oriented Programming)\***

Object Oriented is the methodology to develop any program in any programming language that follow the concept of oop’s.

In this methodology the programming solution of any real world problem can be easily implemented, same as the real world solution.

**The benefits of object oriented methodology:-**

Developers can easily map the real world problems to the programming solution easy development.

The development change & manipulation upon easy as each & everything is a separate unit that is object.

In the object oriented programming, the programming implementation of the real world problem completely around the object.

The concepts of object oriented methodology are prepared from the real world.

Any programming language that follows the concept of oop’s will be considered as the object oriented programming language: such as C++, java, dot (.)net, php etc.

**Concepts of object oriented programming:-**

1. Class & Object
2. Encapsulation & Abstraction
3. Inheritance & Polymorphism

Depending upon the implementation of the above concept further the programming language can be categorized as follows:

1. **Object based programming** language concept of classes & objects exist encapsulation & abstraction also exist but no inheritance & polymorphism

**Example: a.** java script

**b.** VB

1. **Object oriented programming** language:

All concept of OOP’s but that other concepts also can be existed, such as primitive data types, function without class.

Pure object oriented programming language:-

All the programming concepts are available in form of objects. Even number primitive data types

Example**:** small talk.

**\*Class & Objects\***

1. **Classes:**

Class is the categorization of objects.

Class is the definition of the object.

Class is the logical (conceptual) representation of the object

as the definition in the class there are the properties & behaviors of object desired.

1. **Object:**

Object is the physical representation of anything.

Object is the implementation of the class.

Object is the thing that can for ------- their output (result) by their behavior.

Object has Properties & behaviors.

* + - 1. **Property:**

Property of object is used to make the state the object.

Example: color, price, model of a car.

Color of my car is red.

Color is property and red is value of the property.

When we assign the values in the properties of object then the state of object will be prepared.

* + - 1. **Behavior:**

Behavior is the working of the object that means behavior is functionality of the object.

Keyword Identifier

class <class name >

{

Variable declarations

Method declaration

}

1. **Java standard:** Class name start caps letter, $ but not number
2. **Library define:** S caps of String in main method and System.out.println

**Note:**  main method call JVM

System.out.println()

Object method

class Test

{

public static void main(String arg[])

{

int x, y, z;

x= 5;

y= 6;

z= x+y;

System.out.println(“z: ” +z);

}

}

Where same operator like: z= x+y;

System.out.println(“z: ” +z);

Then +z “+” is constant, it’s not user define

There is method overloading.

**\*Encapsulation\***

When affect property from behaviors it’s known as encapsulation.

**“**Properties of the object affected by it behaviors are known as the encapsulation.**”**

**OR**

**“**Wrapping up the data and behaviors in the single unite is known as the encapsulation.**”**

**In the above definition:**

**Unit:** is the object.

**Data:** properties of object.

**Behaviors:** functionality of the object.

**Example:**

1. Written behaviors of the pen will affect the ink property.
2. Eating the nutritive food will increase the calories human body**.**

**\*Abstraction\***

Abstraction means the hiding the complexities and opens up the required user interface.

Abstraction always says that what will be the outcome of any behaviors of any product just we need to know.

How the outcome of a behaviors of product is generated need not to know.

**Example:**

1. After eating the nutritive food the calories will be increased we know it, but how those calories generated in the human body we it don’t know.

**Programming implementation of class and object:**

**class:**  keyword is used to declare the any class.

**Syntax:**

Identifier

class <class name>

{

Variable declaration; (properties of object)

Method declaration (behavior of object)

}

**Variables:** variable are the memory location.

1. Valued type variable
2. Reference type variable

**But only reference type variable used in java always**

**Program: 1**

class Student

{

int roll no, marks;

char grade;

void admission(int r)

{

roll no= r;

System.out.println(roll no+ “She enroll”);

}

void exam(int m)

{

marks=m;

System.out.println(roll no+ “got the ” +marks+ “marks”);

if(marks>80)

grade=’A’;

else

if(marks>=60)

grade=’B’;

else

if(marks>=40)

grade=’C’;

else

grade=’D’;

}

void report()

{

System.out.println(“Roll no: ”+roll no);

System.out.println(“Marks: ”+marks);

System.out.println(“Grade: ”+grade);

}

}

**new keywords:** new keyword is used create the object in java.

**Syntax:**

new <class name>();

**example:**

new Student();

After the creation of object we have to store its reference id (Processed address) in our program to use at object.

In java objects are not the value type whether they are reference type.

**Reference variable:** reference variable always the class type variables that are used to store the reference id of the object.

Declaration of reference variable as follow:

Student st1, st2;

St1, st2: reference variable

new keyword always creates the new object and returns their reference id that we can store reference variable.

Without storing the reference id in the reference variable we can never use object further.

**Example:**

Student st1;

1.3

St1= new Student();

1.4 1.1 1.2

Roll no()

Marks()

Grade()

* 1. new object created
  2. reference id returned
  3. reference id copied into st1 variable
  4. object reference by st1 variable

st1.addmission();

**. (dot) :** association

**Main Method class creates of program 1:**

class School

{

public static void main(String s[])

{

Student st1;

st1= new Student();

st1.addmission(101);

st1.exam(85);

st1.report();

} }

**Output:**

101 she enroll

101 got the 85

Roll no: 101

Marks: 85

Grade: A

**\*Command Line Argument:**

1. Command line argument is the way to get the values at the runtime from side JVM in the argument of the main method.
2. All the command line argument separated by the space.
3. Any number of command line argument begin pass in argument main method.
4. All the command line argument always is in the form of string.

**Java Test ABC XYZ MNO**

**Secondary Memory**

1. 2.  **JVM** 5.

------

3.

4. **Test class**

public static void main(------)

{

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

}

}

**String[] Test class**

0 6.

1

2

1. JVM started.
2. Name of the class provided to the JVM.
3. Class loaded by the JVM from the secondary memory.
4. Array of the type string created with the size equal to the no. of argument.
5. Command line values are copy into the array.
6. Main method invoked and array is passed in the argument.

**Note:** The pattern of using the array in java is say as in the C language but create of array is different on the C.

class Test

{

public static void main(String s[])

{

for(i=0; i>s.length; i++ )

System.out.println( s[i] );

}

}

Command line java Test ABC XYZ MNO

**OUTPUT:**

ABC

XYZ

MNO

**Assignment:**

**class Employee**

**properties: code, salary, tax, netsalary**

**behaviours:**

**- void join (int c)**

**To set the code**

**- void taxcalc ( )**

**To calculate the tax**

**10% of the salary**

**netsalary= salary-tax**

**- void report ( )**

**To show the values**

**Program-2:**

**Class 1**

class Employee

{

int code, salary, tax, netsalary;

void join( int c )

{

code= c;

System.out.println(“Employee Code: ” +code);

}

void taxcalc( int s )

{

salary= s;

tax= salary \* 10/100;

netsalary= salary-tax;

System.out.println(“Employee Netsalary: ” +natsalry);

}

void report( )

{

System.out.println(“Code: ” +code);

System.out.println(“Salary: ” +salary);

System.out.println(“Tax: ” +tax);

System.out.println(“Netsalary: ” +netsalary)

}

}

**Class 2 main method**

class Org

{

Public static void main(String s[]) **// s variable of array**

{

Employee e1, e2; **// two variable create e1 and e2**

e1= new Employee; **// Object perform behaviors**

e2= new Employee;

e1.join(101);

e2.join(102);

e1.taxcalc(30000);

e2. taxcalc(5000);

e1.report();

e2. report();

}

}

**OUTPUT:**

Employee Code: 101

Employee Code: 102

Employee Netsalary: 27000

Employee Netsalary: 4500

Code: 101

Salary: 30000

Tax: 3000

Netsalary: 27000

Code: 102

Salary: 5000

Tax: 500

Netsalary: 4500

**Reference id:** reference id of any one object can be stored in the more than one reference variables.

**Changes in the main method:**

class report

{

Public static void main(String s[])

{

Student st1, st2;

st1= new student();

st2=st1;

st1.addmission(101);

st2.exam(85);

st1.report( );

}

}

In property of the object there are always default values.

In java this no garbage values in any variable.

|  |  |
| --- | --- |
| **Data Types** | **Default Values** |
| **Primitive:**   * Numeric[byte, short, int, long, float, double, char ] * Non-numeric[boolean] | 0  False |
| Reference variable | Null |

**Practice Code:**

class Book

{

int code, price;

boolean status;

void setBook(int c, int p)

{

code=c;

price=p;

status=true;

System.out.println(code+ " book registered successfuul");

}

void issueBook()

{

if(status)

{

System.out.println("Book issued successfuul");

status=false;

return;

}

System.out.println( "Book already successfuul");

}

void returnBook()

{

if(!status)

{

System.out.println("Book returned successfuul");

status=true;

return;

}

System.out.println("Book is already available");

}

void report()

{

System.out.println("Code: " +code);

System.out.println("Price: " +price);

if(status)

{

System.out.println("Status: Avialable");

}

else

System.out.println("Status: Issued");

}

}

}

class Library

{

public static void main(String s[])

{

Book b;

b= new Book();

b.setbook(102, 1000);

b.issuebook();

b.returnbook();

b.report();

}

}

In one reference variable we can store the reference id of only one object.

If any reference variable already has the reference id of any object then we assign the reference id of previous object will be overridden in the reference variable.

class Library

{

Public static void main(String s[])

{

Book b1;

b1= new Book();

b1.setBook(101, 550);

b1.new Book();

}

}

In the execution of the above program the list object becomes the unreferenced object.

Unreferenced object is the object whose reference id is not stored in any reference variable in the program.

That means if any object is not referenced id by any reference variable that object will be known as the unreferenced object.

Garbage collector is the utility in the JVM which is responsible to destroy all the unreferenced objects from the memory.

During the execution of any program garbage collector automatically invoked timely destroy the unreferenced object.

We can never get the reference ID or unreferenced object again in our program.

**\*There are the main calling bases in which any object can be the unreferenced.**

**1)** As discuss in last program.

**2)** Creating the anonymous object.

new Book();

**3)** Storing the null value explicitly in the reference variable.

Book b1;

b1= new Book();

b1= null;

In any reference variable we can store two types of values:

1. Either the reference ID of any object.
2. null value

**4)** Storing the reference ID of object in the local reference variable.

**\*Memory architecture of JVM:**

When the JVM loaded in the main memory for the execution of the program it always prepared the memory in the five different reasons. In which the program and it components will be resides.

**\*There are five memory reasons in the JVM:**

**Native area**

**Stack area**

**Heap area**

**Class / method area**

**PC registration**

1. **class / method area:** All the dot(.) class file / white code of the classes always be stored in this area.

That means whenever the execution of any class is required in the program when JVM always loads white code of that class from secondary storage to this area.

In this area for the separate classes the separate blogs will be created.

**Class area**

**class A class B class N**

Byte code ---------------------------------------------------

Memory for static variables

Byte code ---------------------------------------------------

Memory for static variables

Byte code ---------------------------------------------------

Memory for static variables

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

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

1. **Heap area:** All the objects always be created in this area that means whenever create object with new keywords it always be created in this area.

**Heap area**

Objects

**Note:** there is only one class area and one heap area in the JVM which is shared by all the threads running under the JVM.

1. **Stack area:** stack area is used in the support of any method execution that means whenever any method is called immediately new frame is created in the stack for the execution of the method. In the frame of the method the code of method never be copied rather in that frame the memory to the local variable of the method will be allocated.

As the execution method completed that immoderately the frame of the method destroy stack and all the local variable also destroy from the stack.

In the frame of method the instruction pointer will also be maintained to point the instruction of the method in the class area that is to be executed.

Method calls local variable.

**Stack area**

Code

Price

IP

Heap

`

frames

setBook()

s

b1 110

IP

memory to local

variable of main()

stack

1. **Native area:** native area in this area native code always be placed for the execution.

Through the native code the functions native languages such as- C, C++, perl etc can be invoked.

In java native code always be written within the help of (native keyword) and JNI( Java Native Interfaces).

JNI is one of library java.

With the help of native code JVM always interact with the OS.

1. **PC registration (Program counter register):** This is the smallest area in which always the next instruction loaded that is to be executed.

**Changes in the Library class:**

class Library

{

public static void main(String s[])

{

System.out.println(“Welcome in the main”);

m1(); //calling the m1 method

**1.** System.out.println(“Thank You”);

}

Static void m1()

{

**2.** Book b1;

b1= new Book();

b1.setBook(102, 1000);

**3.** b1.report();

}

}

**heap**

b1 110

**2.2** object of book

m1() **4.** **5.**

1. m1 method is called and new frame created in the stack and memory to the local variable (b1) assigned in the frame.

**2.1** new keyword encounters to create object.

**2.2** object is create and referenced by b1.

**3.** m1 method finished and control is returned into the main.

**4.**  frame is destroyed and local variable also the destroyed.

**5.** object become in reference.

In java we can never compare that two objects according to their data by using any built in operators.

But we can compare the reference variables, in that case only the reference id will be compared.

class Library

{

Public static void main(String s[])

{

Book b1, b2;

b1= new Book();

b2= new Book();

b1.setBook(101, 550);

b2.setBook(101, 550);

System.out.println(b1==b2); **//false**

Book b3;

b3=b1;

System.out.println(b1==b3); **//true**

}

}

In order to compare the two objects according to their data we have to make extra programming efforts.

Creating and extra method in the book class

class Book

{

int code, price;

boolean status;

void setBook(int c, int p)

{

code=c;

price=p;

status=true;

System.out.println(code+ " book registered successfuul");

}

void issueBook()

{

if(status)

{

System.out.println("Book issued successfuul");

status=false;

return;

}

System.out.println( "Book already successfuul");

}

void returnBook()

{

if(!status)

{

System.out.println("Book returned successfuul");

status=true;

return;

}

System.out.println("Book is already available");

}

void report()

{

System.out.println("Code: " +code);

System.out.println("Price: " +price);

if(status)

{

System.out.println("Status: Avialable");

}

else

System.out.println("Status: Issued");

}

boolean compare(Book b)

{

boolean res= code==b.code && price==b.price && status==b.status;

Return(res);

}

}

class Library

{

public static void main(String s[])

{

Book b1, b2;

b1= new Book();

b2= new Book();

b1.setBook(101, 550);

b2.setBook(101, 550);

System.out.println(b1==b2); **//false**

System.out.println(b1.compare(b2));

}

}

**Heap**

**110**

b 220

**3.2**

**3.1** compare()

b1 110

b2 220

**2. 220**

main()

**1.** **stack**

java Library(command in doc)

**3.1** b=b2(internal code)

110 and 220 are reference id of object

**\*Format of the reference id:** The reference ids of object always have the following syntax:

**ClassName@hexadecimalValue**

**hexadecimalValue:** always the generated by the actual address of the object.

class Library

{

Public static void main(String s[])

{

Book b1, b2;

b1=new Book();

b2=new Book();

System.out.println(b1);

System.out.println(b2);

}

}

**OUTPUT:**

Book@15db9742

Book@6d06d69c

At a time of calling of any method in the argument we can call another method but the return type of argument method must be same with the argument of the method.

boolean compare(Book b)

{

}

**We are written this code both type**

boolean res= code==b.code && price==b.price && status==b.status;

Return(res);

**OR**

return(code==b.code && price==b.price && status==b.status);

**code:** stored b1 value

**b.code:**  stored b2 value

and same price & status

**b1: b2:**

code:101 code:101

price: 550 price: 550

status: true status: true

So return print: true because b1 & b2 is equal

**\*Keywords:**

**‘this’ keyword:** In the behavior of any object ‘this’ keyword always use to get the reference id of current object.

**OR**

‘this’ keyword provide the reference id of the current object in the method of object.

Within the method (behavior) of the object the properties and another behaviors of the same class always we use by ‘this’ keyword implicitly.

Compiler automatically placed ‘this’ keyword within the properties of the same class.

class Book

{

int code, price;

boolean status;

void setBook(int c, int p)

{

this.code=c;

this.price=p;

this.status=true;

this.report();

System.out.println(this);

}

void issueBook()

{

if(status)

{

System.out.println("Book issued successfuul");

status=false;

return;

}

System.out.println( "Book already successfuul");

}

void returnBook()

{

if(!status)

{

System.out.println("Book returned successfuul");

status=true;

return;

}

System.out.println("Book is already available");

}

void report()

{

System.out.println("Code: " +code);

System.out.println("Price: " +price);

if(status)

{

System.out.println("Status: Available");

}

else

System.out.println("Status: Issued");

}

}

class Library

{

public static void main(String s[])

{

Book b1, b2;

b1= new Book();

b2= new Book();

b1.setBook(101, 550);

System.out.println(b1);

System.out.println(b2);

}

}

**OUTPUT:**

Code: 101

Price: 550

Status: Available

Book@15db9742

Book@15db9742

Book@6d06d69c

**There are two main practical uses of ‘this’ keyword:**

1. Implicit use: as discussed in previous program.
2. Explicit use: data shadowing

**Data shadowing:** When the name of the local variable and the instance variable (properties) are same then it will be known as the data shadowing.

The property of local variable will always be hire then the instance variable properties.

If we want to use the instance variable by skipping the local variable when we have to use ‘this’ keyword explicitly

void setBook(int code, int p)

{

this.code= code;

price= p;

status= true;

}

// same method

When any method is invoked by the reference variable then the reference variable itself passed as an extra argument and in the signature of the method an extra parameter with name this will be available.

void setBook(int code, int p, Book this)

{

this.code= code;

price= p;

status= true;

}

class Library

{

public static void main(String s[])

{

Book b1, b2;

b1= new Book();

b2= new Book();

b1.setBook(101, 550, );

b2.setBook(102, 600);

}

}

**OUTPUT:**

**Pending**

**‘static’ keyword:**  static keyword can be used with the variables, methods and within the nested class (class within class).

‘static’ keyword makes the member of class (properties and behaviors) or the class itself that means not for the object.

In any class there can be always two types of properties and behaviors.

**Members of class**

static non-static(instance)

properties behaviors properties behaviors

1. **static member:** static member never the accessed with the object rather they always we used with class name.
2. **non-static member:** non-static member are actually members of the object and can never be accessed without the object (reference id of object).

class sample

{

static int x;

int y;

void m1()

{

}

static void main()

{

}

}

**Static variable:**

* These are also known as the class variable.
* Always the declared at the class level not at the local level.

class Test

{

static int x;

void m1()

{

Static int a; **//error**

}

}

That means static keyword can never be used within any method.

* static variable also have the defaults values like the non-static variables.
* static variable always the created in the class area within the JVM.
* Only single copy of the static variable will be created in the memory.

**OR**

Within the JVM static variables always gets the single time memory.

At the time of class loading static variable gets the memory.

* Within the same class static variable can be used in any method directly but outside the class static variable always be used by the class name.

**Example:**

class Test

{

static int x;

int y;

void m1(int val)

{

x= x+val;

y= val;

}

void show()

{

System.out.println(“Y: ”+y);

}

}

class StaticTest

{

Public static void main(String s[])

{ Test t1, t2;

t1= new Test();

t2= new Test();

t1.m1(55);

t2.m2(65);

t1.show();

t2.show();

System.out.println(“Value of x: ”+Test.x);

}

}

**class area**

**class Test class static Test**

Byte code

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

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

Byte code

Static int x

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

x

120

**stack** **Heap**

**110**

55

t1

t2

110

220

**220**

65

main()

**call by t1**

default value of x= 0

val=55

methos m1():

x= x+ val

x= 0+ 55

now, x= 55 its value store in class area because x is static variable so x is common

y= val

y= 55

y is non-static so its value stored in heap area

**call by t2**

x= 55

val=65

methos m1():

x= x+ val

x= 55+ 65

now, x= 012 its value store in class area because x is static variable so x is common

y= val

y= 65

y is non-static so its value stored in heap area

**NOTE:**

static variables always be accessed by the class name outside the class.

Opticianry, we can always use the static variable by the reference variable but compiler automatically change the reference variable in the class name.

**Changes in main method:**

System.out.println(“value of x: ” +t1.x); //Test.x

**Test.x -**  written by the complier in the white code.

**Static method:** Static methods are also known as the class method.

* Outside the class they always be invoked by the class name.
* In the static method we cannot use this keyword.
* In the static method we can use the static member of the same class directly but non-static members either of the same class we always be used by reference id of the object.

**Example:**

class Test

{

static int x;

int y;

static void m1(int val)

{

x= x+val;

y= val; //error

show(); //error

}

void show( )

{

System.out.println(“y: ” +y);

}

}

**Static method always called by class name.**

**NOTE:**

**Q.1 Within the static method we can never use the non-static members directly, why?**

**Ans.** static methods never be invoked by the object rather they always be invoked by the class name. So reference id of any object is not available in the static method implicitly.

**Q.2 In the non-static method we can use the static members directly, why?**

**Ans.**  static member are the class members and only one copy of static members are created in the class area so there is no confusion to the JVM about the accessing of the static members.

**Q.3 How we can get reference id of object in the static method?**

**Ans.** There are number of ways to get the reference id of object in the static method but two common ways as follows:

**1). First way -** Taking the reference id in the argument of the method.

**2). Second way -** Creating the object by new keyword get the reference id of the object.

**First way-** changes in m1() method

class Test

{

static int x;

int y;

static void m1(Test t, int val)

{

x= x+ val;

t.y= val;

t.show( );

}

void show( )

{

System.ot.println(“Y: ” +y);

}

}

class Student

{

// SAME//

public static void main( String s[] )

{

Test t1, t2;

t1= new Test( );

t2= new Test( );

Test.m1( t1, 50 );

}

}

112

**Second way-** changes in m1() method

class Test

{

static int x;

int y;

static void m1(Test t, int val)

{

x= x+ val;

Test t= new Test();

t.y= val;

t.show( );

}

void show( )

{

System.ot.println(“Y: ” +y);

}

}

class Student

{

// SAME//

public static void main( String s[] )

{

Test t1, t2;

t1= new Test( );

t2= new Test( );

Test.m1( t1, 50 );

}

}

**Practice Code**

**1.**

class Salesman

{

int id, nos, netsal, commission;

static int totalsale;

void join (int id)

{

this.id= id;

}

void dosale (int amt)

{

netsal= netsal + amt;

System.out.println(“Sale of ”+id “salesman is ” +amt );

nos++;

}

void totalSalecalc ( )

{

totalsale = totalsale +netsal;

}  
 void report ( )

{

System.out.println(“\nID: ”+id);

System.out.println(“Net sales: ”+netsal);

System.out.println(“No. of sales: ”+nos);

}

void commcalc ( )

{

commission=(int)(netsal\*0.1);

}

static void show totalesal ( )

{

System.out.println(“Total sales: ”+totalsal);

}

}

class Company

{

static int totalcomm;

public static void main( String s[] );

{

Salesman s1, s2;

s1= new Salesman();

s2= new Salesman();

s1.join(111);

s2.join(222);

s1.dosale(10000);

s2.dosale(20000);

s1.dosale(15000);

s2.dosale(20000);

s1.totalSalecalc();

s2.totalSalecalc();

s1.commcalc();

s2.commcalc();

System.out.println(“Report of first salesman: ”);

s1.report();

System.out.println(“Report of second salesman: ”);

s2.report();

totalcomm= totalcomm + s1.commision;

totalcomm= totalcomm + s2.commision;

System.out.println(“\nTotal sales: ” +Salesman.totalSale);

System.out.println(“Total commision: ” +totalcomm);

}

}

**2.**

class Employee

{

int code, salary;

static int totalEmp;

void set(int code, int salary, Department d);

{

this.code= code;

this.salary= salary;

d.join();

totalEmp++;

}

void report( );

{

System.out.println(“\nCode: ”+code);

System.out.println(“Salary: ”+salary);

}

}

class Department

{

int code, count;

void set( int code );

{

this.code= code;

}

void join( );

{

count++;

}

void report( );

{

System.out.println(“\nDepartment code: ”+code);

System.out.println(“Number of employee: ”+count);

}

}

class Company

{

static int totalsalary;

static void newEmployee( Employee e );

{

totalsalary= totalsalary + e.salary;

}

public static void main( String s[] );

{

Department d1, d2;

d1= new Department();

d1.set( 123 );

d2= new Department();

d2.set( 456 );

Employee e1, e2, e3;

e1= new Employee();

e2= new Employee();

e3= new Employee();

e1.set(101, 20000, d1);

e2.set(102, 30000, d1);

e3.set(103, 25000, d2);

newEmployee(e1);

newEmployee(e2);

newEmployee(e3);

d1.report();

d2.report();

e1.report();

e2.report();

e3.report();

System.out.println(“\nTotal Employees: ”+Employee.totalEmp);

System.out.println(“Total salary: ”+totalsalary);

}

}

**\*Constructor\***

constructor is the special type of method that is used to initialize the object.

**Characteristics:**

* 1. Name of constructor always be same as the class name.
  2. Constructors never have any return type.
  3. Constructors are automatically invoked then the object of the class is created.
  4. We can use this keyword in the constructor.
  5. Constructor always be the non-static member of the class that means they never be the static.

class Test

{

int x;

Test() **// constructor**

{

System.out.println(“Constructor of the Test class”);

x= 20;

}

void show()

{

System.out.println(“Value of x= ” +x);

}

public static void mains( String s[] )

{

Test t;

System.out.println(“Welcome in main”);

t= new Test();

t.show();

}

}

**Object initialization:** objectinitialization means not to assign the default values in the properties of object rather to assign the real word data values in the properties of the object.

* In the object initialization it is not necessary to assign the values in all the properties rather to properties rather to assign the values in some selected properties.

**Types of constructor:**  There are two types of constructor-

* 1. **Default constructor:**  They are also known as the no argument constructor.

class Test

{

int x;

Test()

{

System.out.println(“Constructor of the Test class”);

x= 20;

}

void show()

{

System.out.println(“Value of x= ” +x);

}

public static void main( String s[] )

{

Test t;

System.out.println(“Welcome in main”);

t= new Test();

t.show();

}

}

* 1. **Parameterized constructor:** In this type of constructor there is always the specified argument list. At the time of object creation we have to pass the value or these arguments.

class Test

{

int x;

Test( int val )

{

System.out.println(“Constructor of the Test class”);

x= val;

}

void show()

{

System.out.println(“Value of x= ” +x);

}

public static void main( String s[] )

{

Test t;

System.out.println(“Welcome in main”);

t= new Test( 10 );

t.show();

}

}

**Important Note:**

* If programmer does not write any constructor in the class then compiler provides the default constructor in the dot( . ) class file.
* If programmers write any constructor in the class then compiler never provides the default constructor.

**Javap tool:** This tools of the JDK is use to list out the members (variables, methods and constructors) of the class.

javap tool always works upon the .(dot) class file.

javap Test

**Constructor Overloading:** If in any class area more than one constructor then it will be known as the constructor overloading.

The constructor can be overloaded in the following two ways:

* 1. By changing the number of arguments.
  2. By changing the type of arguments.
  3. constructor overloading by changing the number of arguments:

class Area

{

int x, y, z, res, type;

Area (int x)

{

this.x= x;

type= 1;

}

Area (int x, int y)

{

this.x= x;

this.y= y;

type= 2;

}

Area (int x, int y, int z)

{

this.x= x;

this.y= y;

this.z= z;

type= 3;

}

void calc( )

{

if(type==1);

{

res= x\*x;

}

if(type==2);

{

res= x\*y;

}

if(type==3);

{

int s= ( x+y+z)/2;

res= (int) (Math.sqrt(s\*(s-x)\*(s-y)\*(s-z)));

}

}

void show( )

{

if(type==1);

{

System.out.println(“Area of square: ”+res);

}

if(type==2);

{

System.out.println(“Area of rect: ”+res);

}

if(type==3);

{

System.out.println(“Area of triangle: ”+res);

}

}

public static void main( String s[] )

{

Area sqr, rec, tri;

sqr= new Area( 5 );

rec= new Area( 5, 6 );

tri= new Area( 5, 6,7 );

sqr.calc( );

rec. calc( );

tri. calc( );

sqr.show( );

rec. show( );

tri. show( );

}

}

class in the java library

**Math.sqrt()**

Static method in the math class

sqrt() method is used to calculate the square root of any argument value.

**public static double sqrt(double value)**

* 1. constructor overloading by changing the type of arguments

class Test

{

Test (float f)

{

System.out.println(“First constructor”);

}

Test (int x)

{

System.out.println(“Second constructor”);

}

public static void main(String s[])

{

New Test(5.5 f);

New Test(55);

New Test(‘A’);

}

}

First of all exact match will be search if not found then nearest implicit conversion will be application.

**Constructor overloading on the bases of type of arguments with more than one argument:**

class Test

{

Test (int x, float ft)

{

System.out.println(“First Constructor”);

}

Test (char x, float ft)

{

System.out.println(“Second Constructor”);

}

public static void main(String s[])

{

new Test( 5, 5.5f );

new Test( ‘A’, 5.5f );

new Test( 5, 5 );

new Test( 5.5f, 5.5f ); **// compilation error**

new Test( ‘A’, ‘A’ );

}

}

When there are more than one argument in the overloaded constructor then first of all exact match will be searched, if not found then,

**Step 1:** The constructor will be selected that can accept the parameter passed in the object creation.

**Step 2:** After the first step the suitable constructor will be search for the individual arguments.

**Step 3:** If for the individual argument with suitable constructor will be same then it will be invoked.

If for individual arguments the suitable constructor are different then compilation error will be generated as “**Ambiguous calling**” of the constructor.

**Note:**  Above step will be applicable in the constructor overloading based on type of argument with more than one argument.

**Change the above program:**

class Test

{

Test (int x, float ft)

{

System.out.println(“First Constructor”);

}

Test (float ft , char x)

{

System.out.println(“Second Constructor”);

}

public static void main(String s[])

{

new Test( ‘A’, ‘A’ );  **// compilation error**

}

}

**Copy Constructor:** Copy constructor also known as parameterized constructor.

In the argument of the copy constructor the reference id of the same type of object will be received.

**Purpose of Copy Constructor:** Purpose of copy constructor is to initialization the properties of one object with the properties of another object.

**Example:**

class Test

{

int x, y;

Test ( int x, int y )

{

this.x= x;

this.y= y;

System.out.println(“Object created with first constructor”);

}

Test ( Test t )

{

this.x= t.x;

this.y= t.y;

System.out.println(“Object created with copy constructor”);

}

void show ( )

{

System.out.println( “\nx: ”+x );

System.out.println( “y: ”+y );

}

public static void main(String s[])

{

Test t1, t2;

t1= new Test(5, 6);

t2= new Test(t1);

t1.show( );

t2.show( );

}

}

**Understanding the flow of main method:**

Test t1= new Test(5, 6); **1.0**  110 Test(int x, int y)

**1.1** {

**1.3** this.x= x; this 110

**1.2** this.y= y;

}

Test t2= new Test(t1); **2.0** 220 Test(Test t )

**2.1** {

**2.3** this.x= t.x; this 220

**2.2** this.y= t.y; this 110

}

1. First object created.

**1.1** Constructor invoked.

**1.2** Values are copied in the properties of object.

**1.3** Reference id of object returned.

**2.0** Second object created and reference id of first object in passed the argument of constructor.

**2.1** Constructor invoked.

**2.2** values are copied in the properties of the object from the properties of first object.

**2.3** Reference id of second object returned.

**Q.1 What happen of we write the return type before the constructor?**

**Ans.** There will not be any error rather the constructor will be considered as a normal method.

class Test

{

int x, y;

void Test(int x, int y) //normal method

{

this.x= x;

this.y= y;

}

void show( )

{

System.out.println(“/n x: ”+x);

System.out.println(“y: ”+y);

}

public static void main( String s[] )

{

Test t1= new Test( 5,6 ); **//compilation error**

Test t2= new Test( );

t1.Test( 5,6 );

}

}

**Q.2 Why we can not specify the return type of the constructor?**

**Ans.** Because all the constructor always have here implicit return type through which JVM returns the reference id of the object that’s why they cannot be used define return type.

**Q.3 Why compiler provide the default constructor in our class?**

**Ans.** To return the reference id of the object

Test Test(int x, int y )

{

Compiler this.x= x;

generate this.y= y;

code in the

.class file return (this);

}

**Q.4 Does the constructor provide the default values in the properties of the object?**

**Ans.** This is not the responsibility of the constructor rather just after the object creations JVM assign the default value in the properties of object.

class Test

{

int x= getx();

Test( )

{

System.out.println( “\n Constructor invoked ” );

}

int getx()

{

System.out.println( “\n getx( ) invoked” );

System.out.println( “Value of x : ” +x );

return( 5 );

}

public static void main ( String s[] )

{

Test t1= new Test( );

}

}

**Access Specifiers:** Access specifiers are also known as the access modifier.

The purpose of access specifier is to set the accessibility class and its members within the program.

**There are four types of accessibility of member of the class:**

1. Private
2. Default
3. Protective
4. Public
5. **Private:** If any member of any class has the private access specifiers then that member can only the accessed within the same class.
6. **Default:** If any member of any class has the default access specifier can be accessed within the same class or in any other class of the same package.

* Package is the collection of the classes.
* Multiple packages can be used in any program.
* If we do not specifier any access specifier then the default specifies with work.
* Till now all the classes that we have created were in the same package.

1. **Protected:** If any member of any class has the protected access specifier then it can be used in any class of the same package and in the child class in any other package.
2. **Public:** It provides the universal accessibility that means public member can be accessed in any class in any package.

**Note:**

* Any class can only be the public or default but the nested class can be having any access specifier.
* All the members of the class and the class itself always have the **“default”** access specifier by default.
* Without knowing package creation we can only test the private and default access specifier.
* private, protected and public are the keywords and where none of these are present then default access specifier will work.
* Access specifier can always be used member of the class either static member or non-static member.
* Within the method there is no concept off access specifier that means with the local variables we can never use any access specifier.

class Test

{

private static int x;

static int y;

private int a;

int b;

static void m1()

{

System.out.println(“In the m1”);

x=10;

y=20;

}

Static private void m2()

{

System.out.println(“In the m2”);

}

void m3()

{

System.out.println(“In the m3”);

a=30;

b=40;

}

private void m4()

{

System.out.println(“In the m4”);

}

public static void main(String s[])

{

m1();

m2();

Test t= new Test();

t.m3();

t.m4();

}

}

**Static:** Access Modifier.

**private, public, protected:** Access modifier/ Access Specifier.

**OR**

class Demo

{

public static void main( String s[] )

{

Test.m1();

Test t= new Test();

t.m3();

t.m4(); //error

Test.x= 10; //error

}

}

**Q.1 can be make constructor as private in any class?**

**Ans.** Yes, we can make the private constructor in any class but object of that class will only be created from the same class.

That means we can never create object of the class from the other class if the class has the private constructor.

class Test

{

int x;

private Test( int val )

{

x=val;

System.out.println(“Object created”);

}

void show( )

{

System.out.println(“Value of x: ” +x);

}

static Test m1( )

{

Test t= new Test( 5 );

return( t );

}

}

class Demo

{

public static void main(String s[] )

{

Test t1= Test.m1();

t1.show();

}

}

**Factory Pattern:**

This is the pattern do design the class. According to this pattern a class does not allow its instantiation (Object creation) by the new keyword in any other class, rather class itself provides a method to instantiate itself and returned the reference id.

**Singleton Pattern:**

This is the extension of factory pattern and if any class is designed through this pattern when that class can be instantiated only once that means only one object of that class can be created.

**Creating a class based on singleton pattern:**

class Test

{

static Test t;

int x;

private Test(int val )

{

x= val;

System.out.println(“Object created”);

}

void show( )

{

System.out.println(“Value of x: ”+x);

}

static Test m1( )

{

if( t== null)

{

t= new Test(5);

}

return(t);

}

}

class Demo

{

public static void main( string s[] )

{

Test t1= Test.m1();

Test t2= Test.m1();

System.out.println(t1==t2);

t1.show( );

t2.show( );

}

}

If you want second referenced id not create so

Test t1= Test.m1();

Test .t= null;

Test t2= Test.m1();

**OR**

private static Test t;

**Chain of the method calling:**

If any method returns the reference id of any object then with the calling of that method further the other method can be invoked.

class Test

{

Student m1()

{

Student st= new Student();

return(st);

}

public static void main(String s[])

{

Test t= new Test();

Student st1=t.m1();

st1.addmission(101); normal process of method calling

st1.exam(85);

st1.report( );

}

}

**Changes in main method:**

public static void main(String s[])

{

Test t=new Test();

t.m1().addmission(101);

}

In case of the chain of method calling always the returned value of right most method will be assigned to the left of assignment operator (=).

Student st= t.m1().addmission(101);  **//compilation error**

Because admission has return type void

**Changes in student class:**

class Student

{

int rollno, marks;

char grade;

Student addmission(int r)

{

rollno= r;

System.out.println ("Admission invoked”);

return(this);

}

Student add(int m)

{

marks= m;

System.out.println ( rollno+ " of " +name);

}

void exam( int m)

{

marks= m;

System.out.println(“Exam invoked");

if(marks>80)

grade= 'A';

else

if(marks>=60)

grade= 'B';

else

if(marks>=40)

grade= 'C';

}

void report()

{

System.out.println("Roll no: " +rollno);

System.out.println("Marks: " +marks);

System.out.println("Grade: " +grade );

}

}

Class Demo

{

public static void main(String s[])

{

Test t=new Test();

t.m1().addmission(101).exam(85).report();

}

}

**OR**

class Demo

{

public static void main(String s[])

{

new Test().m1().addmission(101).exam(85).report();

}

}

**Nested Method Calling:**

At the time of invocation of any method within the argument we can also invoked the others method but the return type of augmented method must be same with the argument of outer method.

class Test

{

void m1(Student st)

{

System.out.prinltn(“m1 invoked”);

}

}

class Demo

{

Student m2()

{

Student st= new Student();

return( st );

}

public static void main( String s[] )

{

Test t = new Test();

Demo d= new Demo( );

t.m1( d.m2() );

**OR**

t.m1( new Student() );

**OR**

Student st= new Student();

t.m1( st );

}

}

**t.m1( new Student() );**

3. 2. 1.

110

1. new object created.
2. Constructor invoked.
3. Reference id returned to be pass in the argument of m1 method.

**\*Anonymous Object:**

class Sample

{

static Student st;

static void m1( Student st )

{

Sample.st= st;

}

}

class Test

{

public static void main( String s[] )

{

**Anonymous use** new Student().addmission(101);

**Of object**  Sample.m1(new Student() );

}

}

If we use an object in any method without the reference variable then it is known as the anonymous use of that object. It doesn’t mean that any anonymous object becomes the unreferenced after its use.

**\*Association:**

This is the feature of OOP’s that is the implementation of the containership / Has A relationship.

**Example:**

* 1. Car has an engine.
  2. Mobile has a simcard.

There are two types of association:

1. **Aggregation:** Optional association.

Example:

Human body and eyes.

1. **Compassion:** Mandatory association.

Example:

Human body and heart.

In the OOP’s containerships says to store the object in the program.

In java storing the object in the reference variable is the implementation of association.

**According to the scope:**

There can be three types of association-

**1. Method level association:** When the reference variable is the local variable.

**Example:**

class Test

{

public static void main( String s[] )

{

System.out.println( “Welcome” );

m1( );

System.out.println(“Thank You”);

}

static void m1( )

{

Student st

st= new Student( );

}

}

110

st 110

Method level association

m1

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

main( )

stack heap

**2. Object level association:** when the reference variable is the instance variable.

class Test

{

Student st;

void m1( )

{

st= new Student( );

st.addmission(101 );

st.exam(85 );

st.report( );

}

public statc void main( String s[] )

{

Test t= new Test( );

t.m1( );

}

this 110

220

3.3

110

0

3.1 m1( ) 3.2 OLA

st

0

110

main() 1 2 Object of Test

220

stack JVM

In the object level association most common mistake performs by the program as follows, that results the **NullPointerException** at the runtime in the program.

class Test

{

Student st;

void m1( )

{

st=new Student( );

}

void m2( )

{

st.addmission(101 );

st.exam( 85 );

st.report( );

}

public static void main( String s[] )

{

Test t= new Test( );

t.m2( );

t.m1( );

}

}

Local variable never have default variable rather they must be initialize before this use.

public static void main( String s[] )

{

Test t1;

t1.m1( ); **//compilation error**

}

**3. Class level association:** when the reference variable is the static variable.

class Test

{

static Student st;

static void m1( )

{

st= new Student( );

}

public static void main(String s[] )

{

m1( );

st.addmission(101 );

}

}

class Demo

{

public static void main( String s[] )

{

Test.st.addmission(101);

}

}

class Area Heap

110

class Test

0

Byte code-----------------------------------

st

110

0

**1.2**

null

class level association

m1()

main()

**1.1**

JVM

**1.0**

stack

“ System.out.println( ); ” statement depend on class level association.

**Practice code:**

class Product

{

int code, price;

int qty;

void set(int c, int p, int q)

{

code= c;

price= p;

qty= q;

}

void change(int newPrice)

{

price= newPrice;

}

void report()

{

System.out.println("\nCode: "+code);

System.out.println("\nPrice: "+price);

System.out.println("\nQty: "+qty);

}

void sales(int unit, Customer c)

{

if(qty>=unit)

{

qty-=unit;

System.out.println(code+" product sold to "+c.cid+" customer");

}

else

{

System.out.println("Insufficient quantity");

}

}

}

class Customer

{

int cid;

Product p;

void setDetails(int cid, Product p)

{

this.cid=cid;

this.p=p;

}

void doSales(int unit)

{

p.sales(unit, this);

}

void show()

{

System.out.println("\n Customer id: "+cid);

p.report();

}

}

class Market

{

static Product p1;

static Product p2;

static void set(int pNumber, int code, int price, int qty)

{

if(pNumber==1)

{

p1=new Product();

p1.set(code, price, qty);

}

if(pNumber==2)

{

p2=new Product();

p2.set(code, price, qty);

}

}

}

class Test

{

public static void main(String s[])

{

Customer c1, c2;

c1=new Customer();

c2=new Customer();

Market.set(1, 111, 1500, 100);

Market.set(2, 222, 3000, 200);

c1.setDetails(101, Market.p1);

c2.setDetails(102, Market.p2);

c1.doSales(60);

c2.doSales(100);

c1.show();

c2.show();

}

}

**class Area**   **Heap**

110

Object of product

220

Object of product

330

Object of customer

440

Object of customer

class Market

Byte code--------------------------------------------------------------------------------------------

p1

P2

111

100

1500

110

220

222

3000

200

main( )

c1

c2

330

101

440

110

Stack

102

210

**Init block (Initialization Block):**

init block in top is the block that is used to initialize the object.

At the time of object creation init block always be executed before the constructor calling.

**Example:**

class Test

{

init x;

{

init System.out.println(This is the init block);

block x=10;

}

Test ()

{

System.out.println(“Contructor in test”);

x= 20;

}

void show()

{

System.out.println(“Value of x” +x);

}

public static void main(String s[])

{

Test t;

t= new Test( );

0

t.show( );

}

}

After this init block will run, then constructor. Constructor will return id which will be stored in t variable.

t.show will print (value of x)

**NOTE:** We can create more than one unit block in any class and all are executed at the time of object creation in the same sequence in which they are declared in the class (no matter whether created before constructor or after).

init block are non selective = at the time of object of creation all the init blocks are formed.

**Why init block?**

Constructor are selective by nature that means at the time of object creation only one constructor will be invoked depending upon the argument. If we have the common code in our application this we have write it in all the constructors.

init block are non selective that means all the init blocks always be invoked at the time of each object.

**Why multiple init blocks?**

During the compilation the code of all init blocks automatically invoked in the starting of all the constructors.

That means there are no init blocks in the dot class file.

**static block**

static block are the blocks that always be executed at time of loading of the class.

static block always be created with the static keyword.

**Example:**

class Test

{

static

{

System.out.println(“This is static block”);

}

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

Other method

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

}

**Class loading**

Loading of the dot class file from the second storage to the primary memory (RAM) within the JRE in the class area is known as the class loading.

First class is stored then, object is created.

static variable will get memory then, static block will run.

**NOTE:**

* When 1st time the need of any class is resided in our program, at that time class will be loaded.
* When the 1st object of any class is about to the created or 1st time any static number is about to be invoked (which end is earlier) at that time the class will be loaded.
* At the time of class loading following 2 talks will be performed.

1. Memory allocation to the static variable.
2. Invoked of the static blocks.

**Example:**

class Test

{

static int x;

int y;

static

{

System.out.println(“This is static block”);

x=10;

y= 20; **//error**

}

Test()

{

System.out.println(“object created”);

y=30;

}

void show()

{

System.out.println(“x: ” +x);

System.out.println(“y: ” +y);

}

class Demo

{

public static void main(String s[])

{

Test t1;

System.out.println(“Welcome in main”);

t1= new Test( );

Test t2= new Test( );

t1.show( );

t2.show( );

}

}

}

* non-static member cannot be used directly in static blocks.
* static blocks are also the static members of the class. So we can’t access the non-static member in the static method directly.
* Test class is first need object create

**OUTPUT:**

Welcome in main

This is static block

Object created

x: 10

y: 30

x: 10

y: 30

**static block in the main class**

static block runs before the main method.

class Demo

{

static

{

System.out.println(“Demo class loaded”);

}

public static void main(String s[])

{

Test t1;

System.out.println(“Welcome in main”);

t1= new Test();

Test t2= new Test();

t1.show();

t2.show();

}

}

Upto JDK16, JVM first of all load the class and after that stack the existence of main method.

JDK 7 on words, before loading the class JVM checks existence of the main method.

**\*Inheritance\***

Inheritance is one of most important concept of OOP’s.

Inheritance is the concept through which we usability of any class can also be possible.

Another way achieves the usability association through which we can use any class in any other class.

**Example:**

* Child and parents relationship in which the child inherit the number of properties from the parent, such as: money, house, jeans etc.
* Java language is also inherited from the C & C++ in order to get the syntax and basic concept of the programming.

In the object oriented programming any one class can be child of any other class so that members of one class can be use in the other class.

**extends keyword:**

extends keyword is use to make one class child of another class that means extends keyword is used to implement the inheritance in the java programming.

class A

{

int x, y;

void setA( int val )

{

x= val;

y= val+10;

}

}

class B extends A

{

int i, j;

void setB( int val )

{

i= val;

j= val+10 ;

}

void show( )

{

System.out.println(“x: ”+x);

System.out.println(“y: ”+y);

System.out.println(“i: ”+i);

System.out.println(“j: ”+j);

}

}

class Test

{

public static void main( String s[] )

{

B ref= new B();

ref.setA( 50 );

ref.setB( 60 );

ref.show( );

}

}

**Changes in program**

class A

{

private int x, y;

void setA( int val )

{

x= val;

y= val+10;

}

void showA( )

{

System.out.println(“x: ”+this.x);

System.out.println(“y: ”+this.y);

}

}

class B extends A

{

int i, j;

void setB( int val )

{

i= val;

j= val+10 ;

}

void show( )

{

System.out.println(“i: ”+this.i);

System.out.println(“j: ”+this.j);

}

}

class Test

{

public static void main( String s[] )

{

B ref= new B();

ref.setA( 50 );

ref.setB( 60 );

ref.showA( );

ref.show( );

}

}

**Types of Inheritance:**

1. **Single Inheritance:** In this type there is only one parent and its one child class.

A(Parent/ Super)

B(Child/Sub)

1. **Multilevel Inheritance:** In this type one child class can be the parent of other child class.

A

B

C

1. **Multiple Inheritance:** In this type of inheritance only one child can have the multiple parent classes.

A B

C

Java does not support the multiple inheritance.

1. **Hierarchical Inheritance:** In this type of inheritance the one parent class can have the multiple child classes.

A

B C

1. **Hybrid Inheritance:** This combination of all other types of inheritance so it also not be possible in java.

A

B C

D

**Example of Hierarchical Inheritance:**

Employee

Part Time Employee Full Time Employee

**Program**

class Employee

{

int code, tax, netSalary;

void set( int c )

{

code= c;

}

void show( )

{

System.out.println(“Code= ” +code);

System.out.println(“Tax= ” +tax);

System.out.println(“Net salary= ” +netsalary);

}

}

class FullTimeEmp extends Employee

{

int basic, HRA, TA;

FullTimeEmp( int c, int basic )

{

this.set( c );

this.basic= basic;

}

void calc( )

{

HRA= basic \* 12/100;

TA= (basic + HRA + TA)\* 12/100;

netSalary= basic + TA + HRA – tax;

}

void show( )

{

super.show( );

System.out.println(“Basic: ” +basic );

System.out.println(“HRA ” +HRA );

System.out.println(“TA: ” +TA );

}

}

class PartTimeEmp extends Employee

{

int rate, noh;

void set(int c, int r, int noh )

{

set( c );

this.rate= r;

this.noh= noh;

}

void calc( )

{

tax= ( noh\*rate )\*10/100;

netSalary= noh\*rate-tax;

}

void show( )

{

super.show();

System.out.println(“Number of horse: ” +noh);

System.out.println(“Rate per horse: ” +rate);

}

}

class Company

{

public static void main( String s[] )

{

PartTimeEmp pe= new PartTimeEmp( );

FullTimeEmp fe= new FullTimeEmp( );

pe.set (101, 100, 150);

pe.calc( );

fe.calc( );

pe.show( );

fe.show( );

}

}

**“super”** is the wrapper of **“this”** keyword.

**Data Hiding:**

Data hiding is the cases that can only be arise in case of inheritance. According to data hiding the child class can have the variable with the same name as the variables in the parent class.

The priority always the given to the child class variables in the child class.

class A

{

int x, y;

void m1( int val )

{

x= val;

y= val +10;

}

}

Class B extends A

{

int x, z;

void setB( int val )

{

x=val;

z= val +10;

super.x= val + 20;

y= val +30;

}

void show( )

{

System.out.println(“x of A: ”+super.x);

System.out.println(“y of A: ”+x);

System.out.println(“x of B: ”+ x);

System.out.println(“z of B: ”+z);

}

}

class Demo

{

public static oid main(String s[])

{

B ref= new B( );

ref.m1(100);

ref.setB(50);

ref.show( );

}

}

Object of B

x:

y:

**super:**

* **super** automatically convert into **this .**
* super is a part of this.
* We can never print the super.
* super is the casted form of this.
* Whenever we invoked the method of parent class with the child class object then only the super area of child class object will be used.

super = ((name of parent class)this)

**example:** super= ((A)this)

data hiding in case of multilevel inheritance

class A

{

int x;

}

class B extends A

{

int x;

}

class C extends B

{

int x;

void set(int val)

{

((A)this).x= val + 1;

super.super.x= val+1; **// compilation error**

**//no syntax like super.super**

super.x= val+2;

x= val+3;

}

void show()

{

System.out.println(“x of A: ”+((A)this).x);

System.out.println(“x of B: ”+super.x); // **or** +((A)this).x

System.out.println(“x of C: ”+ x);

}

public static void main(String s[])

{

C ref;

ref= new C();

ref.setC(300);

ref.show( );

}

}

**Object of C**

super(A)

x:

0

x:

0

0

super represent intimidate parent

**class loading incase of Inheritance:**

In case of inheritance the parent class always be loaded before the child class.

When we create the first object of the child class then there can be two situations:

**1.** Parent class already has been loaded before in that case only the child class will be load alone.

**2.** The parent class not loaded before so alone with the child class the parent class also will be loaded at the same time (first the parent class and then child class).

class A

{

{

System.out.println(“A class loaded”);

}

static void m1()

{

System.out.println(“m1 of A invoked”);

}

}

class B extends A

{

int x, y;

static

{

System.out.println(“B class loaded”);

}

void show()

{

System.out.println(“This is the show in class B”);

}

}

class Demo

{

public static void main(String s[])

{

System.out.println(“Welcome”);

B ref= ref B();

ref.show( );

}

}

**OUTPUT:**

Welcome

A class loaded

B class loaded

This is the show in class B

**Changes in the main method:**

public static void main(String s[])

{

A.m1();

System.out.println(“Welcome”);

B ref= new B();

ref.show();

}

**OUTPUT:**

A class loaded

m1 of A invoked

Welcome

B class loaded

This is the show in class B

**NOTE:**

* In case of inheritance all the parent classes will be loaded at time of loading of the child class.
* In case association at the time of loading of composite class number need the load all the member class.
* **Association:** Has-A-relationship
* **Inheritance:** Is-A-relationship / Type-of-relationship

**Constructor calling in case Inheritance:**

At the time of creation of the child class object the change with the constructor of child class the constructor of the parent class will also the invoked.

The child class constructor will be invoked after the parent class constructor.

**Example:**

class A

{

A()

{

System.out.println(“Const of A”);

}

class B extends A

{

B()

{

super(); **//hidden**

System.out.println(Const of B);

}

public static void main(String s[])

{

System.out.println(“Welcome”);

new B();

}

}

**OUTPUT:**

Welcome

Const of A

Const of B

**NOTE:**

* In each and every child class constructor the super() statement exist as the first statement that denotes the calling of default constructor of the parent class.
* If in the parent class there is no default constructor then we have to write super() statement explicitly in the child class constructor in the first line.
* super() statement must be the first statement in any child class constructor.

**\*Parameterize Constructor:**

class A

{

A( int val )

{

System.out.println(“Const of A”);

}

class B extends A

{

B()

{

super( 5 ); **//explicit**

System.out.println(“Const of B”);

}

public static void main(String s[])

{

System.out.println(“Welcome”);

new B();

}

}

If any class there is only the private constructor then that class can never be inheritance.

private A()

{

System.out.println(“Const of A”);

}

B()

{

super( 5 ); **// compilation error**

System.out.println(“Const of B”);

}

**\*Object Class:**

* this is the most important class in the java library.
* It can also be considered as a God of all the other java classes.
* All the classes in java are the child of the object class directly or indirectly.
* That means if any class does not have the parent class then it will be direct child of object class.

Object

All classes

There is only one public default constructor in the object class.

**\*Constructor Chaining:**

Within the constructor of any class we can invoke the other constructor of the same class.

class Test

{

Test( )

{

System.out.println(“Default const”);

}

Test( int x )

{

this();

System.out.println(“Param const”);

}

public static void main(String s[])

{

Test t= new Test( 5);

}

}

**NOTE:**

* this() / this(any parameter) statement is used to invoked the other constructor of the same class from the constructor of the same class.
* this() statement also will be the first statement in any constructor.
* this() and super() can never be used together.

Constructor chaining can never be cycling that mean in any constructor we have skip this() statement.

**//compilation error in the program//**

class Test

{

Test( )

{

this( 5);

System.out.println(“Default const”);

}

Test( int x )

{

this();

System.out.println(“Param const”);

}

public static void main(String s[])

{

Test t= new Test( 5);

}

}

**Practice Code**

class A

{

A() {

System.out.println(“Default const of A”);

}

A(int x)

{

this();

System.out.println(“Param const of A”);

}

}

class B extends A

{

B()

{

super( 5);

System.out.println(“Default const of B”);

}

B(int x)

{

this();

System.out.println(“Param const of B”);

}

}

class C extends B

{

C ()

{

super( 5);

System.out.println(“Default const of C”);

}

C (int x)

{

System.out.println(“Param const of C”);

}

public static void main(String s[])

{

new C ();

new C ( 5);

}

}

**OUTPUT:**

Default A class

Param A class

Default B class

Param B class

Default C class

Default A class

Param A class

Default B class

Param C class

**\*Polymorphism\***

Polymorphism term is arrived with two terms poly + morphism.

Poly means many.

Morphism means form.

Complete meaning of the polymorphism is the one name many form.

**Example:**

1. They are the many form of the fruits such as: mango, apple, etc.
2. Many form is the employee: clerk, manager,etc.

**The polymorphism is further categories in the two forms:**

1. 1st is compile time polymorphism.
2. 2nd runtime polymorphism.
3. **Compile time polymorphism:** When the one form is selected at the compile time among the multiple forms perform it will the compile time polymorphism.

**Example:** Railway reservation

1. **Runtime polymorphism:** When the selection of one form among the multiple forms performed at the runtime it will we known as runtime.

**Example:** Selection of any specific shake on the juice corner.

**\*Implementation in java:**

**Polymorphism**

Runtime Polymorphism

Compile Time Polymorphism

method overriding

method overloading

operator overloading

**\*operator overloading:**

Operator overloading is the concept in which one operator will have the multiple forms and the selection of one form always be perform at the compiler by seeing type of operands.

**\*method overloading:**

Method overloading more than a methods can have the same name but with the different argument list.

These methods will be known as the overloading method and can be exist either in the same class or in the child and parent class.

**\*Rules to overload the methods:**

Return type name of method (Argument list)

Must be changed

Must be same

May or may not be changed

**\*Argument list can be change in two ways:**

**1.** By changing the number of argument.

**2.** By changing the type of argument.

**Same as the constructor overloading**

**Q. Why the return type does not affect the method overloading?**

**Ans.** In the programming it is not mandatory to catch the returned value of any method that means the method that have return type and the method without return type can be invoked in the same method.

class Test **// compilation error**

{

int m1(int x)

{

System.out.println(“m1 in Test”);

return( 5);

}

public static void main( String s[] )

{

Test t= new Test( );

t.m1();

int res= t.m1( );

//both above are the correct ways//

}

}

void m1( int x)

{

System.out.printl(“Second m1 in Test”);

}

**Method overloading in the inheritance:**

class A

{

void m1( int x)

{

System.out.println(“First m1 in Test”);

}

}

class B extends A

{

void m1( float x)

{

System.out.println(“Seccond m1 in Test”);

}

public static void main( String s[])

{

B ref= new B();

ref.m1(‘A’);

}

}

**\*Method overriding:**

Method overriding can only be possible in case of inheritance.

The basic requirement of method overriding is to make the signature of the parent class method in the child class.

Through the object t of child class method of child class it will be visible that means we can never invoked the parent class method from the object of child class in case of method overriding.

class A

{

void m1( )  **//overridden method**

{

System.out.println(“m1 in class A”);

}

}

class B extends A

{

void m2( )

{

m1( ); **//m1 of will be invoked**

}

}

class Test

{

public static void main( String s[] )

{

B ref1= new B( );

ref1.m1( );

B ref2= new B( );

ref2.m2( );

}

}

**NOTE:**

By using the super keyword from the non-static method of child class, we can invoke the overridden method of the parent class.

**Changes in the m2 method**

void m2()

{

super.m1();  **//m1 of A will be invoked**

}

**NOTE:** In the multilevel inheritance we can never invoked the overridden method of the parent of parent class that means we can only invoked the overridden method of the immediate parent class by using super keyword.

**\*Virtualization of method calling binding:**

In java at the compile time there can be two types of binding.

1. **Virtual Binding.**
2. **Actual Binding.**
3. **Virtual Binding:** In case of virtual binding compiler just perform the temporary binding that can be change be the JVM at the runtime.
4. **Actual Binding:** In this case compile time binding never be change by the JVM at the runtime.

**NOTE:** In the dot( . ) class file compiler just place the assembly instruction to intimate the JVM about the type of binding.

**There are in total four types of assembly instruction, to will be use in the actual binding and rest to virtual binding.**

|  |  |  |
| --- | --- | --- |
| **Assembly Instruction** | **Type of compilation binding** | **Type of method calling** |
| invokestatic | Actual | calling of all the static method |
| Invokespecial | Actual | non-static private methods, constructors method call with super keyword |
| Invokevirtual | Virtual | non-static non private method |
| Invokeinterface | Virtual | Method of interface |

**\*upcasting:** In the reference variable of parent class the reference id of the object of child class can be stored. This is known as the upcasting.

A ref= new B( ); **//upcasting**

ref.m1( );

**NOTE:** compiler will perform the binding be seeing the reference variable.

If the compile time binding is virtual then JVM will perform the actual binding by seeing the type of object contain by reference variable.

**\*Creating one more child of A:**

class A

{

void m1( )  **//overridden method**

{

System.out.println(“m1 in class A”);

}

}

class B extends A

{

void m2( )

{

m1( );

**//m1 of will be invoked**

}

}

class C extends A

{

void m1( ) **//overriding method**

{

System.out.println(“m1 in C”);

}

}

class Demo

{

static void check(A ref)

{

ref.m1(); **//runtime polymorphism by the method overriding**

}

}

class Test

{

public static void main( String s[] )

{

Demo.check( new A() );

Demo.check( new B() );

Demo.check( new C() );

}

}

**Binding in case of overloading methods:**

If there are the overloading methods in the same class or in the inheritance then at the compile time the selection of one method will be perform by seeing the argument list and appropriate assembly instruction will be placed.

At the runtime JVM can never change the signature of the method because selection of the argument can only be performed by the compiler.

class A

{

void m1( float ft )

{

System.out.println(“m1 in class A”);

}

}

class B extends A

{

void m1( int x )

{

System.out.println(“m1 in class B”);

}

}

class Demo

{

static void check( A ref )

{

ref.m1(‘A’); **//invokevirtual**

}

}

class Mymain

{

public static void main( String s[] )

{

Demo.check( new B() ); **//invokespacial**

}

}

**\*Rule for the method overriding:**

Unique the method overriding we can never use the weaker access privileged with the overriding method in the child class.

private Weakest Accessibility

default

protected

public Strongest Accessibility

**\*Difference between method overloading & method overloading:**

|  |  |  |
| --- | --- | --- |
|  | **Method Overloading** | **Method Overriding** |
| 1. | Name of the method must be same and argument list must be changed. | The complete signature of the method must be same. |
| 2. | Can be possible in same class or the inheritance. | Only be possible in case of inheritance. |
| 3. | Used to achieve the compile time polymorphism. | Used to achieve the runtime polymorphism. |
| 4. | Does not affect by the access specifiers. | Affect by the access specifier. |
| 5. | Dynamic dispatching never be perform that means always be executed the overloaded method selected by the compiler. | Dynamic dispatching will be possible if upcasting is exist. |

**\*Downcasting:**

Downcasting is not the opposite of upcasting rather it always follow to upcasting that means downcasting always be preceded upcasting.

**B ref1= (B)ref;**

**Reference id of parent class can’t come in child class.**

Downcasting is necessary to invoked the members of the child class whose object is already upcasted in the reference variable of parent.

class Mymain

{

public static void main( String s[] )

{

A ref= new B( ); **//upcasting**

B ref1= ( B )ref; **//downcasting**

}

}

**\*Rules for the downcasting:**

1. Downcasting always be possible in the same hierarchy in which the upcasting already being perform.

If we perform downcasting in different hierarchy then **ClasscastingException** will be generated at the runtime.

1. Downcasting always be performed explicitly otherwise compilation error will be generated.

class Mymain

{

public static void main( String s[] )

{

A ref= new B(); **//upcasting**

B ref1= ref; **// compilation error**

}

}

**\*Need of Downcasting:**

Need of downcasting is to invoke the members of the child class with the same object that is already upcasting.

class A

{

void m1() **//overridden**

{

System.out.println(“m1 in class A”);

}

void m2() **//unique method of class A**

{

System.out.println(“m2 in class A”);

}

}

class B extends A

{

void m1() **//overriding**

{

System.out.println(“m1 in class B”);

}

void m3() **//unique method of class B**

{

System.out.println(“m3 in class B”);

}

}

class Demo

{

static void check( A ref)

{

ref.m1();

ref.m2();

B ref1= ( B)ref;

ref1.m3();

}

}

**\*Block Diagram:**

Member of the class B accessible from the reference variable of A

m2()

m1()

m3()

Inherited ✓ compile time + runtime

Overriding method B ✓ runtime

Unique method of B X

class C extends A

{

void m1() **//overriding**

{

System.out.println(“m1 in class C”);

}

void m4() **//unique method of class C**

{

System.out.println(“m4 in class C”);

}

}

class Mymain

{

public static void main(String s[] )

{

A ref= new B();

B ref1= (B) ref;

C ref2= (C)ref; **// ClasscastingException**

}

}

**Changes in the main method:**

class Mymain

{

public static void main(String s[] )

{

B ref= new B();

C ref2= (C)ref; **// Compilation error, B cannot be converted into C**

}

}

**Changes in the main method:**

class Mymain

{

public static void main(String s[] )

{

A ref= new B();

ref.m1();

ref.m2();

ref.m3(); **// Compilation error**

B ref2= (B)ref;

ref2.m3();

}

}

**Changes in the main method:**

class Mymain

{

public static void main(String s[] )

{

A ref= new B();

B ref1= ref; **// Compilation error**

}

}

should be the explicitly conversion

**NOTE:** parent class reference id never stored in child class reference variable

B ref= new A(); **//compilation error**

A ref= new A();

B ref2= ref; **//compilation error**

A ref= new B();

B ref2= (B)ref;

**B ref2= ref;** **//compilation error**

Compile never allows to copy the reference variable of parent into the reference variable of child because at the runtime in the parent class reference variable where can be the object of parent itself or any child.

**B ref= new A();** **//compilation error**

In the child class reference variable we can never store the reference id of any parent object.

**B ref2= (B)ref; // no error**

No compilation error because it instructed to the compiler to copy the reference id from parent to child reference variable. In this case the total responsibility is of the developer that means at runtime this the object of any other child or itself parent then **ClasscastException** will be generate.

**‘instanceof’ operator:**

This also the keyword in java, works as an operator. The purpose of instanceof operator is to identify the type of object contained by the reference variable.

**Syntax:**

refVar instanceof Classname

The result of instance operator always be Boolean type.

A ref= new B();

if(ref instanceof B) **//true**

{

System.out.println(“object of B”);

}

if(ref instanceof C) **//false**

{

System.out.println(“object of C”);

}

class Demo

{

static void check(A ref)

{

ref.m1();

ref.m2();

if(ref indtanceof B)

{

B ref1= (B)ref;

ref1.m3();

}

if(ref indtanceof C)

{

C ref1= (C)ref;

ref1.m4();

}

}

}

class Mymain

{

public static void main(String s[])

{

Demo.check(new A());

Demo.check(new B());

Demo.check(new C());

}

}

**instanceof operator always perform the checking on the basic of type :**

**A**

**B C**

**D**

A a= new D();

System.out.println(a instanceof D); **//true**

System.out.println(a instanceof B); **//true**

System.out.println(a instanceof A); **//true**

System.out.println(a instanceof C); **//false**

**\*method of object class:**

There are number of methods of object class but some common methods as follows –

* hashCode()
* toString()
* equals()
* **hashCode()** method of object class always returns an integer value that is generated by the actual address of the object that means the hashCode() value always be unique for the different object.

**Signature:**

public int hashCode()

**example:**

class Mymain

{

public static void main(String s[])

{

Student st1= new Student();

Student st2= new Student();

System.out.println(st1.hashCode());

System.out.println(st2.hashCode());

}

}

**OUTPUT:**

2018699554

1311053135

* **toString()** method of the object class always returns the reference id of the object that contains.

**Classname@haxadecimalvalue**

**Signature:** public String tostring()

**example:**

class Mymain

{

public static void main(String s[])

{

Student st1= new Student();

Student st2= new Student();

System.out.println(st1.toString());

System.out.println(st2.toString());

}

}

**OUTPUT:**

Student@7852e922

Student@4e25154f

**Hexadecimal:** generated by the integer value returned by the hashCode() method.

**Example:**

class A

{

void m1() //overridden

{ compile time binding(invokevirtual)

System.out.println(“m1 of A”);

}

void m2()

{

System.out.println(“m2 of A”);

m1();

}

}

class B extends A

{

void m1() **//overriding method**

{

System.out.println(“m1 in B”);

}

}

class Mymain

{

public static void main(String s[])

{

B ref= new B();

ref.m2();

}

}

**OUTPUT:**

m2 of A

m1 in B

**\*Overriding the hashCode() method in the student class:**

class Student

{

int rollno, marks;

Student (int rollno)

{

this.rollno= rollno;

}

void exam(int marks)

{

this.marks=marks;

}

void report( )

{

System.out.println(“Roll no:” +rollno);

System.out.println(“Marks:” +marks);

}

public int hashCode()

{

return(roll no);

}

}

class Mymain

{

public static void main(String s[])

{

Student st1= new Student (101);

Student st2= new Student (102);

System.out.println(st1.hashCode());

System.out.println(st2.hashCode());

System.out.println(st1.toString());

System.out.println(st2.toString());

}

}

**OUTPUT:**

101

102

Student@65

Student@75

**Overriding the toString() method in the student class:**

class Student

{

{

int roll no, marks;

Student (int rollno)

{

this.rollno= roll no;

}

void exam(int marks)

{

this.marks=marks;

}

void report( )

{

System.out.println(“Roll no:” +rollno);

System.out.println(“Marks:” +marks);

}

public in hashCode()

{

return(roll no);

}

public String toString()

{

return(roll no+ “:” +marks);

}

}

**OUTPUT:**

101 : 85

102 : 75

class Mymain

{

public static void main(String s[])

{

Student st1= new Student(101);

Student st2= new Student(102);

st1.exam(85);

st2.exam(75);

System.out.println(st1.toString());

System.out.println(st2.toString());

System.out.println(st1);

System.out.println(st2);

}

}

**OUTPUT:**

101 : 85

102 : 75

101 : 85

102 : 75

When we are print reference variable then also call toString().

When we print any reference variable in System.out.println internally toString() method always be invoked by the println() method.

**\*key point about the println() method:**

There are in total 10 println() method in the printscreen class that means all these method will be overloaded form.

These println() method in the object class as follows:

1. public void println( int x )
2. public void println( char ch )
3. public void println( float f )
4. public void println( double d )
5. public void println( long l )
6. public void println( boolean b )
7. public void println( char []ch )
8. public void println( string str )
9. public void println( Object ob )
10. public void println( )

At the time of concatenation reference variable also provide their values via toString() method.

class Mymain

{

public static void main(String s[])

{

Student st1= new Student(101);

Student st2= new Student(102);

st1.exam(85);

st2.exam(75);

String str= ”Hello” +st1; **//str1.toString() invoked internally**

System.out.println(str);

}

}

**OUTPUT:**

Hello 101 : 85

**equals():** equals() method of the object class is use to compare two object based on reference id of the object.

**Signature:**

public Boolean equals(Object ob)

**example:**

class Mymain

{

public staic void main(String s[])

{

//same all

System.out.println(st1==st2); **//false**

System.out.println(st1.equals(st2)); **//false**

}

}

Overriding the equals() method in the student class.

class Student

{

// same method

public Boolean equals(Object ob)

{

if(ob instanceof Student)

{

Student st1= (Student)ob;

return( rollno==st.rollno && marks==st.marks );

}

return(false);

}

}

**OUTPUT:**

False

True

**NOTE:** if we change the signature of the equals( ) methods in our class then it would be the overloaded method not the overriding method, so its binding never be performed at runtime.

class Demo

{

static boolean check( Object ob1, Object ob2)

{

return(ob1.equals(ob2));

}

}

**\*abstract keyword\***

abstract keyword also be known as the access modifier.

abstract keyword can be used with the class and non-static method.

abstract class and abstract method together are used to enforce the runtime binding (runtime polymorphism) of the methods.

**Abstract Class:** if any class is declared as abstract then it can never be instantiated that means that class can never be used in associate.

If any class is abstract when can only be inherited

We can name the reference variable of abstract class.

abstract class A

{

void m1()

{

System.out.println(“m1 in A class”);

}

}

class B extends A

{

}

class Demo

{

static void check(A ref)

{

ref.m1();

}

public static void main(String s[])

{

Demo.check(new A()); **//compilation error, A cannot instantiated**

Demo.check(new B());

}

}

**abstract method:** we can use the abstract keyword with the methods but those methods must be the non-static and non-private.

**NOTE:** If any has abstract method then that class must be declared as the abstract class not a vice versa that means if class is abstract then it is not complexly that it must have any abstract method.

“abstract class without abstract method”- Possible.

“abstract method without abstract class”- Not possible.

abstract method have they only never they have body.

abstract class A

{

abstract void m1();

}

**NOTE:** In the child class of the abstract class all the abstract methods must be overridden otherwise a child class also be declared as the abstract class.

abstract class A

{

abstract void m1();

}

class B extends A

{

void m1() **//overriding**

{

System.out.println(“m1 in B”);

}

}

**NOTE:**

* abstract keyword use with non-static and non-private method.
* abstract means not association.
* We can never create object of abstract class.
* We can create reference variable of abstract class.

**NOTE:**

In any abstract class abstract method as well as non-abstract method both types of method can be accessed.

**Purpose of abstract:**

The purpose of abstract method is just to provide the standard for the child class.

Method calling is abstraction like println ,println what work we know but how do work we don’t know.

**final keyword**

final keyword can be used with the class, methods and also with the variables.

**final class:**

If any class is declared as final then it can never be inherited that means we can use any final class with the association not with the inheritance.

System class and string class in the java library are the final classes.

**Why need of final class:** For stopping multilevel inheritance when we don't override all methods in subclass and don't want extended our class functionality then we use final class.

final class A

{

void m1()

{

}

}

class B extends A **//compilation error**

{

}

**final method:**

If any method is declared as final then that method can never be overridden by the child class.

class A

{

final void m1()

{

}

}

class B extends A

{

void m1() **//compilation error**

{

}

}

**Q.1 Can be make the private method final?**

**Ans.**  Yes, we can make but in the child class we can create the method with the same signature it will not be considered as the overriding method (because in the parent class method is private) .

class A

{

private final void m1()

{

}

}

class B extends A

{

void m1() **//self method ,no error**

{

}

}

**Q.2 Can be make the static method final?**

**Ans.** Yes, we can makethe static method final after that with the same signature we can never create same static method in the child class.

i.e. final static method in the child class can never be hidden.

* static method can never be overridden rather they just hide in the child class.
* It is not complexly to make the class final if already have the final method.

i.e. there is no signification to make the final method in the final class.

final class A

{

final void m1()

{

}

}

class B extends A

{

}

**Q.3 Can be make the final method in the abstract class?**

**Ans.** Yes, we can make.

abstract class A

{

final void m1()

{

System.out.println(“m1 in A, never be overridden”);

}

void m2()

{

System.out.println(“m2 in A, may or may not be overridden”);

}

abstract void m3() **//must be overridden**

}

abstract and final are the opposite in terms of their functionality i.e. they can never be used together.

* **abstract class:** no association only inheritance.
* **Final class:** no inheritance only association.
* **Abstract method:** complexly overriding.
* **Final method:** no overriding.

If a class final then it can be never be the abstract class vice versa.

**Q. Can be make the abstract method in the final class?**

**Ans.** No

**final variables:** In java are used to make the constants to i.e. after one time initialization we can never change the value of final variable.

Final variable never have their default value rather they have to be initialization before their first use.

**Final static variable:** It can be initializing either at the time of declaration or in the static block.

class Test

{

final static int x,y;

final static int z= 3;

static

{

System.out.println( x ); **//error, no default value**

x= 10;

z= 30;  **//error, no multiple time initialization allowed**

}

static void m1()

{

y=20;  **//error, can only initialization at declaration time or in static block**

}

}

**Non-static final variable:** It can be initialization either at the time of declaration in the init block or in the constructor.

**Local final variable:** These variables can be initialization anywhere in the method but before its first use.

class Test

{

void m1()

{

final int x;

System.out.println(x);

x= 10; **//error**

}

}

**OR**

void m1( final int x)

{

x= 10; **//error**

System.out.println( x );

}

**final reference variables:**

class Test

{

final static Student st= new Student();

static

{

st= new Student(); **//error**

}

}

class Demo

{

public static void main(String s[])

{

Test.st= new Student(); **//error**

}

}

out is a static reference variable

**“** out is the public final static reference variable in the system class.**”**

**Practice Code**

interface Account

{

void setAccount(int accno, int bal);

void intrestCalc( );

void report( );

}

class SavingAccount extends Operation

{

int accno;

int bal;

public void setAccount(int accno, int bal)

{

this.accno= accno;

this.bal= bal;

}

void setRate(int rate)

{

this.rate= rate;

}

public void intrestCalc( )

{

this.intrest= bal\*rate/100;

bal= bal + intrest;

}

public void report( )

{

System.out.println(“\nAccount no: ”+accno);

System.out.println(“Balance: ”+bal);

System.out.println(“Type: Saving”);

}

}

class CurrentAccount extends Operation

{

int accno;

int bal, intrest;

public void setAccount(int accno, int bal)

{

this.accno= accno;

this.bal= bal;

}

void setRate(int rate)

{

this.rate= rate;

}

public void intrestCalc( )

{

this.intrest= (bal+intrest)\*rate/100;

bal= bal + intrest;

}

public void report( )

{

System.out.println(“\nAccount no: ”+accno);

System.out.println(“Balance: ”+bal);

System.out.println(“Interest: ”+intrest);

System.out.println(“Type: Saving”);

}

}

abstract class Operation implements Account

{

int rate;

abstract void setRate(int r);

void calculate(int r)

{

this.setRate(r);

this.intrestCalc( );

}

}

class Bank

{

static void open(Account a)

{

int acc\_no= Management.getaccNo();

int bal=0;

if(a instanceof SavingAccount)

{

bal= 1000;

}

if(a instanceof CurrentAccount)

{

bal= 5000;

}

a.setAccount(acc\_no, bal);

}

static void doOperation(Operation op)

{

int rate=0;

if(op instanceof SavingAccount)

{

rate= 5;

}

if(op instanceof CurrentAccount)

{

rate= 10;

}

op.calculate(rate);

}

}

class Management

{

static int count= 100;

static int getAccno()

{

count++;

return(count);

}

public static void main(String s[])

{

SavingAccount account1;

CurrentAccount account2;

account1=new SavingAccount( );

account2=new CurrentAccount( );

Bank.open( account1 );

Bank.open( account2 );

Bank.doOperation( account1 );

Bank. doOperation ( account2 );

account1.report ( );

account2.report ( );

}

}

Account ( I )

Operation (AC)

SA CA

**\*Interface\***

interface is the class like construction with the lots of restriction.

interface is used to prepare the standards in the java i.e. they only have the abstract methods.

**Point about the interfaces:**

* In the interfaces all the methods are the public and abstract implicitly (we can’t change).
* In the interface all the variables are the public, final, static implicitly (never be changed).
* We can never the create object of inheritance but we can make the reference variable of it.
* There can be no constructor in interface.
* There can be no static block in the interface.
* “implements ” keywords is use to child class (implemented class) of the interface.
* “interface” keyword is used to declare the interface.

interface Testable

{

Int x= 10; **//implicitly public, final, static**

public final static int y= 5;

final static int z= 20; **//implicitly public**

void m1(); **//implicitly public, abstract**

public abstract void m2();

public void m3( ); **//implicitly abstract**

}

**NOTE:** Conventionally name of interface should be the adjective.

**Child class of interfaces:**

class Test implements Testable

{

public void m1()

{

}

public void m2()

{

}

public void m3()

{

}

void m4()

{

}

}

class Demo

{

public static void main(String s[])

{

Testable t= new Test();

t.m1();

t.m2();

t.m3();

t.m4(); **//error**

}

}

**NOTE:**

All the variable of the interface must be initialize at the time of declaration.

Testable

Implements

(doted line)

Test

Any class can implement more than one interface.

interface I1

{

int x =10;

void m1();

void m2();

}

interface I2

{

int x =20;

void m1();

void m2();

}

class Test implements I1, I2

{

public void m1()

{

System.out.println(“m1 in the Test: ”);

}

public void m2()

{

System.out.println(“m2 in the Test: ”);

}

public void m3()

{

System.out.println(“m3 in the Test: ”);

}

public void m4()

{

System.out.println(“x of I1: ” +I1.x);

System.out.println(“x of I2: ” +I2.x);

}

}

**OR**

interface I1

{

int x =10;

void m1();

void m2();

}

interface I2 extends I1

{

int x =20;

void m1();

void m2();

}

class Test implements I2

{

public void m1()

{

System.out.println(“m1 in the Test: ”);

}

public void m2()

{

System.out.println(“m2 in the Test: ”);

}

public void m3()

{

System.out.println(“m3 in the Test: ”);

}

public void m4()

{

System.out.println(“x of I2: ” +x);

}

}

I1 I2

Test

* Any interface can extends one or more interface.

I1

extends

I2

* Any interface can never extends and implements to any other class.
* exception is that all the interfaces are the child of the object class.
* All the methods of the object class always be available in the abstract form.

interface I1

{

}

class Test implements I1

{

int val;

Test ( int val );

{

System.out.println(“Test object created”);

this.val =val;

}

public String toString( )

{

return “value: ”+val;

}

}

class Demo

{

static void check(I1 ref)

{

System.out.println( ref.toString() );

}

}

class MainTest

{

public static void main(String s[])

{

Demo.check( new Test(s) );

}

}

* Any class can implement to one or more interfaces along with extends to the other class but first extends then implements.

interface I1

{

void m1();

void m2();

}

class C1

{

public void m1()

{

System.out.println(“m1 in the C1”);

}

}

class C2 extends C1 implements I1

{

public void m2()

{

System.out.println(“m2 in the C2”);

}

}

class Demo

{

static void check( I1 ref)

{

ref.m1();

ref.m2();

}

}

class MainTest

{

public static void main(String s[])

{

Demo.check( new C2() );

}

}

* Any class can take the implementation of the abstract method of the parent interface from their parent class.

**Method Hiding:** Method hiding is the term that works in case of static methods.

If the child class has the static method with the same signature as the static method is available in the parent class then it will be known as the method hiding.

class A

{

static void m1() **//hidden method**

{

System.out.println(“m1 in A”);

}

}

class B extends A

{

static void m1() **//hidding method**

{

System.out.println(“m1 in B”);

A.m1();

}

}

class MainTest

{

Public static void main(String s[] )

{

A ref= new B();

ref.m1();

B.m1();

}

}

* We can never override the non-static method with the static method with the child class.In that case will be compilation error.

class A

{

static void m1()

{

System.out.println(“m1 in A”);

}

}

class B extends A

{

static void m1()

{

System.out.println(“m1 in B”);

}

}

Method overloading by changing the argument from parent class reference to the child class reference.

class First

{

void m2( A ref )

{

ref.m1();

System.out.println(“m2 in First”);

}

}

class Second extends First

{

void m2( B ref ) **//overloaded method**

{

ref.m1();

System.out.println(“m2 in Second”);

}

}

class Demo

{

static void check( First f )

{

f.m2( new B() );

}

public static void main ( String s[] )

{

Demo.check( new Second() );

}

}

**OUTPUT:**  M1 in B

M2 in First

**Co- variant Return Type:** This is the new concept that was introduced in JDK5.

According to this concept we can change the return type of the overriding method in the child class as the sub-type of the overridden method of the parent class.

class A

{

void m1()

{

System.out.println(“m1 in A”);

}

}

class B extends A

{

void m1()

{

System.out.println(“m1 in B”);

}

}

class First

{

A m2()

{

System.out.println(“m2 in First”);

return new A();

}

}

class Second extends First

{

B m2()  **//valid method overriding**

{

System.out.println(“m2 in Second”);

return new B();

}

}

class Demo

{

static void check(First f)

{

A ref= f.m2();

ref.m1();

}

public static void main(String s[])

{

Demo.check(new Second());

}

}

**\*Package\***

Package is the collection of the classes, interfaces and can also contains the other sub-packages.

The purpose of package is to arrange classes and interfaces for their easy accessibility and use.

“package” keyword is use to declare any package.

“import” keyword is use to import and use any package in the class.

In java library all the classes or the interfaces are already arranged in the form of package.

In order to use any class or interface from java library we have do first of all import it.

‘java’ and ’javax(java extention)’ are two main packages in the java library and we can never create our own package with these names.

Packages in the java library as follows:

**Java**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **lang** | **io** | **Util** | **Sql** | **net** | **etc** |
| System | File | Date | Drive | URL |  |
| String | FileReader | Calender | Connection | Socket |  |
| Object |  | Arrangelist | Resultset | Datagram Packet |  |
| Integer |  | Collection |  |  |  |
| Math |  |  |  |  |  |
| Etc. | Etc. | Etc. | Etc. | Etc. |  |

**classes & interfaces**

**packages**

**NOTE:** The classes of the same package can use each other without importing.

class of one package can use the class of other package only after importing it.

Java.lang package is the only package that whose classes automatically imported in all the java classes so we can use any class or interface of lang package without importing it.

**Benefit of the package:**

1. **Easy Accessibility:** The classes can be arranging in the package according to their purpose.

For any specific purpose we can focus only on to the classes of the specific package.

**Example:** In java library all the classes related to the networking exist in the in the java.net package.

1. **Removing the name confliction:** In the industry level multiple developers are involve in the project development so they can make class name but package name will be different according to their module.

**Example:** libmn.member

libmn.transaction

1. **Achieving the another level of security:** If any developer makes the member of the class default then that developer can use that member use in any class in the other developer can’t the use the class in other package.

**Using the ‘import’ keyword:** import keyword is use to import the class and interface of one package into the other package.

**There are the following two syntaxes:**

**Syntax 1:**

Import<package\_name>.<class/interface\_name>;

It will import only one class/interface.

**Example:** import java.util.Date;

java.util is package name and Date is class name.

Only date class will be imported.

**Syntax 2:**

import<package\_name>.\*;

all the class/ interface will be imported.

**Example:** import java.util.\*;

All the classes of java of util package will be import.

**NOTE:** import statement always be placed above the class declaration.

We can place any number of imports statement in any dot java file.

import java.util.Date;

class PackTest

{

public static void main( String s[])

{

Date d= new Date();

System.out.println(“current Date: ”+d); **//d.toString() invoked**

}

}

**Java.util.date class:** An instance of class represent to the date and time.

Default constructor of date class always loads the current system date and time.

In the date class toString() method is available in the overriding format that always returns the encapsulated date & time.

If we do not import the class of other package we have to write the full qualified name of class.

**Java.util.Date d= new java.util.Date();**

If more than one package has the class with same name then we have to import that class along with their name not with \*

**Concept of classpath:** classpath is the location in the file system where the java classes and packages can be resides. Current directory always be considered in the classpath by default and in the **‘**classpath**’** environment variable we can also include the other location of file system.

At the time of compilation or execution all the classpath location always be searched for the classes and packages.

**Difference between the path and classpath environment variable:**

* path environment variable is used to set the location of tools while the classpath environment variable is used to set location of classes and package.
* Like the path environment variable we can also set the values in the classpath either temporary and permanently.

To set the classpath temporary :

**set classpath =d:\abc**

if we set any classpath explicitly then the current directory will not be include in the classpath rather we have to include it explicitly.

**set classpath= d:\abc;.**

. means current directory

; classpath separate

* As we set new classpath the old classpath always be overridden.
* To retain the old classpath of the time of setting new classpath

set classpath = d:\abc;

set classpath= %classpath%;d:\xyz

**%classpath%** to retain the old classpath.

* To view the classpath- **set classpath**
* To remove the classpath- **set classpath=**
* There is no option to remove any specific location from the classpath (in case of temporary way).
* %classpath% only use to retain the user define classpath. We can never retain the default classpath.

**Practice Code**

Create the two classes Student and PackTest

class Student

{

int rollno, marks;

Student (int rollno)

{

this.rollno= rollno;

}

void exam(int marks)

{

this.marks= marks;

}

void report( )

{

System.out.println(“Roll no: ”+rollno);

System.out.println(“Marks: ”+marks);

}

Public String toString( )

{

return(rollno+ “:” +marks);

}

}

class PackTest

{

public static void main(String s[])

{

Student st= new Student(101);

st.exam(85);

System.out.println(st);

}

}

**Step 1:** Place both the .java file in the current directry.

**Step 2:** set the classpath-

set classpath= d:\abc;.

Place the Student.java in d:\abc (classpath location), PackTest.java remains in current directory.

javac PackTest.java

**Step 3:** compiler always uses the classpath environment variable after starting the compilation i.e. the .java file whose name we write with the javac command never be searched by using the classpath environment variable.

**Step 4:** move the PackTest.java also in the d:\abc location (delete the .classfile).

Current directory--> javac PackTest.java

**OUTPUT:** file not found: PackTest.java

Either place the PackTest.java in the current directory or write its name with their location.

javac d:\abc\PackTest.java

**Step 5:** classpath environment variable always be used left to right to get the location.

Create the duplicate Student class in the d:\xyz folder.

class Student

{

Student (int r)

{

}

void exam (int m)

{

}

public String toString ( )

{

return(“Duplicate Student Object”);

}

}

**Compilation of the Student.java-**

javac d:\xyz\Student.java

set classpath= d:\xyz;%classpath%

**Permanent classpath:**

Mycomputer **>** properties **>** advanced system setting **>** environment variable **>** new **>**

**Variable name-** classpath

**Variable value-** d:\abc;.

**Creating the new packages:** By using the package keyword we can define the new package

**Syntax-**

package<name of new package>;

**Example-** package mypack;

package declaration always be the first statement in the .java file.

If there are also import statements then they will be place after package declaration.

**Example-** package mypack;

public class Employee

{

int code, salary;

public Employee(int code, int salary)

{

this.code=code;

this.salary =salary;

}

public String toString( )

{

Return code+ “:” +salary;

}

}

Save the file with the name Employee.java

**Rules-**

1. If any class has the package declaration then its source code (.java) and white code (.class) must be placed within the package folder in any classpath location.
2. If more than one classes have the same package declaration than the all these classes will be considered in the same package and can use to each other without import.
3. Class of one package can use the class of other package only after importing it.
4. If any package there is the class with default access specifier then that class can only be use within other class of same package.
5. A class can have the access specifier either default or public.

public of one package can be used in other package by the import statement.

1. If any .java file has the public class then that java file must be saved with the name of that public class otherwise there will be compilation error.
2. In any .java file there can be only one public class.
3. Place the Employee.java as follows

D:\xyz\mypack\Employee.java

**xyz-** classpath

**myback-** package folder

**Employee.java-** package file

**Creating the main class:**

import mypack.Employee;

class PackTest

{

public static void main(string s[])

{

Employee e= new Employee(10, 1000);

System.out.println(e);

}

}

Save with PackTest.java (optional) in any classpath location

**Practice Code:**

**1 class**

package emp;

public class Employee

{

private int code;

int tax;

public Employee(int code)

{

this.code=code;

}

public void setTax(int tax)

{

this.tax=tax;

}

public void report()

{

System.out.println("\nCode: "+code);

System.out.println("Tax: "+tax);

}

protected int getcode()

{

return(code);

}

protected int gettax()

{

return(tax);

}

}

}

**2 class**

package emp;

public class PartTimeEmp extends Employee

{

private int rate, noh;

public PartTimeEmp(int code, int rate,int noh)

{

super(code);

this.rate= rate;

this.noh= noh;

}

public void calculate()

{

tax= (rate\*noh)\*12/100;

}

public void report()

{

System.out.println("\nCode: "+code);

System.out.println("Tax: "+tax);

System.out.println("Rate: "+rate);

System.out.println("No. of hourse: "+noh);

}

}

**3 class**

package mypack;

import emp.Employee;

public class FullTimeEmp extends Employee

{

private int basic;

public FullTimeEmp(int code, int basic)

{

super(code);

this.basic= basic;

}

public void calculate()

{

int t=basic\*12/100;

setTax(t);

}

public void report()

{

System.out.println("\nCode: "+code);

System.out.println("Basic sal: "+basic);

System.out.println("Tax: "+getTax());

}

}

**4 class**

packag mypack;

imoprt emp.Employee;

import emp.PartTimeEmp;

class Company

{

public static void main(String s[])

{

Employee e= new Employee(101);

e.setTax(5000);

e.report();

// e.getCode(); error

PartTimeEmp pe= new PartTimeEmp(102,100,150);

pe.calculate();

pe.report();

FullTimeEmp pe= new FullTimeEmp(103,25000);

fe.calculate();

fe.report();

}

}

Save all the .java file in the package folders in the classpath location.

If main class is in package then with the java command its name will be placed with the package name d:\abc\Company.java

java mypack.Company

When we import any class with \* then priority always goes to the class of current package then the imported package.

**Create the duplicate class PartTimeEmp**

package mypack;

public class PartTimeEmp

{

public PartTimeEmp(int code, int rate, int noh)

{

}

public void calculate()

{

}

public void report()

{

System.out.println(“Duplicate PartTimeEmp”);

}

}

If Employee.java class in same package like emp and p1 then how to import this class

**Example:**

import emp.Employee;

import P1.\*;

class Company

{

public static void main(String s[])

{

Employee e= new Employee( );

P1.Employee e1= new P1.Employee( );

}

}

Place the PartTimeEmp.java in the mypack package folder

private variable get memory of child class object.

**-d or –desitination option of javac tool :**

-d option is use to specify the destination folder of the .class file generated during the compilation. By default the current directory is the destination of the .class file.

**Command-**

javac –d d:\abc PackTest.java

**or** javac –d . PackTest.java

**-d** destination folder

-d use only compiler time

With the –d it is recommended give the name of classpath folder.

If we compile the class that have the package declaration then –d will also create the package folder on the destination folder.

Student.java class

package p1, p2;

class Student

{

//all same

}

PackTest.java class

Import P1.P2.Student;

class PackTest

{

public static void main(String s[])

{

Student st= new Student(101);

st.exam(85);

System.out.println(st);

}

}

Always place the .java file outside classpath location that have the package declaration and we are creating their package folder with –d

d:\temp\Student.java

**d:\temp** not in classpath location.

javac –d d:\abc d:\temp\Student.java

Place the Packtest.java in the current directory.

javac –d d:\abc PackTest.java

**default package:** If in any .java file there is no package declaration then the class of that .java file will be considered in the default package.

We can also consider the default package as the direct classpath location. All the classes in the classes in the direct classpath location can use their default members.

We can never use the class of the default package in the class of any other package because there is no name of default package so it can’t be imported.

**Exception Handling**

Exception is the abnormal flow of the program. In any program there is always the specify flow of the execution but in some situations the flow of program can be diverted in the worst direction (abnormal flow). That abnormal flow of program is considered as the exception.

Due to the exception the execution program can be never be continue i.e. the execution of program will be terminated immediately.

Handling of exception means to write the code in the program to overcome from the abnormal flow and resume the execution in the normal flow.

Each and every exception at the runtime is the object. The class of exception defines the behavior of the exception.

In java library there is the number of exception classes and we can also define custom exception.

Understanding the generation of exception and handling of exception.

Byte code

----------------------------- **1.** ---------------------------- ---------------------------- **2.**  ----------------------------- **6.** ----------------------------

**3. 5.**

**6.**

JVM

Exception object

X **7.** **4.**

1. Execution of the byte code.
2. Statement encountered that caused the exception.
3. Control is moved to the JVM.
4. Exception object is created.
5. Reference id of exception moved in the program.
6. If exception gets handled then exception of program continued else the reference of unhandled exception returned towards the JVM.
7. If unhandled exception returns from the program to the JVM then immediately JVM will terminate the program.

In the java library there are number of classes known as the exception classes and hierarchy of exception classes started with the java.lang.exception class.

**Object**

**Throwable**

**Exception Error**

Runtime exception(unchecked exception) Sql Exception(Checked Exception)

ArithmeticException FileNotFoundException

NullPointerException IOException

ArrayIndexOutOfBoundsException InterruptedException

NumberFormalException etc

ClassCastException

etc

Error can be arising at the compile time as well as at runtime but exception always be arise at the runtime.

Runtime errors can never be handling programmatically but exception can be handled programmatically.

**Example of runtime error:**

1. **StackOverflowError**
2. **NoClassDefFoundError**
3. **OutOfMemoryError**
4. **StackOverflowError:**

class Test

{

void m1()

{

int x;

m1();

}

pubic static void main(String s[])

{

new Test.m1();

}

}

1. **NoClassDefFoundError:**

If after compile anyone .class delete then this error occurred.

1. **OutOfMemoryError:**

class Test

{

int x;

Test()

{

new Test();

}

public static void main(String s[])

{

new Test();

}

}

**Keywords related to the Exception Handling:**

* try
* catch
* throw
* throws
* finally

**try keyword:** is use to make the block, which is known as try block.

In the try block we should write the code that can produced the exception.

**catch keyword:** also use to make the block known as the catch block and in the catch block we always write the code of exception handling.

**throw keyword:** is use to generate the exception explicitly . (Exception generate library class)

**throws keyword:** is use to propagate the exception in the calling environment(function).

throws keyword mainly used with the checked exception.

**finally keyword:** also use to make the block known as the finally block.

The code written in finally block always be execute either exception is generate or not, if generate then handled or not.

finally block is most secure block in our program.

**ArithmeticException:** This exception always be raised in the program when any value divided from the zero (0).

class ExcTest

{

public static void main(String s)

{

int x, y,z;

x= Integer.parseInt(s[0]);

y= Integer.parseInt(s[1]);

z= x/y;

System.out.println(“Result= ”+z);

System.out.println(“Thank You”);

}

}

**OUTPUT:**

Command line- Java Exctest 4 2

Result= 2

Thank You

**Handling the exception with try & catch block:**

class ExcTest

{

public static void main(String s)

{

int x, y,z;

x= Integer.parseInt(s[0]);

**1.** y= Integer.parseInt(s[1]);

JVM

try **2.**

{ **3.**

Javac ExcTest.java 4 0

z= x/y;

System.out.println(“Result= ”+z); **4.**

**5.** }

catch(ArithmeticException d)

{

System.out.println(“Second value can’t be zero”);

}

System.out.println(“Thank You”);

}

}

1. Starting
2. Statement moved for execution
3. Value of y is zero, so exception object create
4. Reference of exception object returned into program.
5. Exception is catch.

**Rule:**

* Catch block always be placed just blow the try block.
* With one try block at least one catch block or finally block must be placed.
* With one try block multiple catch blocks can be associated.
* Just above the finally block there also must be the catch block or try block.

**Concepts:**

* Generation of the exception just to create the exception object.
* Handling of the exception just to catch the reference id of exception object in reference variable.

**ArrayIndexOfBoundsException:** This exception always be generated by the JVM when in the application the value tried to be accessed from outside the bounds of the array.

class ExcTest

{

public static void main(String s)

{

int x, y,z;

try

{

x= Integer.parseInt(s[0]);

y= Integer.parseInt(s[1]);

z= x/y;

System.out.println(“Result= ”+z);

}

catch(ArrayIndexOfBoundsException d)

{

System.out.println(“Can’t access the value from outside the bounds of array“);

}

catch(ArithmeticException d)

{

System.out.println(“Second value can’t be zero“);

}

catch(NumberFormatException d)

{

System.out.println(“Invalid Format of value“);

}

System.out.println(“Thank You”);

}

}

**NOTE:** when generate exception

NumberFormatException: Javac ExcTest.java 4 a

ArithmeticException: Javac ExcTest.java 4 0

ArrayIndexOfBoundsException: Javac ExcTest.java 4

**Concept:**

* In one try block different types of exception can be generated but at one time only one exception generated.
* As any exception arise in the try block immediately gets beaked and further remaining try block will not be executed.
* We should write at least the statement in the try block from which the exception is generated and other dependent statement should also be written in the try block.

**toString() :** toString() method is available in the overriding format that return the details the exception such as- name of exception , line number exception etc.

catch(ArrayIndexOfBoundsException d)

{

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

System.out.println(“Can’t access the value from outside the bounds of array“);

}

catch(ArithmeticException d)

{

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

System.out.println(“Second value can’t be zero“);

}

catch(NumberFormatException d)

{

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

System.out.println(“Invalid Format of value“);

}

**getMessage() :** This method is original defined in the throwable class and it is overridden in all the exception classes.

catch(ArrayIndexOfBoundsException d)

{

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

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

System.out.println(“Can’t access the value from outside the bounds of array“);

}

catch(ArithmeticException d)

{

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

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

System.out.println(“Second value can’t be zero“);

}

catch(NumberFormatException d)

{

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

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

System.out.println(“Invalid Format of value“);

}

**printStackTrace():** Originally this method is define in the throwable class. It prints the details of exception including their originated point and point from there exception propagate in the program.

catch(ArrayIndexOfBoundsException d)

{

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

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

d.printStackTrace();

System.out.println(“Can’t access the value from outside the bounds of array“);

}

catch(ArithmeticException d)

{

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

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

d.printStackTrace();

System.out.println(“Second value can’t be zero“);

}

catch(NumberFormatException d)

{

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

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

d.printStackTrace();

System.out.println(“Invalid Format of value“);

}

Internally printStackTrace() method always invoked toString() method.

**Polymorphic catch:** By using the catch block of parent exception we can handle the child exception also.

catch block of parent exception can never be place above the catch block of child exception.

**Explicitly generation of the exception (use of the throw keyword):**

By using the throw keyword we can any generate any exception explicitly.

**Syntax:**

throw new <Exception\_class\_name>();

**Example:**

class ExcTest

{

public static void main(String s)

{

int x, y,z;

try

{

x= Integer.parseInt(s[0]);

y= Integer.parseInt(s[1]);

if(y==0)

throw new ArithmeticException();

z= x/y;

System.out.println(“Result= ”+z);

}

catch(Exception d)

{

d.printStackTrace();

}

System.out.println(“Thank You”);

}

}

**In the concept of Exception Handling:** It is never recommended to generate the exception explicitly and to handle that exception in the same function.

**Exception Propagations:** It is the way in which the any unhandled exception propagated and calling environment for their handling and further propagate.

Unhandled unchecked exception automatically propagated in the calling environment but the unhandled checked exception by using the throws keywords.

jVM **1.** main() m1() m2()

{ **2.** {} **3.** {

try{ m2() **4.** throw new or implicitException

m1(); **5.** {} }

} **6.**

catch(\_\_)

{ }

**7.**

}

**Example of exception propagation:**

class ExpPropTest

{

public static void main(String s[])

try

{

int x= Integer.parseInt(s[0]);

int x= Integer.parseInt(s[1]);

int z= calc(x,y);

System.out.println(“Result= ”+z);

}

catch(ArithmeticException d)

{

d.printStackTrace();

}

System.out.println(“Thank You”);

}

static int calc(int a, int b)

{

int c;

if(b<=0)

{

throw new AirtmeticException(“Second value can’t be <-=0”);

}

c= a/b;

return(c);

}

}

**finally block:** It contains the code that will always be executed either exception is generated or not, if generated then handle or not.

* Without try finally block will never be executed or return.
* Execution of finally block doesn’t mean handling of exception.
* If catch block executed for handling the exception then finally block also will be executed.

**Syntax 1:**

try

{

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

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

}

catch(\_\_\_)

{

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

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

}

finally

{

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

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

}

**Syntax 2:**

try

{

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

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

}

finally

{

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

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

}

**Example:**

class FinallyTest

{

public static void main(String s[])

{

int x, y,z;

try

{

x= Integer.parseInt(s[0]);

y= Integer.parseInt(s[1]);

z= x/y;

Sytem.out.println(“Result: ”+z);

}

catch(ArithmeticException e)

{

e.printStackTrace();

}

finally

{

System.out.println(“This is the finally block”);

}

System.out.println(“Thank You”);

}

}

**return:** It is control propagate to JVM

* After the generation of exception and before the handling of the exception no other programming statement can be executed except the finally block.
* In case of return statement finally block also will be executed.

**Changes in try block**

try

{

x= Integer.parseInt(s[0]);

y= Integer.parseInt(s[1]);

z=x/y;

System.out.println(“Result: ”+z);

if(z<5)

return;

}

* In case of System.exit(0); the finally block will not be excuted.

if(z<5)

System.exit(0);

Sytem.exit(0); use to forcefully terminate the program.

In case finally block will not be executed.

* Execution of finally block will only ensured when the control is reached on the associated try block.

try

{

x= Integer.parseInt(s[0]);

y= Integer.parseInt(s[1]);

z=x/y;

System.out.println(“Result: ”+z);

}

* Nested try- catch within one try block another try catch sequence can be prepared as well as within one catch block or finally block also the another try catch sequence can be prepared.

class NestedTryTest

{

public static void main(String s[])

{

int x, y, z;

x=y=z=0;

try

{

try

{

x= Integer.parseInt(s[0]);

System.out.println(“x prepared”);

}

catch(NumberFormatException a)

{

x=10;

}

try

{

y= Integer.parseInt(s[1]);

System.out.println(“y prepared”);

}

catch(NumberFormatException a)

{

y=2;

}

z= x/y;

}

catch(ArrayIndexOfBoundsException a)

{

System.out.printl(“Insufficient supplied values”);

}

catch(ArithmeticException a)

{

z=1;

}

System.out.println(“Result: ”+z);

}

}

**NOTE:** If any exception is not handle in the inner try catch then it can be handled in outer catch block.

**Handling of checked Exception:**

The basic difference between checked & unchecked exception is that, if uncheck exception is not handled then it can be automatically propagated in the poly environment but in case of unhandled checked exception it will never be propagated in the calling environment implicitly rather we have to propagate it by using the throws keyword.

**Rule:** In case of checked exception either we have to handle it by using the try catch statement or we have to propagate it by using throws keyword otherwise there will be the compilation error.

**Que. How we can identify that any particular exception is checked or unchecked?**

**Ans.** The child of the java.lang.RuntimeException class will be the unchecked exception.

And the child of java.lang.exception cass(except the runtimeException) will be the checked exception.

import java.io.IOException;

class CheckedExptest

{

public static void main(String s[])

{

int x, y,z;

try

{

x= Integer.parseInt(s[0]);

y= Integer.parseInt(s[1]);

if(y<=0)

throw new IOException();

z= x/y;

System.out.println(“Result: ”+z);

}

catch(IOException e)

{

e.printSStackTrace();

}

System.out.println(“Thank You”);

}

}

**Use of the throws keyword:**

class CheckExeTest

{

public static void main(String s[]) throws IOException

{

int x,y,z;

try

{

x= Integer.parseInt(s[0]);

y= Integer.parseInt(s[1]);

if(y<=0)

throw new IOException();

z= x/y;

System.out.println(“Result: ”+z);

}

catch(IOException e)

{

e.printSStackTrace();

}

System.out.println(“Thank You”);

}

}

**Note:** If we write both try catch as well as the throws keyword then exception will be handle the try catch and throws keyword will not have any significance.

**\*Propagation of checked exception within the program:**

import java.io.IOException;

class ExpPropTest2

{

public static void main(String s[])

{

try

{

int x= Integer.parseInt(s[0]);

int y= Integer.parseInt(s[1]);

int z= calc(x, y);

System.out.println(“Result= ”+z);

}

catch (IOException e)

{

e.printStackTrace();

}

finally

{

System.out.println(“Finally in main”);

}

System.out.println(“Thank You”);

}

static int calc(int a, int b) throws IOException

{

int c;

try

{

if(b<=0)

{

throws new IOException(“Second value can’t be <=0”)

}

c= a/b;

}

finally

{

System.out.println(“Finally in calc”);

}

return c;

}

}

**NOTE:** In any calling function if we invoke any other function that has the throw clause signature then we have to call function either from the try block with proper catch block or we have to write the throws clause.

**Changes in main method:**

public static void main(String s[])

{

try

{

int x= Ingeter.parseInt(s[0]);

int y= Ingeter.parseInt(s[1]);

int z= calc(x, y);

System.out.println(“Result= ”+z);

}

finally

{

System.out.println(“Finally in main”);

}

}

**NOTE:** In the java.library there are number of function that have throws clause in their signature to propagate the child exception.

**Example:**

**1.**

Java.lang.InterruptedException

Eg: public void join( ) throws InterruptedException

(Method of java.lang.Thread class)

**2.**

Java.io.IOException

Eg: public String readLine( ) throws IOException

(Method of java.io.BufferedReader class)

**3.**

Java.sql.SQLException

Eg: public statement createstatemnet( ) throws SQLException

(Method of java.sql.Connection interface)

**4.**

Java.net.UnkonownHostException

Eg: public static InetAddress getLocalHost( ) throws UnkonownHostException

(Method of java.net.InetAddress class)

* In any method signatures multiple checked exceptions can be return separated by the comma but at one time only one exception can be propagate

static int calc(int a, int b) throws IOException, SQLException

* **Return statement in the finally block:**

If we write the return statement in finally block then the unhandled exceptions generated from the associated try block will never be propagated.

finally

{

System.out.println(“Finally in calc”);

return 1;

}

* If return statement in finally block then after finally block we can also not write any other statement because it will be the unreachable statement.
* It’s never recommend to write return statement in finally.

**Custom Exception:**

package p1.exp;

public class InvalidMarkException extends Exception

{

String msg;

public InvalidMarkException()

{

msg= “ ”;

}

public InvalidMarkException(String msg)

{

this.msg= msg;

}

public String toString()

{

return(“InvalidMarksException: ”+msg);

}

}

import p1.exp.InvalidMarksException;

class Student

{

int rollno, marks;

Student(int rollno)

{

this.rollno= rollno;

}

void exam(int marks) throws InvalidMarksException

{

if(marks<0 || marks>100)

{

throw new InvalidMarksException("Marks should be within 0 -100");

}

this.marks= marks;

}

void report()

{

System.out.println("Roll no: "+rollno);

System.out.println("Marks: "+marks);

}

}

import p1.exp.InvalidMarksException;

class School

{

public static void main(String s[])

{

Student st=new Student(101);

int x= Integer.parseInt(s[0]);

try

{

st.exam(x);

st.report();

}

catch(InvalidMarksException i)

{

i.printStackTrace();

}

}

}

**Difference between the checked & unchecked:**

|  |  |  |
| --- | --- | --- |
|  | **Unchecked Exception** | **Checked Exception** |
| **1.** | All the child of java.lang.runtimeException class | All the child of java.lang.Exception class except the runtimeException class. |
| **2.** | Automatically propagated in calling function if not handled. | Have to be propagated explicitly by using the throws keyword if not handled |
| **3.** | Not known by the compiler. | Known by the compiler. |
| **4.** | Can be generated from the JVM, library or developer code. | Always be generated from library and developer code. |
| **5.** | Working on the unchecked exception optional. | On the checked exception we have write the code either for handling for propagateion. |

Developer’s code

Both checked & unchecked exception generated

Library

JVM

Only unchecked exception generated

**Exception propagation & method overriding:**

In case of method overriding if overridden method in parent class have throws clause to propagate the any checked exception in the overriding method in the child class the same checked exception have to be propagate or any narrower(child of exception) checked exception can be propagated but any new checked exception or any brooder (parent of exception) exception can’t be propagated.

class FirstException extends Exception

{

}

class SecondException extends FirstException

{

}

class A

{

void m1() throws FirstException

{

throw new FirstException();

}

}

class A

{

void m1() throws SecondException **//valid method overriding**

{

throw new SecondException();

}

}

Always narrower(child of exception) throws in method overriding.

**array**

array is the collection of similar data type.

The purpose of array is to store the same type of elements for the same purpose.

**Example:** Storing marks of the 10(ten) students.

In java arrays always be the object.

**Syntax to create the array:**

int a[]; reference variable

a= new int[5];

array object creation

a 110

110

at runtime JVM always creates the proxy class for the array object.

**Format name of proxy classes:**

**int array –** [I

**float array –** [F

**double array –** [D

**Program:** storing the 10 (ten) random values in the array and find out the even & odd values

class ArrayTest

{

public static void main(String s[])

{

int ecount, ocount;

int a[];

a = new int[10];

System.out.println(“Reference id of array object: ”+a);

for(int i=0; i<=a.length; i++)

{

a[i]= (int) (Math.random()\*100);

}

ecount= ocont= 0;

System.out.println(“List of values as follows: ”);

for(int i=0; i<=a.length; i++)

{

System.out.println(a[i]);

if(a[i] %2==0 )

ecount++;

else

ocount++;

}

Systrm.out.println(“No. of even values: ”+econt);

Systrm.out.println(“No. of odd values: ”+ocont);

}

}

**Math.random() -** This is the static function in the math class that always return the random values between 0 – 1 (>=0 && <1).

**Singnature-**

public static void main()

**example- output**

Math.random(); // .6548

Math.random() \* 100; // 65.48

(int)(Math.random\*100); // 65

**Valid syntax to use the array-**

1. int a[];

a= new int[5];

1. int []a;

a= new int [5];

1. int []a= new int[5];
2. int a[]={3, 5, 7, 8, 9};

internal code:

int a[]= new int[5];

a[0]= 3;

a[1]= 5;

a[2]= 7;

a[3]= 8;

a[4]= 9;

1. int a[];

a= new int[]{3, 5, 7, 8, 9} **//array object created with size 5**

All the array element always have default values.

class ArrayUtil

{

static void input(int []a)

{

for(int i=0; i<a.length; i++)

{

a[i]= (int)(Math.random1) \*100);

}

}

static void output(int []a)

{

for(int i=0; i<a.length; i++)

{

System.out.println(a[i]);

}

}

static void sort(int []a)

{

for(int i=0; i<a.length; i++)

{

for(int j=i+1; j<a.length; j++)

{

if(a[i]>a[j])

{

int temp= a[i];

a[i]= a[j];

a[j]=temp;

}

}

}

}

static int search (int []a, int val)

{

**//if value not found then return -1**

}

static int search (int []a, int val)

{

**//home work**

}

static int[] delete (int []a, int val)

{

**/\***

**Create a new array with size -1**

**Copy values from old to new array except the value to be deleted**

**Return reference id of new array**

**\*/**

}

static int[] insert (int []a, int val, int index)

{

**/\***

**If index is valid, then create the new array with size +1**

**Copy values from old array to new array including the new value**

**Return reference id of new array**

**\*/**

}

static int[]

}

class ArrayTest

{

public static void main(String s[])

{

int a[]= new int[5];

ArrayUtil.input(a);

ArrayUtil.output(a);

int x= ArrayUtil.search(a, 6);

if(x== -1)

System.out.println(“6 does not exist in the array”);

else

System.out.println(“6 exist at index” +x);

a= ArrayUtil.delete(a,5);

System.out.println(“\nArray after deleteing 5”);

ArrayaUtil.output(a);

a= ArrayUtil.insert(a, 15, 7);

System.out.println(“\nArray after inserting the 15”);

ArrayaUtil.output(a);

int b[]= {7, 8, 3, 9};

int temp[]= ArrayUtil.intersect(a, b);

System.out.println(“\nArray values after intersection”);

ArrayUtil.output(temp);

int b[]= {7, 8, 3, 9};

int temp[]= ArrayUtil.union(a, b);

System.out.println(“\nArray values after union”);

ArrayUtil.output(temp);

}

}

**Two Dimensional array:**

int a[][]= new int[3][4];

no. of array no. of element in each array

110

0 0 0 0

110 220 330

440

a 220

0 0 0 0

440

330

0 0 0 0

int a[][]= new int [3][4];

for (int i=0; i<a; i++)

{

for(int j=0;j<a[i].lenght; i++)

{

a[i][j]= (int)(Math.random()\*100);

}

}

System.out.println(“Values as follows: ”);

for(int i=0; i<a.lenght; i++)

{

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

{

System.out.println(“\t” +a[i][j]);

}

System.out.println( );

}

**Assignment 1:**

Create one dimensional array of 4 rows and 3 columns and store the random values \*8. Find out the followings:

1. sum of the element in each row.
2. Max value in each row
3. Find out the row number that have the highest some of values.
4. Find out the columns number that have the highest some of values.
5. Find out the largest values among the all values stored I the two dimensional array.
6. Find out the row number that have maxima mum number of repeat values.

**Array of object:** we can also create the array of object but actually it will be the array of reference variable that will hold the reference id of object.

Student st[]

st=new Student[5];

reference variable of Student class

st

null null null null null

110

110

Student st[];

st= new Student[5];

for(int i=0; i<st.length; i++)

{

st[i]= new Student();

}

110

0

0

0

220

0

0

st

0

110 220 330 440 550

550

0

0

0

0

440

0

0

0

330

0

0

**Assignment 2:**

Create the array for 10 student object except their names as command line argument and store the marks randomly and store the roll no. sequence 101, 102, 103,……..110 calculate the grade of students. Find out the followings:

1. Student that have the highest marks.
2. No. of Student have the ‘A’ grades.
3. Find out the Students that have the highest marks grade wise.

**NOTE:**  array encapsulate in string class.

**String Handling**

String is the collection of characters. In the real word in the string format we always represents the name of the person, address, email id etc.

Programming implementation of the string is the char[].

Collection of characters is most frequently used type of date in any program so each and every programming language provides the special provision for storing and working on the string.

In java there are the three classes java.lang package for string handling

1. String (immutable)
2. StringBuffer(mutable)
3. StringBuilder(mutable)

**Immutable:** Its means the non- editable.

**Mutable:** Its means the editable.

In case of string object we can never change the content if we try to change the content in the string object then automatically new object string will be created.

class StringTest

{

public static void main(String s[])

{

String st1= new String(“ABC”);

System.out.println(st1); **//toStringmethod is available in overriding format in string class**

String st2= st1.toLowercase(); **//new string object create and return**

System.out.println(st1==st2);

}

}

**IMP NOTE:** Whenever in the java program any string constant(“ ”) is encountered then immediately internally new string object is created in the string constant pool.

**String constant pool:** It is the logical area within the class area in which all the string object are stored that are created by the string constant.

The main feature of this area is that it never contents any duplicate object with expect of contant.

public static void main(String s[])

{

String st1= “ABC”; **//new string object created in the string pool**

String st2= “ABC”; **//reference id of existing string object return**

System.out.println(“st1==st2”);

St2= st2.toLowcase();

System.out.println(“st1==st2”);

}

**Pool**

110

String Object

st1

110

220

st2

**NOTE:**  When the new object is created as the result of changing the content in String object it always be created in the heap area.

If any constant write in “ ”then object create in pool otherwise object created in heap area

st2= st2.tolowcase(); **//new object with contents small abc will created in heap area**

**NOTE:**

String st1= new String (“ABC”);

In heap in pool

It will create the two objects one in the heap and second in the pool.

class StringTest

{

public static void main(String s[])

{

String st1= “ABC”;

String st2= “abc”;

String st3= “abc”;

st1= st1.tolowercase();

System.out.println(st1==st2);

System.out.println(st1==st3);

System.out.println(st2==st3);

}

}

**javadoc tool:** it is used to prepared the documentation from the java source code.

This tool always reads the documentation comment from the dot (.)java file.

**Step to create the documentation of java library:**

**Step1:** Extract the src.rar in JDK folder src.rar exist.

**Step2:** Open the src folder on the doc prompt

**Step3:** Run the following command

Javadoc java.lang  **//it will create the documentation of all class of java.lang package**

**Practice code:** (how many objects will be created)

class StringTest

{

public static void main(String s[])

{

String st1= new String (“ABC”);

String st2= “ABC”;

String st3= st1.concat(“abc”);

st1= st1.toLowercase();

String st4= “abc”;

}

}

**NOTE:** We can also use the String constant as String object.

**“ABC”.lenght();**

**StringBuffer:** constant pool never works in case of StringBuffer. Always then new keywords.

The default size of internal char[] will be the 16.

As the StringBuffer is mutable the growing policy of char[] is twice of current size +2.

**Constructor:**

public StringBuffer() -size of char[] is 16

public StringBuffer(int capacity) -size of char[] is argument capacity

public StringBuffer(String str) -size of char[] is 16+ size of argument str.

**Example:**

class StringBufferTest

{

public static void main(String s[])

{

StringBuffer sb1= new StringBuffer();

StringBuffer sb2= new StringBuffer(40);

StringBuffer sb3= new StringBuffer(“ABC”);

System.out.println(sb1.capacity); **//16**

System.out.println(sb2.capacity); **//40**

System.out.println(sb3.capacity); **//19**

}

}

**NOTE:** capacity() of StringBuffer return the current size the internal char[]

**Example:**

public static void main(String s[])

{

StringBuffer sb1= new StringBuffer(5);

System.out.println(sb1.capacity); //5

Sb1.append(“ABCDEFG”);

System.out.println(sb1.capacity); //12

}

**Method of StringBuffer:**

**P**

1. **public StringBuffer append(String str) :** This method will append the argument String into this StringBuffer object and returns the reference id of current StringBuffer.
2. **public StringBuffer reverse( )**
3. **public StringBuffer insert(int index, String str)**
4. **public StringBuffer delete(int startIndex, int endIndex)**

**NOTE:** All the above methods change the content in the current object and also returns the reference id of same object.

We can prepare chain of the method calling with the methods of StringBuffer as they return the reference id of same object.

StringBuffer sb= new StringBuffer(“ABC”);

sb.append(“ayz”).insert(2, “MNO”).delete(2, 4).reverse();

System.out.println(sb);

**StringBuilder:** This class is identical to the StringBuffer class but have the following changes-

1. StringBuilder class was edit in JDK file and StringBuffer is part of JDK from the starting.
2. StringBuffer is thread safe while StringBuilder is not thread safe.

**Assignment:**

1. Accept the string from end user and find out the number of words.
2. Accept the string from end user and find out the number of vowels.
3. Accept the string from end user and check whether string palindrome or not.
4. Accept the string from end user and reverse all the words in the string.
5. Accept the string from end user and check string palindrome or not with expect or not(12321)
6. Accept the string from end user and whether string contains all the alphabets at least once or not.

**AWT (Abstract Window Toolkit)**

* This is the first approach in java introduce in JDK 1.1.
* Java.awt package contains classes and interfaces for the window programming.
* In the AWT terminology the GUI always be prepare by using the components such as text field, button, label, radio button etc.

Each component on the GUI is an object.

**Sample GUI**

Frame Lables TextField Buttons



**Java.awt.componet class:** This is the root class from all GUI components. It defines the common functionality required for all the components.

java.awt.component

etc.

Container button label TextField

Normal component

Window Panel

Frame Applet

Container like component

**Method of component class:**

1. public void setSize(int width, int height) **//(height width will be in pixels)**
2. public void setVisible(boolean b)
3. public void setBounds(int x, int y, int width, int height)
4. public void enabled(boolean b)

**Container class:** It provides the functionality of the containership.

1. public void add(component c)
2. public void remove(component c)

**Window class:** It is the child class of the container class and it have the boarder but does not have the menu bar.

**Frame class:** It is the child of window and also have the menu bar.

**Panel:** It is the hidden container it does not have their visibility rather it have to be edit on to the other contain to make it visible.

**Applet:** It is child of panel and will be displayed on to the web browser or applet viewer.

**Step to prepare frame:**

1. create the object of java.awt.frame class or it child class.
2. Set the size.
3. Makes the frame visible.

**Example:**

import java.awt.\*;

class Myframe

{

Frame fr;

MyFrame()

{

fr= new Frame(“Test Frame”); **//step1**

fr.setSize( 400, 300); **//step2**

fr.setVisible(true); **//step3**

}

public static void main(String s[])

{

new MyFrame();

}

}

**Example:**

import java.awt.\*;

class MyFrame2 extends Frame

{

MyFrame2()

{

super(“Test Frame”);

setSize(400, 300); **//step2**

setVisible(true); **//step3**

}

public static void main(String s[])

{

new MyFrame2(); **//step1**

}

}

**Adding the component into the frame:** There are two base to adds the components onto the frame.

1. With layout manager.
2. Without layout manager.

**Layout Manager:** It I the predefine way to arranged the component onto the frame and other containers. Each and every container always have default layout manager. In the java.awt package there are number of classes for the layout managers.

* FlowLayout
* BorderLayout
* GridLayout
* CardLayout

If you want to add the components our desire location(coordinates) when we have to go without layout manager.

**Step to add the component without layout manager:**

1. Create the object of container or it child such as frame.
2. Remove the predefine layout manager from the container
3. Create the object of the components.
4. Set the bounds (coordinates of the components).
5. Add the components onto the frame.
6. Set the size of frame.
7. Make the frame visible.

import java.awt.\*;

class StudentFrame

{

Frame fr;

TextField tf1, tf2;

Lable lb1, lb2;

Button b1;

StudentFrame()

{

fr.new Frame(“Student Frame”); **//step1**

fr.setLayout(null); **//step2**

tf1= new TextField();

tf2= new TextField();

lb1= new Label(“Name”);

lb2= new Label (“Marks”);

b1= new Button(“Submit”);

lb1.setBounds(30, 50, 100, 50);

tf1.setBounds(150, 50, 100, 50);

lb2.setBounds(30, 120, 100, 50);

tf1.setBounds(150, 120, 100, 50);

b1.setBounds(70, 200, 100, 50);

fr.add(lb1); fr.add(tf1);

fr.add(lb2); fr.add(tf1);

fr.add(b1);

fr.setSize(300, 300); **//step6**

fr.setVisible(true); **//step7**

}

public static void main(String s[])

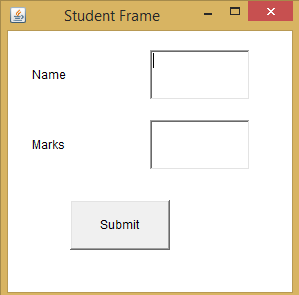
{

New StudentFrame();

}

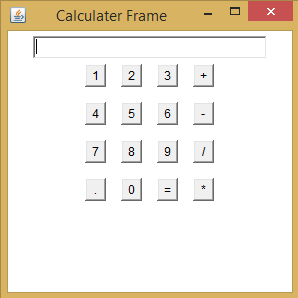
}

**O/P:**



object of StudentFrame

**Assignment:**



Make button by arrary.

**Code:**

import java.awt.\*;

class CalculaterFrame

{

Frame fr;

TextField tf;

Panel p;

GridLayout g;

Button b[]= new Button[16];

String str[]= {"1","2","3","+","4","5","6","-","7","8","9","/",".","0","=","\*"};

CalculaterFrame()

{

fr= new Frame("Calculater Frame");

fr.setLayout(null);

p= new Panel();

tf= new TextField(30);

tf.setBounds(20, 50, 250, 30);

fr.setLayout(new FlowLayout());

for(int i= 0; i<16;++i)

{

b[i]= new Button(str[i]);

}

g= new GridLayout(5, 5, 15, 15);

p.setLayout(g);

fr.add(tf);

for(int i= 0; i<16;++i)

{

p.add(b[i]);

}

fr.add(p);

fr.setSize(300, 300);

fr.setVisible(true);

}

public static void main(String s[])

{

new CalculaterFrame();

}

}

**Event Handling**

An event is the change in the atate of any component such as- clicking on the button, writing in the TextField etc.

**State of any component can be changed in two ways-**

1. By the user interaction.
2. By the application itself.

java.awt.event package provides the number of event classes whose objects are created when components changing their state.

**List of event classes as follows-**

1. **ActionEvent:** An object of this event classes created by the environment when any button is clicked or any menu item is selected.
2. **ItemEvent:** This event class object is created when the radio button, check box or dropdown menu is selected.
3. **FocusEvent:** This event is generated when any component gained and lost the keyboard focus.
4. **TextEvevnt:** This event is generated when the text in the TextField is changed.
5. **MouseEvent:** Object of this event class is created when the mouse is clicked, pressed, released, moved or tracked, entered or exited in any component area.
6. **KeyEvent:** This event class object is created when the key of the keyboard is pressed, typed or released.
7. **WindowEvent:** This event object is created when any window is opened, closed, activated, deactivated, maximized and minimized.
8. **ComponentEvent:** This event object is created when any component is added of removed from the container.

**Event Delegation Model:** This is the working pattern in java for the event handling. In this model there are two participants:

1. Source
2. Listener
3. **Source:** It is the GUI component which is responsible to generated the event that means in the changes in the state of component the event will be generated.
4. **Listener:** It is the object of user defined class that contains the place of code that the programmer want to be executed at time of event generation.

Listener contains the code in the form of callback method.

**callback method:** are the method whose name provided by the environment, method developed by the programmer finally method invoked by the environment.

**Working of event delegation model:**

**user Interface** **Java app**

Listener class

{

callback Method()

{

Java code for event handling

}

}

2.1

2.2 2.3

2.4 1.1

2.0 **call back()** 1.0

**user**

2.5

**1.0** Listener object created

**1.1** Listener object is registered with the source object

**2.0** User interaction with the GUI & state of the source is changed.

**2.1** Source object is modified to generate the event.

**2.2** The event object is created.

**2.3** Reference id of source object itself stored in the event object.

2.4

**2.4** callback method is invoked and reference id of event object is passed.

**2.5** Execution of callback() method is started.

Creation of listener class always be perform under the standard developed in the AWT API.

Java.awt.event package provides the number of interfaces known as the listener interfaces corresponding to each event separately.

GUI developer have to implement the listener interface in order to prepare the listener class.

**Listener interfaces as follows:**

|  |  |  |
| --- | --- | --- |
| **Event classes** | **Listener interfaces** | **callback methods** |
| ActionEvent | ActionListener | public void actionPerformed(ActionEvent e) |
| ItemEvent | ItemListener | public void stateChanged(ItemEvent e) |
| TextEvent | TextListener | public void textChanged(TextEvent e) |
| FocusEvent | FocusListener | public void focusGained(FocusEvent e)  public void focusLost(FocusEvent e) |
| MouseEvent | MouseListener  MouseMotionListener | public void mouseClicked(MouseEvent e)  public void mousePressed(MouseEvent e)  public void mouseReleased(MouseEvent e)  public void mouseEntered(MouseEvent e)  public void mouseExited(MouseEvent e)  public void mouseDregged(MouseEvent e)  public void mouseMoved(MouseEvent e) |
| WindowEvent | WindowListener | public void windowOpened(WindowEvent e)  public void windowClosed(WindowEvent e)  public void windowClosing(WindowEvent e)  public void windowActivated(WindowEvent e)  public void windowDeactivated(WindowEvent e)  public void windowIconified(WindowEvent e)  public void windowDeiconified(WindowEvent e) |

Each & every source provides the registration method to registered the listener object with the source object.

**Format of the registration method as follows:**

public void add (name of the listener interface) (reference var of listener interface)

**Example:**

1. method in the Button class

public void addActionListener(ActionListener ref)

1. method in the TextField class

public void addTextListener(TextListener ref)

**Program1.**

import java.awt.event.\*;

class MyListener implements ActionListener

{

public void actionPerformed(ActionEvent e)

{

System.out.println(“Hello Everyone”);

}

}

**Program2.**

import java.awt.\*;

class EventTest

{

Frame fr;

TextField tf;

Button b;

EventTest()

{

fr= new Frame(“Frame Test”);

fr.setLayout(null);

tf= new TextFeld();

b= new Button(“Submit”);

tf.setBounds(50, 100, 100,50);

b.setBounds(50, 180, 100, 50);

fr.add(tf); fr.add(b);

MyListener listener= new MyListener();

**// registration of listener object with source**

b.addActionListener(listener);

fr.setSize(400, 400);

fr.setVisible(true);

}

public static void main(String s[])

{

new EventTest();

}

}

It is recommend to make the listener class of our designer class so that GUI component can be used in the callback method directly.

**Change in the program:**

import java.awt.\*;

import java.awt.event.\*;

class EventTest2 implements ActionListener

{

Frame fr;

TextField tf;

Button b1, b2;

EventTest()

{

fr= new Frame(“Frame Test”);

fr.setLayout(null);

tf= new TextFeld();

b= new Button(“Submit”);

tf.setBounds(50, 100, 100,50);

b.setBounds(50, 180, 100, 50);

fr.add(tf); fr.add(b);

MyListener listener= new MyListener();

**// Listener object will not be created separately**

b.addActionListener(this);

b.addActionListener(this);

**// registration of listener object with source object**

fr.setSize(400, 400);

fr.setVisible(true);

}

public void actionPerformed(ActionEvent e)

{

String text= e.getActionCommand(); **//to get the caption written above the component**

if(text.equals(“Submit”))

{

tf.setText(“Hello”);

}

if(text.equals(“Cancel”))

{

tf.setText(“Good Bye Everyone”);

}

}

public static void main(String s[])

{

new EventTest();

}

}

**Assignment:**

**Changes in the actionPerformed:**

public void actionPerformed(ActionEvent e)

{

if(e.getsource()==b1)

{

tf.setText(“Hello”);

}

if(e.getsource()==b2)

{

tf.setText(“Good Bye Everyone”);

}

}

**Window Listener :**

//WindowEventTest.java

Import java.awt.\*;

Import java.awt.event.\*;

Class WindowEventTest implements WindowListener

{  
 Frame fr;

TextField tf;

WindowEventTest()

{

fr=new Frame(“window event test”);

fr.setlayout(null);

tf.new TextField();

tf.setBounds(50,50,100,50);

fr.add(tf);

fr.addWindowListener(this);

fr.setSize(400,400);

fr.setVisible(true);

}

public void windowOpened(WindowEvent e)

{

System.out.println(“Window is opened”);

}

public void windowClosing( WindowEvent e)

{

System.out.println("window is closing");

fr.dispose();

}

public void windowClosed ( WindowEvent e)

{

System.out.println("window is closed");

}

public void windowActivated ( WindowEvent e)

{

System.out.println("window is Activated");

tf.setText("window is activated");

}

public void windowDeactivated( WindowEvent e)

{

System.out.println("window is deactivated");

}

public void windowIconified ( WindowEvent e)

{

System.out.println("window is minimized");

tf.setText("window is minimized");

}

public void windowDeiconified ( WindowEvent e)

{

System.out.println("window is maximized");

tf.setText("window is maximized");

}

public static void main(String s[])

{

new WindowEventTest();

}

}

**Adapter classes :**

There are no. of listener interfaces that have the multiple callback methods such as WindowListener, FocasListener, MouseListener, KeyListener , etc.

In java.awt.event package, there are the adapter classes that provides the default implementation of the method s of the listener interface.

Adapter classes are as follows:-

* Window Adapter
* KeyAdapter
* FocusAdapter
* etc.

|  |  |
| --- | --- |
| WindowLIstener  WindowAdapter  MyListener class  (Indirect implementation of Listener Interface) | WindowListerner(I)  MyLIstener class  (Direct implementation of Listener Interface) |