***DAY 1***

1. Inside class - methods... Independently - functions.

C++ Java

a. Object - real world entity, contain memory reference or address

b. class - template/blue print

- contains variables, methods and constructors

- accessed by creating the object

c. Encapsulation - wrapping of data into single unit

- Data hiding/Information hiding

- class contain private variables and public methods

d. Abstraction - without knowing the background details of the class, we are using that class

e. Inheritance - accessing the properties of one class into another class

- code reusability

In C++, single, multiple, In Java, we don’t have multiple

multilevel, hybrid, hierarchical inheritance, instead we have interface

f. Polymorphism - one class which takes many forms - 2types

1. static/compile time - using overloading

In C++, we can overload operator, methods, const. In Java, no operator overloading

2. Dynamic/runtime - using method overriding

1. Constructor

- What is constructor?

- Whenever we create an object for a class, memory will be allocated so the allocation of memory is called constructor

- When constructor will be invoked?

- Whenever we create an object for a class, the constructor will be invoked.

- Why constructor?

- used to initialize value to variables at time of object creation

- By default all class will contain one default constructor(ie) constructor without any argument, and we can create our own constructor where your constructor name and class name should be same in that case it will destroy the already present default constructor

- Constructor can be created with or without arguments

- Constructor can have access specifiers

- Constructor should not have return type

***Syntax -***

class Box {

Box() { //default constructor

}

void Box(){ //method

}

}

1. Destructor 3. No destructor, instead we have

- deallocation of memory - Automatic Garbage collection

- ~(tilde) operator - System.gc()/Runtime.gc() where it will internally call protected

void finalize(){

//define all resources to be deallocated and called only once in lifecycle of prg }

1. String is datatype 4. String is a class/literal

**Fundamentals of Java**

1. **Identifiers** - names given for variables, methods and class should always start with letters, can contain digits, \_ , $ are allowed.

class - starting letter of each word should be capital letter, SampleEmployee, ExampleProgram

method - camelCase, from 2nd word onwards starting letter should be capital(), setEmployeeName()

variables - smallcase

constants - upper case

1. **Keywords - 51 keywords**

- goto, const are the keywords of Java but if we try to use, it will show compilation error

goto - continue

const- final

- 3 reserved words - true, false, null

boolean - true, false

object - null

1. **Datatype** - how we store the values into variables

|  |
| --- |
| datatypes byte size- min max range |
| 1. byte 1 -27 27 -1 -128 to 127 |
| 2. short 2 -215 215-1 |
| 3. int 4 -231 231-1 |
| 4. long 8 -263 263-1 |
| 5. float 4 -3.4e38 3.4e38 |
| 6. double 8 -1.7 e38 1.7e38 |
| 7. char 2(16bit 0 65535 unsigned integer) |
| 8. boolean 1 bit N.A{VM dependent} N.A{allowed true/false} |

1. **Literals** - how we can define value to the variable

a. Numeric Literal - 3 ways.. use \_ between no. after 1.7v

1. Decimal literal - base 10

int a=4;

int b=846;

2. Octal literal - base 8 - always preceded with 0

int a=01; //1

int b=02; //2

03,04,05,06,07

int c==010; //8

0\*8^0=0

1\*8^1=8

0\*8^2=0

int d=011; //9

1\*8^0=1

1\*8^1=8

0\*8^2=0

int e=012; //10

3. Hexadecimal literal - base 16 - always preceded with 0X or 0x

int a=0x16; //22

6\*16^0=6;

1\*16^1=16

By default any number it is consider as int

b. Float literal

float a1=3.13; //error

float a2=3.13f; //correct

float a3=5.14F; //correct

float a4 = 1.2e3f //correct

c. Double literal - By default all decimal values are double

double a1=3.13; //correct

double a2=3.13d; //correct

double a3=3.13D; //correct

double a4 = 1.2e3 //correct

d. Long literal

long a1=10; //correct

long a2=10l; //correct

long a3=10L; //correct

e. short literal

short s=10; //correct

short s1=10s; //error

short s2=10S; //error

f. boolean literal

boolean b1=true; //correct

boolean b2=false; //correct

boolean b3=True; //error

g. Character literal - single char single quote .. till 66535.. Unicode ‘\u0061’.. ‘\n’

char c1='a'; //correct

char c2="a"; //error

char c3='ab'; //error

char c4="ab"; //error

char c5=100; //correct(ascii value)

char c6=37373; //correct

char c7=-25; //error

char c8=(char)70000; //correct

Character can also be represented as unicode representation

char a='\u0001';

h. String literal - sequence of char with double quotes. String is a class/literal. if no obj. is formed then it is literal.

String s="hello";

i. null literal - can be defined only for object

String s=null;

class Box{ }

Box b=null;

5. Variables

- identifier used to store the values - 2types

a. Instance/class variable

- any variable that is declared inside the class and outside the method

class Example{

int a; //instance variable

void add(){

}}

- No need to initialize the instance variable, it will take default value depending on the datatype

int, short, byte, long - 0

float, double - 0.0

boolean - false

char - /u0000

object - null

b. Local variable

- Any variable that is declared inside the method and it should be compulsorily initialized otherwise it leads to compilation error.

***Syntax -***

class Example{

void add(){

int a=0; //Local variable

}

}

6. Access Specifiers/Access Modifiers - 4 types

1. public - it can be accessed anywhere

2. private - it can be accessed only within the class in which it is declared

3. protected - it can be accessible within the class as well as in inherited class

4. default - if we are not specifying public or private or protected then by default it is default access specifier

accessed only in same package and same/diff class.

Visibility default private protected public

1. same package yes yes yes yes

same class

package com;

class Sample {

int a;

private int b;

protected int c;

public int d; }

2. Same package yes no no yes

different class

package com;

class Sample {

int a;

private int b;

protected int c;

public int d;

}

class Sample1{ }

3. Different pkg no no yes yes

subclass

package com;

class Sample {

int a;

private int b;

protected int c;

public int d;

}

package com1;

class Sample1 extends Sample{

}

4. Different pkg no no no yes

non subclass

package com;

class Sample {

int a;

private int b;

protected int c;

public int d;

}

package com1;

class Sample1 {

}

7. Type conversion

- converting from one datatype to another type - 2types

1. Implicit conversion

- converting from lower datatype to higher datatype

float a=31.4f;

double d=a;

2. Explicit conversion

- converting from higher datatype to lower datatype

int i=128;

byte b; //-128 to 127

b=(byte)i; //explicit conversion

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

int i=129; //-127

***DAY 2***

**Operators**

1. Arithmetic Operators +, -, \*, /

2. Modulus operator %

3. Relational operator >, <, >= , <=

4. Assignment operator =

5. Conditional assignment operator +=, -=, \*=, /=

a+=2; a=a+2;

a-=3; a=a-3;

6. Equality and Inequality operator ==, !=

7. Ternary operator ?:

z=a>b?a:b; z=10>20?true:false; //false

8. Increment, Decrement operator ++, --

int a=10, b=6;

int c=a++ + b++; 10+6//postfix - first assign and then in nextline only it will incremented

SOP(a+" "+b+" "c);//11 7 16

int d=++a + ++b; 12+8//prefix - first increment then assign

SOP(a+" "+b+" "d); //12 8 20

9. Bitwise operator - always works on truth table mostly 0,1

& - Bitwise AND

| - Bitwise OR

^ - Bitwise XOR

~ - 1's complement

>> - Right shift - n/2^s

<< - Left shift - n\*2^s

a=5,b=6

a b a&b=4 a|b=7 a^b=3 ~a

0 0 0 0 0 1

1 1 1 1 0 0

0 1 0 1 1 1

1 0 0 1 1 0

a=8,b=2

a>>b = 8/2^2=8/4=2 (a is n and b is s)

a<<b = 8\*2^2=32

10. Boolean logical operator - works on true or false

& - Logical AND

| - Logical OR

! - Logical NOT

11. Shortcircuit logical operator - used to check the condition

&&

||

12. new operator - used to create an object for a class - 3ways

***Syntax -***

//Base class

class Box {

int a=10;

void add(){

int b; } }

1. Box b=new Box();

- We are creating an object called "b", we are allocating the memory using new operator and stores inside the object, and invokes ur default constructor

Box b=new Box(10); //invoke parameterized constructor that takes 10 as arg

Box b=new Box("hello");

Box b1=new Box(); //

Box b2=b1;

b2.add();

2. Box b; //object declaration contain null reference

b=new Box(); //creating an object, allocated memory and invoke default constructor //1000

b.add();

SOP(b.a);//10

SOP(b); //memory reference

3. We dont want to access methods or variables, we just want to allocate memory and invoke constructor

new Box().add(); //new ref and invoke add()

SOP(new Box().a); //10 //new ref and invoke 'a' variable

13. instanceof operator - check whether an object is a instance of a class

***Syntax -***

Box b=new Box();

if(b instanceof Box){

//logic

}

Execution control statements in Java - 3 types

1. Conditional statement - 2types

a. if/if-else/if else if/nested if - always check for condititon

In C and C++, other than 0 if we provide any number then the condition is true, but this is not possible in Java

In java only condition is always checked.

***Syntax -***

if(10){ //error

}

else{ }

int a=10;

if(a=11){ //error

}

if(a==11){ //condition

}

b. switch case - used to check multiple condition

***Syntax -***

switch(expr){

case arg:

//stmt;

break;

--

--

default:

//stmt

break;

}

-expr can be either int/byte/short/char/enum/String(JDK1.7)

Cannot switch on a value of type double. Only convertible int values, strings or enum variables are permitted

-arg should be always final(ie) we cant repeat case arg

switch(1) {

case 1:

SOP("1");

break;

case 2:

SOP("2");

break;

case 1: //error

SOP("3");

break

}

- default stmt can be present anywhere in switch case

- If we want to come out from switch case we have to use break stmt, dont ever use continue stmt inside switch case it leads compilation error

2. Looping statement - 3 types

a. for loop

for(initializtion;condition;inc/dec operator){

}

b. do while

do{

//stmt

}

while(condition);

c. while

while(condition);

3. Flow breaking statement

a. break - used to stop the entire iteration

b. continue - used to stop the current iteration

c. return - used to transfer the call back to calling prg

public class Main {

public static void main(String[] args) {

if(10==10) {

System.out.println("Hello");

}

if(true) {

System.out.println("Hello1");

}

int a=10;

if(a>10) {

System.out.println("Yes");

}

if(a>10) {

System.out.println("Yes");

}else {

System.out.println("No");

}

if(a==11 && a<15) {

System.out.println("yes");

}

if(a==10)

System.out.println("10");

System.out.println("15");

switch(10) {

case 10:

System.out.println("10");

//break;

default:

System.out.println("nothing");

//break;

case 15:

System.out.println("15");

break;

}

char c='b';

switch(c)

{

case 'a':

System.out.println("a");

break;

case 'b':

System.out.println("b");

break;

}

String c1="three";

switch(c1)

{

case "one":

System.out.println("a");

break;

case "two":

System.out.println("b");

break;

default:

System.out.println("No string");

break;

}

for(int i=0;i<10;i++) {

if(i%2==0)

//continue;

break;

System.out.println(i+" "); //1 3 5 7 9

}

}

}

**Array** - collection of similar datatypes

1. int a[]={1,2,3}; //correct - declaring and initializing values to an array and its size is fixed depending upon the number of elts

2. int a[3]={1,2,3}; //error - if we want to provide the size of the array then we have to compulsorly provide new operator

3. int a[]=new int[3]; //correct - declare an array of size 3, memory allocated is (4\*3)12 bytes

4. int[] a=new int[3]; //correct

5. int a[]=new int[]{1,2,3}; //correct - Anonmyous array where we can declare and initialize the array in same line

6. int a[]=new int[3]{1,2,3}; //error - In anonmyous array we should not specify its size

7. int a[]=new int[-3]; //correct - In Java we can declare an array with negative size but at runtime we will get an exception called NegativeArraySizeException

8. int a[]=new int[3]; //correct - declare an array of size 3

a[0]=1; a[1]=2; a[2]=3; //initializing an array

9. int a[]={1,2,3}; //declaring and initializing values to an array

sop(a); //reference

10. We want to access individual elements of the array - 2 ways

arrayname.length - used to find the length of the array

1. for loop

i - loopcounter variable

int a[]={1,2,3};

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

System.out.println(a[i]); //1 2 3

// when printed just a the reference gets printed

}

double d[]={1.2,3.4,5.6};

int i;

for(i=0;i<d.length;i++){

System.out.println(d[i]); //1.2 3.4 5.6

}

2. for each statement - avaialble from JDK1.5 onwards, used to access individual elements of array.

***Syntax -***

for(loopcounter variable: array){

}

- Loopcounter variable should be declared only inside foreach stmt, and it should be same datatype as an array

- 1D array should be stored in a variable, 2D array should be stored in a 1D array, 3D array should be stored in a 2D array and so on

1. int a[][]={{1,2},{3,4}}; //correct - declaring and initializing 2d array

2. int a[2][2]={{1,2},{3,4}}; //error

3. int a[][]=new int[3][3]; //correct

4. int[] a[]=new int[5][5]; //correct

5. int[][] a=new int[3][3]; //correct

6. int[] a[]=new int[5][5]; //correct - declare an array and totally allocated 100 bytes (25\*4)=100bytes

a[0][0]=1; a[0][1]=2; //initialized only 8bytes, remaining 92 bytes will be wasted

7. int[][] a=new int[5][] - Arrays of array - we have to declare only row bytes and based on requirement we can allocate columns

a[0]=new int[1]; a[0][0]=1;

a[1]=new int[2]; a[1][0]=2; a[1][1]=3;

a[4]=new int[2]; a[4][0]=2; a[4][1]=4;

Now it allocates only 20 bytes of memory

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

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

**double** a[]= {1.2,3.4,5.6,7.8};

**int** a1[]=**new** **int**[3];

//For loop

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

System.***out***.println(a[i]);

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

//For each stmt

**for**(**double** d:a)

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

**float** f1[][]= {{1.2f,3.4f},{4.5f,6.7f}};

**for**(**int** i=0;i<2;i++) {

**for**(**int** j=0;j<2;j++) {

System.***out***.println(f1[i][j]);

}

}

//for each stmt

**for**(**float** f2[]:f1)

**for**(**float** f3:f2)

System.***out***.println(f3);

}

}

***DAY 3***

***2 types of class***

1. Main class

- entry point of program, there will be always only 1 main class

- if a class contain public static void main(String[] args)

- used to create an object for base class and only print the output

2. Base class

We have to write all logic inside base class, there may be multiple base classes. Class contains variables, methods, constructors and be accessed by use of objects.

java prg (Sample.java)- compile(javac - convert source code to byte code and generate class file) - Sample.class (execute ur class file)

// base class

**class** Box{

**double** width,height,length; //instance variables

}

**public** **class** Baseclass{

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

Box b1= **new** Box(); //create an object 'b1' which contains memory address, invoke default constructor

Box b2;

b2 = **new** Box();

// Assigning values for Box instance 1

b1.width=10;

b1.height=20;

b1.length=30;

Double vol = b1.width\*b1.height\*b1.length; //vol is a local variable

System.***out***.println("Volume of Box 1 = "+vol);

// Assigning values for Box instance 2

b2.width=21;

b2.height=2;

b2.length=39;

vol = b2.width\*b2.height\*b2.length;

System.***out***.println("Volume of Box 2 = "+vol);

}

}

Method - used to write any logic

Syntax: returntype methodname(arg list){

//logic }

// using method for logic logic

**class** Box{

**double** width,height,length; //instance variables

**void** volume(){

System.***out***.println(width\*height\*length);

}

}

**public** **class** Baseclass{

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

Box b1= **new** Box();

b1.width=10;

b1.height=20;

b1.length=30;

b1.volume();

}

}

// using method for logic - DEFAULT

**class** Box{

**double** width,height,length; //instance variables

**double** volume(){

**return**(width\*height\*length);

}

}

**public** **class** Baseclass{

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

Box b1= **new** Box();

b1.width=10;

b1.height=20;

b1.length=30;

**double** vol = b1.volume(); //vol is a local variable

System.out.println(vol);

}

}

Parameterized methods - method that takes an argument

//using method for logic PARAMETTERIZED method

**class** Box{

**double** width,height,length; //instance variables

**double** volume(){

**return**(width\*height\*length);

}

**void** setDim(**double** w,**double** h,**double** l) {

width = w;

height = h;

length = l;

}

}

**public** **class** Baseclass{

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

Box b1= **new** Box();

b1.setDim(2,32,1);

**double** vol = b1.volume(); //vol is a local variable

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

} }

***2 types of methods***

1. Accessor method - used to return the value - getter method - getSize(), getAge(), getName()

2. Mutator method - used to set the value - setter method - setSize(), setAge(), setName()

when there is no parameterised constructor then to set values for variables it is being used getters and setters.

***Constructor***

- What is constructor?

- Whenever we create an object for a class, memory will be allocated so the allocation of memory is called constructor

- When constructor will be invoked?

- Whenever we create an object for a class, the constructor will be invoked

- Why constructor?

- used to initialize value to variables at time of object creation

- By default all class will contain one default constructor(ie) constructor without any argument, and we can create our own contructor where ur constructor name and class name should be same in that case it will destroy the already present default constructor

- Constructor can be created with or without arguments

- Constructor can have access specifiers

- Constructor should not have return type

//// using DEFAULT CONSTRUCTOR

**class** Box{

**double** width,height,length; //instance variables

**double** volume(){

**return**(width\*height\*length);

}

Box(){ //default constructor

width=-1;

height=-1;

length=-8;

}

}

**public** **class** Baseclass {

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

Box b1=**new** Box(); //create an object 'b1' which contains memory address, invoke default constructor

**double** vol; //local variable

vol=b1.volume();

System.***out***.println("Volume of b1 is "+vol);

}}

**Parameterized constructor**

// use of PARAMETERIZED CONSTRUCTOR

**class** Box{

**double** width,height,length; //instance variables

**double** volume() {

**return**(width\*height\*length);

}

Box(**double** w,**double** h,**double** l){

width = w;

height = h;

length = l;

}

}

**public** **class** Baseclass {

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

Box b1= **new** Box(2,21,21);

**double** vol = b1.volume();

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

}

}

Another way to initialize values without parameterized constructor using getter and setter methods. (IN ECLIPSE - Right click on the program - Source - Generate Getter and setter)

// using getters and setters

**class** Box{

**double** width,height,length;

**double** volume() {

**return**(width\*height\*length);

}

**public** **double** getwidth() {

**return** width;

}

**void** setwidth(**double** width) {

**this**.width=width;

}

**public** **double** getheigth() {

**return** height;

}

**void** setheigth(**double** height) {

height=height;

}

**public** **double** length(){

**return** length;

}

**void** setlength(**double** length) {

**this**.length=length;

}

}

**public** **class** Baseclass{

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

Box b1=**new** Box();

**double** vol;

b1.setwidth(10);

b1.setheigth(20);

b1.setlength(30);

vol = b1.volume();

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

}

}

**this keyword**

- used to refer the current class variable

- it is used in the case if ur instance variable name and parameter name in constructor or methods are same

// PARAMETERIZED CONSTRUCTOR with this keyword

**class** Box{

**double** width,height,depth;

**double** volume() {

**return**(width\*height\*depth);

}

**public** Box(**double** width, **double** height, **double** depth) {

**super**();

**this**.width = width;

**this**.height = height;

**this**.depth = depth;

}

}

**public** **class** Baseclass {

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

Box b1=**new** Box(10,20,30); //create an object 'b1' which contains memory address, invoke default constructor

Box b2=**new** Box(1,2,3);

**double** vol; //local variable

vol=b1.volume();

System.***out***.println("Volume of b1 is "+vol);

vol=b2.volume();

System.***out***.println("Volume of b2 is "+vol);

}

}

**this constructor**

- used to invoke different constructor of same class, it should be always present in first line other compilation error

**class** A{

A(){

**this**("hello");

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

}

A(**int** i){

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

}

A(String s){

**this**(10);

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

} }

**public** **class** Baseclass{

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

A a1=**new** A(); //231 }

}

class A {

A(){

this(4);

System.out.println("1");

}

A(int a){

System.out.println("2");

}

A(String s){

this();

System.out.println("3");

} }

public class Main {

public static void main(String[] args) {

A a1=new A("hi"); //213

}

}

***Polymorphism*** - one class takes many forms - 2types

1. Static/compile time

2. Dynamic/Runtime

***Static/Compile time polymorphism***

- using overloading

- 2 types - Method overloading, Constructor overloading

in java operator overloading is not possible.

**Method overloading**

- same method name but different number, order and datatype of the argument, different return type and present in same class

- No need of returntype for overloading.

**class** OverloadDemo {

**void** test() {

System.***out***.println("No parameters");

}

**void** test(**int** a) {

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

}

**void** test(**int** a,**int** b) {

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

}

**double** test(**double** a) {

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

**return** a\*a;

}

}

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

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

OverloadDemo ob=**new** OverloadDemo();

ob.test();

ob.test(10);

ob.test(10,20);

**double** result=ob.test(123.4);

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

}

}

**2. Constructor overloading**

- same constructor name, but different number, order and datatype of the argument present in the same class

**class** Box{

**double** width, height, depth;

**double** volume() {

**return** width\*height\*depth;

}

//Constructor overloading

Box(){

width=-1;

height=-1;

depth=-1; }

Box(**double** width,**double** height,**double** depth){

**this**.width=width;

**this**.height=height;

**this**.depth=depth;

}

Box(**double** len){

width=height=depth=len;

} }

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

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

Box b1=**new** Box();

Box b2=**new** Box(10,20,30);

Box b3=**new** Box(7);

**double** vol;

vol=b1.volume();

System.***out***.println("Volume of b1 is "+vol);

vol=b2.volume();

System.***out***.println("Volume of b2 is "+vol);

vol=b3.volume();

System.***out***.println("Volume of b3 is "+vol);

}

}

Passing object as an argument

**class** Box{

**double** width;

**double** height;

**double** depth;

**double** volume() {

**return** width\*height\*depth;

}

//Constructor overloading

Box(){

width=-1;

height=-1;

depth=-1;

}

Box(**double** width,**double** height,**double** depth){

**this**.width=width;

**this**.height=height;

**this**.depth=depth;

}

Box(**double** len){

width=height=depth=len;

}

Box(Box ob){

width=ob.width;

height=ob.height;

depth=ob.depth;

}

}

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

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

Box b1=**new** Box();

Box b2=**new** Box(10,20,30);

Box b3=**new** Box(7); //pass by value

Box b4=**new** Box(b2); //pass by reference

**double** vol;

vol=b1.volume();

System.***out***.println("Volume of b1 is "+vol);

vol=b2.volume();

System.***out***.println("Volume of b2 is "+vol);

vol=b3.volume();

System.***out***.println("Volume of b3 is "+vol);

vol=b4.volume();

System.***out***.println("Volume of b4 is "+vol);

}

}

class Example {

void add(int a,int b) {

SOP(a+b);

}

void add(int a,int b,int c) {

SOP(a+b+c);

}

void add(int a,int b,int c,int d) {

SOP(a+b+c+d);

}

void add(int a,int b,int c,int d,int e) {

SOP(a+b+c+d+e);

}

}

**var args**

- used to define variable number of arguments

- available only from JDK1.5

- Using ...

- If we are passing combination of var arg and normal parameters, then ur var args should be always present as last argument

- One method can have only one var args and that also should be present as last argument

void add(int...a){ //can take 0 or any number of int argument

}

void add(boolean...b){

}

class Demo{

void add(int...a) {

for(int a1:a)

System.out.println(a1);

}

void add(String...s) {

for(String s1:s)

System.out.println(s1);

}

void add(int a,boolean...b) {

System.out.println(a);

for(boolean b1:b)

System.out.println(b1);

}

}

public class Main {

public static void main(String...args) {

Demo d=new Demo();

//d.add(); //nothing will be printed

d.add(1,2,3,4,5,6);

d.add(2,4);

d.add("hello","hi","wel");

d.add(10,true,false,true);

}

}

***Scanner class***

- used to read input from the user based on particular datatype

- present in java.util.\*

- If u use combination of numbers and string either u can define dummy line sc.nextLine() or we can get all input as string (ie) using nextLine(), and based on requirement we convert that to related datatype

Methods

1. int nextInt()

2. float nextFloat()

3. double nextDouble()

4. long nextLong()

5. boolean nextBoolean()

6. short nextShort()

7. String next() - read single word without space

8. String nextLine() - read words with space

public class Main {

public static void main(String...args) {

Scanner sc=new Scanner(System.in);

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

String name=sc.nextLine();

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

int age=sc.nextInt();

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

double sal=sc.nextDouble();

System.out.println(name+" "+age+" "+sal);

} }

public class Main {

public static void main(String...args) {

Scanner sc=new Scanner(System.in);

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

int age=Integer.parseInt(sc.nextLine());

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

String name=sc.nextLine();

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

double sal=Double.parseDouble(sc.nextLine());

//sc.nextLine();

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

String address=sc.nextLine();

System.out.println(name+" "+age+" "+sal);

} }

class Hexy {

public static void main (String[] args) {

Integer i = 42;

String s = (i<40)?"life":( i>50)?"universe":"everything";

System.out.println(s);

}

}

***DAY 4***

**Access specifiers** - public, private, protected, default - can be applied for all (i.e.) class, variable, interface, method.

Private – can’t be accessed outside the class. To set values for particular variable.

**Non Access specifiers** - it has its own restrictions (i.e.) cannot be applied for all.

**Static keyword**

- It is a non-access specifier

- Whenever a class is declared as static, there is no need to create an object for that class, outer class cannot be declared as static only inner class can be declared as static

class A //outer class

class B {

//inner class } }

- Whenever a method is declared as static, if it is present in same class then we can access using "methodname", if it is present in different class then we can access using "classname.methodname"

- Whenever a variable is declared as static it acts as global variable, if it is present in same class then we can access using "variablename", if it is present in different class then we can access using "classname.variablename"

- static method can access only static content, if it is not static then we have to create an object and access it

class Sample {

int a;

public static void main(String[] r){

System.out.println(a); //compilation error, static method can access only static content

}

}

class Sample {

static int a;

boolean b;

public static void main(String[] r){

System.out.println(a); //0

Sample s=new Sample();

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

}

}

- When we want to execute any task before main(), then we can go for static block which will be executed before main()

Syntax: static {

//logic }

- static method can override only another static method

public class Main {

static int a=4;

static int b;

static void method(int x) {

System.out.println(x); //30

System.out.println(a); //4

System.out.println(b); //16

}

static {

System.out.println("Static block Initialized");

b=a\*4;

}

public static void main(String[] args) {

method(30);

} }

class Base{

static int a=10;

static int b=20;

static void show() {

System.out.println(b);

}

}

public class Main {

public static void main(String[] args) {

Base.show(); //20

System.out.println(Base.a); //10

}

}

**System.out.println()/System.in**

* System is a predefined class
* out is an object of PrintStream class and it is a static variable in System class
* println() is a instance method of PrintStream class

public class System {

static PrintStream out;

static InputStream in;

}

static import

- Available only from JDK1.5

- Normally we can invoke static method or static variable using classname.methodname or classname.variablename, but if we want to invoke static method and static variable without classname.methodname or classname.variable we can use static import

import static java.lang.Math.sqrt;

import static java.lang.System.out;

public class Main {

public static void main(String[] args) {

double d=sqrt(16);

out.println(d);

}

}

Inheritance

- accessing the properties of one class into another class, if it is not present

- code reusability

- using "extends" keyword

- Single, Multilevel, Hybrid, Hiearchial

class Parent { //base class/super class/ parent class

}

class Child extends Parent { //derived/subclass/child class

}

//Base class

**class** Box{

**double** width, height, depth;

**double** volume() {

**return** width\*height\*depth;

}

//Constructor overloading

Box(){

width=-1;

height=-1;

depth=-1;

}

Box(**double** width,**double** height,**double** depth){

**this**.width=width;

**this**.height=height;

**this**.depth=depth;

}

Box(**double** len){

width=height=depth=len;

}

Box(Box ob){

width=ob.width;

height=ob.height;

depth=ob.depth;

}

}

// derived class

**class** BoxWeight **extends** Box{

**double** weight;

BoxWeight(){

width = -1;

height = -1;

depth = -1;

}

BoxWeight(**double** w, **double** h,**double** d, **double** w1){

width = w;

height = h;

depth = d;

weight = w1;

}

BoxWeight(**double** len, **double** w){

width = height = depth = len;

weight = w;

}

BoxWeight(BoxWeight ob){

width=ob.width;

height=ob.height;

depth=ob.depth;

weight = ob.weight;

}

}

**public** **class** Inheritance {

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

BoxWeight b1=**new** BoxWeight();

**double** vol;

vol = b1.volume();

System.***out***.println("Volume of b1 is "+vol);

System.***out***.println("Weight is "+b1.weight);

BoxWeight b2=**new** BoxWeight(10,20,30,40);

vol = b2.volume();

System.***out***.println("Volume of b2 is "+vol);

System.***out***.println("Weight is "+b2.weight);

BoxWeight b3=**new** BoxWeight(10,20);

vol = b3.volume();

System.***out***.println("Volume of b3 is "+vol);

System.***out***.println("Weight is "+b3.weight);

BoxWeight b4=**new** BoxWeight(b2);

vol = b4.volume();

System.***out***.println("Volume of b4 is "+vol);

System.***out***.println("Weight is "+b4.weight);

}

}

super keyword

- used to access base class constructor, base variables and base methods

- Only in the case accessing **base class constructor**, it should always present in the **first line**

- super and this cant used simultaneously

- if package is diff then also base class can be accessed using super

//Base class

class Box {

private double width, height, depth;

double volume() {

return width\*height\*depth;

}

Box(){

width=-1;

height=-1;

depth=-1;

}

Box(double width,double depth,double height){

this.width=width;

this.depth=depth;

this.height=height;

}

Box(double len){

width=depth=height=len;

}

Box(Box ob){

width=ob.width;

depth=ob.depth;

height=ob.height;

}

}

//Derived class

class BoxWeight extends Box {

double weight;

BoxWeight(){

super();

weight=-1;

}

BoxWeight(double w,double d,double h,double w1){

super(w,h,d);

weight=w1;

}

BoxWeight(double len,double w){

super(len);

weight=w;

}

BoxWeight(BoxWeight ob){

super(ob);

weight=ob.weight;

}

}

public class Main {

public static void main(String[] args) {

BoxWeight b1=new BoxWeight();

double vol;

vol=b1.volume();

System.out.println("Volume is "+vol);

System.out.println("Weight is "+b1.weight);

BoxWeight b2=new BoxWeight(10,20,30,40);

vol=b2.volume();

System.out.println("Volume is "+vol);

System.out.println("Weight is "+b2.weight);

BoxWeight b3=new BoxWeight(10,20);

vol=b3.volume();

System.out.println("Volume is "+vol);

System.out.println("Weight is "+b3.weight);

BoxWeight b4=new BoxWeight(b2);

vol=b4.volume();

System.out.println("Volume is "+vol);

System.out.println("Weight is "+b4.weight);

}

}

Multilevel inheritance

//Base class

class Box {

private double width;

private double height;

private double depth;

double volume() {

return width\*height\*depth;

}

Box(){

width=-1;

height=-1;

depth=-1;

}

Box(double width,double depth,double height){

this.width=width;

this.depth=depth;

this.height=height;

}

Box(double len){

width=depth=height=len;

}

Box(Box ob){

width=ob.width;

depth=ob.depth;

height=ob.height;

}

}

//Derived class

class BoxWeight extends Box {

double weight;

BoxWeight(){

super();

weight=-1;

}

BoxWeight(double w,double d,double h,double w1){

super(w,h,d);

weight=w1;

}

BoxWeight(double len,double w){

super(len);

weight=w;

}

BoxWeight(BoxWeight ob){

super(ob);

weight=ob.weight;

}

}

class Shipment extends BoxWeight {

String address;

Shipment(double w,double d,double h,double w1,String address){

super(w,d,h,w1);

this.address=address;

}

}

public class Main {

public static void main(String[] args) {

Shipment s1=new Shipment(10,20,30,40,"Chennai");

double vol;

vol=s1.volume();

System.out.println("Volume is "+vol);

System.out.println("Weight is "+s1.weight);

System.out.println("Address is "+s1.address);

}

}

access same var name by using super keyword

class A {

int i;

}

class B extends A {

int i;

B(int a,int b){

i=a;

super.i=b;

}

void show() {

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

}

}

public class Main {

public static void main(String[] args) {

B b=new B(1,2);

b.show();

// System.out.println(b.i); // main being static method need to access using obj…. only access b class

// System.out.println(b.j); // if var name is diff. then can be accessed from child otherwise create obj of parent class

}

}

class A {

int i;

int j;

A(int a,int b){

i=a;

j=b;

}

void show() {

System.out.println("i="+i+" j="+j);

}

}

class B extends A {

int k;

B(int a,int b,int c){

super(a,b);

k=c;

}

void show() {

super.show();

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

}

}

public class Main {

public static void main(String[] args) {

B b=new B(1,2,3); //i=1 j=2 k=3

b.show();

}

}

**Method Overriding** - same method name, same number, order and datatype of argument, same return type and present in different class and it should be inherited.

No Constructor overriding – as to override it has to be present in diff. class but Constructor should have same name as the class name so no constructor overriding is present.

**Dynamic Method Dispatch**

- In order to have effective method overriding, we go for Dynamic Method dispatch

- We always create an object for base class but we store the reference of derived class but not vice versa, so the compiler thinks that the object is created for base class but only at runtime it comes to know that it contains the reference of derived class, in this way we can achieve runtime polymorphism

class Sphere {

void volume() {

System.out.println("Sphere volume");

}

}

class Hemisphere extends Sphere {

void volume() {

System.out.println("Hemisphere volume");

}

}

public class Main {

public static void main(String[] args) {

/\* Hemisphere h=new Hemisphere();

h.volume();

Sphere s=new Sphere();

s.volume(); \*/

Sphere s=new Hemisphere(); //DMD

s.volume(); //Hemisphere volume

s=new Sphere(); //Hemisphere reference is garbage collected and s contain Sphere reference

s.volume(); //Sphere volume

}

}

**//Hierachial**

class A {

void callback() {

System.out.println("A's callback");

}

}

class B extends A {

void callback() {

System.out.println("B's callback");

}

}

class C extends A {

void callback() {

System.out.println("C's callback");

}

}

public class Main {

public static void main(String[] args) {

A a=new A();

a.callback();

a=new B(); //DMD

a.callback();

a=new C();

a.callback();

}

}

**How constructor is invoked in inheritance?**

**- In top down approach**

**//Multilevel**

class A {

A(){

System.out.println("1");

}

}

class B extends A {

B(){

System.out.println("2");

}

}

class C extends B {

C(){

System.out.println("3");

}

}

public class Main {

public static void main(String[] args) {

C c=new C(); //123

}

}

**//Multilevel**

class A {

A(int a){

System.out.println("1");

}

}

class B extends A {

B(){

System.out.println("2");

}

}

class C extends B {

C(){

System.out.println("3");

}

}

public class Main {

public static void main(String[] args) {

C c=new C(); //Compilation error

}

}

First it will go to the related class constructor (ie) C class constructor and check whether we have use "this()" or "super()" then it works according to that, if it is not it will by default try to invoke default constructor of base class

**//Multilevel**

class A {

A(int a){

System.out.println("1");

}

/\*A() {

}\*/

}

class B extends A {

B(){

super(10);

System.out.println("2");

}

}

class C extends B {

C(){

System.out.println("3");

}

}

public class Main {

public static void main(String[] args) {

C c=new C(); //Compilation error

}

}

**final keyword**

- It is a non access specifier

- When we declare a class to be final, inheritance is not possible

final class A {

A(){

System.out.println("1");

}

}

class B extends A { //error

B(){

System.out.println("2");

}

}

- when a method is declared as final, overriding is not possible

class A {

final void show() {

}

}

class B extends A {

void show() { //error

}

}

- When a variable is declared it will acts like constant

Relationship

1. is-a relationship = when we perform inheritance

class Animal{

}

class Dog extends Animal {

}

Dog is a Animal

2. has-a relationship - Aggregation

- When we try to create object of one class into another class

class Wheel {

}

class Car {

Wheel w=new Wheel(); //Car has a Wheel

}

***DAY 5***

Covariant return type

- Before JDK1.5 it is not possible to override any method by changing its return type

- From JDK1.5, it is possible to override method by changing its return type (ie) only non primitive

- Avoid type casting , wrong typecasting leads to ClassCastException at runtime

class A {

A show(){

return this;

}

void print() {

System.out.println("Inside A's method");

}

}

class B extends A {

A show(){

return this;

}

void print() {

System.out.println("Inside B's method");

}

}

class C extends B {

A show(){

return this;

}

void print() {

System.out.println("Inside C's method");

}

}

class Main {

PSVM {

A a1=new A();

a1.show().print(); //Inside A's method

B b1=new B();

((B)b.show()).print(); //Inside B's method

b.show().print(); //Inside B's method

C c1=new C();

((C)c1.show()).print();

}

}

Compound block/Instance block

- It will be invoked before ur constructor invokes

Syntax: { //logic }

In case of inheritance, first static block will be invoked from top to bottom

**class** Baseclass {

**static** {

System.***out***.println("1 - static in base c");

}

Baseclass(){

System.***out***.println("2 - default constructor in base c");

}

Baseclass(**int** a){

System.***out***.println("3 - parameterized constructor");

}

Baseclass(**float** b){

**this**();

System.***out***.println("4 - param constructor float");

}

{

System.***out***.println("5 - compound block");

}

{

System.***out***.println("6 - compound block");

}

}

**class** Compoundblockk **extends** Baseclass{

**static** {

System.***out***.println("10 - static in derived c");

}

Compoundblockk(){

System.***out***.println("11 - default constructor in base c");

}

{

System.***out***.println("12 - compound block");

}

}

**public** **class** Compoundblock{

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

Compoundblockk cb = **new** Compoundblockk();

{

System.***out***.println("7 - call to derived c");

{

System.***out***.println("8 - compound block");

}

}

Baseclass b = **new** Baseclass(2f);

{

System.***out***.println("13 - call to base c");

}

}

**static** {

System.***out***.println("9 - Static in main");

} }

***abstract keyword***

- It is a non-access specifier

- When a class is declared as abstract, we can’t create an object for that class

- When a method is declared as abstract, it does not contain any definition it just ends with semicolon

- Variable cannot be declared as abstract

- When a class contain abstract method then that class should be declared as abstract otherwise error, but not necessary all abstract class should contain abstract methods

- Abstract class can also contain non abstract methods

- Abstract class can also be inherited, so when u inherit any abstract class then compulsorily we have to provide the implementation of abstract method in the inherited class or if we not defining the definition then inherited class itself should be declared as abstract

- Abstract class contains constructor but we can’t invoke it

abstract class A {

abstract void add();

void show() {

System.out.println("Concrete implementation");

}

}

/\*abstract\*/ class B extends A {

void add() {

System.out.println("Abstract method");

}

}

public class Main {

public static void main(String[] args) {

B b=new B();

b.add();

b.show();

}

}

**Anonymous Inner class**

- used to access normal methods / predefined normal methods of abstract class without inheritance.

**abstract** **class** A {

**abstract** **void** add();

**void** show() {

System.***out***.println("Concrete implementation");

}

}

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

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

//Anonmyous Inner class

A a=**new** A() {

**public** **void** show() //but can get warning as A must implement the inherited

abstract method A.add()

{

System.***out***.println("Hello world");

}

**public** **void** add() {

System.***out***.println("Abstract method");

}

};

a.show(); //Hello world

a.add();

}

}

***Interface***

- No multiple inheritance instead we use interface

class A extends B,C,D { //error

}

- Interface are syntactically similar to class but contains method declaration and variable declaration and initialization

**Syntax**: <<accessspecifier>> interface interfacename {

//method declaration

//variable declaration and initialization

}

- By default all interface are abstract, so we can't create an object for interface

- By default all methods in interface are public and abstract, so interface method doesn’t contain any definition, it just ends with semicolon

- By default all variables in interface are public, static and final, so we can access interface variable using "interfacename.variablename"

- Using "implements" keyword we can use interface in class

- When a class implements an interface, then compulsorily we have to provide the definition of all methods declared in the interface with public access specifier or define the class itself to be abstract

- The implemented class can also contain non interface method. Interface can also be inherited

1 class extends 1 class

1 class implements n interface

1 interface extends n interface

- Marker interface - interface which doesn’t contain anything like Cloneable, Serializable, Remote interface

interface A {

}

- From Java8 onwards, interface will also contain default and static methods also

**interface** Arithmetic {

**void** process(**int** a,**int** b);

**int** ***sum***=10;

}

/\*abstract\*/ **class** A **implements** Arthimetic {

**public** **void** process(**int** a,**int** b) {

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

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

}

}

**class** B **implements** Arthimetic {

**public** **void** process(**int** a,**int** b) {

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

}

}

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

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

/\*A a=new A();

a.process(3, 4);

B b=new B();

b.process(5, 3);\*/

Arthimetic a=**new** A(); //DMD

a.process(3, 4);

a=**new** B();

a.process(4, 2);

}

}

**interface** A {

**void** meth1();

}

**interface** B {

**void** meth2();

}

**interface** C **extends** A,B{

**void** meth3();

}

**class** Sample **implements** C {

@Override

**public** **void** meth1() {

System.***out***.println("Inside Meth1");

}

@Override

**public** **void** meth2() {

System.***out***.println("Inside Meth2");

}

@Override

**public** **void** meth3() {

System.***out***.println("Inside Meth3");

}

}

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

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

Sample s=**new** Sample();

s.meth1();

s.meth2();

s.meth3();

}

}

Nested class - one class inside another class

//Nested class

class A {

class B {

}

}

Inside class can be either static or non static, non static inner class is called inner class and static inner class is called static class

class A {

class B { //Inner class

}

}

class A {

static class B { //static class

}

}

- we can access properties of outer class inside inner class only if it is instance variable or declared as final

- We cant access properties of inner class in outer class

class Outer {

int outer = 100;

void test() {

Inner i=new Inner();

i.display();

}

class Inner {

int y=10;

void display() {

System.out.println(y+" "+outer);

}

}

}

public class Main {

public static void main(String[] args) {

Outer o=new Outer();

o.test();

}

}

class Outer {

int outer = 100;

class Inner {

int y=10;

void display() {

System.out.println(y+" "+outer);

}

}

}

public class Main {

public static void main(String[] args) {

Outer.Inner o=new Outer().new Inner();

o.display();

}

}

class Outer {

static int outer = 100;

static class Inner {

int y=10;

void display() {

System.out.println(y+" "+outer);

}

}

}

public class Main {

public static void main(String[] args) {

Outer.Inner o=new Outer.Inner();

o.display();

}

}

Predefined package

1. java.lang.\* - Language package - only one optional pkg

1. Object class

- super class of all classes (ie) any predefined or userdefined class by default will inherit Object class

void add(int a) - int as argu

void show(String s)- String as arg

void show(Box b) - object of only Box class as arg

void show(Test t) - object of only Test class as arg

void show(Object o) - object of any class as arg

Constructor:

1. Object()

Method

1. String toString() - String representation of an object

String s=new String("Hello"); //all predefined class by default contains toString, so if we print object of any predefined class it automatically invokes toString() and print the content

SOP(s); //Hello

Sample s1=new Sample("Hello"); //userdefined

SOP(s1); //memory reference

So whenever we print any object it will print only memory reference. But we dont want to print memory reference instead we want to print some content then in that case we have to override toString()

// toString()

**class** Tostring{

String s1="";

Tostring(String s){

s1 =s;

}

**public** String toString() {

**return** s1;

}

public class Main {

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

Tostring ts = **new** Tostring("Hello ");

System.***out***.println(ts); //when used without toString() returns memory ref. //System.out.println(a); //memory reference

System.out.println(a.toString()); //Hello

} }

2. boolean equals(Object o) - to check equality of content

3. == (equals versus) - to check equality of object reference, by using new operator. If we are not using new operator then == also works like equals()

**public class Main {**

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

**String s1=new String("Hello"); //class**

**String s2=new String("Hello"); //class**

**System.out.println(s1.equals(s2)); //true**

**System.out.println(s1==s2); //false**

**String s3=s2;**

**System.out.println(s2.equals(s3)); //true**

**System.out.println(s2==s3); //true**

**String s4="hello"; //literal**

**String s5="hello"; //literal**

**System.out.println(s4.equals(s5)); //true**

**System.out.println(s4==s5); //true**

**}**

**}**

4. int hashCode()

- When we store some value into an object, internally an address will be generated for that value, to return that address we use hashCode()

- If two object are equal according to equals(), then hashCode will return same result

**public class Main {**

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

**String s1=new String("Hello");**

**String s2=new String("Hello");**

**System.out.println(s1.hashCode());**

**System.out.println(s2.hashCode());**

**}**

**}**

5. final void wait()

6. final void wait(long msec)

7. final void notify()

8. final void notifyAll()

9. protected void finalize()

- used to perform cleanup activity before destroying the object. called by default for every object before its deletion and only once in lifecycle of the program

Different ways

1. By anonymous object - are thos object which are created without any reference variable

new Sample();

2. By nulling reference

Sample s=new Sample();

s=null;

3. By assigning reference to another object

Student s1=new Student(); //1000

Student s2=new Student(); //1001

s1=s2;

To explicitly call garbage collector using System.gc(), that invokes garbage collector which destroy unused object, that internally call finalize() only once for each object

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

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

Main m=**new** Main();

m=**null**;

System.*gc*();

}

**protected** **void** finalize() {

System.***out***.println("Finalize called");

}

}

public class Main {

public static void main(String[] args) {

String s1=new String("Hello");

s1=null;

System.gc();

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

}

protected void finalize() {

System.out.println("Finalize called");

}

}

public class Main {

public static void main(String[] args) {

Main s1=new Main();

s1=null;

System.gc();

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

}

protected void finalize() {

System.out.println("Finalize called");

}

}

finalize() is called only once for a object

public class Main {

public static void main(String[] args) {

Main m1=new Main();

m1=null;

System.gc();

System.gc();

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

}

protected void finalize() {

System.out.println("Finalize called");

}

}

public class Main {

public static void main(String[] args) {

Main m1=new Main();

Main m2=new Main();

m1=m2;

m1.finalize();

System.gc();

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

}

protected void finalize() {

System.out.println("Finalize called");

}

}

10. Object clone()

- used to create the exact copy of the object

- clone() always follow shallow copy

- If we want to take copy of object using clone() then that class should implement an interface called Cloneable which is a marker interface, in case if it is not implemented Cloneable interface it will throw CloneNotSupportedException

Shallow copy

- will have exact copy of all fields of original object (ie) any changes made to thos objects through clone object will be reflected in original object and vice versa

class Address {

String street;

String country;

String city;

public Address(String street, String country, String city) {

super();

this.street = street;

this.country = country;

this.city = city;

}

}

class Employee implements Cloneable{

int id;

String name;

Address address;

public Employee(int id, String name, Address address) {

super();

this.id = id;

this.name = name;

this.address = address;

}

protected Object clone() throws CloneNotSupportedException{

return super.clone();

}

}

public class Main {

public static void main(String[] args) {

Address addr=new Address("xyz","India","ABC");

Employee e1=new Employee(100,"Ram",addr);

Employee e2=null;

try {

e2=(Employee)e1.clone();

}

catch(CloneNotSupportedException e) {

e.printStackTrace();

}

System.out.println(e1.address.country); //India

e2.address.country="Russia";

System.out.println(e1.address.country); //Russia

} }

Deep copy

- will have exact copy of all fields of original object, and any changes made to clone object will not be reflected in original object and vice versa

class Address implements Cloneable {

String street;

String country;

String city;

public Address(String street, String country, String city) {

super();

this.street = street;

this.country = country;

this.city = city;

}

protected Object clone() throws CloneNotSupportedException{

return super.clone();

}

}

class Employee implements Cloneable{

int id;

String name;

Address address;

public Employee(int id, String name, Address address) {

super();

this.id = id;

this.name = name;

this.address = address;

}

protected Object clone() throws CloneNotSupportedException{

Employee e=(Employee)super.clone();

e.address=(Address)address.clone();

return e;

}

}

public class Main {

public static void main(String[] args) {

Address addr=new Address("xyz","India","ABC");

Employee e1=new Employee(100,"Ram",addr);

Employee e2=null;

try {

e2=(Employee)e1.clone();

}

catch(CloneNotSupportedException e) {

e.printStackTrace();

}

System.out.println(e1.address.country); //India

e2.address.country="Russia";

System.out.println(e1.address.country); //India

} }

2. Wrapper class

- classes that supports the primitive datatype, to perform any operation on the datatype

- All wrapper classes are immutable(u cant change)

int a=10;

Datatype Wrapper class

int - Integer

float - Float

double - Double

short - Short

byte - Byte

long - Long

boolean - Boolean

char - Character

1. Float class

- used to perform operation on float datatype

Syntax: public class Float extends Number implements Comparable

Constructor

1. Float(double)

Float f1=new Float(3.14);

2. Float(float)

Float f2=new Float(3.14f);

3. Float(String) throws NumberFormatException

Float f3=new Float("3.14");

Float f4=new Float("abc"); //throws NFE

**Methods -** int a=5, b=5; a==b

float a1=3.14, b1=3.14;

1. static int compare(float f1,float f2) - compare two float datatype, if equal it returns 0, if greater 1, if lesser means -1

2. int compareTo(Float f) - compare 2 Float object, if equal it returns 0, if greater 1, if lesser means -1

3. static boolean isInfinite() - check float datatype to be infinite or not

4. boolean isInfinite() - check Float object

5. static boolean isFinite() - check float datatypes to be finite or not

6. boolean isFinite() - check Float object

7. static boolean isNaN() - check float datatype as not a number

8. boolean isNaN() - check Float object

9. static float parseFloat(String s) throws NumberFormatException - used convert String to float datatype

10. static Float valueOf(String s) throws NFE - used to convert String to Float wrapper class

11. float floatValue() - used to convert Float wrapper class to float datatype

12. int intValue()

13. double doubleValue()

14. long longValue()

15. short shortValue()

16. byte byteValue()

**Contants**

1. Float.MAX\_VALUE

2. Float.MIN\_VALUE

3. Float.POSITIVE\_INFINITY

4. Float.NEGATIVE\_INFINITY

5. Float.TYPE

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

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

**float** f1=3.14f;

**float** f2=3.14f;

System.***out***.println(Float.*compare*(f1, f2)); //0

Float f3=**new** ~~Float~~(3.14f);

Float f4=**new** ~~Float~~(3.14f);

System.***out***.println(f3.compareTo(f4)); //0

**float** f5=(**float**)(1/0.);

System.***out***.println(Float.*isInfinite*(f5)); //true

Float f6=**new** ~~Float~~(1/0.);

System.***out***.println(f6.isInfinite()); //true

**float** f7=(**float**)Math.*sqrt*(-4);

System.***out***.println(Float.*isNaN*(f7)); //true

Float f8=**new** ~~Float~~(Math.*sqrt*(-4));

System.***out***.println(f8.isNaN()); //true

String s="3.14";

**float** f9=Float.*parseFloat*(s);

System.***out***.println(f9); //3.14

Float f10=Float.*valueOf*(s);

System.***out***.println(f10); //3.14

**float** f11=f10.floatValue();

System.***out***.println(f11); //3.14

**int** i=f10.intValue();

System.***out***.println(i); //3

System.***out***.println(Float.***MAX\_VALUE***);

System.***out***.println(Float.***MIN\_VALUE***);

System.***out***.println(Float.***POSITIVE\_INFINITY***);

System.***out***.println(Float.***NEGATIVE\_INFINITY***);

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

}

}

**2. Double class**

- used to perform operation on double datatype

**Constructor**

1. Double(double)

2. Double(String) throws NFE

3. Integer, Short, Byte, Long wrapper class

- used to perform operation on int, short, byte, long datatype respectively

**Constructor**

1. Integer(int i)

- Integer i=new Integer(30);

Integer(String s) throws NFE

- Integer i=new Integer("30");

2. Short(short s)

- Short s1=new Short(23); //error

- Short s1=new Short((short)23); //correct

Short(String s) throws NFE

- Short s2=new Short("23");

3. Byte(byte b)

- Byte b1=new Byte(20); //error

- Byte b2=new Byte((byte)20); //correct

Byte(String s) throws NFE

- Byte b3=new Byte("20");

4. Long(long g)

- Long l1=new Long(10);

- Long l2=new Long(10l);

Long(String s) throws NFE

- Long l3=new Long("10");

**Methods**

1. static int parseInt(String d)

2. static int parseInt(String s, int radix)

public class Main {

public static void main(String[] args) {

int i1=Integer.parseInt("42");

System.out.println(i1); //42

int i2=Integer.parseInt("42", 5);

//2\*5^0=2

//4\*5^1=20

System.out.println(i2); //22

System.out.println(Integer.toBinaryString(2)); //10

System.out.println(Integer.toOctalString(9)); //11

System.out.println(Integer.toHexString(10)); //a

System.out.println(Integer.MAX\_VALUE);

System.out.println(Integer.MIN\_VALUE);

System.out.println(Integer.TYPE); //int

}

}

**3. Boolean class**

- used to perform operation on boolean datatype

**Constructor**

1. Boolean (boolean b)

Boolean b1=new Boolean(true);

System.out.println(b1); //true

Boolean b2=new Boolean(false);

System.out.println(b2); //false

Boolean b3=new Boolean(True); //error

System.out.println(b3);

Boolean b4=new Boolean(hello); //error

System.out.println(b4);

2. Boolean(String s) - we can give only true in any format it will return true, other than true if we give anything it return false

Boolean b1=new Boolean("true");

System.out.println(b1); //true

Boolean b2=new Boolean("True");

System.out.println(b2); //true

Boolean b3=new Boolean("false");

System.out.println(b3); //false

Boolean b4=new Boolean("hello");

System.out.println(b4); //false

Boolean b5=new Boolean("TRUE");

System.out.println(b5); //true

**4. Character class**

- used to perform operation on char datatype

**Constructor**

1. Character(char c)

**Methods**

1. static boolean isDigit(char c)

2. static boolean isLetter(char c)

3. static boolean isLetterOrDigit(char c)

4. static boolean isLowerCase(char c)

5. static boolean isUpperCase(char c)

6. static boolean isSpace(char c)

7. static char toLowerChar(char c)

8. static char toUpperCase(char c)

***Autoboxing and Unboxing***

- Introduced from JDK1.5

- Autoboxing is automatic conversion of primitive datatype to wrapper class

- Unboxing is automatic conversion of wrapper class to datatype

with autoboxing without autoboxing

Integer i; Integer i;

int j; int j;

i=10; //Autoboxing i=new Integer(10);

j=i; //unboxing j=i.intValue();

Short s;

Boolean b=(s instanceof Short);

Integer i=10;

int j=i;

***3. String class***

- String is final class in java.lang.\*

- Fixed length of character, hence it is immutable class (ie) we cant increase or decrease its size at runtime

Syntax: public final class String extends Object implements Serializable, Comparable, CharSequence

Constructor

1. String()

2. String(String s)

3. String(byte[] b)

4. String(byte[] b,int start, int offset)

5. String(char[] c)

6. String(char[] c,int start,int offset)

7. String(StringBuffer s)

Methods

1. String toString() - String representation of object

2. char charAt(int position) - return a single char

3. void getChars(int start,int end, char buf[], int targetstart) - return group of char

4. byte[] getBytes() - convert string to byte[]

5. char[] toCharArray() - convert string to char[]

6. boolean equals(String d) - check equality of content taking case into consideration

7. boolean equalsIgnoreCase(String d) - check equality of content without taking case into consideration

8. == - check equality of object reference

9. boolean startsWith(String s)

10. boolean endsWith(String s)

11. String toLowerCase()

12. String toUpperCase()

13. int compareTo(String s) - compare 2 string taking case into consideration and if both are equal return o, greater return +ve, lesser return -ve

14. int compareToIgnoreCase(String s) - compare 2 string without taking case into consideration and if both are equal return o, greater return +ve, lesser return -ve

public class Main {

public static void main(String[] args) {

byte[] b= {65,66,67,68,69};

String s1=new String(b);

System.out.println(s1); //ABCDE

String s2=new String(b,1,3); //BCD

char[] c= {'J','A','V','A'};

String s3=new String(c);

System.out.println(s3); //JAVA

String s4=new String(c,0,2);

System.out.println(s4); //JA

String s5=new String(s4);

System.out.println(s5); //JA

String s6="Hello"; //literal

System.out.println(s6.charAt(1)); //e

String s7="This is a demo of getChars method";

int start=10; int end=14;

char buf[]=new char[end-start]; //4

s7.getChars(start, end, buf, 0);

System.out.println(buf); //demo

String s8="ABCD";

byte[] b1=s8.getBytes();

for(byte b2:b1)

System.out.println(b2); //65 66 67 69

String s9="ABCD";

char c1[]=s9.toCharArray();

for(char c2:c1)

System.out.println(c2); //A B C D

String s10=new String("Hello");

String s11=new String("Hello");

String s12=new String("hello");

System.out.println(s10.equals(s11)); //true

System.out.println(s10.equals(s12)); //false

System.out.println(s10.equalsIgnoreCase(s12)); //true

System.out.println(s11==s12); //false

String s13=s12;

System.out.println(s12==s13); //true

System.out.println("Foobar".startsWith("Foo")); //true

System.out.println("Foobar".endsWith("bar")); //true

System.out.println("Foobar".toLowerCase());

System.out.println("Foobar".toUpperCase());

System.out.println("hello".compareTo("hello")); //0

System.out.println("check".compareTo("hello")); //-5

System.out.println("hello".compareTo("check")); //5

System.out.println("hell".compareTo("heck")); //9

}

}

***DAY 7***

**Methods**

14. int compareToIgnoreCase(String s) - compare 2 string without taking case into consideration and if both are equal return o, greater return +ve, lesser return -ve

15. int indexOf(char c)

int indexOf(String s)

int indexOf(int charvalue,int startindex)

int indexOf(String s,int start)

- used to return the position of first occurence of char in given string, return -1

16. int lastIndexOf(char c)

int lastIndexOf(String s)

int lastIndexOf(int start,char c)

- used to return the position of last occurrence of char in given string

17. String substring(int start)

String substring(int start, int end) //start to end-1

- used to return part of the string

18. int length()

19. String concat(String s) or + operator

20. String trim() - remove leading and trailing space

21. String replace(char old, char new)

22. String[] split(String delimiter)

String[] split(String delimter,int limit)

- used to split the string based on delimiter

23. boolean matches(String regex)

- used to match the string based on regex

24. boolean regionMatches(boolean ignorecase,int toffset, String other, int offset, int length)

- used to match part of string with another string

25. int codePointAt(int index)

26. int codePointBefore(int index)

27. int codePointAfter(int index)

28. int codePointCount(int beginindex,int endindex)

29. boolean contains(String s)

30. static String join(String delimiter,String...s) - used to join string based on delimiter

31. static String copyValueOf(char[] c)

static String copyValueOf(char[] c,int offset,int count)

32. static String format(String format,Object val)

%+-0wspecifier.precision

+ - add space before number

- - add space after number

0 - pad with 0

specfier - d,s,c,o,x,f

precision - after decimal place

public class Main {

public static void main(String[] args) {

String s1="It is the time for all good men to come to their country and pay their due tax";

System.out.println(s1.indexOf('t')); //1

System.out.println(s1.indexOf("the")); //6

System.out.println(s1.indexOf(116, 5)); //6

System.out.println(s1.indexOf("the", 10)); //43

System.out.println(s1.indexOf('z')); //-1

System.out.println(s1.lastIndexOf('t')); //75

System.out.println(s1.lastIndexOf("the")); //65

System.out.println(s1.lastIndexOf("the", 65));//43

String s2="helloworld";

System.out.println(s2.substring(3)); //loworld

System.out.println(s2.substring(2, 6)); //start to end-1 llow

String s3="Hello";

System.out.println(s3.concat("World")); //HelloWorld

System.out.println(s3); //Hello

String s4=s3.concat("World");

System.out.println(s4); //HelloWorld

String s5=" Hello World ";

System.out.println(s5.length()); //13

System.out.println(s5.trim()); //Hello World

System.out.println(s5.length()); //13

String s6=s5.trim();

System.out.println(s6.length()); //11

System.out.println("Hello".replace('e','i')); //Hillo

String s7="one-two-three";

String temp1[]=s7.split("-");

for(String t1:temp1)

System.out.println(t1); //one two three

String s8="one.two.three";

String temp2[]=s8.split("\\.");

for(String t2:temp2)

System.out.println(t2); //one two three

String s9="A\*bunch\*of\*stars";

String temp3[]=s9.split("\\\*");

for(String t3:temp3)

System.out.println(t3);

String temp4[]=s9.split("\\\*",3);

for(String t4:temp4)

System.out.println(t4); // A bunch of\*stars

String s10="string is a class";

String temp5[]=s10.split("s");

for(String t5:temp5)

System.out.println(t5);

String s11="No concession, no concillation, no comprise and just give and take policy";

String temp6[]=s11.split("concession|concillation|comprise|(give and take)");

for(String t6:temp6)

System.out.println(t6); //no , no , no and just policy

String s12="Welcome to Java";

System.out.println(s12.matches("(.\*) to PHP")); //false

System.out.println(s12.matches("Welcome to (.\*)")); //true

System.out.println(s12.matches("Java")); //false

System.out.println(s12.matches("(.\*) to (.\*)")); //true

String s13="ABC Windows test";

System.out.println(s13.regionMatches(true, 4, "windows", 0, 7)); //true

System.out.println("abcd".codePointAt(0)); //97

System.out.println("abcd".codePointBefore(2)); //98

System.out.println("abcdefg".codePointCount(0, 4)); //4

System.out.println("hello".contains("e")); //true

System.out.println("hello".contains("E")); //false

System.out.println(String.join("-", "one","two","three")); //one-two-three

char c[]= {'a','b','c','d','e','f','g'};

String s14="";

System.out.println(s14.copyValueOf(c)); //abcdefg

System.out.println(s14.copyValueOf(c, 3, 2)); //de

int a=23;

System.out.println(String.format("%d",a)); //23

System.out.println(String.format("|%d|", a)); //|23|

System.out.println(String.format("|%5d|", a)); //| 23| //5-total width

System.out.println(String.format("|%-5d|", a)); //|23 |

System.out.println(String.format("|%06d|",a)); //|000023|

//System.out.println(String.format("|%-06d|",a)); //exception

System.out.println(String.format("|%f|", 345.334));//|345.334000|

System.out.println(String.format("|%.2f|", 345.334)); //|345.33|

System.out.println(String.format("|%8.2f|", 345.334)); //| 345.33|

} }

4. StringBuffer class

- present in java.lang.\*

- It is a mutable class, variable length of char (ie) we can increase or decrease its size at runtime

- we cant overide equals() in StringBuffer, using toString() we convert StringBuffer to String and then apply equals()

- By default StringBuffer is synchronized or threadsafe

- Default capacity of StringBuffer is 16

Constructor

1. StringBuffer()

2. StringBuffer(String s)

3. StringBuffer(int capacity)

Methods

1. int length()

2. char charAt(int position)

3. void setCharAt(int index,char c)

4. void setLength(int length)

5. int capacity()

6. StringBuffer append(int i)

StringBuffer append(char c)

StringBuffer append(String s)

- adding at end of StringBuffer

7. StringBuffer insert(int index,int n)

StringBuffer insert(int index,char c)

StringBuffer insert(int index,String s)

8. StringBuffer reverse()

9. StringBuffer replace(int start,int end, String s)

10. StringBuffer delete(int start,int end) - delete group of char

11. StringBuffer deleteCharAt(int index) - delete single char

public class Main {

public static void main(String[] args) {

StringBuffer sb1=new StringBuffer("Hello");

System.out.println(sb1); //Hello

System.out.println(sb1.length()); //5

System.out.println(sb1.capacity()); //16+5=21

System.out.println(sb1.charAt(1)); //e

sb1.setCharAt(1, 'i');

System.out.println(sb1); //Hillo

sb1.setLength(2);

System.out.println(sb1); //Hi

System.out.println(sb1.length()); //2

int b=20;

StringBuffer sb2=new StringBuffer();

String s1=sb2.append("a=").append(b).append("!").toString();

System.out.println(s1); //a=20!

StringBuffer sb3=new StringBuffer("I Java");

System.out.println(sb3.insert(2, "like ")); //I like Java

StringBuffer sb4=new StringBuffer("hello");

System.out.println(sb4.reverse()); //olleh

StringBuffer sb5=new StringBuffer("This is a test");

System.out.println(sb5.replace(5, 7, "was ")); //This was a test

StringBuffer sb6=new StringBuffer("This is a test");

System.out.println(sb6.delete(5, 7)); //This a test

System.out.println(sb6.deleteCharAt(0)); //his a test

} }

5. StringBuilder class

- Available from JDK1.5

- Similar to StringBuffer class, it is not synchronized or thread safe so it gives better performance than ur StringBuffer class

6. Throwable class

- used to handle exception in Java

2 types of exception

1. Checked exception

- all subclasses of Exception class excluding RuntimeException. Eventhough we have written 100% correct program also, ur code has to be surrounded by try/catch block or throws keyword otherwise your program will not compile.

- Will insist the programmer to surround the code by try/catch block or throws keyword otherwise your program will not compile

2. Unchecked Exception

- all subclassess of RuntimeException class

- Eventhough your program contains exception, the compiler will just compile the program but at runtime we get related exception

- Will not insist the programmer to surround the code by try/catch block or throws keyword, the program will compile, but at runtime we get related exception

Constructor

1. Throwable()

2. Throwable(String msg)

3. Throwable(String msg,Throwable t)

Types of Exception

1. ArthimeticException - divide anything by 0

2. ArrayIndexOutOfBoundsException - datatype array

int a[]=new int[3];

a[3]=5;

3. StringIndexOutOfBoundsException - only String array

String a[]=new String[3];

a[3]="5";

4. NegativeArraySizeException

int a[]=new int[-3];

5. NumberFormatException

int a=Integer.parseInt("abc");

6. ArrayStoreException

int a[]=new int[3];

a[0]="hello";

7. NullPointerException

class A {

void show() {

}

}

class Main { PSVM {

A a=new A();

a.show();

a=null;

a.show(); //NPE

***DAY 8***

1. Exception 2(ArrayIndexOutOfBoundsException And NegativeArraySizeException)

Write a program to get the number of overs and the runs scored in each over. Get the over number from the user and display number of runs scored in that over. Let

• number of overs be the array size

• over number be the index of the array+1

• runs be the array elements

This program may generate

1. NegativeArraySize Exception when the number of overs is negative

2. ArrayIndexOutOfRange Exception when the over number that is searched is beyond the specified over numbers.

Use exception handling mechanisms to handle these exceptions. Use a single catch block. In the catch block, print the class name of the exception thrown.

Input and Output Format:

**Sample Input/Output 1:**

Enter the number of overs - 3

Enter the number of runs for each over 8 15 12

Enter the over number - 2

Runs scored in this over - 15

**Sample Input/Output 2:**

Enter the number of overs - 3

Enter the number of runs for each over 8 15 12

Enter the over number - 4

java.lang.ArrayIndexOutOfBoundsException

**Sample Input/Output 3:**

Enter the number of overs - -1

Enter the number of runs for each over

java.lang.NegativeArraySizeException

2.Custom Exceptions [Age]

Write a program to get the name and age of the player from the user and display it.

player name is a string

player age is an integer value

Note : The player is eligible to participate in IPL when their age is 19 and above

This program may generate

1. InvalidAgeRange Custom Exception when the player's age is below 19

Use exception handling mechanisms to handle these exceptions.

Create a class called CustomException which extends Exception and it includes constructor to initialize the message. Use appropriate exception handling mechanisms to handle these exceptions

**Input and Output Format:**

**Sample Input/Output 1:**

Enter the player name - Albie Morkel

Enter the player age - 35

Player name : Albie Morkel

Player age : 35

**Sample Input/Output 2:**

Enter the player name - Ishan Kishan

Enter the player age - 16

CustomException: InvalidAgeRangeException

3. TeamNameNotFound Exception

Write a program to get the two team names i.e expected Runner and Winner team of IPL season 4 and display it. Team name is a string

Note : The team name given below are only eligible to take part in IPL season 4

Chennai Super Kings

Deccan Chargers

Delhi Daredevils

Kings XI Punjab

Kolkata Knight Riders

Mumbai Indians

Rajasthan Royals

Royal Challengers Bangalore

This program may generate TeamNameNotFound Custom Exception when the expected team entered is not present in the above eligible teams list for IPL season 4.

Use exception handling mechanisms to handle these exceptions

**Input and Output Format:**

**Sample Input and Output 1:**

Enter the expected winner team of IPL Season 4

Chennai Super Kings

Enter the expected runner Team of IPL Season 4

Mumbai Indians

Expected IPL Season 4 winner: Chennai Super Kings

Expected IPL Season 4 runner: Mumbai Indians

**Sample Input and Output 2:**

Enter the expected winner team of IPL Season 4

Pune Warriors

TeamNameNotFoundException: Entered team is not a part of IPL Season 4

***Exception Handling***

5 keywords

1. try

- Program to be monitored for exception has to be put inside try block

2. catch

- used to catch the exception generated, mainly used to print userdefined messages when an exception occurs

3. finally

- optional stmt, it will be executed every time irrespective of exception occurs or not

- finally stmt used for closing resources,release memory etc

- used in 3 cases - file, database, socket programming

try {

fp=fopen("a.txt","r");

read operation }

catch(Exception e){

}

finally {

fclose(fp); }

3 ways

1. try { 2. try{ 3. try{

} } }

catch(Exception e){ catch(Exception e){ finally{

} } }

finally{

}

- We should not write any code between try catch and finally

**Command line argument**

- we give input while running the prg in command line

- When we use args argument inside the prg, then for that prg we need to give input thru command line

- each command line args should be separated by space

public class Main {

public static void main(String[] args) {

try {

int a=Integer.parseInt(args[0]);

int c=10/a;

System.out.println(c);

}

catch(ArithmeticException e) {

System.out.println("Number divided by 0: "+e);

} } }

**Multi catch statement**

- single try can contain multiple catch blocks

public class Main {

public static void main(String[] args) {

try {

int a=Integer.parseInt(args[0]);

int c=10/a;

System.out.println(c); //5

int b[]= {10};

b[5]=20;

}

catch(ArithmeticException e) {

System.out.println("Number divided by 0: "+e);

}

catch(ArrayIndexOutOfBoundsException e) {

System.out.println("Array Index: "+e);

} } }

If we want to invoke any other exception then we can define catch block with general class called Exception or Throwable

public class Main {

public static void main(String[] args) {

try {

int a=Integer.parseInt(args[0]);

int c=10/a;

System.out.println(c); //5

int b[]= {10};

b[5]=20;

}

catch(ArithmeticException e) {

System.out.println("Number divided by 0: "+e);

}

catch(ArrayIndexOutOfBoundsException e) {

System.out.println("Array Index: "+e);

}

catch(NullPointerException e) {

System.out.println(e);

}

catch(NumberFormatException e) {

System.out.println(e);

}

catch(Exception e) { //catch(Throwable e)

System.out.println(e);

}

} }

- Whenever we define general class called Exception or Throwable, it should be always present in last catch block otherwise it leads to compilation error

public class Main {

public static void main(String[] args) {

try {

int a=Integer.parseInt(args[0]);

int c=10/a;

System.out.println(c); //5

int b[]= {10};

b[5]=20;

}

catch(Exception e) { //catch(Throwable e)

System.out.println(e);

}

catch(ArithmeticException e) { //error

System.out.println("Number divided by 0: "+e);

}

catch(ArrayIndexOutOfBoundsException e) { //error

System.out.println("Array Index: "+e);

}

catch(NullPointerException e) { //error

System.out.println(e);

}

catch(NumberFormatException e) { //error

System.out.println(e);

} } }

- From JDK1.7 version, we define multiple exception in a single catch block using | symbol

public class Main {

public static void main(String[] args) {

try {

int a=Integer.parseInt(args[0]);

int c=10/a;

System.out.println(c); //5

int b[]= {10};

b[5]=20;

}

catch(ArithmeticException | ArrayIndexOutOfBoundsException | NullPointerException | NumberFormatException e) {

System.out.println(e);

}

catch(Exception e) {

System.out.println(e);

}}}

4. throw keyword

- used to manually throw an exception

- whenever it invokes throw keyword it will automatically goes to related catch block

Syntax: throw new Exception("messages");

public class Main {

static void demo() {

try {

throw new NullPointerException("Demo");

}

catch(NullPointerException e) {

System.out.println("Caught");

throw e;

} }

public static void main(String[] args) {

try {

demo();

}

catch(NullPointerException e) {

System.out.println("Recaught");

} } }

5. throws keyword

- throws keyword used to declare an exception and used only in methods

- used to indicate that the methos might throw one of the exception

public class Main {

static void demo() throws NullPointerException {

throw new NullPointerException("Demo");

}

public static void main(String[] args) {

try { demo(); }

catch(NullPointerException e) {

System.out.println("Recaught");

}}}

public class Main {

static void demoA() {

try {

System.out.println("Inside demoA");

throw new RuntimeException("Hello");

}

finally {

System.out.println("Inside demoA finally");

}

}

static void demoB() {

try {

System.out.println("Inside demoB");

return;

}

finally {

System.out.println("DemoB finally");

}

}

public static void main(String[] args) {

try {

demoA();

}

catch(RuntimeException e) {

System.out.println("caught");

}

demoB(); }}

Userdefined Exceptions

- Your userdefined exception class should extend Exception class and override toString()

class NotValidAgeException extends Exception {

String s1="";

public NotValidAgeException(String s1) {

super();

this.s1 = s1;

}

@Override

public String toString() {

return s1;

}

}

public class Main {

static void validateAge() throws NotValidAgeException {

Scanner sc=new Scanner(System.in);

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

int age=sc.nextInt();

if(age<18) {

throw new NotValidAgeException("Your age is not eligible");

}

else {

System.out.println("You are eligible");

}

}

public static void main(String[] args) {

try {

validateAge();

} catch (NotValidAgeException e) {

System.out.println(e);

} } }

Assertions

- Available from JDK1.5

- Used to validate the boolean conditions at runtime

Syntax: assert <<expression>>;

assert <<expression>>:String message;

- By default assertions is disabled in Java, while running we have to enable assertion using -ea

- Provided with assert keyword and AssertionError class is unchecked exception because it inherits from Error class

- If assert condition fails it throws AssertionError class

public class Main {

static double withdraw(double balance, double amount) {

assert (balance>=amount):"Balance is insufficient";

return (balance-amount);

}

public static void main(String[] args) {

System.out.println(withdraw(1000,500)); //500.0

System.out.println(withdraw(1000,2000)); //-1000.0

} }

try with resources

- Available from JDK1.7

- Used to close automatically all the resources at the end of stmt, using AutoCloseable interface

Syntax:

try(resources) {

}

catch(Exception e){

}

public class Main {

public static void main(String[] args) {

String line;

try (BufferedReader br=new BufferedReader(new FileReader("Main.java"));

PrintWriter pw=new PrintWriter(new File("a.txt"))){

while((line=br.readLine()) != null) {

System.out.println(line);

}

}

catch(Exception e) {

System.out.println(e);

}

}

}

In JDK1.9, enhancement in try with resouces, we can define resources outside the try and use it

public class Main {

public static void main(String[] args) throws IOException {

String line;

BufferedReader br=new BufferedReader(new FileReader("Main.java"));

PrintWriter pw=new PrintWriter(new File("a.txt"));

try (br;pw){

while((line=br.readLine()) != null) {

System.out.println(line); } }

catch(Exception e) {

System.out.println(e);

} } }

java.util.\*

- Utility framework or Collection framework

- used to store collection of objects

1. Collection interface - core interface in order to store of collection of objects

2. Collections class - provided with static algorithm/methods that supports util package

Collection interface

- core interface in order to store of collection of objects

Methods

1. boolean add(Object obj) - add single object

2. boolean addAll(Collection c) - add multiple objects

3. boolean remove(Object obj) - remove single object

4. boolean removeAll(Collection obj) - remove all object

5. boolean contains(Object obj) - check single object present in collection or not

6. boolean containsAll(Collection c) - check multiple object present in collection or not

7. boolean retainAll(Collection c) - remove from the target collection all the elements that are not contained in the specified collection

8. int size() - return number of object present in collection

9. Object[] toArray() - return an array containing all elts in collection

10. Iterator iterator()

11. ListIterator listIterator()

12. boolean isEmpty()

13. boolean equals(Object o)

List interface

- Ordered and duplicate elements

Methods

1. void add(int index,Object obj) - add single object at particular index position

2. boolean addAll(int index,Collection c) - add multiple object at particular index position

3. Object get(int index) - return single object at particular index position

4. int indexOf(Object o) - return position of first occurence of object in collection

5. int lastIndexOf(Object o) - return position of last occurence of object in collection

6. Object remove(int index) - remove single object present in particular index

7. Object set(int index, Object o) - set is something like replace

8. List subList(int start,int end) - return part of list from start to end-1

Set interface

- unordered and no duplicates elt

SortedSet interface

- used to sort the elements in Set interface

Methods

1. Object first() - return first object

2. Object last() - return last object

3. SortedSet subSet(Object start,Object end) - part of set from start to end-1

4. SortedSet headSet(Object o) - return all elts present before specified elt

1 2 3 4 5.headSet(3); //1 2

5. SortedSet tailSet(Object o) - return all elts present after specified elt

1 2 3 4 5.tailSet(3); //4 5

***DAY 9***