java language originally developed by sun micro system. Which was initiated by james glosing and released by 1995

java is:

* dynamic
* secure
* independent
* robust
* interpreted
* simple
* portable
* high performance
* object oriented
* distributed
* architeture-nuetral

syntax:

* object
* class
* method
* variable

first java program:

public class classname{

public static void main(String[] args){

system.out.println(""Hello World");

}

}

java modifiers:

it is possible to modify class, methods, ect

types of modifiers:

1.access modifiers

2.Non-access Modifiers

1.Access Modifiers:

* default
* public
* private
* protected

Non-access Modifier:

* final
* abstract
* strictfp

Java Variables:

types of variables

1. local variable
2. static variables
3. instance variable(Non-static variable)

Java Arrays:

It is an object on the heap.

Arrays are object that stored in multiple variables of same type.

Java enums:

enums restricted variable to have one only a few predefined values.

the values in this enumerated is called enums.

example:

**class** FreshJuice {

**enum** FreshJuiceSize{***SMALL***,***LARGE***,***MEDIUM***};

FreshJuiceSize size;

}

**public** **class** FreshJuiceTest{

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

FreshJuice juice=**new** FreshJuice();

juice.size=FreshJuice.FreshJuiceSize.***SMALL***;

juice.size=FreshJuice.FreshJuiceSize.***MEDIUM***;

juice.size=FreshJuice.FreshJuiceSize.***LARGE***;

System.***out***.println("Size:"+juice.size);

}

}

Inheritance :

class can be derived from classes.

the existing class is called **super** class and the derived class is called **subclass**.

Interface:

interface is defines as a contract between object to communicate with each other( object).

methods, derived class but the implementation of method is subclass.

java-Objects and classes

java is a object oriented programming language.

fundamental concepts:9

* polymorphism
* encapsulation
* abstraction
* inheritance
* object
* classes
* method
* instance
* message parsing

1.Oject:

it is state and behaviour

example:

a dog

here,

dog is---> state

dog 's----> color, name, breed is ---->behaviour.

2.class:

a class is a blueprint.

state/behaviour that object of its type.

**public** **class** Dog {

String bread;

**int** agec;

String color;

**void** hungry() {

}

**void** barking() {

}

**void** sleeping() {

}

}

class can contain variable types:

1.Local Variable

2.Instance Variable

3.Class Variable

Local Variable:

Variables defined inside methods, Constructors or blocks are called local variable.

Instance Variable:

Variables within a class but outside any method

Class Variable:

Variables within a class, outside any method, with the static keyword.

**Constructor:**

Every class has a constructor.

Each time new object is created, at least one constructor will be invoked.

The main rule of constructor is that they should have the same name as the class.

A class can have more than one constructor.

**public** **class** Puppy {

**public** Puppy() {

}

**public** Puppy(String name) {

}

}

Create an Object: **new Keyword**

a class provides the blueprint for object.

the new keyword used to create new objects.

Basically, An object is created from a class.

three steps:

1.declaration

2.inistantiation

3.initialization

**Accessing instance Variable and Method:**

these are accessed via created objects.

fully qualified path:

1. first create an object

**Object Reference=new constructor();**

2. call a variable

**Object Refference.varianleName;**

3.call a class Method

Object Reference.MethodName();

**public** **class** Puppy {

**int** puppyage;

**public** Puppy(String name) {

System.***out***.println("Name choosen is: "+name);

}

**public** **int** getPuppyage() {

System.***out***.println("Puppy's age is: "+puppyage);

**return** puppyage;

}

**public** **void** setPuppyage(**int** puppyage) {

**this**.puppyage = puppyage;

}

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

Puppy mypuppy=**new** Puppy("tommy");

mypuppy.setPuppyage(2);

mypuppy.getPuppyage();

System.***out***.println("Variable value: "+mypuppy.puppyage);

}

}

source file declaration rules:

1.there can be only one public class per source file.

2.A source file can have multiple non-public classes.

3. A source file should be as classname.java

4.a. if the class defined inside package

b. The package statement should be the first statement in the source code .

5.a. if import statement is present, then they must be written between the package statement and the class declaration.

b. if there is no package statement then import statement should be the first line in the source file.

6.

there are different types of classes:

1.abstract classes

2.final classes

3,inner classes

4. Anonymous classes

**java Packages:**

A way of categorizing the class and interface.

When developing applications in java, hundreds of classes and interfaces will be written

**Import statement:**

import statement is a way of giving a proper location for the complier to find that particular location.

Example:

import java.io.\*;

java-data types

based on the data types of a variable, the operating system allocating memory and decided what can be stored in the reserved memory.

There are two data types in java:

1.primitive data types

2.Refernce/Object data types

**Primitive data types:**

Primitive data types are predefined by the language and named by a keywords.

they are eight(8) Primitive data types support by java.

1. byte
2. short
3. int
4. long
5. float
6. double
7. boolean
8. char

**Reference data type:**

String

**Java Literals:**

A literal is a source code representation of fixed length.

primitive data types:

byte a=68;

byte, short, int and long can be expressed in decimal(based 10) or hexadecimal(based 16) or octal(based 8) number system as well

prefix 0(zero) is used to indicate octal.

prefix 0x is used to indicate hexadecimal.

when using these number system for literal .

example:

int decimal=100;

int octal=0144;

int hexdecimal=0x64;

char a ='A';

String a="A";

exmple:

char a='u0001';

String a="u0001";

String and char notation:

Notation ----character---represent

\n------Newline----(0x0a)

\r-------carriage return---(0x0d)

\f-------formfeed-----(0x0c)

\b----Backspace----(0x08)

\s-----Space----(0x20)

\"-------double Quote

\'--------Single Quote

\\-----Backslash

\ddd----octal character (ddd)

\uxxxx------hexadecimal unicode character (xxxx)

java-Variable Types

the range of values that can be stored within that memory.

The set of operations that can be applied to the variable.

form of a variable :

data type variable [ = value][,variable [=value]....];

Here, data type is one of java's data type.

variable is the name of the variable.

Example:

int a, b, c;

int a=10, b=20;

byte B=22;

double pi=3.14159;

char a='a';

three kind of variables:

1.local variable

2.instance variable

3.class 0r static variable

Local Variable:

* local variables are declared in methods, blocks and constructors.
* Access modifiers can't be used for local variables.
* Local variables are visible only within the declared method, constructor , or blocks.
* A Local variable should be declared and an initial value should be assigned before the first use.
* Local variables are implemented at stack level internally.

Instance Variable:

* Instance variable are declared in a class, but outside a method, constructor or any block.
* When a space is allocated for an object in the heap, instance variable value is created.
* Access modifiers can be given for instance variables.
* Instance variable is created, when an object is created within use of the keyword 'new'.
* Instance variable must be referenced by more than one methods, constructor, or blocks
* Instance variable have default values .

i.e number is 0(zero),

boolean is false

String( Object Reference) is null

* Instance variables can be accessed directly by calling the variable name inside the class
* Instance variable within static methods(when instance variables are given accessibility), they should be called using the fully qualified name

i.e., ObjectRefernece.VariableName

Class (or) Static Variable:

* Static variables are declared with the static keyword in a class
* Only one copy of each variable per class.
* static variable rarely used other than being declared as constants.
* Constants are declared as public/private, final and static.
* Constants variables never change from their initial value.
* static variable are stored in the static memory.

java-Modifier Types

Modifiers are keywords

types of Modifiers:

1.Access control Modifiers

2.Non-Access control Modifier

**Access Modifiers:**

java provides number of access control modifiers to set access levels for classes, variables, methods and constructors.

four(4) access control modifiers level are

1.default

2.public

3.private

4.protected

default:

* visible to the package
* no modifiers are needed

example:

public:

* visible to the world

example: public class className{

//....

}

private:

* visible to the class only

example: private boolean myFlag;

protected:

* visible to the packages and all subclasses

example: protected static final int boxwidth=42;

Non-Access control modifier:

java provides number of non-access control modifiers to achieve many other funtionality.

1.static modifier

2.final modifier

3.abstract modifier

4.volatile modifier

5.synchronized modifier

**static modifier:**

creating class, methods and variables.

**final modifier:**

finalizing the implementation of classes, methods and variables.

**abstract modifier:**

creating abstract classes and methods.

**volatile modifier:**

it is used for thread.

**synchronized modifier:**

it is used for thread

java-Operator

java provides a rich set of operators to manipulate variables.

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

Arithmetic Operator:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.

|  |  |  |
| --- | --- | --- |
| Operator | description | example |
| +(Addition) | Adds values on either side of the operator. | A+B will give 30 i.e.,10+20 |
| -(subtraction) | Subtraction right-hand operand from left-hand operand. | A-B will give -10 i.e., 10-20 |
| \*(multiplication) | Multiplies values on either side of the operator. | A\*B will give 200 |
| /(Division) | Divides left-hand operand by right-hand operand | B/A will give 2 |
| %(modulus) | Divides left-hand operand by right-hand operand and returns remainder. | B % A will give 0 |
| ++(Increment) | Increases the value of operand by 1 | B++ gives21  A++ gives 11 |
| --(Decrement) | Decreases the value of operand by 1 | B-- gives 19  A-- gives 9 |

Relational Operator:

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| ==(equal to) |  | (A == B) is not true. |
| !=(not equal to) |  | (A!=B) is true |
| > (greater than) |  | (A>B) is not true. |
| < (less than) |  | (A<B) is true. |
| >=( greater than or equal to) |  | (A>=B) is not true. |
| <=(less than or equal to) |  | (A<=B) is true. |

Bitwise operator:

Bitwise operator works on bits and performs bit-by-bit operation.

|  |  |  |
| --- | --- | --- |
| Operation | Description | Example |
| &( bitwise and) |  |  |
| |(bitwise or) |  |  |
| ^(bitwise XOR) |  |  |
| ~(bitwise compliment) |  |  |
| <<(left shift) |  |  |
| >>(right shift) |  |  |
| >>>(Zero fill right shift) |  |  |

Logical Operators:

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| && (logical and) |  | (A && B) is false |
| || (logical or) |  | (A || B) is true |
| ! (logical not) |  | !(A && B) is true |

Assignment Operator:

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| = |  |  |
| += |  |  |
| -= |  |  |
| \*= |  |  |
| /= |  |  |
| %= |  |  |
| <<= |  |  |
| >>= |  |  |
| &= |  |  |
| ^= |  |  |
| |= |  |  |
|  |  |  |

**Miscellaneous Operators:**

1.conditional (? :)

2.instanceof

3.precedence

|  |  |  |
| --- | --- | --- |
| Operator |  |  |
| Postfix |  |  |
| Unary |  |  |
| Multiplicative |  |  |
| Additive |  |  |
| Shift |  |  |
| Equality |  |  |
| Bitwise AND |  |  |
| Bitwise XOR |  |  |
| Bitwise OR |  |  |
| Logical AND |  |  |
| Logical OR |  |  |
| Conditional |  |  |
| Assignment |  |  |

java-Loop

|  |  |  |
| --- | --- | --- |
| Loop | Description | syntax |
| While |  |  |
| For |  |  |
| do-while |  |  |

|  |  |  |
| --- | --- | --- |
| Control | description | Syntax |
| Break |  |  |
| Continue |  |  |

Enhanced for loop in java:

this is mainly used traverse collection of elements in including arrays.

Syntax:

for(declaration :expression){

// statements

}

java-Decision Making

|  |  |  |
| --- | --- | --- |
| Statement | Description | syntax |
| If |  |  |
| If-else |  |  |
| Nested if |  |  |
| Switch |  |  |

The ? : Operator:

syntax:

Exp1 ? Exp2 : Exp3;

java-Numbers Class or wrapper classes

we use primitive data types such as byte, int, long, double, etc.

Example:

int i=5000;

float gpa=13.65;

double mask=0xaf;

we to use objects instead of primitive data types

wrapper classes:

All the wrapper classes are subclasses of the abstract class Number.

Numbers:

1.Byte

2.Int

3.Double

4.Float

5.Long

6.Short

wraps its respective data type.

Converting primitive types into object is called **boxing,** and this is taken care by the complier.

To pass the value of the primitive data type to the constructor of the wrapper class.

The wrapper object will be converted back to a primitive data type, and this process is called **unboxing**

**T**he number class is part of the java.lang package.

Number Method:

The list of the instance methods that all the subclasses of the number class implement-

|  |  |
| --- | --- |
| Method | Description |
| xxxValue() |  |
| compareTo() |  |
| Equals() |  |
| valueOf() |  |
| toString() |  |
| parseInt() |  |
| Abs() |  |
| Ceil() |  |
| Floor() |  |
| Rint() |  |
| Round() |  |
| Min() |  |
| Max() |  |
| Exp() |  |
| Log() |  |
| Pow() |  |
| Sqrt() |  |
| Sin() |  |
| Cos() |  |
| Tan() |  |
| Asin() |  |
| Acos() |  |
| Atan() |  |
| Atan2() |  |
| toDegrees() |  |
| toRandians() |  |
| Random() |  |

java-Character Class or wrapper class

we use primitive data types char

Example:

char ch='a';

Array of char:

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

we need to use objects instead of primitive data types

Character ch= new Character('a');

Escape sequence:

|  |  |
| --- | --- |
| Escape Sequence | Description |
| \t |  |
| \b |  |
| \n |  |
| \r |  |
| \f |  |
| \'  \''  \\ |  |

Character Method:

The important instance methods that all the subclasses of the charater class

|  |  |
| --- | --- |
| Method | Description |
| isLetter() |  |
| isDigit() |  |
| isWhiteSpace() |  |
| isUpperCase() |  |
| isLowerCase() |  |
| toUpperCase() |  |
| toLowerCase() |  |
| toString() | Return a String object representing the specified character value that is, a one-character string. |

**Java-String Class**

Strings are a sequence of characters.

Strings are treated as object.

The string class to create and manipulate strings.

**Creating Strings:**

example:

String greeting="hello world";

**string Length:**

Methods used to obtain information about an object are known as **accessor methods.**

you can use with string is the **length()** method, which returns the number of character contained in the string object.

Example:

public class StringDemo{

public static void main (String args[]) {

String palindrome="Dot saw i was Tod";

int len=palindrome.length();

system.out.println("String Length is:"+len);

}

}

**Concatenating String:**

The String class includes a method for concatenating two string.

syntax:

string1.concat(string2);

example:

"My name is".concat("Zara");

strings are more commonly concatenated with the + operator.

"hello,"+"world"+"!"

which result is in "hello,world!"

Example:

public class StringDemo{

public static void main(String args[]) {

String string1="saw I was";

system.out.println("Dot"+string1+"Tod");

}

}

**Creating Format Strings:**

**example:**

System .out .printf("The value of the float variable is"+ "%f, which the value of the integer"+ "is%",floatVar, intVar, stringVar);

**Example:**

String fs;

fs=String.format("The value of the float variable is"+%f"+ "%f, which the value of the integer"+ "is%",floatVar, intVar, stringVar);

System .out.println(fs);

**String Methods:**

|  |  |
| --- | --- |
| Method | Description |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Java-Arrays

Arrays are used to store a collection of data and variables.

java provides a data structure, the array , which stores a fixed-size sequential collection of elements of the same type.

Example:

Instead of declaring individual variables, such as number0,number1,number2,................number99.

one array variable such as numbers and use number[0],number[1],number[2],...............number[99].

Declaring Array Variables:

you must declare a variable to reference the array , and you must specify the type of array the variable can reference.

syntax:

dataType[] arrayRefVar;

or

dataType arrayRefVar[];

**Example:**

double[] myList;

or

double myList[];

Creating Arrays:

you can create an array by using the **new** operator

syntax:

arrayRefVar = new dataType[arraysize];

two things:

1.It creates an array using new data Type[arraySize].

2.It assign the reference of the newly created array to the variable arrayRefVar.

datatype[] arrayRefVar = new dataType[arraySize];

dataType[] arrayRefVar= {value0,value1,value2......,valueN};

Array indicates are 0- based;

start from 0 to **arrayRefVar.length-1**.

Example:

double[] myList = new double[10];

Processing Arrays:

we often use either **for** loop or **foreach** loop because all of the elements in an array of the same type and the size of the array.

for loop:

syntax:

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

foreach loop:

syntax:

for(double element : myList )

Passing Arrays to Methods:

you can also pass arrays to method

example:

void printArray(int[] array){

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

system.out.println(array[i]+" ");

}

}

you can invoke it by passing an array .

example:

the printArray method to display 3,1,2,6,4, and 2

printArray(new int[] {3,1,2,6,4,2});

**Returning an Array from a Method:**

A method may also return an array.

Example:

The method returns an array that is the reverse of another array.

public static int[] reverse(int[] list) {

int[] result= new int[list.length];

for(int i=0,j=result.length-1; i< list.length; i++,j--) {

result[j]=list[i];

}

return result;

}

The Array Class:

The java.util.Arrays class contains various static methods for sorting and searching arrays, comparing arrays and filling array elements.

these methods are overloaded for all primitive types(data types).

|  |  |
| --- | --- |
| Methods | Description |
|  |  |
|  |  |
|  |  |
|  |  |

java-Date and Time

java provides the date and time class available in java.util package, this class encapsulates the current date and time.

|  |  |
| --- | --- |
| Constructor | Description |
| Date() |  |
| Date(long millisec) |  |

|  |  |
| --- | --- |
| Method | Description |
| boolean after(Date date) |  |
| booleanbefore(Date date) |  |
| Object clone() |  |
| int compareTo(Date date) |  |
| int compareTo(Object obj) |  |
| boolean equals(Object obj) |  |
| long getTime() |  |
| int hashCode() |  |
| void setTime(long time) |  |
| String toString() |  |

Getting Current Date and Time:

this is a very easy method to get current date and time in java. You can use a simple Date object with toSting() method .

program:

import java.util.date;

public class dateDemo {

public static void main(string args[]) {

Date date = new Date();

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

}

}

Date Comparison:

* getTime()
* compareTo()
* before(),after() and equals()

Date Formatting Using SimpleDateFormat:

It is a concrete class for formatting and parsing dates in a locale-sensitive manner.

it is also used any user-defined patterns for date-time formatting.

program:

import java.util.\*;

import java.util.\*;

public class DateDemo{

public static void main(String args[]){

Date dnow=new Date();

SimpleDateFormat ft= new SimpleDateFormat("E yyyy.mm.dd 'at' hh.mm.ss. a zzz");

System.out.println("Current Date:" +ft.format(dnow));

}

}

Simple DateFormat Format Codes:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Character | Description | | Example | |
| G |  | |  | |
| Y |  | |  | |
| M |  | |  | |
| D |  | |  | |
| H |  | |  | |
| M |  | |  | |
| S |  | |  | |
| S |  | |  | |
| E |  | |  | |
| D |  | |  | |
| F |  | |  | |
| W |  | |  | |
| W |  | |  | |
| A |  | |  | |
| K |  | |  | |
| K |  | |  | |
| Z |  | |  | |
| ' |  | |  | |
| '' |  | |  | |
|  | |  | |

Date formatting using printf:

you use a two-letter format, Staring with t and ending in one of the letters of the table.

program:

import java.util.Date;

public class DateDemo {

Date date = new Date();

String str = String. format("Current Date/Time : %tc ", date);

System.out.printf(str);

}

}

A format string can indicate the index of the argument to be formatted.

The index must immediately follow the % and it must be terminated by a $.

Example:

import java.util.Date;

public class DateDemo {

public static void main(String args[]) {

Date date=new Date();

System.out.printf("%1$s %2$tB %2$td, %2$tY", "Due date:",date);

}

}

you can use the < flag. It indicates that the same argument as in the preceding format.

import java.util.Date;

public class DateDemo {

public static void main(String args[]) {

Date date= new Date();

System.out.printf("%s %tB %<te, %<tY", "Due date:", date);

}

}

Date and Time Conversion Characters:

|  |  |  |
| --- | --- | --- |
| Character | Description | Example |
| C |  |  |
| F |  |  |
| D |  |  |
| T |  |  |
| R |  |  |
| R |  |  |
| Y |  |  |
| Y |  |  |
| C |  |  |
| B |  |  |
| B |  |  |
| M |  |  |
| D |  |  |
| E |  |  |
| A |  |  |
| A |  |  |
| J |  |  |
| H |  |  |
| K |  |  |
| I |  |  |
| I |  |  |
| M |  |  |
| S |  |  |
| L |  |  |
| N |  |  |
| P |  |  |
| P |  |  |
| Z |  |  |
| Z |  |  |
| S |  |  |
| Q |  |  |

Parsing Strings into Dates:

program:

import java.util.\*;

import java.text.\**;*

*public class DateDemo{*

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

*SimpleDateFormat ft= new SimpleDateFormat("yyyy-MM*-dd");

String input= args.length == 0 ? "1818-11-11" : args[0];

System.out.print(input+"Parses as");

Date t;

try {

t= ft.parse(input);

System.out.println(t);

}

catch(ParseException e) {

System.out.println("Unparseable using "+ft);

}

}

}

Sleeping for a While:

sleep for any period of time from one millisecond up to the lifetime of your computer.

program:

import java.util.\*;

public class sleepDemo{

public static void main(String args[]) {

try{

system .out .println(new Date( ) + "\n");

Thread .sleep(5\*60\*10);

System .out .println("Got an exception!");

}

}

}

**Measuring Elapsed Time:**

you may need to measure point in time in milliseconds.

program:

import java.util.\*;

public class DiffDemo{

public static void main(String args[]) {

try{

long start=Syste.currentTimeMilli( );

system .out .println(new Date( ) + "\n");

Thread .sleep(5\*60\*10);

system .out .println(new Date( ) + "\n");

long end=System.current.TimeMillis( );

long diff=end-start;

System.out.println("Difference is: "+diff);

} catch (Exception e) {

System .out .println("Got an exception!");

}

}

}

Gregorian Calendar Class:

It is a concrete implementation of a calendar class that implementation the normal gregorian calendar with which you are familiar.

getInstance( ) Method:

Calendar returns a gregorian Calendar initialized with the current date and time in the default locale and time zone.

two fields:

1.AD

2.BC

|  |  |
| --- | --- |
| Constructor | Description |
| GregorianCalendar( ) |  |
| GregorianCalendar(int year,int month, int date) |  |
| GregorianCalendar(int year, int month, int date, int hour, int minute) |  |
| GregorianCalendar(int year,int month,int date, int hour, int minute, int second) |  |
| GregorianCalendar(Locale alocale) |  |
| GregorianCalendar(TimeZone zone) |  |
| GregorianCalendar(TimeZone zone, Locale alocale) |  |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

java-Regular Expression

java provides the java.util.regex package for pattern matching with regular expressions.

java regular expressions are very similar to the perl programming language .

A special sequence of characters

find other string

sets of strings

They can be used to search, edit or manipulate text and data.

three classes:

1.Pattern class:

first invoke one of its public static **compile()** methods, which will then return a pattern object.

These methods accept a regular expression as the first argument.

The pattern class provides no public constructors

2.Matcher Class:

A Matcher object by invoking the **matcher()** method on a pattern object.

Matcher defines no public Constructors.

A matcher object is the engine that interprets the pattern and performs match operations against an input string.

3.PatternSyntaxException:

object is an unchecked exception that indicates a syntax error in a regular expression pattern.

**Capturing Groups:**

Capturing groups are a way to treat multiple characters as a single unit.

They are created by placing the characters to be grouped inside a set of parentheses.

Capturing groups are numbered by counting their opening parentheses from the left to the right.

In the expression ((A)(B(c)))

these are four groups:

1.(A(B(C)))

2.(A)

3.(B(C))

4.(c)

Regular Expression syntax:

|  |  |  |
| --- | --- | --- |
| Subexpression | Matches |  |
| ^ |  |  |
| $ |  |  |
| .  [...] |  |  |
| [^...] |  |  |
| \A |  |  |
| \z |  |  |
| \Z |  |  |
| re\* |  |  |
| re+ |  |  |
| re? |  |  |
| re{n} |  |  |
| re{n,} |  |  |
| re{n,m} |  |  |
| a|b |  |  |
| (re) |  |  |
| (?:re) |  |  |
| \w |  |  |
| \W |  |  |
| \s |  |  |
| \S |  |  |
| \d |  |  |
| \D |  |  |
| \A |  |  |
| \Z |  |  |
| \z |  |  |
| \G |  |  |
| \n |  |  |
| \b |  |  |
| \B |  |  |
| \n,\t, etc. |  |  |
| \Q |  |  |
| \E |  |  |

Methods of the Matches:

a list of useful instance methods

Index Methods:

Index methods provide useful index values that show precisely where the match was found in the input string.

|  |  |
| --- | --- |
| Method | Description |
| public init start( ) |  |
| public init start(int group) |  |
| public int end |  |
| public int end(int group) |  |

study Methods:

Study methods review the input string and return a Boolean indicating whether or not the pattern is found.

|  |  |
| --- | --- |
| Method | Description |
| public boolean lookingAt( ) |  |
| public boolean find( ) |  |
| public boolean find(int start) |  |
| public boolean matches( ) |  |

Replacement Methods:

Replacement methods are useful methods for replacing text in an input string.

|  |  |
| --- | --- |
| Method | Description |
| public Matcher append Replacement( StringBuffer sb, String replacement) |  |
| public StringBuffer appendTail(StringBuffer sb) |  |
| public String replaceFirst(String replacement) |  |
| public String replaceAll(String replacement) |  |
| Public static String quoteReplacement(String s) |  |

The start and end Method:

count the number of times the word "cat" appers in the input.

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches {

private static final String REGEX = "\\bcat\\b";

private static final String INPUT = "cat cat cat cattie cat";

public static void main( String args[] ) {

Pattern p = Pattern.compile(REGEX);

Matcher m = p.matcher(INPUT); // get a matcher object

int count = 0;

while(m.find()) {

count++;

System.out.println("Match number "+count);

System.out.println("start(): "+m.start());

System.out.println("end(): "+m.end());

}

}

}

The matches and lookingAt Methods:

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches {

private static final String REGEX = "foo";

private static final String INPUT = "fooooooooooooooooo";

private static Pattern pattern;

private static Matcher matcher;

public static void main( String args[] ) {

pattern = Pattern.compile(REGEX);

matcher = pattern.matcher(INPUT);

System.out.println("Current REGEX is: "+REGEX);

System.out.println("Current INPUT is: "+INPUT);

System.out.println("lookingAt(): "+matcher.lookingAt());

System.out.println("matches(): "+matcher.matches());

}

}

The replaceFirst and replaceALL Methods:

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches {

private static String REGEX = "dog";

private static String INPUT = "The dog says meow. " + "All dogs say meow.";

private static String REPLACE = "cat";

public static void main(String[] args) {

Pattern p = Pattern.compile(REGEX);

// get a matcher object

Matcher m = p.matcher(INPUT);

INPUT = m.replaceAll(REPLACE);

System.out.println(INPUT);

}

}

The appendReplacement and appendTail Methods:

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches {

private static String REGEX = "a\*b";

private static String INPUT = "aabfooaabfooabfoob";

private static String REPLACE = "-";

public static void main(String[] args) {

Pattern p = Pattern.compile(REGEX);

// get a matcher object

Matcher m = p.matcher(INPUT);

StringBuffer sb = new StringBuffer();

while(m.find()) {

m.appendReplacement(sb, REPLACE);

}

m.appendTail(sb);

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

}

}

PatternSyntaxException Class Methods:

A PatternSyntaxException is an unchecked exception that indicates a syntax error in a regular exception pattern.

|  |  |
| --- | --- |
| Method | Description |
| public String getDescrition( ) |  |
| public int getIndex( ) |  |
| public String getPattern( ) |  |
| public String getMessage( ) |  |

HTML5Methods

A java method is a collection of statements that are grouped together to perform an opertion.

System.out.**println()** Methods:

System--> The system actually executes several statements in order to display a message on the console.

**how to create your own method:**

1.your own methods with or within return values.

2.invoke a method with or without parameters ,

3.apply method abstraction in the program design

**Creating Method:**

syntax:

public static int methodName(int a, int b) {

//body

}

Here,

* **public static---->** modifier
* **int** --->return type
* **methodName--->**name of method
* **a,b--**>**formal parameters**
* **int a, int b-->**list of parameters

Syntax:

**modifier returnType nameofMethod (Parameter List) {**

**//method body**

**}**

* **modifier**-It defines the access type of the method and it is optional to use.
* **return -**
* **nameofMethod-**
* **Parameter-**
* **method body-**

example:

public static int minFunction(int n1, int n2) {

int min;

if(n1>n2)

min=n2;

else

min=n1;

return min;

}

**Method calling:**

The process of method calling is simple.

* the return statement is executed.
* it reaches the method ending closing brace.

The methods returning void is considered as call to a statement.

**System.out.println("This is kalpana");**

The method returning value .

**int result = sum(6,9);**

Example:

public class MinNumber {

public static void main(String[] args) {

int a=11;

int b=6;

int c=minFunction(a,b);

system.out.println("Minimum Value=" + c);

}

public static int minFunction(int n1,int n2)

int min;

if( n1 > n2)

min =n2;

else

min = n1;

return min;

}

}

**The Void keyword:**

The void keyword allows us to create methods which do not return a value.

public class ExampleVoid {

public static void main(String[] args) {

methodRankPoints(255.7);

}

public static void methodRankPoints(double points) {

if (points >= 202.5) {

System.out.println("Rank:A1");

}else if (points >= 122.4) {

System.out.println("Rank:A2");

}else {

System.out.println("Rank:A3");

}

}

}

**Passing Parameters by value:**

public class ExampleVoid {

public static void main(String[] args) {

methodRankPoints(255.7);

}

**Passing Parameters by Value:**

While working under calling process, arguments is to be passed.Parameters can be passed by value or by reference.

Passing Parameters by Value means calling a method with a parameter. Through this, the argument value is passed to the parameter.

Example:

public class swappingExample {

public static void main(String[] args) {

int a = 30;

int b = 45;

System.out.println("Before swapping, a = " + a + " and b = " + b);

// Invoke the swap method

swapFunction(a, b);

System.out.println("\n\*\*Now, Before and After swapping values will be same here\*\*:");

System.out.println("After swapping, a = " + a + " and b is " + b);

}

public static void swapFunction(int a, int b) {

System.out.println("Before swapping(Inside), a = " + a + " b = " + b);

// Swap n1 with n2

int c = a;

a = b;

b = c;

System.out.println("After swapping(Inside), a = " + a + " b = " + b);

}

}

Method overloading:

when a class has two or more methods by the same name but different parameters, it is known as method overloading.

In overriding, a method has the same method name, type, number of parameters, etc.

public class ExampleOverloading {

public static void main(String[] args) {

int a = 11;

int b = 6;

double c = 7.3;

double d = 9.4;

int result1 = minFunction(a, b);

// same function name with different parameters

double result2 = minFunction(c, d);

System.out.println("Minimum Value = " + result1);

System.out.println("Minimum Value = " + result2);

}

// for integer

public static int minFunction(int n1, int n2) {

int min;

if (n1 > n2)

min = n2;

else

min = n1;

return min;

}

// for double

public static double minFunction(double n1, double n2) {

double min;

if (n1 > n2)

min = n2;

else

min = n1;

return min;

}

}

Using Command -Line Argument:

A command-line argument is the information that directly follows the program's name on the command line when it is executed.

To access the command-line arguments inside a java program is quite easy.

They are stored as strings in the String array passed to main( ).

Constructor:

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method.

however, constructor have no explicit return type.

Java automatically provides a default constructor that initializes all member variables to zero.

However, once you define your own constructor, the default constructor is no longer used.

Example:

// A simple constructor.

class MyClass {

int x;

// Following is the constructor

MyClass() {

x = 10;

}

}

to call constructor to initialize objects:

public class ConsDemo {

public static void main(String args[]) {

MyClass t1 = new MyClass();

MyClass t2 = new MyClass();

System.out.println(t1.x + " " + t2.x);

}

}

Parameterized Constructor:

A constructor that accepts one or more parameters.

Parameters are added to a constructor in the same way that they are added to a method.

Inside the parentheses after the constructor's name.

**The This keyword:**

**this** is a keyword in java

which is used as a reference to the object of the current class, with in an instance method or a constructor.

Using **this**  you can refer the members of a class such as constructors, variables and methods.

NOTE:

it is used only within instance methods or constructors.

|  |
| --- |
| This |

|  |
| --- |
| Instance variable |
| Constructor |
| Methods |

----------->

In general,

* differentiate the instance variable from local variable

If they have same names, within a constructor or a method.

class Student {

int age;

Student(int age) {

this.age = age;

}

}

* parametrized constructor or default:

Call one type of constructor from other in a class. It is known as explicit constructor invocation.

class Student {

int age

Student() {

this(20);

}

Student(int age) {

this.age = age;

}

}

**Variable Arguments( var-args):**

a variable number of arguments of the same type to a method

The parameter in the method is declared

syntax:

typeName....parameterName

public class VarargsDemo {

public static void main(String args[]) {

// Call method with variable args

printMax(34, 3, 3, 2, 56.5);

printMax(new double[]{1, 2, 3});

}

public static void printMax( double... numbers) {

if (numbers.length == 0) {

System.out.println("No argument passed");

return;

}

double result = numbers[0];

for (int i = 1; i < numbers.length; i++)

if (numbers[i] > result)

result = numbers[i];

System.out.println("The max value is " + result);

}

}

The finalize( ) Method:

An object's final destruction by the garbage collector. This is called **finalize() ,**and it can be used to ensure that an object terminates cleanly.

For example, you might use finalize( ) to make sure that an open file owned by that object is closed.

syntax:

protected void finalize( ) {

//finalization code here

}

here,

HTML-fILES and I/O

All these streams represent an input source and an output destination

The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in java.

**Stream:**

A stream can be defined as a sequence of data.

These are two kinds of stream

* InPutStream
* OutPutStream

**Destination**

**Program**

**source**

Java provides strong but flexible support for I/O related to files and networks.

**Byte Streams:**

Java byte streams are used to perform input and output of 8-bit bytes.

The most frequently used classes are, **FileInputStream**  and **FileOutputStream.**

There are many classes related to byte streams .

**Character Streams:**

java character streams are used to perform input and output for 16-bit unicode.

The most frequently used classes are, **FileReader** and **FileWriter.**

internally Filereader uses FileInputStream and FileWriter uses FileOutputStream .

The major difference is that FileReader reads two bytes at a time and FileWriters writes two bytes at a time.

**Standard Streams:**

C,C++ programming languages, then you must be aware of three standard devices STDIN,STDOUT,STDERR

java languages, then you must be aware of three standard devices

* **Standard Input**
* **Standard Output**
* **Standard Error**

Reading and Writing Files:

A stream can be defined as a sequence of data.

The **InputStream** is used to read data from a source .

The **OutputStream** is used to writing data to a destination.

FileInputStream

ByteArrayInputStream

FilterInputStresm

InputStream

ObjectInputStreammm

object

OutputStream

The two important streams are **FileInputStream** and **FileOutputStream**

**FileInputStream:**

This stream is used for reading data from the files.

Objects can be created using the keyword **new .**

There are several types of constructors.

Constructor takes a file name as a string to create an input stream object to read the file.

syntax:

InputStream f = new FileInputStream("C:/java/hello");

Constructor takes a file object to create an input stream object to read the file.

First we create a file object using File( ) method

syntax:

File f= new File("C:/java/hello");

InputStream f = new FileInputStream(f);

|  |  |
| --- | --- |
| Methods | Descrition |
| publicvoid close( ) throw IOException{} |  |
| protected void finalize( ) throws IOException |  |
| public int read(int r) throws IOException{} |  |
| public int read(byte[] r) throws IOException{} |  |
| public int available( ) throws IOException{} |  |

FileOutputStream:

It is used to create a file and write data into it.

* The stream would create a file.
* If it doesn't already exit
* Before opening it for output.

Two constructors which can be used to create a FileOutputStream object.

constructor takes a file name as a string to create an input stream object to write a file.

OutputStream f=new FileOutputStream("C:/java/hello")

constructor takes a file object to create an output stream object to write a file.

File f=new file("C:/java/hello");

OutputStream f= new FileOutputStream(f);

|  |  |
| --- | --- |
| Method | Description |
| public void close( ) throws IOException{ } |  |
| protected void finalize( ) throws IOException{ } |  |
| public void write(int w)throws IOException{ } |  |
| public void write(byte[] w) |  |

import java.io.\*;

example of both InputStream and OutputStream:

public class fileStreamTest {

public static void main(String args[]) {

try {

byte bWrite [] = {11,21,3,40,5};

OutputStream os = new FileOutputStream("test.txt");

for(int x = 0; x < bWrite.length ; x++) {

os.write( bWrite[x] ); // writes the bytes

}

os.close();

InputStream is = new FileInputStream("test.txt");

int size = is.available();

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

System.out.print((char)is.read() + " ");

}

is.close();

} catch (IOException e) {

System.out.print("Exception");

}

}

}

File Navigation and I/O:

the basic of file navigation and I/O

* File Class
* FileReader class
* FileWriter class

Directories in java:

A directory is a File which can contain a list of other files and directories.

You use **File** object

* create directories
* list down files available in a directory.

**Creating Directories:**

There are two useful **File** utility methods

1. **mkdir( )** method
2. **mkdirs( )** method
3. Following example creates "/tmp/user/java/bin" directory −
4. **Example**
5. import java.io.File;
6. public class CreateDir {
7. public static void main(String args[]) {
8. String dirname = "/tmp/user/java/bin";
9. File d = new File(dirname);
11. // Create directory now.
12. d.mkdirs();
13. }
14. }

Compile and execute the above code to create "/tmp/user/java/bin".

**Note:** java automatically takes care of path separators on UNIX and windows as per conventions. If you use a forward slash (/) on a windows version of java, the path will still resolve correctly.

Listing Directories:

you can use **List( )** method provided by  **File** object to list down all the files and directories available in a directory.

import java.io.File;

public class ReadDir {

public static void main(String[] args) {

File file = null;

String[] paths;

try {

// create new file object

file = new File("/tmp");

// array of files and directory

paths = file.list();

// for each name in the path array

for(String path:paths) {

// prints filename and directory name

System.out.println(path);

}

} catch (Exception e) {

// if any error occurs

e.printStackTrace();

}

}

}

JAVA-Exception

An exception is a problem that arises during the execution of a program.

When an exception occurs the normal flow of the program is disrupted and the program/Application.

An exception can occur for many different resons:

1. A user has entered an invalid data.
2. A file that needs to be opened can't found.
3. A network connection has been lost in the middle of communications or the JVM has run out of memory.

We have three categories of Exceptions:

* Checked Exception
* unchecked Exception
* Errors

**Checked Exception:**

A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions.

For example

if you use **FileReader**  class in your program to read data from a file.

If the file doesn't exist, then a FileNotFoundException occurs, and the complier programmer to handle the exception.

import java.io.File;

import java.io.FileReader;

public class FilenotFound\_Demo {

public static void main(String args[]) {

File file = new File("E://file.txt");

FileReader fr = new FileReader(file);

}

}

C:\>javac FilenotFound\_Demo.java

FilenotFound\_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown

FileReader fr = new FileReader(file);

^

1 error

Unchecked Exception:

An unchecked exception is an exception that occurs at the time of execution.

These are also called as **Runtime Exceptions**.

Runtime exceptions are ignored at the time of compilation.

These include programming bugs, such as logical errors or improper use of an API.

**Errors:**

Errors are typically ignored in your code because you can rarely do anything about an error.

For example ,

If a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

**Exception Hierarchy:**

All exception classes are subtypes of the java.lang.Exception class.

The exception class is a subclass of the Throwable class.

The exception class is another subclass called Error which is derived from the Throwable class.

The Exception class has two main subclasses:

* IOException class
* RuntimeException class

java.lang

Object

Throwable

Errors

Exception

RunTime Exception

other Exception

|  |  |
| --- | --- |
| Method | Description |
| public String getMessage() |  |
| public Throwable getCause() |  |
| public String toString() |  |
| public void printStackTrace() |  |
| public StackTraceElement[] getStackTrace() |  |
| public Throwable fillInStackTrace() |  |

Catch Exception:

A method catches an exception using a combination of the **try** and **catch** keywords.

Code within a try/catch block is referred to as protected code.

Syntax:

try{

//protected code

} catch (ExceptionName e1) {

//Catch block

}

Every try block should be immediately followed either by a catch block or finally block.

If an exception occurs in protected code, the catch block that follows the try is checked.

If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example:

// File Name : ExcepTest.java

import java.io.\*;

public class ExcepTest {

public static void main(String args[]) {

try {

int a[] = new int[2];

System.out.println("Access element three :" + a[3]);

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Exception thrown :" + e);

}

System.out.println("Out of the block");

}

}

Multiple Catch Blocks:

syntax:

try {

// protected code

} catch (ExceptionType1 e1){

//Catch block

} catch (ExceptionType2 e2) {

// Catch block

} catch (ExceptionType3 e3) {

//Catch block

}

If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops exception and the exception is thrown down to the privous method on the call stack.

Exception:

try {

file = new FileInputStream(fileName);

x=(byte) file.read( );

} catch (IOException i) {

i.printStackTrace( );

return -1;

} catch (FileNotFoundException f)// not valid!

f.printStackTrace( );

return -1;

}

Throws/Throw keywords:

If a methods does not handle a checked ecception, the method must declare it using the  **throws**  keyword.

The throws keyword appears at the end of a method's signature.

You can throw an exception , either a newly instantiated one or an exception that you just caught, by using the  **throw**  keyword.

Question:

The difference between throws and throw keywords?

|  |  |
| --- | --- |
| Throws | Throw |
| It is used to postpone the handling of a checked exception | It is used to invoke an exception explicitly |

Example:

method declares that it throws a RemoteException:

import java.io.\*;

public class className {

public void deposit(double amount) throws RemoteException {

//Method implementation

throw new Remote Exception( );

}

//Remainder of class definition

}

method declares that it throws a RemoteException and an InsufficientFundsException:

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas.

Example:

import java.io.\*;

public class className {

public void withdraw(double amount) throws RemoteException, InsufficientFundsException {

// Method implemention

}

//Remainder of class definition

}

Finally Block:

A finally block of code always execute, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

syntax:

try{

//protected code

} catch (ExceptionType1 e1){

// Catch block

} catch (ExceptionType2 e2) {

//Catch block

} catch (exceptionType3 e3) {

//Catch block

}finally {

// the finally block always executes.

}

Example:

public class ExcepTest {

public static void main(String args[]){

int a[]=new int[2];

try{

system.out.println("Access element three :" + a[3]);

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Exception thrown :" +e);

} finally {

a[0]=6;

System.out.println("First element value: " +a[0]);

System.out.println("The finally statement is executed");

}

}

}

Note:

1. A catch clause cannot exist without a try statement.
2. It is not compulsory to have finally clauses whenever a try/catch block is present.
3. The try block cannot be present without either catch clause or finally clause.
4. Any code cannot be present in between the try, cat, finally blocks.

try-with-resource:

When we use any resources like streams, connections etc. We have to close them explicitly using finally block.

Example:

import java.io.File;

import java.io.FileReader;

import java.io.IOException;

public class ReaderData\_Demo {

public static void main(String args[]){

FileReader fr=null;

File file=new File("file.txt");

fr=new FileReader(file); char [] a =new char[50];

fr.read(a);

for(char c:a)

System.out.print(c);

}catch (IOException e){

e.printStackTrace();

}finally {

try{

fr.close();

}catch (IOException ex) {

ex.printStackTrace();

}

}

}

}

**try-with-resources,** also referred as  **automatic resource management:**

it is a new exception handling mechanism that was introduced in java 7, which automatically closes the resources used within the try catch block.

to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block.

Syntax:

try(FileReader fr= new FileReader("file path")) {

// use the resource

} catch( ){

//body of catch

}

}

Example:

import java.io.FileReader;

import java.io.IOException;

public class Try\_withDemo{

public static void main(string args[]) {

try(FileReader fr= new FileReader("E://file.text")){

char[] a= new char[50];

fr.read(a);

for(char c:a)

System.out.print(c);

} catch (IOException e) {

e.printStackTrace();

}

}

}

Following points are to be kept oin mind while working with  **try-with-resources:**

1. To use a class with try-with-resources statement it should implement b **AutoCloseable** interface and the **close()**  method of it gets invoked automatically at runtime.
2. you can declare more than one class in try-with-resources statement.
3. while you declare multiple classes in the try block to try-with-resources statement these classes are closed in reverse order.
4. Except the declaration of resource within the parenthesis everything is the samer as normal try/catch block of a try block.
5. The resourcendeclared in try gets instantiated just before the start of the try-block.
6. The resource declared at the try block is implicitly declared as final.

User-defined Exception:

You can create your own exceptions in java:

* All exception must be a child of Throwable
* If you want to write a checked exception that is automatically enforced by the Handler or Declare Rule, you need to extend the Exception class.
* If you want to write a runtime exception, you need to extend the Runtime Exception class.

syntax:

class MyException extends Exception {

}

Common Exception:

it is possible to define two categories of Exception and Errors:

* **JVM Exception**
* **Programmatic Exceptions**

JVM Exception:

These are exceptions/errors that are exclusively or logically thrown by the JVM.

Example:

NullPointerException

ArrayIndexOutOfBoundsException

ClassCastException

Programmatic Exception:

These exception are thrown explicitly by the application or the API programmers.

Example:

IllegalArgumentException

IllegalStateException.

JAVA-INNER CLASS

The class written within is called the **nested class,** and the class that holds the inner class is called the **outer class.**

syntax:

class Outer\_Demo{

class Nested\_Demo{

}

}

Nested classes are divided into two types

* **Non-static nested classes**
* **static nested classes**

Nested class

static classes

Non-static classes

Method local

inner classes

Inerr

Inner classes

Inner classes (or) Non-static classes:

An inner class can be **private** cannot and once you declare an inner class private, it can't be accessed from an object outside the class.

Example:

class Outer\_Demo{

int num;

// inner class

private class Inner\_Demo {

public void print() {

system.out.println("this is an inner class");

}

}

// Accessing the inner class from the method within

void display\_Inner() {

Inner\_Demo inner= new Inner\_Demo();

inner.print();

}

}

public class My\_class {

public static void main(String args[]) {

// Instantiating the outer class

Outer\_Demo outer=new Outer\_Demo();

//Accessing the display\_Inner() method.

outer.display\_Inner();

}

}

Here,

**Outer\_Demo -->** it is the outer class

**Inner\_Demo --->** it is the inner class

**display\_inner()** ---> it is the method inside which we are instantiating the inner class, and this method is invoked from the **main** method.

**Output:**

This is an inner class

**Accessing the private Members:**

A class is having private members to access them.

return the private members from a method within the inner class, **getValue()** , and finnaly from another class call the getValue() method of the inner class.

Initially you have to instantiate the outer class.

using the object of the outer class

**Outer\_Demo outer=new Outer\_demo();**

**Outer\_Demo.Inner\_Demo inner= outer.new Inner\_Demo();**

To access the private members of a class using inner class:

Eample:

class Outer\_Demo {

//private variable of the outer class

private int num=175;

//inner class

public class Inner\_Demo {

public class Inner\_Demo {

public int getNum() {system.out.println("This is the getnum method of the inner class");

return num;

}

}

}

public class My\_class2 {

public static void main(String args[]) {

// Instantiating the outer class

Outer\_Demo outer= new Outer\_Demo( );

//Instantiating the inner class

Outer\_Demo .Inner\_Demo inner = outer.new Inner\_Demo( );

System.out.println(inner.getNum( ));

}

}

**Output:**

This is the getnum method of the inner class:175

Method-local Inner class:

Like local variable, the scope of the inner class is retricted within the method.

Example:

public class Outerclass {

// instance method of the outer class

void my\_Method( ) {

int num=23;

// method-local inner class

class MethodInner\_Demo {

public void print( ) {

System.out.println("This is method inner class "+num);

}

} // end of inner class

//Accessing the inner class

MethodIner\_Demo inner= new MethodInner\_Demo ( );

inner.print( );

}

public static void main(String args[ ]) {

Outerclass outer= new Outerclass( );

outer.my\_Method ( );

}

}

**Output:**

This is method inner class 23

Anonymous Inner class:

An inner class declared without a classname .

Generally, They are used whenever you need to override the method of a method of a class or an interface.

Syntax:

AnonymousInner an\_inner = new AnonymousInner( ) {

public void my\_method( ) {

}

};

Example:

abstact class AnonymousInner {

public abstract void mymethod( );

}

public class Outer\_class {

public static void main(String args[ ] ) {

AnonymousInner inner = new AnonymousInner( ) {

public void myMethod( ) {

system.out.println("This is an example of anonymous inner class");

}

};

inner.mymethod( );

}

}

Anonymous Inner Class as Argument:

Generally, if a method accepts an object of an interface, an abstract class, or a concrete class, then we can implement the interface, extend the abstract class, and pass the object to the method.

Syntax:

passing an anonymous inner classs as a method argument-

obj.my\_Method(new My\_Class() {

public void Do( ) {

}

});

Example:

// interface

interface Message {

String greet();

}

public class My\_class {

// method which accepts the object of interface Message

public void displaymessage(Message m) {

System.out.println(m.greet( ) +", This is an example of anonymous inner class as an argument");

}

public static void main(String args[ ]) {

//Instantiating the class

My\_class obj= new My\_class( );

// Passing an anonymous inner class as an argument\obj.displayMessage(new Message( ) {

public String greet() {

return "Hello";

}

});

}

}

Static Nested Class:

It can be accessed without instantiating the outer class, using other static members. A static nested class does not have access to the instance variables and methods of the outer class.

syntax:

class MyOuter {

static class Nested\_Demo {

}

}

Instantiating a static nested class is a bit different from instantiated an inner class.

Example:

public class Outer {

ststic class Nested\_Demo {

public void my\_method( ) {

System.out.println("This is my nested class");

}

}

public static void main(String args[ ]) {

Outer.Nested\_Demo nested=new Outer.Nested\_Demo( );

nested.my\_method( );

}

}

Java-Inheritance

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another.

The class which inherits the properties of other is known as subclass(derived class, child class) and the class whose properties are inherited is known as super class(base class, parent class).

extends keyword:

**extends** is the keyword used to inherit the properties of a class.

syntax:

class Super {

}

class Sub extends Super {

}

Example:

class Calculation {

int z;

public void addition(int x, int y) {

z = x + y;

System.out.println("The sum of the given numbers:"+z);

}

public void Subtraction(int x, int y) {

z = x - y;

System.out.println("The difference between the given numbers:"+z);

}

}

class My\_Calculation extends Calculation {

public void multiplication(int x, int y) {

z = x \* y;

System.out.println("The product of the given numbers:"+z);

}

public static void main(String args[]) {

int a = 20, b = 10;

My\_Calculation demo = new My\_Calculation();

demo.addition(a, b);

demo.Subtraction(a, b);

demo.multiplication(a, b);

}

}

**Output**

The sum of the given numbers:30

The difference between the given numbers:10

The product of the given numbers:200

super Keyword:

The super keyword is similar to this keyword.

the super keyword is used:

* It is used to **differentiate the members** of super class from the members of subclass, if they have same names.
* It is used to  **invoke the super class**  constructor from subclass.

Differentiate the members:

If a class is inheriting the properties of another class .

If the members of the super class have the names same as the sub class, to differentiate these variables .

super.variable;

super.method( );

**Example:**

**class** super\_class{

**int** num=20;

**void** display() {

System.***out***.println("This is the display method of super class");

}

}

**public** **class** SuperDemo **extends** super\_class{

**int** num=10;

**void** display() {

System.***out***.println("This is the display method of sub class");

}

**void** My\_method() {

SuperDemo sup=**new** SuperDemo();

sup.display();

**super**.display();

System.***out***.println("value of the variable named num is sub class"+sup.num);

System.***out***.println("value of the variable named num is super class"+**super**.num);

}

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

SuperDemo obj = **new** SuperDemo();

obj.My\_method();

}

}

Output:

This is the display method of sub class

This is the display method of super class

value of the variable named num is sub class10

value of the variable named num is super class20

Invoking Super Class Constructor:

If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the super class .

If you want to call a parameterized constructor of the super class.

super(values);

Example:

**class** Superclass{

**int** age;

**public** Superclass(**int** age) {

**this**.age=age;

}

**public** **void** getAge() {

System.***out***.println("the value of the variable named age is super class is"+age);

}

}

**class** SuperConstructor **extends** Superclass {

**public** SuperConstructor(**int** age) {

**super**(age);

}

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

SuperConstructor s=**new** SuperConstructor(24);

s.getAge();

}

}

Output:

the value of the variable named age is super class is 24

IS-A Relationship:

This object is a type of that object.

it is used to **extends keyword**

public class Animal{

}

Public class Mammal extends Animal{

}

public class Reptile extends Mammal{

}

public class Dog extends Mammal{

}

In Object-Oriented terms:

* Animal is the superclass of Mammal class.
* Animal is the superclass of Reptile class.
* Mammal and Reptile are subclasses of Animal class.
* Dog is the subclass of both Mammal and Animal classes.

Now, IS-A relationship:

* Mammal IS-A Animal
* Reptile IS-A Animal
* Dog IS-A Mammal
* Hence: Dog IS-A Animal as well

With the use of the extends keyword, the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclas.

Example:

**class** Animal{

}

**class** Mammal **extends** Animal{

}

**class** Reptile **extends** Animal{

}

**class** Dog **extends** Mammal{

}

**public** **class** InstanceofOperator **extends** Dog {

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

Animal a=**new** Animal();

Mammal m=**new** Mammal();

Reptile r=**new** Reptile();

Dog d= **new** Dog();

System.***out***.println(m **instanceof** Animal);

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

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

System.***out***.println(r **instanceof** Animal);

}

}

Output:

true

true

true

true

**implements** keyword:

It is used to get the IS-A relationship.

It is used with classes to inherit the properties of an interface

Interfaces can never be extended by a class.

Example:

**interface** Wildanimal{

}

**class** Supermammal **implements** Wildanimal{

}

**class** Reptail **implements** Wildanimal{

}

**class** Sweetydog **extends** Supermammal{

}

**public** **class** InstanceofOperatorUsingimplements **extends** Sweetydog {

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

InstanceofOperator i=**new** InstanceofOperator();

Supermammal m =**new** Supermammal();

Sweetydog d =**new** Sweetydog();

Reptile r =**new** Reptile();

System.***out***.println(i **instanceof** Wildanimal);

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

System.***out***.println(r **instanceof** Wildanimal);

}

}

Output:

false

true

false

HAS-A relationship:

These relationships are mainly based on the usage.

This relationship helps to reduce duplication of code as well as bugs.

Example:

Types of inheritance:



 This means that a class cannot extend more than one class.

Example:

public class extends Animal, Mammal{}

A class can implement one or more interfaces.

Java- Overriding

Overriding means to override the functionality of an existing method.

If a class inherits a method from its super class, then there is a chance to override the method provided that it is not marked final.

The benefit of overriding is ability to define a behaviour that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.

Example:

**class** Animal{

**public** **void** move() {

System.***out***.println("Animals can move");

}

}

**class** Dog **extends** Animal{

**public** **void** move() {

System.***out***.println("Dogs can walk and run ");

}

}

**public** **class** TestDog {

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

Animal a= **new** Animal();

Animal b=**new** Dog();// up casting//Animal reference but dog object

a.move();// runs the method in animal class

b.move();// runs the method in dog class

}

}

Output:

Animals can move

Dogs can walk and run

**Rules for Method Overriding:**

1. the argument list should be exactly the same as that of the overriden method.
2. the return type should be the same or a subtype of the return type declared in the original overriden method in the super class.
3. If the super class method is declared public then the overridding method in the sub class cannot be either private or protected.
4. Instance method can be overriden only if they are inherited by the subclass.
5. A method declared final can't be overriden.
6. A method declared static can't be overridden but can be re-declared.
7. If a method can't be inherited, then it can't be overridden.
8. A subclass within the same package as the instance's super class can override any super class method that is not declared private or final.
9. a subclass in a different package can only override the non-final methods declared public or protected.
10. Constructors can't be overridden.
11. An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.

Using **super** keyword in override:

Example:

**class** Animal

{

**public** **void** move() {

System.***out***.println("Animal can move");

}

}

**class** Dog **extends** Animal{

**public** **void** move() {

**super**.move();

System.***out***.println("Dog can walk and run");

}

}

**public** **class** TestDogUsingsuper {

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

Animal b= **new** Dog();// Animal reference but dog object

b.move();

}

}

Output:

Animal can move

Dog can walk and run

JAVA-POLYMORPHISM

polymorphism :

It is ability of an object to take on many forms.

When a parent class reference is used to reference is used to refer to a child class object.

Any java object that can pass more than one IS-A test is consider to be polymorphic.

To access an object is through a reference variable. The type of reference variable can't be changed.

The reference variable can be reassigned to other objects provided that it is not declared final.

Example:

public interface Vegetarian{}

public class Animal{}

public class Deer extends Animal implements Vegetarian{}

the deer class is considered to be polymorphic .

this has multi inheritance.

* A Deer IS-A Animal
* A Deer IS-A Vegetarian
* A Deer IS-A Deer
* A Deer IS-A Object

The reference variable facts to a Deer object reference.

Example:

Deer d=new Deer();

Animal a=d;

Vegetarian v=d;

object o=d;

All the reference variables d, a, v, o refer to the same Deer object in the heap.

Virtual Methods:

Example:

/\* File name : Employee.java \*/

public class Employee {

private String name;

private String address;

private int number;

public Employee(String name, String address, int number) {

System.out.println("Constructing an Employee");

this.name = name;

this.address = address;

this.number = number;

}

public void mailCheck() {

System.out.println("Mailing a check to " + this.name + " " + this.address);

}

public String toString() {

return name + " " + address + " " + number;

}

public String getName() {

return name;

}

public String getAddress() {

return address;

}

public void setAddress(String newAddress) {

address = newAddress;

}

public int getNumber() {

return number;

}

}

Now suppose we extend Employee class as follows −

/\* File name : Salary.java \*/

public class Salary extends Employee {

private double salary; // Annual salary

public Salary(String name, String address, int number, double salary) {

super(name, address, number);

setSalary(salary);

}

public void mailCheck() {

System.out.println("Within mailCheck of Salary class ");

System.out.println("Mailing check to " + getName()

+ " with salary " + salary);

}

public double getSalary() {

return salary;

}

public void setSalary(double newSalary) {

if(newSalary >= 0.0) {

salary = newSalary;

}

}

public double computePay() {

System.out.println("Computing salary pay for " + getName());

return salary/52;

}

}

Now, you study the following program carefully and try to determine its output −

/\* File name : VirtualDemo.java \*/

public class VirtualDemo {

public static void main(String [] args) {

Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);

Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);

System.out.println("Call mailCheck using Salary reference --");

s.mailCheck();

System.out.println("\n Call mailCheck using Employee reference--");

e.mailCheck();

}

}

This will produce the following result −

Output

Constructing an Employee

Constructing an Employee

Call mailCheck using Salary reference --

Within mailCheck of Salary class

Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--

Within mailCheck of Salary class

Mailing check to John Adams with salary 2400.0

JAVA-abstraction

Abstraction:

In oop's

it is a process of hiding the implementation details from the user, only the functionality will be provided to the user.

The user will have the information on what the object does instead of how it does it.

In Java,

abstraction is achieved using abstract classes and interfaces.

Abstract Class:

\* File name : Employee.java \*/

public abstract class Employee {

private String name;

private String address;

private int number;

public Employee(String name, String address, int number) {

System.out.println("Constructing an Employee");

this.name = name;

this.address = address;

this.number = number;

}

public double computePay() {

System.out.println("Inside Employee computePay");

return 0.0;

}

public void mailCheck() {

System.out.println("Mailing a check to " + this.name + " " + this.address);

}

public String toString() {

return name + " " + address + " " + number;

}

public String getName() {

return name;

}

public String getAddress() {

return address;

}

public void setAddress(String newAddress) {

address = newAddress;

}

public int getNumber() {

return number;

}

}

You can observe that except abstract methods the Employee class is same as normal class in Java. The class is now abstract, but it still has three fields, seven methods, and one constructor.

Now you can try to instantiate the Employee class in the following way −

/\* File name : AbstractDemo.java \*/

public class AbstractDemo {

public static void main(String [] args) {

/\* Following is not allowed and would raise error \*/

Employee e = new Employee("George W.", "Houston, TX", 43);

System.out.println("\n Call mailCheck using Employee reference--");

e.mailCheck();

}

}

When you compile the above class, it gives you the following error −

Employee.java:46: Employee is abstract; cannot be instantiated

Employee e = new Employee("George W.", "Houston, TX", 43);

^

1 error

Inheriting the Abstract class:

inherit the properties of Employee class just like concrete class

Example:

/\* File name : Salary.java \*/

public class Salary extends Employee {

private double salary; // Annual salary

public Salary(String name, String address, int number, double salary) {

super(name, address, number);

setSalary(salary);

}

public void mailCheck() {

System.out.println("Within mailCheck of Salary class ");

System.out.println("Mailing check to " + getName() + " with salary " + salary);

}

public double getSalary() {

return salary;

}

public void setSalary(double newSalary) {

if(newSalary >= 0.0) {

salary = newSalary;

}

}

public double computePay() {

System.out.println("Computing salary pay for " + getName());

return salary/52;

}

}

Here, you cannot instantiate the Employee class, but you can instantiate the Salary Class, and using this instance you can access all the three fields and seven methods of Employee class as shown below.

/\* File name : AbstractDemo.java \*/

public class AbstractDemo {

public static void main(String [] args) {

Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);

Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);

System.out.println("Call mailCheck using Salary reference --");

s.mailCheck();

System.out.println("\n Call mailCheck using Employee reference--");

e.mailCheck();

}

}

This produces the following result −

Output

Constructing an Employee

Constructing an Employee

Call mailCheck using Salary reference --

Within mailCheck of Salary class

Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--

Within mailCheck of Salary class

Mailing check to John Adams with salary 2400.0

Abstract Method:

The actual implementation of that method to be determined by child classes, the method in the parent class as an abstract:

* **abstract**  keyword is used to declare the method as abstract.
* the  **abstract** keyword before the method name in the method declaration
* the **abstract**  method contains a method signature, but no method body
* Instead of curly braces, an  **abstract**  method will have a semi colon ( ; )at the end.

Example:

public abstract class Employee {

private String name;

private String address;

private int number;

public abstract double computePay();

// Remainder of class definition

}

A method as abstract has two consequences:

* The class containing it must be declared as abstract.
* Any class inheriting the current class must either override the abstract method or declare itself as abstract.

Note:

Eventually , a descendant class has to implement the abstract method, otherwise you would have a hierarchy of abstract classes that can't be instantiated.

Salary class inherits the Employee class, it should implement the  **computePay( )**  method.

/\* File name : Salary.java \*/

public class Salary extends Employee {

private double salary; // Annual salary

public double computePay() {

System.out.println("Computing salary pay for " + getName());

return salary/52;

}

// Remainder of class definition

}

Java-Encapsulation

Encapsulation :

A Mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.

Data hiding:

The variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class.

To achieve encapsulation in java:

* Declare the variable of **a class as private**
* provide public **setter and getter methods** to modify and view **the variables values.**

Example:

\* File name : EncapTest.java \*/

public class EncapTest {

private String name;

private String idNum;

private int age;

public int getAge() {

return age;

}

public String getName() {

return name;

}

public String getIdNum() {

return idNum;

}

public void setAge( int newAge) {

age = newAge;

}

public void setName(String newName) {

name = newName;

}

public void setIdNum( String newId) {

idNum = newId;

}

}

/\* File name : RunEncap.java \*/

public class RunEncap {

public static void main(String args[]) {

EncapTest encap = new EncapTest();

encap.setName("James");

encap.setAge(20);

encap.setIdNum("12343ms");

System.out.print("Name : " + encap.getName() + " Age : " + encap.getAge());

}

}

Benefits of Encapsulation:

* The fields of a class can be made read-only or write-only.
* A class can have total control over what is stored in its feilds.

JAVA-Interface

An interface is a reference type in java.

it is similar to class,

I t is a collection of abstraction methods.

A class implements an interface, thereby inheriting the abstract methods of the interface.

interface along with abstract methods.

An interface may also contain:

* constants
* default methods
* static methods
* nested types

Method bodies exist only default methods and static methods.

A class describes the attributes and behaviours of an object.

An interface contains behaviours that a class implements.

An interface is similar to a class :

* An interface can contain any number of methods.
* An interface is written in a file with a.java extension, with the name of the interface matching the name of the file.
* the byte code of an interface appears in a .class file.
* Interface appear in packages, and their corresponding byte code file must be in a directory structure that matches the package name.

An interface is different to a class:

* you can't instantiate an interface
* An interface doesn't contain any constructors.
* All of the methods in an interface are abstract.
* An interface can't contain instance fields. The only fields that can appear in an interface must be declared both static and final.
* An interface is not extended by a class.
* It is implemented by a class
* An interface can extend multiple interfaces.

Declaring Interface:

The  **interface**  keyword is used to declare an interface.

Example:

/\* File name : NameOfInterface.java \*/

import java.lang.\*;

// Any number of import statements

public interface NameOfInterface {

// Any number of final, static fields

// Any number of abstract method declarations\

}

Interface have the following properties:

1. An interface is implicitly abstract.
2. you do not to use the **abstract** keyword while declaring an interface.
3. Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
4. Methods in an interface are implicitly public.

Example:

/\* File name : Animal.java \*/

interface Animal {

public void eat();

public void travel();

}

Implementing Interfaces:

A class uses the **implements**  keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

/\* File name : MammalInt.java \*/

public class MammalInt implements Animal {

public void eat() {

System.out.println("Mammal eats");

}

public void travel() {

System.out.println("Mammal travels");

}

public int noOfLegs() {

return 0;

}

public static void main(String args[]) {

MammalInt m = new MammalInt();

m.eat();

m.travel();

}

}

When override method defined in interface:

* Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method

(or)

Subclasses of those declared by the interface method.

* The signature of the interface method and the same type

(or)

Subtype should be maintained when override the methods.

* An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interface:

* A class can implement more than one interface at a time.
* A class can extend only one class, but implement many interfaces.
* An interface can extend another interface, in a similar way as a class can extend another class.

Extending Interfaces:

An interface can extend another interface in the same way that a class can extend another class.

The  **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

Example

// Filename: Sports.java

public interface Sports {

public void setHomeTeam(String name);

public void setVisitingTeam(String name);

}

// Filename: Football.java

public interface Football extends Sports {

public void homeTeamScored(int points);

public void visitingTeamScored(int points);

public void endOfQuarter(int quarter);

}

// Filename: Hockey.java

public interface Hockey extends Sports {

public void homeGoalScored();

public void visitingGoalScored();

public void endOfPeriod(int period);

public void overtimePeriod(int ot);

}

Extending Multiple interfaces:

A java class can only extend one parent class.

Multiple inheritance is not allowed.

Interface are not classes

An interface can extend more than one parent interface.

**The extend keyword** is used once, and **the parent interface** are declared in a **comma-separated list.**

Example:

public interface Hockey extends Sports, Event

Tagging Interface:

The most common use of extending interfaces occurs when the parent interface does not contain any methods.

Example:

the MouseListener interface in the java.awt.event package extended java.util.EventListener:

package java.util;

public interface EventListener

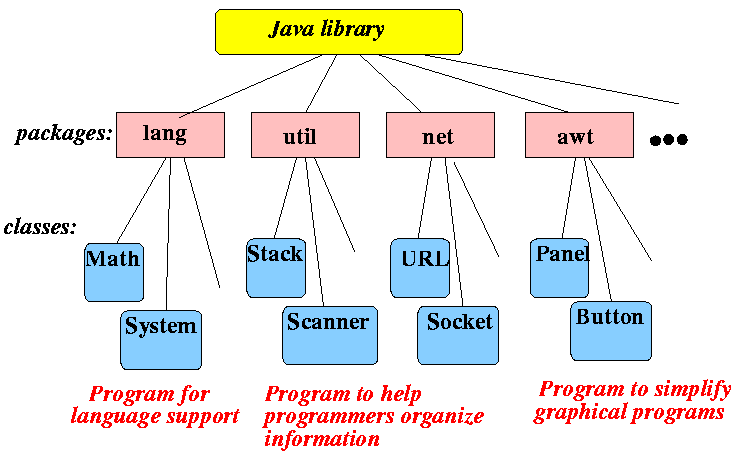
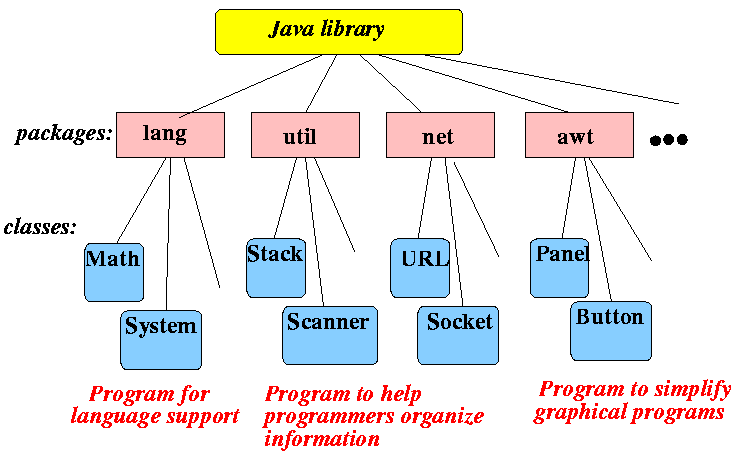
{}

An interface with no methods in it is referred to as a tagging interface.

Two basic design :

* Creates a common parent
* Adds a data type to a class
* **Creates a common parent** − As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.
* **Adds a data type to a class** − This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

JAVA-PACKAGES



**packages:**

they are used in java in order to prevent

naming conflicts,

to control access,

to make searching/locating and using of classes,

interfaces,

enumerations and annotation easier, etc.

A packages can be defined as a grouping of related types providing access protection and namespace management.

packages in java:

* **java.lang:**

bundles the fundamental classes

* **java.io**

classes for input , output functions are bundled in this package.

Creating a Package:

the package statement should be the first line in the source file. there can be only one package statement in each source file, and it applies to all types in the file.

While creating a package, you should choose a name for the package and include a **package**  statement a long with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that include in the package.

To compile the java programs with package statement:

javac -d Destination\_folder file\_name.java

Example:

/\* File name : Animal.java \*/

package animals;

interface Animal {

public void eat();

public void travel();

}

package animals;

/\* File name : MammalInt.java \*/

public class MammalInt implements Animal {

public void eat() {

System.out.println("Mammal eats");

}

public void travel() {

System.out.println("Mammal travels");

}

public int noOfLegs() {

return 0;

}

public static void main(String args[]) {

MammalInt m = new MammalInt();

m.eat();

m.travel();

}

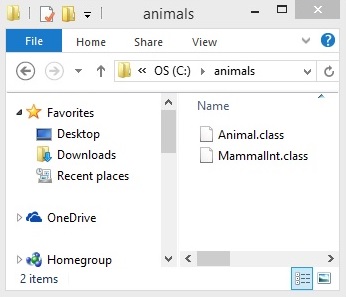
}

compile the java files:

$ javac -d . Animal.java

$ javac -d . MammalInt.java

Now a package/folder with the name **animals** will be created in the current directory and these class files will be placed in it as shown below.



You can execute the class file within the package

import keyword:

If a class wants to use another class in the same package, the package name need not be used.

Example:

package payroll;

public class Boss {

public void payEmployee(Employee e) {

e.mailCheck();

}

}

The fully qualified name of the class can be used.

payroll.Employee

The package can be imported using the import keyword, and the wild card (\*):

import payroll.\*;

The class itself can be imported using the import keyword.

import payroll.Employee;

Note:

A class file can contain any number of import statements. the import statements must appear after the package statement and before the class declaration.

Directory Structure of packages:

Two major results occur when a class is placed in a package:

1. The name of package becomes a part of the name of the class, as we just discussed in the previous section.
2. The name of the package must match the directory structure where the corresponding byte code resides.

Example:

package vehicle;

public class Car {

// Class implementation.

}

The source file in a directory whose name reflects the name of the package to which the class .

....\vehicle\Car.java

The qualified class name and pathname :

* class name-->vehicle.car
* path name---> vehicle\car.java

Example:

A company's Internet domain name is apple.com , then all its package names would start with **com.apple**. Each component of the package name corresponding to a subdirectory.

Example:

The company had a com.apple .computers package that contained a Dell.java source file, it would be in a series of subdirectories like

....\com\apple\computers\Dell.java

At the time of compilation, the compiler creates a different output file for each class, interface and enumeration defined in it.

The base name of the output file is the name of the type, and its extension is . **class.**

Example:

// File Name: Dell.java

package com.apple.computers;

public class Dell {

}

class Ups {

}

compile:

$javac -d . Dell.java

files will be compiled :

.\com\apple\computers\Dell.class

.\com\apple\computers\Ups.class

import all the classes and interfaces:

import com.apple.computers.\*;

The .java source files, the compiled .class files should be in a series of directories that reflect the package name.

<path-one>\sources\com\apple\computers\Dell.java

The path to the .class files does not have to be the same as the path to the .java source files.

path-two>\classes\com\apple\computers\Dell.class

Set CLASS PATH system variable:

To display the current CLASS PATH variable in Windows and UNIX (Bourne shell)

* In Window ---> C:\> set CLASS PATH
* in UNIX---> % echo $CLASSPATH

To delete the current contents of the CLASSPATH variable, use −

* In Windows → C:\> set CLASSPATH =
* In UNIX → % unset CLASSPATH; export CLASSPATH

To set the CLASSPATH variable −

* In Windows → set CLASSPATH = C:\users\jack\java\classes
* In UNIX → % CLASSPATH = /home/jack/java/classes; export CLASSPATH

Java-Data Structure

The data structures provided by the java utility package are very powerful and perform a wide range of functions.

* Enumeration
* BitSet
* Vector
* Stack
* Dictionary
* Hashtable
* properties

All these classes are now legacy and java-2 introduced in a new Framework is called as a collections Framework.

**Enumaeration: Interface**

The Enumeration interface defines the methods by which you can enumerate ( obtain one at a time) the elements in a collection of objects.

This legacy interface has been superceded by Iterator.

It is used by several methods defined by the legacy classes such as **Vector and peoperties** , is used by several other API classes, and is currently in widespread use in application code.

|  |  |
| --- | --- |
| Methods | Description |
| boolean hasMoreElements( ) |  |
| Object nextElement( ) |  |

program:

import java.util.Vector;

import java.util.Enumeration;

public class EnumerationTester {

public static void main(String args[]) {

Enumeration days;

Vector dayNames = new Vector();

dayNames.add("Sunday");

dayNames.add("Monday");

dayNames.add("Tuesday");

dayNames.add("Wednesday");

dayNames.add("Thursday");

dayNames.add("Friday");

dayNames.add("Saturday");

days = dayNames.elements();

while (days.hasMoreElements()) {

System.out.println(days.nextElement());

}

}

}

**BitSet: class**

The BitSet class creates a special type of array that holds bit values .

The BitSet array can increase in size as needed.

This makes it similar to a vector of a bits.

|  |  |
| --- | --- |
| Constructor | Description |
| BitSet( ) |  |
| BitSet(int size) |  |

BitSet implements the Cloneable interface

|  |  |
| --- | --- |
| **Method** |  |
| **void and(BitSet bitSet)** |  |
| **void andNot(Bitset bitSet)** |  |
| **int cardinality( \_** |  |
| **void clear()** |  |
| **void clear( int index)** |  |
| **void clear(int startIndex , int endIndex)** |  |
| **Object clone( )** |  |
| **boolean equals(Oject bitSet)** |  |
| **void flip(int index)** |  |
| **void flip(int startIndex, int endIndex)** |  |
| **boolean get(int index)** |  |
| **BitSet get(int startIndex, int endIndex)** |  |
| **int hashCode( )** |  |
| **boolean intersects(BitSet bitSet)** |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

program:

import java.util.BitSet;

public class BitSetDemo {

public static void main(String args[]) {

BitSet bits1 = new BitSet(16);

BitSet bits2 = new BitSet(16);

// set some bits

for(int i = 0; i < 16; i++) {

if((i % 2) == 0) bits1.set(i);

if((i % 5) != 0) bits2.set(i);

}

System.out.println("Initial pattern in bits1: ");

System.out.println(bits1);

System.out.println("\nInitial pattern in bits2: ");

System.out.println(bits2);

// AND bits

bits2.and(bits1);

System.out.println("\nbits2 AND bits1: ");

System.out.println(bits2);

// OR bits

bits2.or(bits1);

System.out.println("\nbits2 OR bits1: ");

System.out.println(bits2);

// XOR bits

bits2.xor(bits1);

System.out.println("\nbits2 XOR bits1: ");

System.out.println(bits2);

}

}

Vector:

Vector implements a dynamic array. It is similar to ArrayList but with two differences

* Vector is synchronized.
* Vector contains many legacy methods that are not part of the collections framework.

The size of the array in advance or you just need one that can change sizes over the lifetimeof a program.

|  |  |
| --- | --- |
| Constructor | Description |
| Vector( ) |  |
| Vector( int size) |  |
| Vector(int size, int incr)   |  |  | | --- | --- | |  |  | |  |
| Vector(Collection c |  |

|  |  |
| --- | --- |
| Method | Description |
| void add(int index, Object element) |  |
| boolean add(Object o) |  |
| boolean addAll(Collection c) |  |
|  |  |

Program:

Stack:

Stack is a subclass of Vector that implements a standard **last-in, first-out stack.**

Stack only defines the default constructor, which creates an empty stack.

Stack( )

Stack includes all the methods inherited from its Parent class Vector, Stack defines

|  |  |
| --- | --- |
| Mthods |  |
| boolean empty( ) |  |
| Object peek() |  |
| Object pop( ) |  |
| Object push(Object element) |  |
| int search(Object element) |  |

Example:

import java.util.\*;

public class StackDemo {

static void showpush(Stack st, int a) {

st.push(new Integer(a));

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

System.out.println("stack: " + st);

}

static void showpop(Stack st) {

System.out.print("pop -> ");

Integer a = (Integer) st.pop();

System.out.println(a);

System.out.println("stack: " + st);

}

public static void main(String args[]) {

Stack st = new Stack();

System.out.println("stack: " + st);

showpush(st, 42);

showpush(st, 66);

showpush(st, 99);

showpop(st);

showpop(st);

showpop(st);

try {

showpop(st);

} catch (EmptyStackException e) {

System.out.println("empty stack");

}

}

}

output

stack: [ ]

push(42)

stack: [42]

push(66)

stack: [42, 66]

push(99)

stack: [42, 66, 99]

pop -> 99

stack: [42, 66]

pop -> 66

stack: [42]

pop -> 42

stack: [ ]

pop -> empty stack

Dictionary:

It is an abstract class that represents a key/value storage repository and operates much like Map.

Given a key and value, you can storage the value in a dictionary object.

Once the value is stored, you can retrieve it by using its key.

|  |  |
| --- | --- |
| Methods |  |
| Enumeration elements( ) |  |
| Object get(Object key) |  |
| boolean isEmpty( ) |  |
| Enumeration keys( ) |  |
| Object put(Object key, Object value) |  |
| Oject remove(Object key) |  |
| Object remove(Object key) |  |
| int size() |  |

The dictionary class is obsolete. you should implement the Map interface to obtain key /value storage functionality.

Map interface:

The Map interface maps unique keys to values.

A key is an object that you use to retrieve a value at a later date:

* Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.
* Several methods throw a No such Element Exception when no items exist in the invoking map.
* A classcastException is thrown when an object is incompatible with the elements in a map.
* A NullPointer Exception is thrown if an attempt is made to use a null object and null is not allowed in the map.
* An Unsupported OPerationException is thrown when an attempt is made to change an unmodifiable map.

|  |  |
| --- | --- |
| Methods |  |
| void clear( ) |  |
| boolean containsKey(Object k) |  |
| boolean containsValue(Object v) |  |
| Set entrySet( ) |  |
| boolean equals(Object obj) |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

import java.util.\*;

public class CollectionsDemo {

public static void main(String[] args) {

Map m1 = new HashMap();

m1.put("Zara", "8");

m1.put("Mahnaz", "31");

m1.put("Ayan", "12");

m1.put("Daisy", "14");

System.out.println();

System.out.println(" Map Elements");

System.out.print("\t" + m1);

}

}

Output:

Map Elements

{Daisy = 14, Ayan = 12, Zara = 8, Mahnaz = 31}

Hashable:

Hashtable was part of the original java.util and it is a concrete implementation of a Dictionary.

Java 2 re-engineered Hashtable so that it also implements the Map interface.

Hashtable is now integrated into the collections framework.

It is similar to HashMap but It is synchronized.

Like HashMap, Hashtable stores key/value pairs in a hash table.

When using a Hashtable, you specify an object that is used as a key , and the value that you want linked to that key.

The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

|  |  |
| --- | --- |
| Constructor |  |
| HAshtable( ) |  |
| Hashtable(int size) |  |
| Hashtable(int size, float fillRatio) |  |
| Hashtable(Map <? extends K, ?extends V> t) |  |

|  |  |
| --- | --- |
| Methods |  |
| void clear( ) |  |
| Object clone( ) |  |
| boolean contains(Object value) |  |
| boolean containsKey(Object key) |  |
| boolean ContainValue(Object value) |  |
| Enumeration elements( ) |  |
| Object get(Object key) |  |
| boolean isEmpty( ) |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

import java.util.\*;

public class HashTableDemo {

public static void main(String args[]) {

// Create a hash map

Hashtable balance = new Hashtable();

Enumeration names;

String str;

double bal;

balance.put("Zara", new Double(3434.34));

balance.put("Mahnaz", new Double(123.22));

balance.put("Ayan", new Double(1378.00));

balance.put("Daisy", new Double(99.22));

balance.put("Qadir", new Double(-19.08));

// Show all balances in hash table.

names = balance.keys();

while(names.hasMoreElements()) {

str = (String) names.nextElement();

System.out.println(str + ": " + balance.get(str));

}

System.out.println();

// Deposit 1,000 into Zara's account

bal = ((Double)balance.get("Zara")).doubleValue();

balance.put("Zara", new Double(bal + 1000));

System.out.println("Zara's new balance: " + balance.get("Zara"));

}

}

output:

Qadir: -19.08

Zara: 3434.34

Mahnaz: 123.22

Daisy: 99.22

Ayan: 1378.0

Zara's new balance: 4434.34

Properties Class:

Properties is a subclass of Hashtable .

It is used to maintain lists of values in which **the key is a String and value is a String**.

The properties class is used by many other java classes.

Example:

It is the type of object returned by System.getProperties( ) when obtaining environment values.

Properties define instance variable.

This variable holds a default property list associated with a Properties object.

Properties defaults;

|  |  |
| --- | --- |
| Constructor |  |
| Properties( ) |  |
| Properties( Properties propDefault) |  |

|  |  |
| --- | --- |
| Methods |  |
| String getProperty(String key) |  |
| String getProperty(String key, String defaultProperty) |  |
| void list(PrintStream streamOut) |  |
| Void load(InputStream streamIn) throws IOException |  |
|  |  |
|  |  |
|  |  |
|  |  |

import java.util.\*;

public class PropDemo {

public static void main(String args[]) {

Properties capitals = new Properties();

Set states;

String str;

capitals.put("Illinois", "Springfield");

capitals.put("Missouri", "Jefferson City");

capitals.put("Washington", "Olympia");

capitals.put("California", "Sacramento");

capitals.put("Indiana", "Indianapolis");

// Show all states and capitals in hashtable.

states = capitals.keySet(); // get set-view of keys

Iterator itr = states.iterator();

while(itr.hasNext()) {

str = (String) itr.next();

System.out.println("The capital of " + str + " is " +

capitals.getProperty(str) + ".");

}

System.out.println();

// look for state not in list -- specify default

str = capitals.getProperty("Florida", "Not Found");

System.out.println("The capital of Florida is " + str + ".");

}

}

output:

The capital of Missouri is Jefferson City.

The capital of Illinois is Springfield.

The capital of Indiana is Indianapolis.

The capital of California is Sacramento.

The capital of Washington is Olympia.

The capital of Florida is Not Found.

The collection framework was designed to meet several goals:

* 1. The framework had to be high-performance.

2.The implementations for the fundamental collections( dynamic arrays, linked lists, trees and hashtable) were to be high efficient.

* The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
* The framework had to extend and /or adapt a collection easily.

The collections framework is a unified architecture for representing and manipulating collections.

* Interface:

1.These are abstract data types that represent collections.

2.Interface allow collections to be manipulated independently of the details of their representation.

3.In Object-oriented languages, interfaces, generally form a hierarchy.

* Implementations, i.e., classes:

1.These are the concrete implementations of the collection interfaces.

2. In essence, they are reusable data structure.

* Algorithms:

1.These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.

* 2.The algorithms are said to be **polymorphic:** The same method can be used on many different implementations of the appropriate collection interface.

|  |  |
| --- | --- |
| Interfaces |  |
| collection Interface |  |
| List interface |  |
| Set |  |
| SortedSet |  |
| Map |  |
| Map.Entry |  |
| SortedMap |  |
| Enumeration |  |

Collection Interface:

These method are throw an **UnsupportedOperationException.**

|  |  |
| --- | --- |
| Methods |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| int size( ) |  |
| Object[ ]toArray( ) |  |
| Object[ ] toArray(Objecty array [ ]) |  |

Program

import java.util.\*;

public class CollectionsDemo {

public static void main(String[] args) {

// ArrayList

List a1 = new ArrayList();

a1.add("Zara");

a1.add("Mahnaz");

a1.add("Ayan");

System.out.println(" ArrayList Elements");

System.out.print("\t" + a1);

// LinkedList

List l1 = new LinkedList();

l1.add("Zara");

l1.add("Mahnaz");

l1.add("Ayan");

System.out.println();

System.out.println(" LinkedList Elements");

System.out.print("\t" + l1);

// HashSet

Set s1 = new HashSet();

s1.add("Zara");

s1.add("Mahnaz");

s1.add("Ayan");

System.out.println();

System.out.println(" Set Elements");

System.out.print("\t" + s1);

// HashMap

Map m1 = new HashMap();

m1.put("Zara", "8");

m1.put("Mahnaz", "31");

m1.put("Ayan", "12");

m1.put("Daisy", "14");

System.out.println();

System.out.println(" Map Elements");

System.out.print("\t" + m1);

}

}

Output:

ArrayList Elements

[Zara, Mahnaz, Ayan]

LinkedList Elements

[Zara, Mahnaz, Ayan]

Set Elements

[Ayan, Zara, Mahnaz]

Map Elements

{Daisy = 14, Ayan = 12, Zara = 8, Mahnaz = 31}