What is an object?

An object can be considered a "thing" that can perform a set of related activities. The set of activities that the object performs defines the object's behavior.

In pure OOP terms an object is an instance of a class.

Object Oriented Principles

Abstraction

Encapsulation

Inheritance

Polymorphism

Abstracion

Abstraction is an emphasis on the idea, qualities and properties rather than the particulars . The importance of abstraction is derived from its ability to hide irrelevant details and from the use of names to reference objects

Encapsulation

The encapsulation is the inclusion within a program object of all the resources need for the object to function - basically, the methods and the data. In OOP the encapsulation is mainly achieved by creating classes

Inheritance

Ability of a new object to be created, from an existing object, is called inheritance. Simply a classs can exibit properties of another class.

Polymorphism

Polymorphisms is a generic term that means 'many shapes'. More precisely Polymorphisms means the ability to request that the same operations be performed by a wide range of different types of things.

Java Character Set

Java programs are a collection of whitespace, identifiers,comments, literals, operators, separators, and keywords

Java Keywords

Key words are reserved words in the language, with special meaning.

Eg:- class, package, int,...

Literals

A constant value in Java is created by using a literal representation of it.

Eg:

integer constant: 89, 45

Float constant: 34.78, 32.12

Character constant: 'X', 'k'

String constant: “helo to” , “in india”

Whitespaces

In Java, whitespace are blank space, tab, or newline.

Identifiers

Identifiers are used for class names, interface names, method names,package names and variable names.

Rules for create an identifier

I. First character should be an alphabet, underscore, or dollar

ii. Remaining characters should be any of the above or digits.

iii. No any special character except dollar and underscore

iv. No keywords

v. No spacing

vi. case sensitive

Comments

Comments are non executebale statements . Two types of comments:-

Single line comments

Eg:- //int x=20,y;

/* Multiline comments

Eg:-/* int x=5, y=20,z;

z=x+y;

System.out.println(“Sum:”+z);

***/**

Separators

In Java, there are a few characters that are used as separators.

( )	Parentheses
{ }	Braces
[ ]	Brackets
;	Semicolon
,	Comma
.	Period

Data Types

Primitive data types:-

Java defines eight simple (or elemental) types of data:

byte, short, int, long, char, float, double, and boolean.

Variables

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer.

Variable declaration syntax

type var1[=value1], var2[=value2]... ;

Eg:int x,y=123,z;

Arrays

An array is a group of like-typed variables that are referred to by a common name.

One-Dimensional Arrays

Syntax: type array-var [ ]= new type[size];

```
int a1[ ] = new int[10];

int[ ] a2 = new int[20];
```

Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays.

Eg: int twoD[ ][ ] = new int[4][5];

Operators

Operators can be divided into the following three groups:

Arithmetic,

Relational,

and Logical.

Arithmetic Operators

Arithmetic operators are used in mathematical expressions

+ Addition

- Subtraction (also unary minus)

* Multiplication

/ Division

% Modulus

Arithmetic Operators

++ Increment

+= Addition assignment

-= Subtraction assignment

***= Multiplication assignment**

/= Division assignment

%= Modulus assignment

-- Decrement

Relational Operators

The relational operators determine the relationship that one operand has to the other.

== Equal to

!= Not equal to

> Greater than

< Less than

>= Greater than or equal to

<= Less than or equal to

Logical Operators

The Boolean logical operators operate only on boolean operands.

|| Short-circuit OR

&& Short-circuit AND

! Logical unary NOT

Control Statements

A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program.

*if statements

* switch

* loopings

If statements

*simple if:

```
if (condition) statement1;
```

* if...else:

```
if (condition) statement1;
 else statement2;
```

If statements

if...else if chain:

```
        if (condition1) statement 1;
         elseif(condition2) statement 2;
         ...
        elseif(conditionn( statement n);
```

*Nested if

switch

Switch statement is Java's multiway branch statement

```
switch (expression) {
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
...
case valueN:
// statement sequence
break;
default:
// default statement sequence
}
```

Loops(Iteration Statements)

While: It repeats a   block while it is true.

```
while(condition) {
// body of loop
}
```

Do-while:The do-while loop always executes its body at least once

```
do {
// body of loop
} while (condition);
```

Loops(Iteration Statements)

```
for(initialization; condition; iteration) {
// body
}
```

Methods

Method is just a block of code

Syntax:

```
type <method_name>(parameterlist){
   ......
   [return value];
}
```

Here, *value* is the value returned by the method.

class

A class which binds data and methods together to form a single unit

It is simply a representation of a type of object. It is the blueprint/ plan/ template that describe the details of an object.

Class is composed of three things: a name, attributes, and operations.

Class syntax

```
access specifier class <class_name>{
          access specifier type instance_vari1;
          access specifier type instance_vari2;
                . . .
          access specifier type <method1>(parameterlist1){
                      . . .
                     [return value];
                 }
         access specifier type <method2>(parameterlist2){

                            . . .
                    [return value];
                }
                . . .
}
```

Sample Class

```
class Sample{
    int x;
    int y;
    void setData(int a,int b){
        x=a;
        y=b;
        }
}
```

Creating and invoking of objects

An instance of a class is known as object.

Syntaxt for object creation:-classname <object_name> = new classname( );

Eg:- Sample ob=new Sample( );

Object invokation:-object_name.member;

Where member is a data or a method.

ob.x=10;

ob.y=20;

ob.setData(5,6);

Sample java program

```
class Example {

     public static void main(String ar[ ]) {

          System.out.println("Welcome to java”);

      }

}
```

Explanation on Sample.java

* Execution is always start from main() method

*public:Access specifier,it is must, for call code outside of its class when the program is started.

*static:allows main( ) to be called without having to instantiate a particular instance of the class.

*String ar[ ]: ar receives any command-line arguments

present when the program is executed.

Explanation on Sample.java

*System: System is a predefined class from java

*out: It is the output stream that is connected to the console

Constructors

It is just like a method. A *constructor* initializes an object immediately upon creation

* Its name is same as that of class name

* No return type

*May or may not have parameters

*Can write any number of constructors in a class

Syntax of constructor

```
access_specifier class <class_name>{
      . . .
      class_name(parameterlist1){
          ...
       }
       class_name(parameterlist2){
         ...
       }
       . . .
}
```

this operator

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword. this can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked. You can use this any where a reference to an object of the current class.

“this” Example

```
BoxShape(double w, double h, double d) {
    this.width = w;
    this.height = h;
    this.depth = d;
}
```

Also avoids name collission

```
BoxShape(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

Overloading Methods

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading.

Java Access specifiers

Java's access specifiers are public, private, and protected. Java also defines a default

access level(no modifier). The protected applies only when inheritance is involved.

Inheritance

Ability of a new class to be created, from an existing class by extending it, is called inheritance.

Private members willnot inherit. Because they have validity, within same class only.

Java does not support the inheritance of multiple superclasses into a single subclass.

Inheritance syntax

```
acceeess_specifier class <superclass_name>{

...

}

access_specifier class <subclass_name> extends
   <superclass_name>{

...

}
```

Method Overriding

when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.

Method Overriding

When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Syntax of Method Overriding-1

```
class SuperClass{
...
            type <method1>(parameterlist1){
             //body of method1.
            }
   ...
}
```

Syntax of Method Overriding-2

```
class SubClass extends SuperClass{
   ...
            type <method1>(parameterlist1){
                  //new body of method1.
            }
...
}
```

“super” keyword

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super. The super has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass

First form of super

Using super to Call Superclass Constructors:-

A subclass can call a constructor method defined by its superclass by use of the following form of super:

super(*parameter-list*);

Here, *parameter-list* specifies any parameters needed by the constructor in the superclass. super( ) must always be the first statement executed inside a subclass constructor.

Second form of “super”

The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.member

Here, member can be either a method or an instance variable.

Abstract Classes

When ever a class is unable to implement all of its methods such class is considred as an abstract class.

*** declare class and undefined methods with the keyword *abstract***

***may or may not contain abstarct and non abstract methods**

***Unable to define objects**

***Just acting as base class for other classes.**

***Can extend with other abstract classes**

***Must override all abstract methods when extends with a normal class.**

Abstract class syntax

```
abstract class <abstract_class>{
    abstract type abst_method1(parameterlist1);
    abstract type abst_method2(parameterlist2);
    ...
    type method1(para_list1){
     ...
    }
    type method2(para_list2){
   ...
   }
   ...
}
```

Extending an abstract class

```
class <class_name> extends <abstract_class>{

             type abst_method1(parameterlist1){...}

             type abst_method2(parameterlist2){...}

            ...

            type method1(para_list1){...}

            type method1(para_list2){...}

            ...

}
```

“final” keyword

*prevent overrding

*prevent inheritance

“final” prevent overrding

```
 class <super_class>{
      final type <method1>(parameter_list)>{
      }
}
class <sub_class> extends <super_class>{
     type <method1>(parameter_list)>{ //Error
        //New body
     }
}
```

“final” prevent inheritance

```
final class <super_class> {
    ...
}
//Following is an incorrect class
class <sub_class> extends <super_class>{
    ...
}
```

Interfaces

Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.Once it is defined, any number of classes can implement an interface. One class can implement any number of interfaces.

Interface syntax

An interface is defined much like a class.

```
access specifier interface <interface-name> {
    final type var-name1 = value;
    final type var-name2 = value;
     ...
    return-type <method-name1>(parameter-list1);
    return-type <method-name2>(parameter-list2);
    ...
}
```

Implementing interfaces

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface.

During implementing time we must override all the methods that were available from the interfaces as public.

Interface implementation syntax

```
access specifier class <classname> [extends
  superclass] implements interface1,... {
    public type  <method1>(parameterlist1){
    }
    public type  <method2>(parameterlist2){
    }
    ...
}
```

Interfaces can be extended

We can extend an interfaces with more than one interfaces at a time.

```
interface I1{

}

interface I2{

}

interface I extends I1,I2,..{

}
```

packages

A package is just a room for store classes and interfaces. It is simply a folder. A package may contain any number of classes and interfaces.

The package is both a naming and a visibility control mechanism. We can define classes inside a package that are not accessible by code outside that package.

Synax for package

```
package <package-name>;

access specifier class | interface  <name>{

   ...

}
```

Package name must match with the folder name it holds.

Package statement should be the first statement of the program.

Nested Packages

We can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period operator

Syntax:

```
package pkg1.pkg2....;

access specifier class|interface <name>{

}
```

Importing packages

We can import one or more packages in our program at a time

Syntax: import <pkgname>.*;

This command imports only classes and interfaces of the package, not sub packages.

OR

import <pkgname>.<class-name> | <interface-name>;

static concept

It is possible to create a member that can be used by itself, without reference to a specific instance. The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist.

Methods declared as static have several restrictions:

• They can only call other static methods.

•They must only access static data.

• They cannot refer to this or super in any way.

More about static

A static member inside a same class ,can be call with out use object of that class.

Also, static member from outside a class can be call by the following syntax.

classname.member.

(In both cases, static member can call with the object too);

String class(java.lang)

In java a string is a sequence of characters. Java implements strings as objects of type **String.**

The String class supports several constructors.

Eg: String(), String(String str), String(char ch[ ])

String(byte bt[ ]),...

String objects

eg:

```
String str=”helo to java”; //static initialization

String sts=new String( );

String st=new String(“Hai to”);

char [ ]ch={'I','N','D','I','A'};

String sp=new String(ch);
```

Some methods of String class

int length( ), char charAt(int index), byte[ ] getBytes( ),

char[ ] toCharArray( ), boolean equals(String str),

boolean equalsIgnoreCase(String str)

boolean startsWith(String str), boolean endsWith(String str)

, int indexOf(String str)

int lastIndexOf(String str), String [ ] split(String)

More String methods

String substring(int startIndex, int endIndex)

String substring(int startIndex),

String concat(String stp)

String replace(char original, char replacement),

String trim( ), String toLowerCase( ),

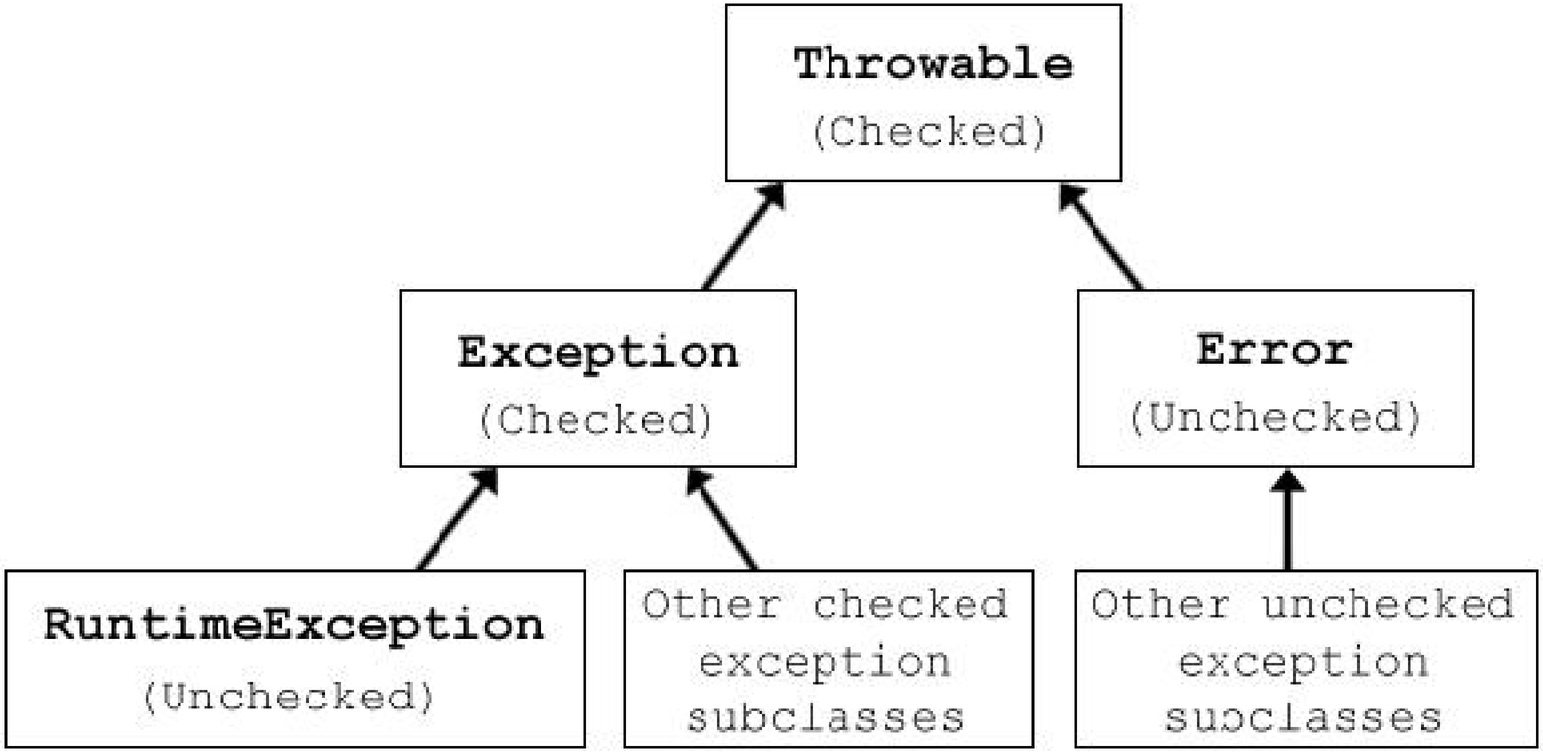
String toUpperCase( )

Exception Handling

Exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error.

Java exception handling is managed via five keywords: try, catch, throw, throws, andfinally.

Exception Hierarchy



Try-catch syntax

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
/ ...
finally {
// block of code to be executed before try block ends}
```

Try- catch explanation

Program statements that you want to monitor exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Our code can catch this exception (using catch) and handle it in some rational manner.

Try- catch explanation

System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed before a method returns is put in a finally block.

throw

It is possible for our program to throw an exception explicitly, using the throw statement. The general form of throw is shown below:

```
throw ThrowableInstance;
```

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

Eg: throw new NullPointerException("demo");

throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. We cando this by including a throws clause in the method's declaration.

throws

A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException.

Throws syntax

This is the general form of a method declaration that includes a throws clause:

type <method-name>(parameter-list) throws *exception-list*{

// body of method

}

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

finally

finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. What ever happened inside the try-catch block finally block will execute just before leave from, try-catch block. So we can put our necessary task inside the finally block.

Wrapper classes(java.lang)

Java provides classes that correspond to each of the simple types(primitive). Inessence, these classes encapsulate, or wrap, the simple types within a class. Thus, they are commonly referred to as type wrappers. They are, Double, Float, Byte, Short, Integer, Long, Boolean and Character

Common methods of numerical wrappers

int intValue(), float floatValue(), double doubleValue(), short shortValue(), long longValue(), byte byteValue()

And for,

Float , static float parseFloat(String num)

Integer , static int parseInt(String num)

Double , static double parseDouble(String num)

Short , static short parseShort(String num)

Long , static long parseLong(String num)

Byte , static byte parseByte(String num),...

Math class(java.lang)

The Math class contains all the floating-point functions that are used for geometry and trigonometry, as well as several general-purpose methods

Math functions

static double sqrt(double)

static double pow(double y, double x)

static int max(int x, int y)

static int min(int x, int y)

static int abs(int arg),

Static long round(double)

static double ceil(double)

static double floor(double)

StringTokenizer(java.util)

The class helps to tokenize a string based on delimiter.

StringTokenizer(String str, String delimiters)

Methods:-

boolean hasMoreTokens()

String nextToken()

Date class(java.util)

The Date class encapsulates the current date and time. Date supports the following constructors:

Date( )

Date(long millisec)

The first constructor initializes the object with the current date and time. The second constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970.

Some Date methods

boolean after(Date date) ,

boolean before(Date date), long getTime( ),

int getDate(), int getMonth(), int getYear(),

int getHours(), int getMinutes(),

int getSeconds()

File class(java.io)

It deals directly with files and the file system. That is, the File class does not specify how information is retrieved from or stored in files. It describes the properties of a file itself.

More about File

A File object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.

File constructors

The following constructors can be used to create File objects:

File(String directoryPath)

File(String directoryPath, String filename)

File(File dirObj, String filename) ...

Major File methods

String getAbsolutePath()

String getParent(), String getName(),

boolean exists(), boolean isFile(),

boolean isDirectory(), boolean canRead(),

boolean canWrite(), long lastModified()

More File Methods

String[ ] list(), File [ ] listFiles(),

static File [ ] listRoots(), boolen mkdir(),

boolen mkdirs(), boolean renameTo(File obj),

boolean delete(), boolean isHidden() ,

boolean createNewFile()

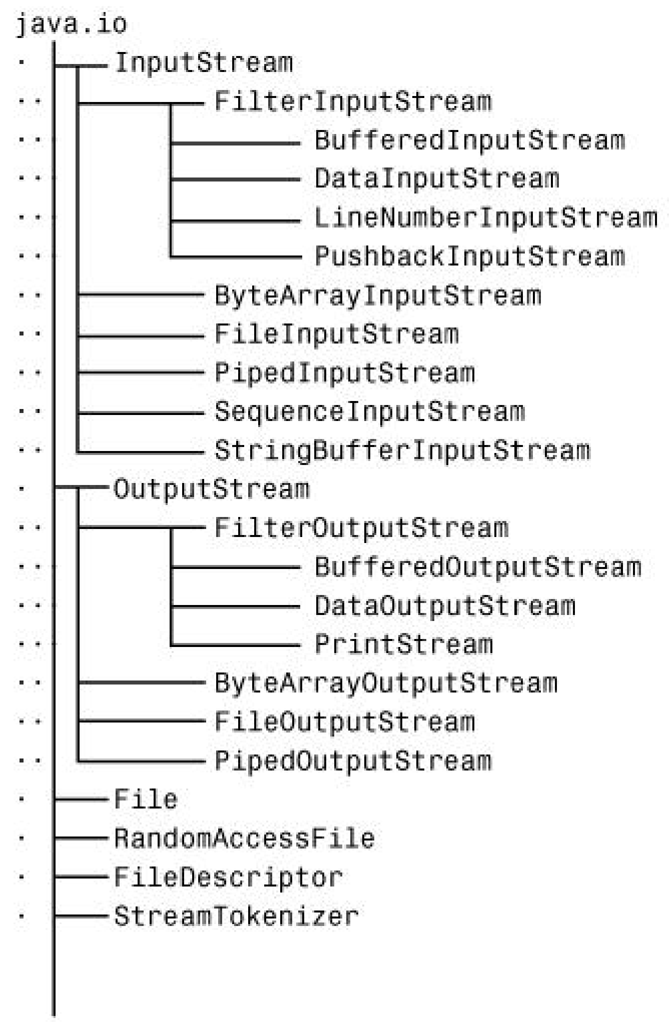
Streaming(java.io)

Streaming is flow of data from a source to destination as in a form of byte or characters. Java defines two types of streams: byte and character.

Streaming

Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, reading or writing binary data. Character streams provide a convenient means for handling input and output of characters.

Byte Streaming classes



Details of some byte stream classes

InputStream(abstract class):

int read(), int read(byte[ ]), int available(), void close(), ...

OutputStream(abstract class):

void write(int),void write(byte[]), void close()

DataInputStream(InputStream):

int read(), int read(byte[]), String readLine(), int readInt(),...

DataOutputStream(OutputStream):

void write(int), void write(byte[ ]), void writeInt(int),...

Stream classes and methods

FileInputStream():

FileInputStream(String path)

FileInputStream(File obj)

Methods:

int read(), int read(byte[ ]),

int available(), void close()

Stream classes and methods

FileOutputStream():

FileOutputStream(String path)

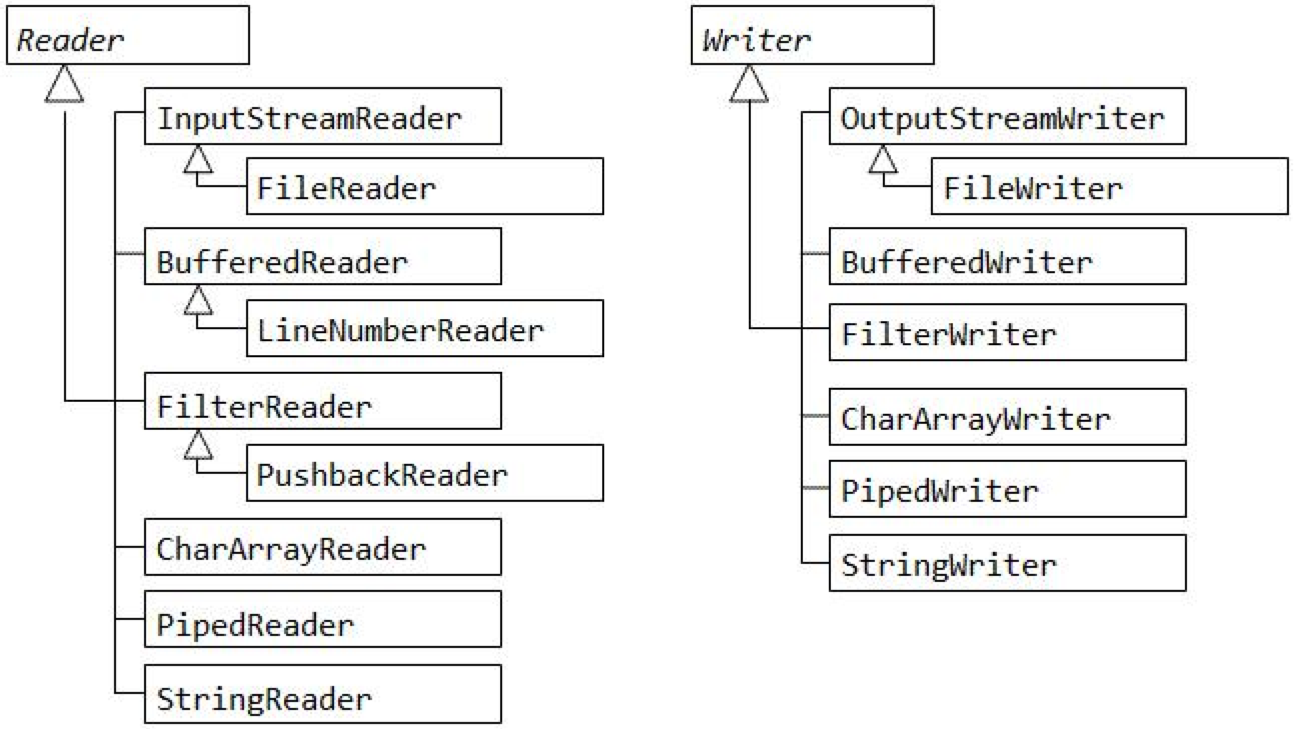
FileOutputStream(String path,boolean type)

FileOutputStream(File obj)

FileOutputStream(File obj, boolean type)

void write(int), void write(byte[ ]), void close()

Character Streaming classes(java.io)



Character reader classes

Reader(): abstract class:-

Int read(),Int read(char [ ]), void close( )

InputStreamReader:

InputStreamReader(InputStream),...

Int read( ), int read(char [ ],int,int), void close( )

Character reader classes

BufferedReader:

BufferedReader(Reader)

Int read(),

int read(char [ ],int,int),

void close( ),

String readLine( )

Character writer classes

Writer(): abstract class:-

void write(int), void write(char [ ]), void close()

OutputStreamWriter:

OutputStreamWriter(OutputStream),...

    void write(int), void write(char [ ],int,int), void close()

Character writer classes

BufferedWriter:

BufferedWriter(Writer)

void write(int), void write(char [ ],int,int),

void close()

JDBC

Java Database Connectivity or in short JDBC is a technology that enables the java program to manipulate data stored into the database. JDBC is Java application programming interface that allows the Java programmers to access database management system from Java code.

JDBC

JDBC is a java API which enables the java programs to execute SQL statements. It is an application programming interface that defines how a java programmer can access the database in tabular format from Java code using a set of standard interfaces and classes written in the Java programming language.

More about JDBC

JDBC provides methods for querying and updating the data in Relational Database Management system such as MySQL, SQLServer,Oracle etc.The Java application programming interface provides a mechanism for dynamically loading the correct Java packages and drivers and registering them with the JDBC Driver Manager

About Driver Manager

Driver Manager is used as a connection factory for creating JDBC connections which supports creating and executing statements such as SQL INSERT, UPDATE and DELETE.

About JDBC API

API is available in a package named java.sql

The JDBC Driver Manager:

The JDBC Driver Manager is a very important class that defines objects which connect Java applications to a JDBC driver.

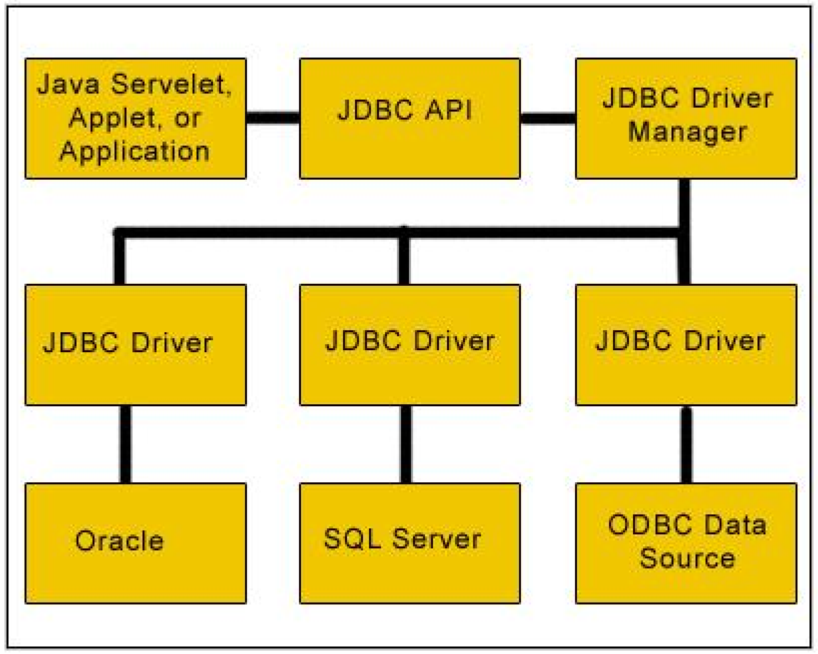
Driver Responsibility

The main responsibility of JDBC database driver is to load all the drivers found in the system properly as well as to select the most appropriate driver from opening a connection to a database.

JDBC Architecture

The Driver Manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Layers of the JDBC Architecture



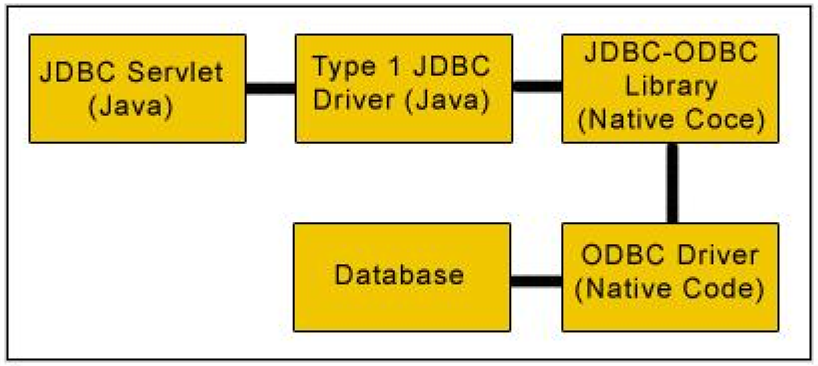
Type1 drivers

Type 1 JDBC-ODBC Bridge: Type 1 drivers act as a "bridge" between JDBC and another database connectivity mechanism such as ODBC.

Type I

he JDBC- ODBC bridge provides JDBC access using most standard ODBC drivers.

Type 1 JDBC Architecture


JDBC Servlet (Java)
Type 1 JDBC Driver (Java)
JDBC-ODBC Library (Native Coce)
ODBC Driver (Native Code)
Database

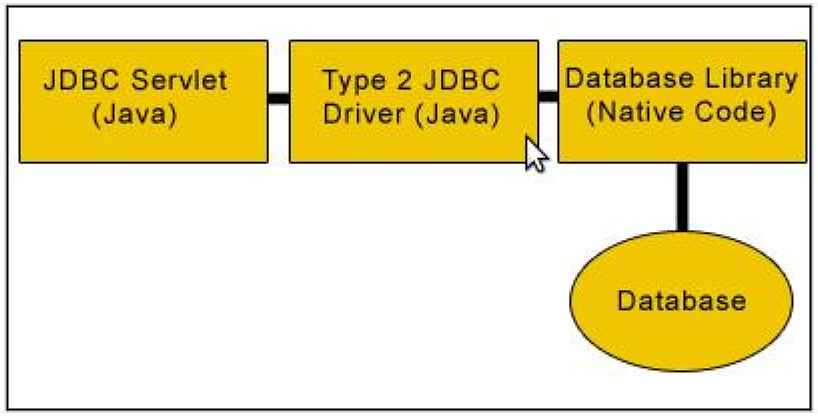
Type2 drivers

Type 2 Java to Native API: Type 2 drivers use the Java Native Interface (JNI) to make calls to a local database library API. This driver converts the JDBC calls into a database specific call for databases such as SQL, ORACLE etc.

Type II

This driver communicates directly with the database server. It requires some native code to connect to the database.

Type 2 JDBC Architecture



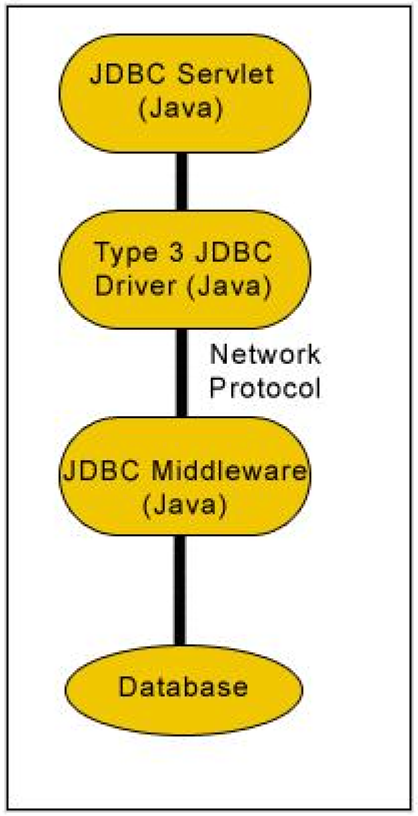
Type3 drivers

Type 3 Java to Network Protocol Or All-Java Driver: Type 3 drivers are pure Java drivers that use a proprietary network protocol to communicate with JDBC middleware on the server.

Type III

The middleware then translates the network protocol to database-specific function calls.

Type 3 JDBC Architecture



Type4 drivers

Type 4 Java to Database Protocol. Type 4 drivers are pure Java drivers that implement a proprietary database protocol to communicate directly with the database.

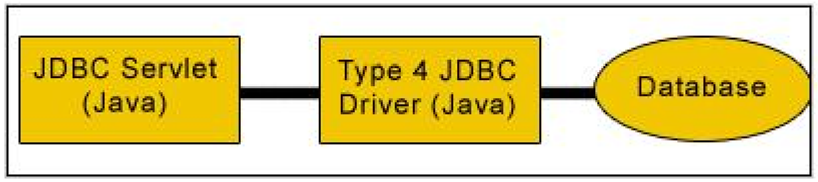
Type4 drivers

Like Type 3 drivers, they do not require native database libraries and can be deployed over the Internet without client installation. Type drivers communicate directly with the database engine rather than through middleware or a native library.

Type IV

Type 4 drivers are usually the fastest JDBC drivers available.

Type 4 JDBC Architecture



Accessing Database using Java and JDBC

Class.forName(String driver): It loads the driver.

DriverManager: This class controls a set of JDBC drivers. Each driver has to be register with this class.

DriverManager class

static Connection getConnection(String url, String userName, String password):

This method establishes a connection to specified database url.

DriverManager class

url: - Database url where stored or created your database

username: - User name of MySQL, password: - Password of MySQL

Connection interface

Connection: This interface specifies connection with specific databases like: MySQL, Ms-Access, Oracle etc and java files. The SQL statements are executed within the context of this interface.

Connection interface

Statement createStatement():

Creating a statement object for apply sql queries.

Statement:-

ResultSet   executeQuery(String sql)(select commands...)

Connection interface

int executeUpdate(String ql)

(insert,update,delete..)

ResultSet interface

It generates collection of records based on an sql query.

Major methods of ResultSet interface:

boolean next()

String getString(String fieldname)

String getString(String fieldindex)

int getInt(String fieldname),

int getInt(String fieldindex),...

Database Connectivity

Class.forName(String driver_url);

For mysql: com.mysql.cj.jdbc.Driver

Connection con=DriverManager.getConnection

(String DB_url,String username,String pass);

Database Connectivity

For mysql:

jdbc:mysql://localhost/smec?autoReconnect=true&useSSL=false","root","root

Statement stmt = con.createStatement();

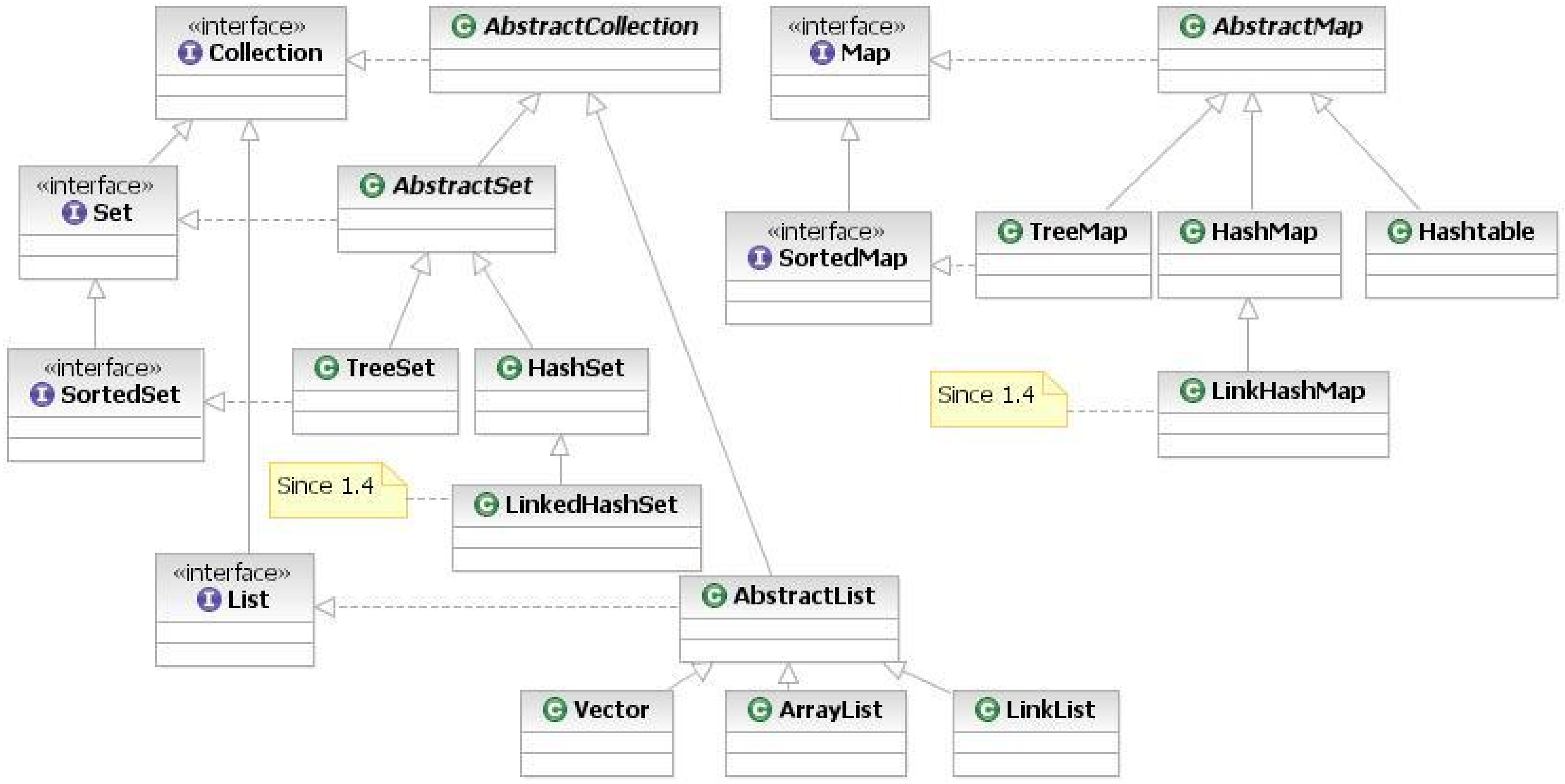
Then for ResultSet rs=stmt.executeQuery(String sql);

Or int x=stmt.executeUpdate(String sql);

Collection Framework(java.util)

A collection is a group of objects.The Collections Framework provides a well-designed set of interfaces and classes for storing and manipulating groups of data as a single unit. The framework provides a convenient API to many of the abstract data types. This API contains set of collection classes and interfaces.

Collection Structure



Collection Interface(java.util)

The Collection interface is used to represent any group of objects, or elements. We use the interface when we wish to work with a group of elements in as general a manner as possible.

Collection interface(Methods)

boolean add(Object obj),

boolean addAll(Collection c)

void clear(),

boolean contains(Object obj),int size()

boolean containsAll(Collection c),

Object [ ] toArray( )

Collection interface(Methods)

boolean equals(Object obj),

boolean isEmpty()

Iterator iterator(),

boolean remove(Object obj),

boolean removeAll(Collection c) ,

boolean retainAll(Collection c)