B.TECH VI SEM
IARE – R16

ELECTRONICS AND COMMUNICATION ENGINEERING
JAVA PROGRAMMING

Prepared by,
Mr. G Chandra Sekhar
Assistant Professor

UNIT - I

OOP concepts: Classes and objects, data abstraction, encapsulation, inheritance, benefits of inheritance, polymorphism, constructors, methods, data types, variables, constants, scope and life time of variables, operators, operator hierarchy, expressions, type conversion and casting, enumerated types, control flow statements, arrays, parameter passing.

Need for OOP Paradigm

- OOP is an approach to program organization and development, which attempts to eliminate some of the drawbacks of conventional programming methods by incorporating the best of structured programming features with several new concepts.
- OOP allows us to decompose a problem into number of entities called objects and then build data and methods (functions) around these entities.
- The data of an object can be accessed only by the methods associated with the object.

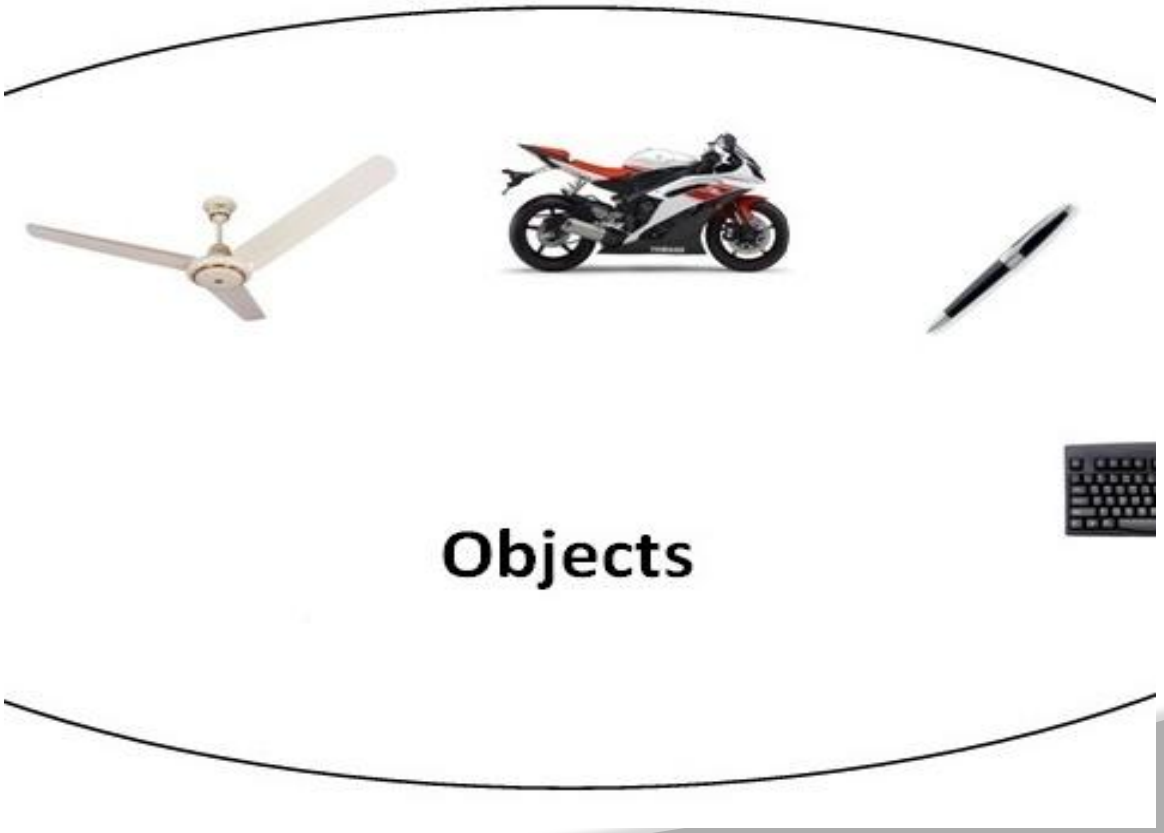
Object-oriented programming (OOP) is a programming paradigm that uses "Objects "and their interactions to design applications.

It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Data Abstraction & Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

OBJECT

- Objects are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.



CLASS

- The entire set of data and code of an object can be made of a user defined data type with the help of a class.
- In fact, Objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class.
- **Classes** are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represents a set of individual objects.
- Characteristics of an object are represented in a class as Properties. The actions that can be performed by objects become functions of the class and is referred to as Methods.
- A class is thus a collection of objects of similar type . for example: mango, apple, and orange are members of the class fruit . ex: fruit mango; will create an object mango belonging to the class fruit.

Example for class

```
class Human

{

 private:  EyeColor IColor;

 NAME personname;

public:

void SetName(NAME anyName);

 void SetIColor(EyeColor eyecolor);

}
```

Data abstraction

- Abstraction refers to the act of representing essential features without including the background details or explanations. since the classes use the concept of data abstraction ,they are known as *abstraction data type(ADT).*
- For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only know how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other.

An example of Data abstraction

- Humans manage complexity through abstraction. When you drive your car you do not have to be concerned with the exact internal working of your car(unless you are a mechanic).
- What you are concerned with is interacting with your car via its interfaces like steering wheel, brake pedal, accelerator pedal etc. Various manufacturers of car has different implementation of car working but its basic interface has not changed (i.e. you still use steering wheel, brake pedal, accelerator pedal etc to interact with your car). Hence the knowledge you have of your car is abstract.

Some of the Object-Oriented Paradigm are:

- Emphasis is on data rather than procedure.
- Programs are divided into objects.
- Data Structures are designed such that they Characterize the objects.
- Methods that operate on the data of an object are tied together in the data structure.
- Data is hidden and can not be accessed by external functions.
- Objects may communicate with each other through methods.

Encapsulation

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

To achieve encapsulation in Java –

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

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 fields.

Inheritance

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the IS-A relationship which is also known as a parent-child relations

Why use inheritance in java

For Method Overriding (so runtime polymorphism can be achieved).

For Co

Inheritance

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the IS-A relationship which is also known as a parent-child relations

Why use inheritance in java

For Method Overriding (so runtime polymorphism can be achieved).

For Co

Inheritance

Terms used in Inheritance

- Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.de Reusability.hip.

Benefit of using inheritance:

- A code can be used again and again
- Inheritance in Java enhances the properties of the class, which means that property of the parent class will automatically be inherited by the base class
- It can define more specialized classes by adding new details.

Polymorphism

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Runtime Polymorphism in Java.

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

Need for OO Paradigm

Differences between Procedural and OO Programing

Procedural	OO programming
Code is placed into totally distinct functions or procedures	Everything treated as an Object
Data placed in separate structures and is manipulated by these functions or procedures	Every object consist of attributes(data) and behaviors (methods)
Code maintenance and reuse is difficult	Code maintenance and reuse is easy
Data is uncontrolled and unpredictable (i.e. multiple functions may have access to the global data)	The data of an object can be accessed only by the methods associated with the object
You have no control over who has access to the data	Good control over data access
Testing and debugging are much more difficult	Testing and debugging are much easy
Not easy to upgrade	Easy to upgrade
Not easy to partition the work in a project	Easy to partition the work in a project

Constructor

- A constructor initializes the instance variables of an object.
- It is called immediately after the object is created but before the new operator completes.
    1) it is syntactically similar to a method:
    2) it has the same name as the name of its class
    3) it is written without return type; the default return type of a class
- constructor is the same class When the class has no constructor, the default constructor automatically initializes all its instance variables with zero.

```
class Box {
double width;
double height;
double depth;
Box() {
System.out.println("Constructing Box");
width = 10; height = 10; depth = 10;
}
double volume() {
return width * height * depth;
}}
```

Parameterized Constructor

Parameterized Constructor – A **constructor** is called **Parameterized Constructor** when it accepts a specific number of parameters. To initialize data members of a class with distinct values. ... With a **parameterized constructor** for a class, one must provide initial values as arguments, otherwise, the compiler reports an error.

```
class Box {
double width;
double height;
double depth;
Box(double w, double h, double d) {
width = w; height = h; depth = d;
}
double volume()
 { return width * height * depth;
}
}
```

Methods

- General form of a method definition:

```
type name(parameter-list) {
                                    … return value;
                                …
                              }
```

- Components:

  1) type - type of values returned by the method. If a method does not return any value, its return type must be void.

  2) name is the name of the method

  3) parameter-list is a sequence of type-identifier lists separated by commas

  4) return value indicates what value is returned by the method.

Classes declare methods to hide their internal data structures, as well as for their own internal use: Within a class, we can refer directly to its member variables:

```
class Box {
double width, height, depth;
void volume() {
System.out.print("Volume is ");
System.out.println(width * height * depth);
}}
```

Parameterized Methods

Parameters increase generality and applicability of a method:

1) method without parameters

```
int square() { return 10*10; }
```

2) method with parameters

```
int square(int i) { return i*i; }
```

- Parameter: a variable receiving value at the time the method is invoked.
- Argument: a value passed to the method when it is invoked.

Access Control: Data Hiding and Encapsulation

- Java provides control over the *visibility* of variables and methods.
- *Encapsulation,* safely sealing data within the *capsule* of the class Prevents programmers from relying on details of class implementation, so you can update without worry
- Helps in protecting against accidental or wrong usage.
- Keeps code elegant and clean (easier to maintain)

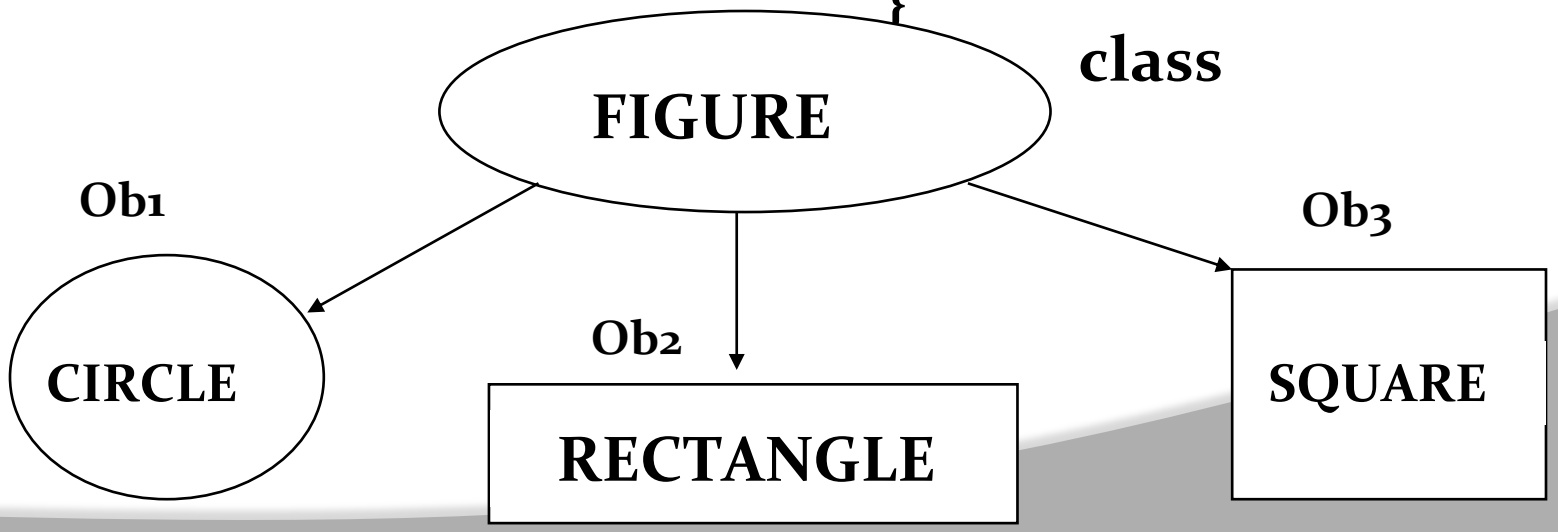
Access Modifiers: Public, Private, Protected

- *Public:* keyword applied to a class, makes it available/visible everywhere. Applied to a method or variable, completely visible.
- Default(No visibility modifier is specified): it behaves like public in its package and private in other packages.
- *Default Public* keyword applied to a class, makes it available/visible everywhere. Applied to a method or variable, completely visible.
- *Private* fields or methods for a class only visible within that class. Private members are *not* visible within subclasses, and are *not* inherited.
- *Protected* members of a class are visible within the class, subclasses and *also* within all classes that are in the same package as that class.

Classes

- Class is blue print or an idea of an Object
- From One class any number of Instances can be created
- It is an encapsulation of attributes and methods

```
class <ClassName>
{
        attributes/variables;
        Constructors();
        methods();
}
```



Instance

- Instance is an Object of a class which is an entity with its own attribute values and methods.

Creating an Instance

```
ClassName  refVariable;
refVariable = new Constructor();
                or
ClassName  refVariable = new  Constructor();
```

Java Class Hierarchy

In Java, class “Object” is the base class to all other classes

- If we do not explicitly say extends in a new class definition, it implicitly extends Object
- The tree of classes that extend from Object and all of its subclasses are is called the class hierarchy
- All classes eventually lead back up to Object
- This will enable consistent access of objects of different classes.

Method Binding

- Objects are used to call methods.
- **MethodBinding** is an object that can be used to call an arbitrary public method, on an instance that is acquired by evaluating the leading portion of a method binding expression via a value binding.
- It is legal for a class to have two or more methods with the same name.
- Java has to be able to uniquely associate the invocation of a method with its definition relying on the number and types of arguments.
- Therefore the same-named methods must be distinguished:

  1) by the number of arguments, or

  2) by the types of arguments
- Overloading and inheritance are two ways to implement polymorphism.

History of Java

Computer language innovation and development occurs for two fundamental reasons:

1) to adapt to changing environments and uses

2) to implement improvements in the art of programming

- The development of Java was driven by both in equal measures.
- Many Java features are inherited from the earlier languages:

B → C → C++ → Java

Before Java: C

- Designed by Dennis Ritchie in 1970s.
- Before C: BASIC, COBOL, FORTRAN, PASCAL
- C- structured, efficient, high-level language that could replace assembly code when creating systems programs.
- Designed, implemented and tested by programmers.

Before Java: C++

- Designed by Bjarne Stroustrup in 1979.
- Response to the increased complexity of programs and respective improvements in the programming paradigms and methods:
  - 1) assembler languages
  - 2) high-level languages
  - 3) structured programming
  - 4) object-oriented programming (OOP)
- OOP – methodology that helps organize complex programs through the use of inheritance, encapsulation and polymorphism.
- C++ extends C by adding object-oriented features.

Java: History

- In 1990, Sun Microsystems started a project called Green.
- Objective: to develop software for consumer electronics.
- Project was assigned to James Gosling, a veteran of classic network software design. Others included Patrick Naughton, ChrisWarth, Ed Frank, and Mike Sheridan.
- The team started writing programs in C++ for embedding into
  - toasters
  - washing machines
  - VCR's
- Aim was to make these appliances more "intelligent".

Java: History (contd.)

- C++ is powerful, but also dangerous. The power and popularity of C derived from the extensive use of pointers. However, any incorrect use of pointers can cause memory leaks, leading the program to crash.
- In a complex program, such memory leaks are often hard to detect.
- Robustness is essential. Users have come to expect that Windows may crash or that a program running under Windows may crash. (“This program has performed an illegal operation and will be shut down”)
- However, users do not expect toasters to crash, or washing machines to crash.
- A design for consumer electronics has to be *robust*.
- Replacing pointers by references, and automating memory management was the proposed solution.

Java: History (contd.)

- Hence, the team built a new programming language called Oak, which avoided potentially dangerous constructs in C++, such as pointers, pointer arithmetic, operator overloading etc.
- Introduced automatic memory management, freeing the programmer to concentrate on other things.
- Architecture neutrality (Platform independence)
- Many different CPU's are used as controllers. Hardware chips are evolving rapidly. As better chips become available, older chips become obsolete and their production is stopped. Manufacturers of toasters and washing machines would like to use the chips available off the shelf, and would not like to reinvest in compiler development every two-three years.
- So, the software and programming language had to be *architecture neutral*.

Java: History (contd.)

- It was soon realized that these design goals of consumer electronics perfectly suited an ideal programming language for the Internet and WWW, which should be:
  - ❖ object-oriented (& support GUI)
  - ❖– robust
  - ❖– architecture neutral
- Internet programming presented a BIG business opportunity. Much bigger than programming for consumer electronics.
- Java was “re-targeted” for the Internet
- The team was expanded to include Bill Joy (developer of Unix), Arthur van Hoff, Jonathan Payne, Frank Yellin, Tim Lindholm etc.
- In 1994, an early web browser called WebRunner was written in Oak. WebRunner was later renamed HotJava.
- In 1995, Oak was renamed Java.
- A common story is that the name Java relates to the place from where the development team got its coffee. The name Java survived the trade mark search.

Java: History

- Designed by James Gosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan at Sun Microsystems in 1991.
- The original motivation is not Internet: platform-independent software embedded in consumer electronics devices.
- With Internet, the urgent need appeared to break the fortified positions of Intel, Macintosh and Unix programmer communities.
- Java as an “Internet version of C++”? No.
- Java was not designed to replace C++, but to solve a different set of problems.

The Java Buzzwords

- The key considerations were summed up by the Java team in the following list of buzzwords:
  - Simple
  - Secure
  - Portable
  - Object-oriented
  - Robust
  - Multithreaded
  - Architecture-neutral
  - Interpreted
  - High performance
  - Distributed
  - Dynamic

The Java Buzzwords

- **Simple** – Java is designed to be easy for the professional programmer to learn and use.
- **Object-oriented:** a clean, usable, pragmatic approach to objects, not restricted by the need for compatibility with other languages.
- **Robust:** restricts the programmer to find the mistakes early, performs compile-time (strong typing) and run-time (exception-handling) checks, manages memory automatically.
- **Multithreaded:** supports multi-threaded programming for writing program that perform concurrent computations.
- **Architecture-neutral:** Java Virtual Machine provides a platform independent environment for the execution of Java byte code
- **Interpreted and high-performance:** Java programs are compiled into an intermediate representation – byte code:

  a) can be later interpreted by any JVM

  b) can be also translated into the native machine code for efficiency.

The Java Buzzwords

- **Distributed:** Java handles TCP/IP protocols, accessing a resource through its URL much like accessing a local file.
- **Dynamic:** substantial amounts of run-time type information to verify and resolve access to objects at run-time.
- **Secure:** programs are confined to the Java execution environment and cannot access other parts of the computer.
- **Portability:** Many types of computers and operating systems are in use throughout the world—and many are connected to the Internet.
- For programs to be dynamically downloaded to all the various types of platforms connected to the Internet, some means of generating portable executable code is needed. The same mechanism that helps ensure security also helps create portability.
- Indeed, Java's solution to these two problems is both elegant and efficient.

Comments

- The java comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code for specific time.

Types of Java Comments

- There are 3 types of comments in java.
  - Single Line Comment
  - Multi Line Comment
  - Documentation Comment

Comments

1) Java Single Line Comment

The single line comment is used to comment only one line.

- **Syntax:**
- //This is single line comment

Example:

```
public class CommentExample1 {
public static void main(String[] args) {
 int i=10;//Here, i is a variable
System.out.println(i);
}
}
```

Output:

10

Comments

2) Java Multi Line Comment

The multi line comment is used to comment multiple lines of code.

Syntax:

```
/*
This
is
multi line
comment
*/
```

Example:

```
public class CommentExample2 {
public static void main(String[] args) {
/* Let's declare and
 print variable in java. */
   int i=10;
   System.out.println(i);
} }  Output: 10
```

Comments

3) Java Documentation Comment

The documentation comment is used to create documentation API. To create documentation API, you need to use **javadoc tool**.

Syntax:

```
/**
This
is
documentation
comment
*/
```

Data Types

- Java defines eight simple types:
  1)byte – 8-bit integer type
  2)short – 16-bit integer type
  3)int – 32-bit integer type
  4)long – 64-bit integer type
  5)float – 32-bit floating-point type
  6)double – 64-bit floating-point type
  7)char – symbols in a character set
  8)boolean – logical values true and false

Data Types

- **byte:** 8-bit integer type.
  Range: -128 to 127.
  Example: byte b = -15;
  Usage: particularly when working with data streams.
- **short:** 16-bit integer type.
  Range: -32768 to 32767.
  Example: short c = 1000;
  Usage: probably the least used simple type.
- **int:** 32-bit integer type.
  Range: -2147483648 to 2147483647.
  Example: int b = -50000;
  Usage:
  1) Most common integer type.
  2) Typically used to control loops and to index arrays.
  3) Expressions involving the byte, short and int values are promoted to int before calculation.

Data Types

- **long:** 64-bit integer type.
  Range: -9223372036854775808 to 9223372036854775807.
  Example: long l = 10000000000000000;
  Usage: 1) useful when int type is not large enough to hold the desired value

- **float:** 32-bit floating-point number.
  Range: 1.4e-045 to 3.4e+038.
  Example: float f = 1.5;
  Usage:
  1) fractional part is needed 2) large degree of precision is not required

- **double:** 64-bit floating-point number.
  Range: 4.9e-324 to 1.8e+308.
  Example: double pi = 3.1416;
  Usage:
  1) accuracy over many iterative calculations
  2) manipulation of large-valued numbers

Data Types

char: 16-bit data type used to store characters.

Range: 0 to 65536.

Example: char c = ‘a’;

Usage:

1) Represents both ASCII and Unicode character sets; Unicode defines a character set with characters found in (almost) all human languages.

2) Not the same as in C/C++ where char is 8-bit and represents ASCII only.

boolean: Two-valued type of logical values.

Range: values true and false.

Example: boolean b = (1<2);

Usage:

1) returned by relational operators, such as 1<2

2) required by branching expressions such as if or for

Variables

- declaration – how to assign a type to a variable
- initialization – how to give an initial value to a variable
- scope – how the variable is visible to other parts of the program
- lifetime – how the variable is created, used and destroyed
- type conversion – how Java handles automatic type conversion
- type casting – how the type of a variable can be narrowed down

- Java uses variables to store data.
- To allocate memory space for a variable JVM requires:
  1) to specify the data type of the variable
  2) to associate an identifier with the variable
  3) optionally, the variable may be assigned an initial value
- All done as part of variable declaration.

Variable Declaration

- datatype identifier [=value];
- datatype must be
  - A simple datatype
  - User defined datatype (class type)
- Identifier is a recognizable name confirm to identifier rules
- Value is an optional initial value.

We can declare several variables at the same time:

type identifier [=value][, identifier [=value] …];

Examples:

```
int a, b, c;
int d = 3, e, f = 5;
byte g = 22;
double pi = 3.14159;
char ch = 'x';
```

Variable Scope

- Scope determines the visibility of program elements with respect to other program elements.
- In Java, scope is defined separately for classes and methods:

  1) variables defined by a class have a global scope

  2) variables defined by a method have a local scope

  A scope is defined by a block:

  {

  …

  }

  A variable declared inside the scope is not visible outside:

  {

  int n;

  }

  n = 1;// this is illegal

Variable Lifetime

- Variables are created when their scope is entered by control flow and destroyed when their scope is left:
- A variable declared in a method will not hold its value between different invocations of this method.
- A variable declared in a block looses its value when the block is left.
- Initialized in a block, a variable will be re-initialized with every re-entry. Variables lifetime is confined to its scope!

Operators Types

- Java operators are used to build value expressions.
- Java provides a rich set of operators:
    1) assignment
    2) arithmetic
    3) relational
    4) logical
    5) bitwise

Arithmetic assignments

- Arithmetic assignments

+=	v += expr;	v = v + expr ;
-=	v -=expr;	v = v - expr ;
*=	v *= expr;	v = v * expr ;
/=	v /= expr;	v = v / expr ;
%=	v %= expr;	v = v % expr ;

Basic Arithmetic Operators

+	op1 + op2	ADD
-	op1 - op2	SUBSTRACT
*	op1 * op2	MULTIPLY
/	op1 / op2	DIVISION
%	op1 % op2	REMAINDER

Relational operator

==	Equals to	Apply to any type
!=	Not equals to	Apply to any type
>	Greater than	Apply to numerical type
<	Less than	Apply to numerical type
>=	Greater than or equal	Apply to numerical type
<=	Less than or equal	Apply to numerical type

Logical operators

&	op1 & op2	Logical AND
|	op1 | op2	Logical OR
&&	op1 && op2	Short-circuit AND
||	op1 || op2	Short-circuit OR
!	! op	Logical NOT
^	op1 ^ op2	Logical XOR

Bitwise operators

~	~op	Inverts all bits
&	op1 & op2	Produces 1 bit if both operands are 1
|	op1 |op2	Produces 1 bit if either operand is 1
^	op1 ^ op2	Produces 1 bit if exactly one operand is 1
>>	op1 >> op2	Shifts all bits in op1 right by the value of op2
<<	op1 << op2	Shifts all bits in op1 left by the value of op2

Operator Hierarchy

Operators	Precedence
postfix increment and decrement	++ --
prefix increment and decrement, and unary	++ -- + - ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>=>>>=

Ternary Operator

- Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements.
- This operator is the **?**, and it works in Java much like it does in C, C++, and C#. It can seem somewhat confusing at first, but the **?** can be used very effectively once mastered.

The **?** has this general form:

- expression1 **?** expression2 **:** expression3
- Here, expression1 can be any expression that evaluates to a **boolean** value. If expression1 is **true**, then expression2 is evaluated; otherwise, expression3 is evaluated.
- The result of the **?** operation is that of the expression evaluated. Both expression2 and expression3 are required to return the same type, which can't be **void**.

Expressions

- An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value.
- Examples of expressions are in bold below:

```
int number = 0;  anArray[0] = 100;
System.out.println ("Element 1 at index 0: " + anArray[0]);
int result = 1 + 2; // result is now 3 if(value1 == value2)
System.out.println("value1 == value2");
```

- The data type of the value returned by an expression depends on the elements used in the expression.
- The expression number = 0 returns an int because the assignment operator returns a value of the same data type as its left-hand operand; in this case, number is an int.
- As you can see from the other expressions, an expression can return other types of values as well, such as boolean or String. The Java programming language allows you to construct compound expressions from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other.
- Here's an example of a compound expression: 1 * 2 * 3

Type Conversion

- Size Direction of Data Type
  - Widening Type Conversion (Casting down)
    - Smaller Data Type → Larger Data Type
  - Narrowing Type Conversion (Casting up)
    - Larger Data Type → Smaller Data Type
- Conversion done in two ways
  - Implicit type conversion
    - Carried out by compiler automatically
  - Explicit type conversion
    - Carried out by programmer using casting

Type Conversion

- Widening Type Convertion
  - Implicit conversion by compiler automatically

byte -> short, int, long, float, double
short -> int, long, float, double
char -> int, long, float, double
int -> long, float, double
long -> float, double
float -> double

Type Conversion

- Narrowing Type Conversion
  - Programmer should describe the conversion explicitly

byte -> char
short -> byte, char
char -> byte, short
int -> byte, short, char
long -> byte, short, char, int
float -> byte, short, char, int, long
double -> byte, short, char, int, long, float

Type Casting

- byte and short are always promoted to int
- if one operand is long, the whole expression is promoted to long
- if one operand is float, the entire expression is promoted to float
- if any operand is double, the result is double

- General form: (targetType) value
- Examples:
- 1) integer value will be reduced module bytes range:

```
int i;
byte b = (byte) i;
```

- 2) floating-point value will be truncated to integer value:

```
float f;
int i = (int) f;
```

Type Casting

- byte and short are always promoted to int
- if one operand is long, the whole expression is promoted to long
- if one operand is float, the entire expression is promoted to float
- if any operand is double, the result is double

- General form: (targetType) value
- Examples:
- 1) integer value will be reduced module bytes range:

```
int i;
byte b = (byte) i;
```

- 2) floating-point value will be truncated to integer value:

```
float f;
int i = (int) f;
```

Enumerated types

The **Enum in Java** is a data type which contains a fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc.

According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

Points to remember for Java Enum

- Enum improves type safety
- Enum can be easily used in switch
- Enum can be traversed
- Enum can have fields, constructors and methods
- Enum may implement many interfaces but cannot extend any class because it internally extends Enum class

Enumerated types

Simple Example of Java Enum

```
1.class EnumExample1{
2.//defining the enum inside the class
3.public enum Season { WINTER, SPRING, SUMMER, FALL }
4.//main method
5.public static void main(String[] args) {
6.//traversing the enum
7.for (Season s : Season.values())
8.System.out.println(s);
9.}}
```

Output:
WINTER
SPRING
SUMMER
FALL

Control Statements

- Java control statements cause the flow of execution to advance and branch based on the changes to the state of the program.
- Control statements are divided into three groups:
    1. selection statements allow the program to choose different parts of the execution based on the outcome of an expression
    2. iteration statements enable program execution to repeat one or more statements
    3. jump statements enable your program to execute in a non-linear fashion

Selection Statements

Java selection statements allow to control the flow of program's execution based upon conditions known only during run-time.

Java provides four selection statements:

1) if
2) if-else
3) if-else-if
4) switch

Control Statements

Iteration Statements

- Java iteration statements enable repeated execution of part of a program until a certain termination condition becomes true.
- Java provides three iteration statements:

  1) while
  2) do-while
  3) for

Jump Statements

Java jump statements enable transfer of control to other parts of program.
Java provides three jump statements:

1) break
2) continue
3) return

In addition, Java supports exception handling that can also alter the control flow of a program.

Simple Java Program

A class to display a simple message:

```
class MyProgram
{
        public static void main(String[] args)
        {
                System.out.println(“First Java program.");
        }
}
```

Arrays

- An array is a group of liked-typed variables referred to by a common name, with individual variables accessed by their index.
- Arrays are:
  1) declared
  2) created
  3) initialized
  4) used
- Also, arrays can have one or several dimensions.

Array Declaration

Array declaration involves:

1) declaring an array identifier
2) declaring the number of dimensions
3) declaring the data type of the array elements

Two styles of array declaration:

type array-variable[];

or

type [] array-variable;

Array Creation

- After declaration, no array actually exists.
- In order to create an array, we use the new operator:

  type array-variable[];

  array-variable = new type[size];
- This creates a new array to hold size elements of type type, which reference will be kept in the variable array-variable.

Array Indexing

- Later we can refer to the elements of this array through their indexes:
- array-variable[index]
- The array index always starts with zero!
- The Java run-time system makes sure that all array indexes are in the correct range, otherwise raises a run-time error.

Array Initialization

- Arrays can be initialized when they are declared:
- int monthDays[] = {31,28,31,30,31,30,31,31,30,31,30,31};

Note:

1) there is no need to use the new operator

2) the array is created large enough to hold all specified elements

Multidimensional Arrays

Multidimensional arrays are arrays of arrays:

1) declaration: int array[][];

2) creation: int array = new int[2][3];

3) initialization: int array[][] = { {1, 2, 3}, {4, 5, 6} };

Advantages of Arrays:

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

SINGLE DIMENSIONAL ARRAY

```
1.//Java Program to illustrate how to declare, instantiate, initialize
2.//and traverse the Java array.
3.class Testarray{
4.public static void main(String args[]){
5.int a[]=new int[5];//declaration and instantiation
6.a[0]=10;//initialization
7.a[1]=20;
8.a[2]=70;
9.a[3]=40;
10.a[4]=50;
11.//traversing array
12.for(int i=0;i<a.length;i++)//length is the property of array
13.System.out.println(a[i]);
14.}}
```

MULTI DIMENSIONAL ARRAY

```
1.//Java Program to illustrate the use of multidimensional array
2.class Testarray3{
3.public static void main(String args[]){
4.//declaring and initializing 2D array
5.int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
6.//printing 2D array
7.for(int i=0;i<3;i++){
8. for(int j=0;j<3;j++){
9.   System.out.print(arr[i][j]+" ");
10. }
11. System.out.println();
12.}
13.}}
```

PARAMETER PASSING

- There are different ways in which parameter data can be passed into and out of methods and functions.
- Let us assume that a function *B()* is called from another function *A()*. In this case *A* is called the ***"caller function"*** and *B* is called the ***"called function or callee function"***.
- Also, the arguments which *A* sends to *B* are called *actual arguments* and the parameters of *B* are called *formal arguments*.

Important methods of Parameter Passing

1. **Pass By Value:** Changes made to formal parameter do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. This method is also called as ***call by value*****.** Java infact it is **s**trictly **call by value.**

PARAMETER PASSING

Example of call by value in java

```
1.class Operation{
2. int data=50;
3.
4. void change(int data){
5. data=data+100;//changes will be in the local variable only
6. }
7.
8. public static void main(String args[]){
9.   Operation op=new Operation();
10.
11.   System.out.println("before change "+op.data);
12.   op.change(500);
13.   System.out.println("after change "+op.data);
14.
15. }
16.} Output: before change 50
              after change 50
```

PARAMETER PASSING

2. Pass By Value

- In case of call by reference original value is changed if we made changes in the called method.
- If we pass object in place of any primitive value, original value will be changed. In this example we are passing object as a value.

Let's take a simple example:

```
class Operation2{
 int data=50;
   void change(Operation2 op){
 op.data=op.data+100;//changes will be in the instance variable
 }
   public static void main(String args[]){
   Operation2 op=new Operation2();
     System.out.println("before change "+op.data);
   op.change(op);//passing object
   System.out.println("after change "+op.data);
   }
}
```

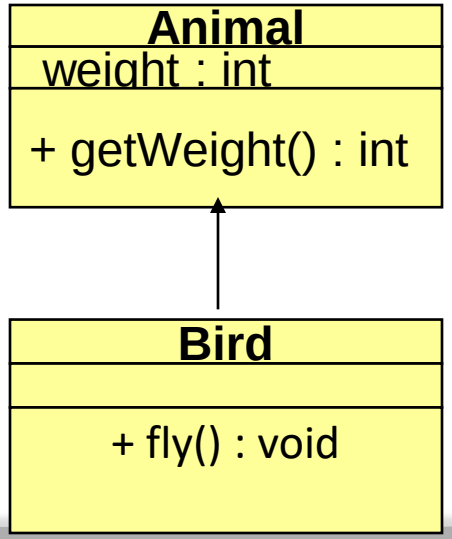
Output: before change 50 after change 150

UNIT – II

Inheritance: Inheritance hierarchies, super and subclasses, member access rules, Polymorphism: Dynamic binding, method overriding, abstract classes and methods.

Inheritance

- Methods allows a software developer to reuse a sequence of statements
- *Inheritance* allows a software developer to reuse classes by deriving a new class from an existing one
- The existing class is called the *parent class,* or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent
- That is, the child class *inherits* the methods and data defined for the parent class.
- Inheritance relationships are often shown graphically in a *class diagram*, with the arrow pointing to the parent class



Inheritance should create an *is-a relationship*, meaning the child *is a* more specific version of the parent

Deriving Subclasses

- In Java, we use the reserved word extends to establish an inheritance relationship

```
class Animal
    {
      // class contents
       int weight;
public void int getWeight() {…}
    }

    class Bird extends Animal
    {
      // class contents
public void fly() {…};
    }
```

INHERITANCE

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

Class Hierarchy

- A child class of one parent can be the parent of another child, forming *class hierarchies.*

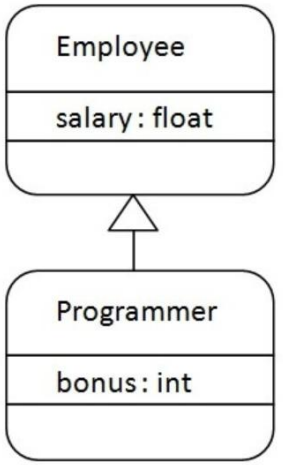
The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

- The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
- In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

As displayed in the figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

Java Inheritance Example



Inheritance

Example program of Inheritance:

```
1.class Employee{
2. float salary=40000;
3.}
4.class Programmer extends Employee{
5. int bonus=10000;
6. public static void main(String args[]){
7.   Programmer p=new Programmer();
8.   System.out.println("Programmer salary is:"+p.salary);
9.   System.out.println("Bonus of Programmer is:"+p.bonus);
10.}
11.}
```

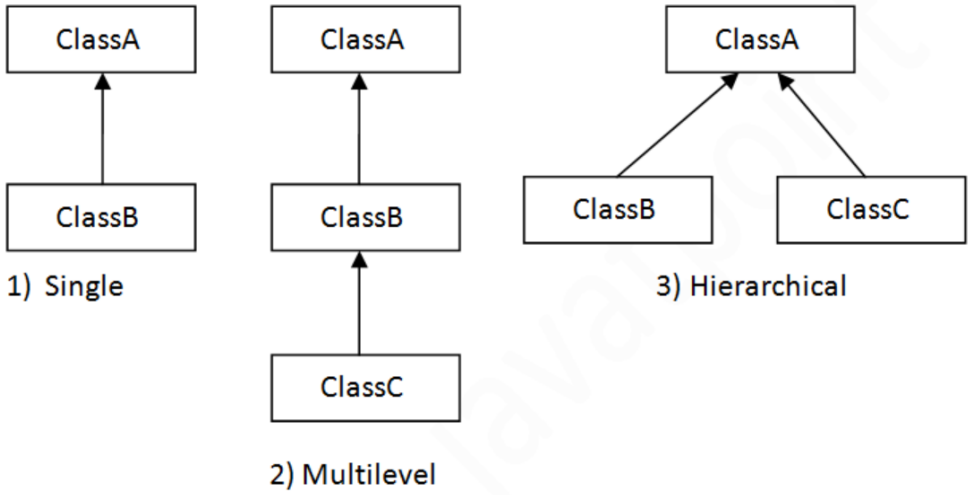
Output:

Programmer salary is:40000.0

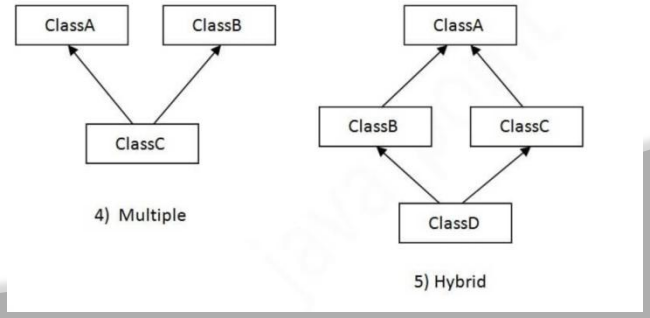
Bonus of programmer is:10000

Types of Inheritance

- On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
- In java programming, multiple and hybrid inheritance is supported through interface only.



When one class inherits multiple classes, it is known as multiple inheritance. For Example:



Types of Inheritance

Single Inheritance Examples:

```
1.class Animal{
2.void eat(){System.out.println("eating...");}
3.}
4.class Dog extends Animal{
5.void bark(){System.out.println("barking...");}
6.}
7.class TestInheritance{
8.public static void main(String args[]){
9.Dog d=new Dog();
10.d.bark();
11.d.eat();
12.}}
```

Output:

```
barking...
eating...
```

Types of Inheritance

Why multiple inheritance is not supported in java?

- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
- Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.
- Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B
{//suppose if it were
   public static void main(String args[]){
  C obj=new C();
  obj.msg();//Now which msg() method would be invoked?
} }
```

Member Access Rules

- Visibility modifiers determine which class members are accessible and which do not
- Members (variables and methods) declared with public visibility are accessible, and those with private visibility are not
- Problem: How to make class/instance variables visible only to its subclasses?
- Solution: Java provides a third visibility modifier that helps in inheritance situations: protected

Visibility Modifiers for class/interface:

Private: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

Default: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Protected: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

Public: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Member Access Rules

- Understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Polymorphism

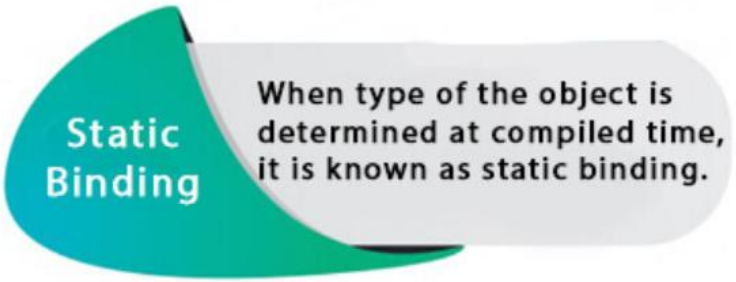
- Polymorphism is one of three pillars of object-orientation.
- Polymorphism: many different (poly) forms of objects that share a common interface respond differently when a method of that interface is invoked:
  1) a super-class defines the common interface
  2) sub-classes have to follow this interface (inheritance), but are also permitted to provide their own implementations (overriding)
- A sub-class provides a specialized behaviors relying on the common elements defined by its super-class.

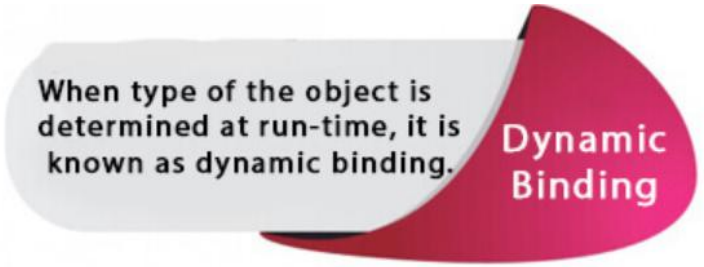
Connecting a method call to the method body is known as binding.

There are two types of binding

1.Static Binding (also known as Early Binding).

2.Dynamic Binding (also known as Late Binding).





Polymorphism

1. Static binding

- When type of the object is determined at compiled time(by the compiler), it is known as static binding.
- If there is any private, final or static method in a class, there is static binding.
- The binding which can be resolved at compile time by compiler is known as static or early binding. The binding of static, private and final methods is compile-time. **Why?** The reason is that the these method cannot be overridden and the type of the class is determined at the compile time. Lets see an example to understand this:

Example of static binding

```
1.class Dog{
2. private void eat()
3.{
4.System.out.println("dog is eating...");
5.}
6. public static void main(String args[]){
7.  Dog d1=new Dog();
8.  d1.eat();
9. }
10.}
```

Polymorphism

2. Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

```
1.  class Animal{
2.   void eat(){System.out.println("animal is eating...");}
3.  }
4.   class Dog extends Animal{
5.   void eat(){System.out.println("dog is eating...");}
6.    public static void main(String args[]){
7.    Animal a=new Dog();
8.    a.eat();
9.   }
10. }  Output: dog is eating...
```

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal. So compiler doesn't know its type, only its base type.

Polymorphism

Understanding Type: Let's understand the type of instance.

1) variables have a type : Each variable has a type, it may be primitive and non-primitive.

```
1.int data=30;  Here data variable is a type of int.
```

2) References have a type

```
1.class Dog{
2. public static void main(String args[]){
3.  Dog d1;//Here d1 is a type of Dog
4. } }
```

3) Objects have a type

An object is an instance of particular java class,but it is also an instance of its superclass.

```
1.class Animal{}
2. class Dog extends Animal{
3. public static void main(String args[]){
4.  Dog d1=new Dog();
5. } }
```

Here d1 is an instance of Dog class, but it is also an instance of Animal.

Method Overloading

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.
- Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

1.By changing number of arguments

2.By changing the data type

Method Overloading

1) Method Overloading: changing no. of arguments

In this example, created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.
In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
1.class Adder{
2.static int add(int a,int b){return a+b;}
3.static int add(int a,int b,int c){return a+b+c;}
4.}
5.class TestOverloading1{
6.public static void main(String[] args){
7.System.out.println(Adder.add(11,11));
8.System.out.println(Adder.add(11,11,11));
9.}}
```

Output:

22 33

Method Overloading

2) Method Overloading: changing data type of arguments

In this example, created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
1.class Adder{
2.static int add(int a, int b){return a+b;}
3.static double add(double a, double b){return a+b;}
4.}
5.class TestOverloading2{
6.public static void main(String[] args){
7.System.out.println(Adder.add(11,11));
8.System.out.println(Adder.add(12.3,12.6));
9.}}
```

Output:

22 24.9

Method Overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
- In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism.

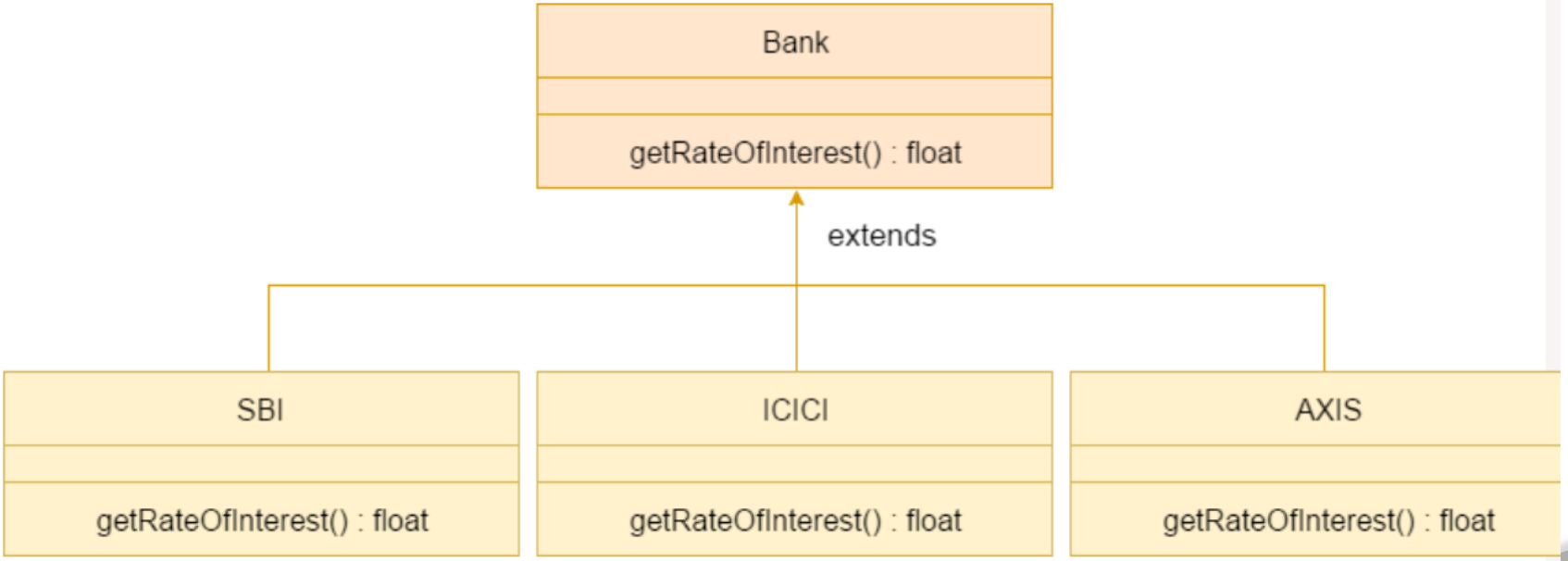
Rules for Java Method Overriding

1.The method must have the same name as in the parent class
2.The method must have the same parameter as in the parent class.
3.There must be an IS-A relationship (inheritance).

Method Overriding

A real example of Java Method Overriding

- Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks.
- For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.



Method Overriding

```
//Java Program to demonstrate the real scenario of Java Method Overriding
class Bank{
int getRateOfInterest(){return 0;}
}
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}
 class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
} }
```

Differences between Overloading and Overriding

Method Overloading	Method Overriding
Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Abstract classes and methods

Abstract class in Java

- A class which is declared with the abstract keyword is known as an abstract class in Java.
- It can have abstract and non-abstract methods (method with the body).

Abstraction in Java

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
- Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1.Abstract class (0 to 100%)

2.Interface (100%)

Abstract classes and methods

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Abstract classes and methods

Example of abstract class: **abstract class** A{}

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

abstract void printStatus();//no method body and abstract

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
} }
```

Output: running safely

Abstract classes and methods

Example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes. Mostly, don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.
A **factory method** is a method that returns the instance of the class.
In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.
File: TestAbstraction1.java

```
abstract class Shape{
abstract void draw();
}  //In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}  //In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() m
ethod
s.draw();
} }
```

Output: drawing circle

UNIT – III

Exception Handling: Benefits of exception handling, the classification of exceptions, usage of try, catch, throw, throws and finally.

Multithreading: Differences between multiple processes and multiple threads, thread states, creating threads, interrupting threads.

Exception Handling

What is Exception

- Exception is an abnormal condition that arises when executing a program.
- In the languages that do not support exception handling, errors must be checked and handled manually, usually through the use of error codes.
- In contrast, Java:
  1) provides syntactic mechanisms to signal, detect and handle errors
  2) ensures a clean separation between the code executed in the absence of errors and the code to handle various kinds of errors
  3) brings run-time error management into object-oriented programming

What is Exception Handling

- An exception is an object that describes an exceptional condition (error) that has occurred when executing a program.
- Exception handling involves the following:
  1) when an error occurs, an object (exception) representing this error is created and thrown in the method that caused it
  2) that method may choose to handle the exception itself or pass it on
  3) either way, at some point, the exception is caught and processed

Exception Handling

Exception Sources

- Exceptions can be:

  1) generated by the Java run-time system Fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.

  2) manually generated by programmer's code Such exceptions are typically used to report some error conditions to the caller of a method.

Benefits of exception handling

- Separating Error-Handling code from "regular" business logic code
- Propagating errors up the call stack
- Grouping and differentiating error types

Exception Handling

Exception Hierarchy

- All exceptions are sub-classes of the build-in class Throwable.
- Throwable contains two immediate sub-classes:

1) Exception – exceptional conditions that programs should catch

The class includes:

a) Runtime Exception – defined automatically for user programs to include: division by zero, invalid array indexing, etc.

b) use-defined exception classes

2) Error – exceptions used by Java to indicate errors with the runtime environment; user programs are not supposed to catch them

General form:

```
try { … }
catch(Exception1 ex1) { … }
catch(Exception2 ex2) { … }
…finally { … }
```

where:

1) try { … } is the block of code to monitor for exceptions
2) catch(Exception ex) { … } is exception handler for the exception Exception
3) finally { … } is the block of code to execute before the try block ends

Exception Handling

Exception Hierarchy

- All exceptions are sub-classes of the build-in class Throwable.
- Throwable contains two immediate sub-classes:

1) Exception – exceptional conditions that programs should catch

The class includes:

a) Runtime Exception – defined automatically for user programs to include: division by zero, invalid array indexing, etc.

b) use-defined exception classes

2) Error – exceptions used by Java to indicate errors with the runtime environment; user programs are not supposed to catch them

General form:

```
try { … }
catch(Exception1 ex1) { … }
catch(Exception2 ex2) { … }
…finally { … }
```

where:

1) try { … } is the block of code to monitor for exceptions
2) catch(Exception ex) { … } is exception handler for the exception Exception
3) finally { … } is the block of code to execute before the try block ends

Exception Handling

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Exception Handling

Java Exception Handling Example

Let's see an example of Java Exception Handling where we using a try-catch statement to handle the exception.

```
1.public class JavaExceptionExample{
2.  public static void main(String args[]){
3.   try{
4.      //code that may raise exception
5.      int data=100/0;
6.   }catch(ArithmeticException e){System.out.println(e);}
7.   //rest code of the program
8.   System.out.println("rest of the code...");
9.  }
10.}
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero rest of the code...

Exception Handling

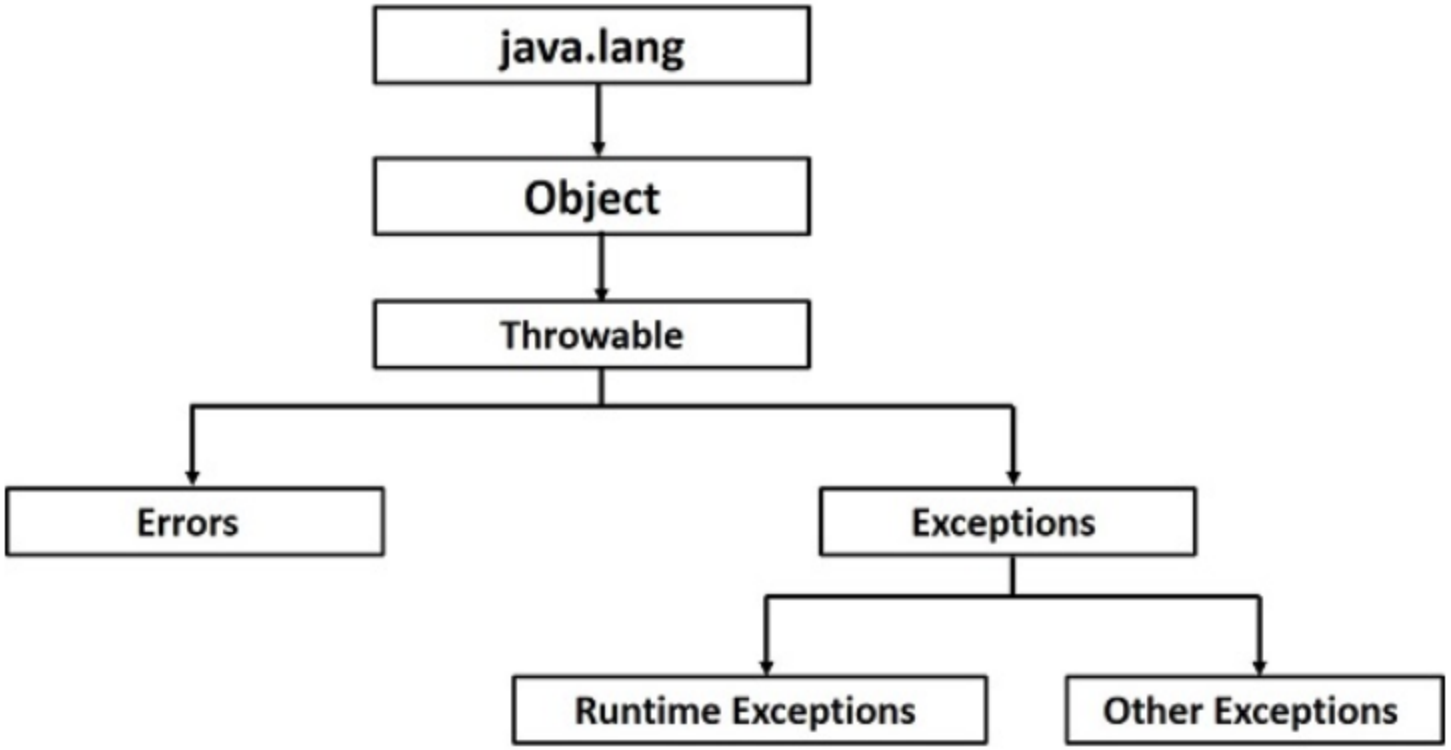
```
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        try {
            int a = 30, b = 0;
            int c = a/b;  // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch(ArithmeticException e) {
            System.out.println ("Can't divide a number by 0");
        }
    }
}
```

Output:

```
Can't divide a number by 0
```

Exception Handling

The Exception class has two main subclasses: IOException class and RuntimeException Class.



Exception Handling

Usage of try block

- Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.
- If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keeping the code in try block that will not throw an exception.
- Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
1. try{
2. //code that may throw an exception
3. }catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
1. try{
2. //code that may throw an exception
3. }finally{}
```

Exception Handling

The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
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"); } }
```

OutPut: Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block

Exception Handling

Usage of catch block

- Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.
- The catch block must be used after the try block only. You can use multiple catch block with a single try block.

```
public class TryCatchExample3 {
    public static void main(String[] args) {
     try
     {
     int data=50/0; //may throw exception
                    // if exception occurs, the remaining statement will not exceute
     System.out.println("rest of the code");
     }
     catch(ArithmeticException e)
     {
        System.out.println(e);
     } } }
```

Exception Handling

Multiple catch block

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following −

Syntax

```
try
{ // Protected code }
catch (ExceptionType1 e1)

{ // Catch block }
catch (ExceptionType2 e2)

{ // Catch block }
catch (ExceptionType3 e3)
{ // Catch block }
```

Exception Handling

Here is code segment showing how to use multiple try/catch statements.

```
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;
}
```

Exception Handling

Java Exception Handling Example

Example of Java Exception Handling where by using a try-catch statement to handle the exception.

```
1.public class JavaExceptionExample{
2.  public static void main(String args[]){
3.   try{
4.      //code that may raise exception
5.      int data=100/0;
6.   }catch(ArithmeticException e){System.out.println(e);}
7.   //rest code of the program
8.   System.out.println("rest of the code...");
9.  }
10.}
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero rest of the code...

Exception Handling

Throw Keyword

- So far, we were only catching the exceptions thrown by the Java system.
- In fact, a user program may throw an exception explicitly:

  throw ThrowableInstance;

- ThrowableInstance must be an object of type Throwable or its subclass.

Once an exception is thrown by:

  throw ThrowableInstance;

Two ways to obtain a Throwable instance:

1) creating one with the new operator

   All Java built-in exceptions have at least two Constructors:

   One without parameters and another with one String parameter:

   throw new NullPointerException("demo");

2) using a parameter of the catch clause

   try { … } catch(Throwable e) { … e … }

Exception Handling

Throw Keyword example

In this example, creat the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
1.public class TestThrow1{
2.   static void validate(int age){
3.     if(age<18)
4.      throw new ArithmeticException("not valid");
5.     else
6.      System.out.println("welcome to vote");
7.   }
8.   public static void main(String args[]){
9.      validate(13);
10.       System.out.println("rest of the code...");
11.  }
12.}
```

Output:

Exception in thread main java.lang.ArithmeticException:not valid

Exception Handling

Throws Keyword example

- The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of java throws

1.return_type method_name() **throws** exception_class_name{
2.//method code
3.}

- The throwOne method throws an exception that it does not catch, nor declares it within the throws clause.

```
class ThrowsDemo {
static void throwOne() {
System.out.println("Inside
    throwOne.");
throw new
    IllegalAccessException("demo");
}
public static void main(String args[])
    {
throwOne();
}
}
```

- Therefore this program does not compile.

- Corrected program: throwOne lists exception, main catches it:

```
class ThrowsDemo {
static void throwOne() throws
    IllegalAccessException {
System.out.println("Inside
    throwOne.");
throw new
    IllegalAccessException("demo");
}
public static void main(String args[])
    {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
} } }
```

Exception Handling

finally Keyword

- The try/catch statement requires at least one catch or finally clause, although both are optional:

```
try { … }
catch(Exception1 ex1) { … } …
finally { … }
```

- Executed after try/catch whether of not the exception is thrown.
- Any time a method is to return to a caller from inside the try/catch block via:

  1) uncaught exception or

  2) explicit return

  the finally clause is executed just before the method returns.

- **FileNotFoundException:** Signals that an attempt to open the file denoted by a specified pathname has failed.
- **InterruptedIOException:** Signals that an I/O operation has been interrupted
- **InvalidClassException:** Thrown when the Serialization runtime detects one of the following problems with a Class.
- **InvalidObjectException:** Indicates that one or more deserialized objects failed validation tests.
- **IOException:** Signals that an I/O exception of some sort has occurred.

Multithreading

- **Multithreading in java** is a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
- However, multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Advantages of Java Multithreading

1. It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
2. You **can perform many operations together, so it saves time**.
3. Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multithreading

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

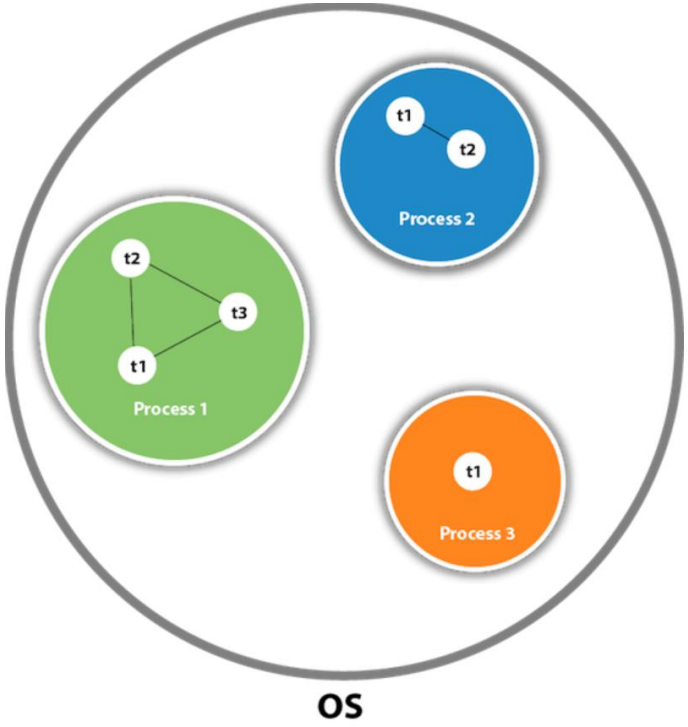
- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

Multithreading

What is Thread

- A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.
- Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

As shown in the figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.



Multithreading

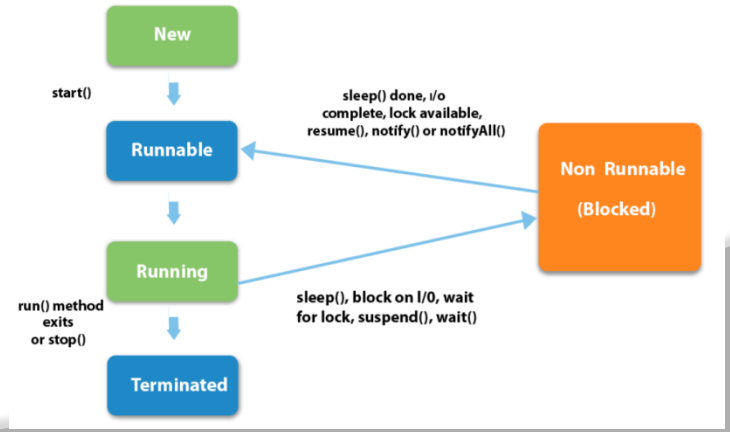
Thread States

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

Thread exist in several states:

1) ready to run
2) running
3) a running thread can be suspended
4) a suspended thread can be resumed
5) a thread can be blocked when waiting for a resource
6) a thread can be terminated

•Once terminated, a thread cannot be resumed.



Multithreading

Thread States

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

Multithreading

How to create thread

There are two ways to create a thread:

1.By extending Thread class

2.By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

1. Thread() 2. Thread(String name) 3. Thread(Runnable r) 4. Thread(Runnable r,String name)

Runnable interface:

1. The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().**public void run():** is used to perform action for a thread.

Starting a thread:

- **start() method** of Thread class is used to start a newly created thread. It performs following tasks:A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

Multithreading

1) Java Thread Example by extending Thread class

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}
```

Output:thread is running...

Multithreading

1) Java Thread Example by extending Thread class

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}
```

Output:thread is running...

Multithreading

1) Java Thread Example by extending Thread class

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}
```

Output:thread is running...

2) Java Thread Example by implementing Runnable interface

```
1.class Multi3 implements Runnable{
2.public void run(){
3.System.out.println("thread is running...");
4.}
5.
6.public static void main(String args[]){
7.Multi3 m1=new Multi3();
8.Thread t1 =new Thread(m1);
9.t1.start();
10. }
11.}
```

Output: thread is running...

If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.

Multithreading

2) Java Thread Example by implementing Runnable interface

```
1.class Multi3 implements Runnable{
2.public void run(){
3.System.out.println("thread is running...");
4.}
5.
6.public static void main(String args[]){
7.Multi3 m1=new Multi3();
8.Thread t1 =new Thread(m1);
9.t1.start();
10. }
11.}
```

Output: thread is running...

If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.

Multithreading

Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.3 constants defined

in Thread class:

1.public static int MIN_PRIORITY
2.public static int NORM_PRIORITY
3.public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Multithreading

Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
 public void run(){
  System.out.println("running thread name is:"+Thread.currentThread().getName())
;
  System.out.println("running thread priority is:"+Thread.currentThread().getPriorit
y());
   }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();
   } }
```

Output:running thread name is:Thread-0 running thread priority is:10 running thread name is:Thread-1 running thread priority is:1

Multithreading

Interrupting Thread

- If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException.
- If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true.
- Methods provided by the Thread class for thread interruption.

The 3 methods provided by the Thread class for interrupting a thread

- **public void interrupt()**
- **public static boolean interrupted()**
- **public boolean isInterrupted()**

Example of interrupting a thread that stops working

In this example, after interrupting the thread, we are propagating it, so it will stop working. If we don't want to stop the thread, we can handle it where sleep() or wait() method is invoked. Let's first see the example where we are propagating the exception.

```
class TestInterruptingThread1 extends Thread{
public void run(){
try{
Thread.sleep(1000);
System.out.println("task");
}catch(InterruptedException e){
throw new RuntimeException("Thread interrupted..."+e);
} } public static void main(String args[]){
TestInterruptingThread1 t1=new TestInterruptingThread1();
t1.start();
try{
t1.interrupt();
}catch(Exception e){System.out.println("Exception handled "+e);}}}
```

Output: Exception in thread-0 java.lang.RuntimeException: Thread interrupted...
java.lang.InterruptedException: sleep interrupted at A.run(A.java:7)

Multithreading

Example of interrupting a thread that doesn't stop working

In this example, after interrupting the thread, handled the exception, so it will break out the sleeping but will not stop working.

```
class TestInterruptingThread2 extends Thread{
public void run(){
try{
Thread.sleep(1000);
System.out.println("task");
}catch(InterruptedException e){
System.out.println("Exception handled "+e);
}  System.out.println("thread is running...");  }
 public static void main(String args[]){
TestInterruptingThread2 t1=new TestInterruptingThread2();
t1.start();
 t1.interrupt();  }}
```

Output: Exception handled java.lang.InterruptedException:
sleep interrupted thread is running...

Example of interrupting thread that behaves normally

If thread is not in sleeping or waiting state, calling the interrupt() method sets the interrupted flag to true that can be used to stop the thread by the java programmer later.

```
1.class TestInterruptingThread3 extends Thread{
2.public void run(){
3.for(int i=1;i<=5;i++)
4.System.out.println(i);
5.}
6.public static void main(String args[]){
7.TestInterruptingThread3 t1=new TestInterruptingThread3();
8.t1.start();
9.t1.interrupt();
10.}
11.}
```

Output:1 2 3 4 5

UNIT – IV
INTERFACES AND PACKAGES

Interface: Interfaces vs Abstract classes, defining an interface, implement interfaces, Packages: Defining, creating and accessing a package, importing packages.

Differences between classes and interfaces

- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- One class can implement any number of interfaces.
- Interfaces are designed to support dynamic method resolution at run time.
- Interface is little bit like a class... but interface is lack in instance variables....that's u can't create object for it.....
- Interfaces are developed to support multiple inheritance...
- The methods present in interfaces r pure abstract..
- The access specifiers public,private,protected are possible with classes, but the interface uses only one spcifier public.....
- interfaces contains only the method declarations.... no definitions.....
- A interface defines, which method a class has to implement. This is way - if you want to call a method defined by an interface - you don't need to know the exact class type of an object, you only need to know that it implements a specific interface.
- Another important point about interfaces is that a class can implement multiple interfaces.

Differences between interfaces vs Abstract classes

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

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Defining a interface

- Using interface, we specify what a class must do, but not how it does this.
- An interface is syntactically similar to a class, but it lacks instance variables and its methods are declared without any body.
- An interface is defined with an interface keyword.

❖ An interface declaration consists of modifiers, the keyword interface, the interface name, a comma-separated list of parent interfaces (if any), and the interface body.
For example:

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {
// constant declarations double E = 2.718282;
// base of natural logarithms //
//method signatures
 void doSomething (int i, double x);
 int doSomethingElse(String s);
 }
```

❖ The public access specifier indicates that the interface can be used by any class in any package. If you do not specify that the interface is public, your interface will be accessible only to classes defined in the same package as the interface.

❖ An interface can extend other interfaces, just as a class can extend or subclass another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces. The interface declaration includes a comma-separated list of all the interfaces that it extends

Implementing interface

General format:

interface interface name {

type method-name1(parameter-list);

type method-name2(parameter-list);

…

type var-name1 = value1;

type var-nameM = valueM;

…

}

- Interface methods have no bodies – they end with the semicolon after the parameter list.
- They are essentially abstract methods.
- An interface may include variables, but they must be final, static and initialized with a constant value.
- In a public interface, all members are implicitly public.

Implementing interface

- A class implements an interface if it provides a complete set of methods defined by this interface.

  1) any number of classes may implement an interface

  2) one class may implement any number of interfaces

- Each class is free to determine the details of its implementation.
- Implementation relation is written with the implements keyword.

General format of a class that includes the implements clause:

- Syntax:

  access class name extends super-class implements interface1, interface2, …, interfaceN {

  …

  }

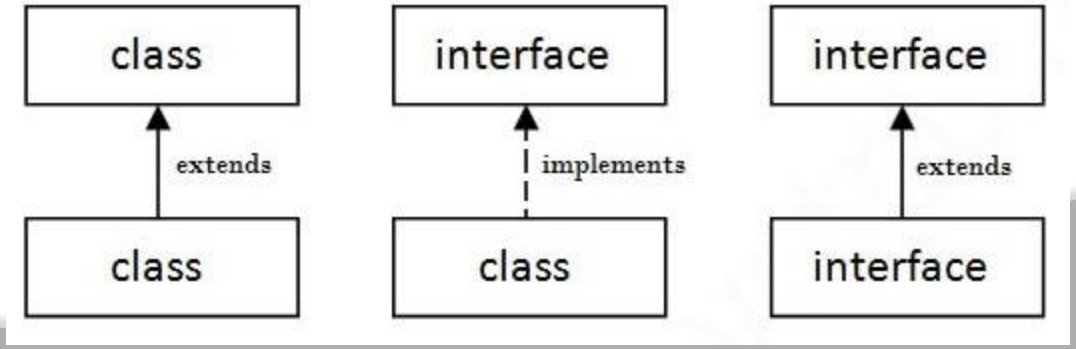
- Access is public or default.

Implementation Comments

- If a class implements several interfaces, they are separated with a comma.
- If a class implements two interfaces that declare the same method, the same method will be used by the clients of either interface.
- The methods that implement an interface must be declared public.
- The type signature of the implementing method must match exactly the type signature specified in the interface definition.

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Implementation Comments

Java Interface Example

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

```
1.interface printable{
2.void print();
3.}
4.class A6 implements printable{
5.public void print(){System.out.println("Hello");}
6.
7.public static void main(String args[]){
8.A6 obj = new A6();
9.obj.print();
10. }
11.}
```

Output:

Hello

Implementation Comments

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

File: TestInterface1.java

```
//Interface declaration: by first user
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
d.draw();
}}
```

Output: drawing circle

Extending interfaces

- One interface may inherit another interface.
- The inheritance syntax is the same for classes and interfaces.

```
interface MyInterface1 {
void myMethod1(…) ;
}
interface MyInterface2 extends MyInterface1 {
void myMethod2(…) ;
}
```

- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.
- Consider interfaces A and B.

```
interface A {
void meth1();
void meth2();
}
```

B extends A:

```
interface B extends A {
void meth3();}
```

Extending interfaces

- **MyClass must implement all of A and B methods:**

```
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");
} }
```

- **Create a new MyClass object, then invoke all interface methods on it:**

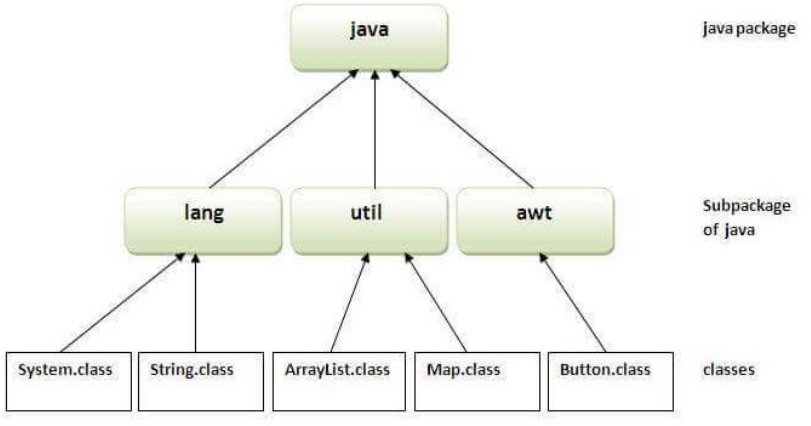
```
class IFExtend {
public static void main(String arg[]) {
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}}
```

Packages

- A **java package** is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.



Packages

Simple example of java package

The **package keyword** is used to create a package in java.

```
1.//save as Simple.java
2.package mypack;
3.public class Simple{
4. public static void main(String args[]){
5.    System.out.println("Welcome to package");
6.   }
7.}
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

javac -d directory javafilename

For example

```
javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

Packages

How to run java package program

Need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.SimpleOutput:Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

Creating a Package

- A package statement inserted as the first line of the source file:
  package myPackage;
  class MyClass1 { … }
  class MyClass2 { … }

- means that all classes in this file belong to the myPackage package.
- The package statement creates a name space where such classes are stored.
- When the package statement is omitted, class names are put into the default package which has no name.

Packages and Directories

- Java uses file system directories to store packages.
- Consider the Java source file:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

- The byte code files MyClass1.class and MyClass2.class must be stored in a directory myPackage.
- Case is significant! Directory names must match package names exactly.

Package Hierarchy

- To create a package hierarchy, separate each package name with a dot:

  package myPackage1.myPackage2.myPackage3;

- A package hierarchy must be stored accordingly in the file system:
  1) Unix myPackage1/myPackage2/myPackage3
  2) Windows myPackage1\myPackage2\myPackage3
  3) Macintosh myPackage1:myPackage2:myPackage3
- You cannot rename a package without renaming its directory!

Accessing a Package

There are three ways to access the package from outside the package.

1.import package.*; 2. import package.classname; 3. fully qualified name.

1) Using packagename.*

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
1.//save by A.java
2.package pack;
3.public class A{
4.  public void msg(){System.out.println("Hello");} }
5.//save by B.java
6.package mypack;
7.import pack.*;
8. class B{
9.  public static void main(String args[]){
10.   A obj = new A();
11.   obj.msg();
12. } } Output: Hello
```

Accessing a Package

2) Using packagename.classname

- If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.A;
  class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Output: Hello

Accessing a Package

3) Using fully qualified name

- If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

Output: Hello

Importing a Package

- Since classes within packages must be fully-qualified with their package names, it would be tedious to always type long dot-separated names.
- The import statement allows to use classes or whole packages directly.

Importing of a concrete class:

```
import myPackage1.myPackage2.myClass;
```

Importing of all classes within a package:

```
import myPackage1.myPackage2.*;
```

UNIT – V
FILES AND CONNECTING TO DATABASE

Files: streams – byte streams, character stream, text input/output, binary input/output, file management;
Connecting to Database: Connecting to a database, querying a database and processing the results, updating data with JDBC.

Streams – byte streams

- **Java I/O** (Input and Output) is used *to process the input* and *produce the output*.
- Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.
- We can perform **file handling in Java** by Java I/O API.

Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.
In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) System.out: standard output stream
2) System.in: standard input stream
3) System.err: standard error stream

Streams – byte streams

Byte Streams

- Java byte streams are used to perform input and output of 8-bit bytes.
- There are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**.

OutputStream vs InputStream

The explanation of OutputStream and InputStream classes are given below:

File Output Stream

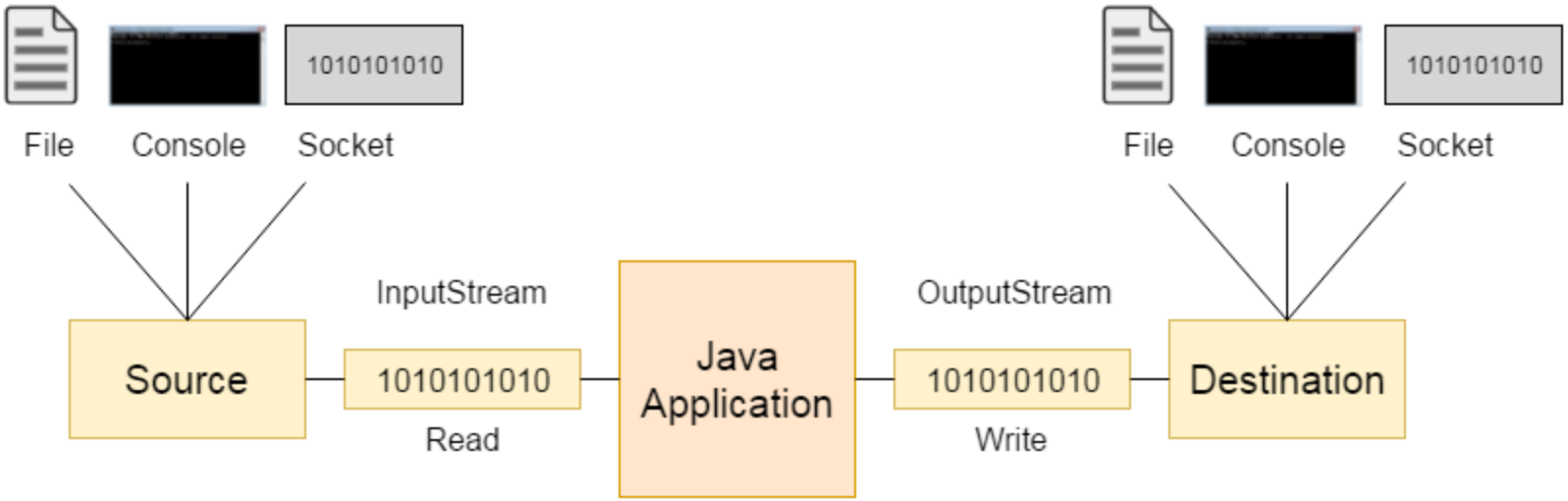
Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

File Input Stream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Streams – byte streams

- Let's understand the working of Java OutputStream and InputStream by the figure given below.



Streams – byte streams

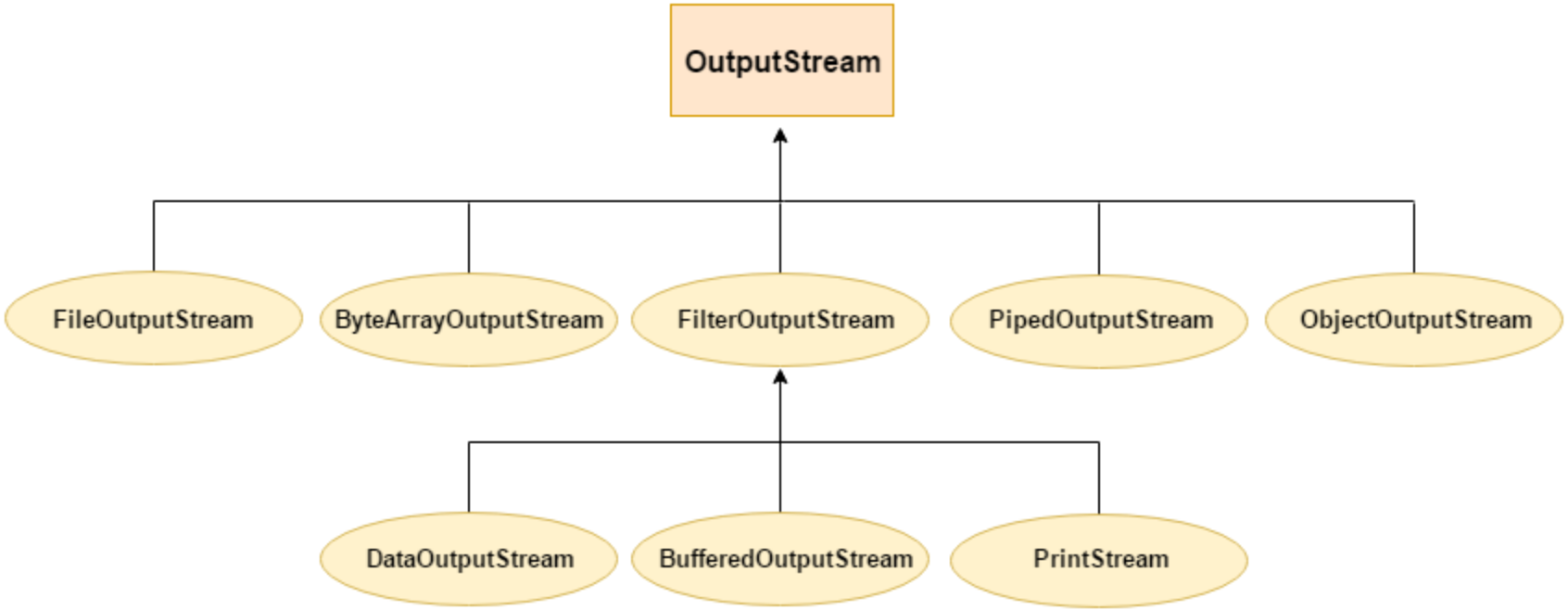
File Output Stream class

- This class is used to create a file and write data to it. It creates a new file if and only if the file does not exist. New file can be created using the keyword 'new'. OutputStream f = new FileOutputStream("d/java/sia");
- An output stream accepts output bytes and sends them to some sink.

Method	Description
1) void write(int b)	This method writes the byte specified by b
2) void write(byte[] buffer)	This method write an array containing bytes the output.
3) void write(byte[] buffer, int offset, int numBytes)	This method writes the number of bytes specified by numBytes from buffer to output stream. It begins writing from buffer[offset]
3)void flush()	This method flushes the output that has to be sent and contained in the buffer.
4) void close()	is used to close the current output stream.

Streams – byte streams

OutputStream Hierarchy



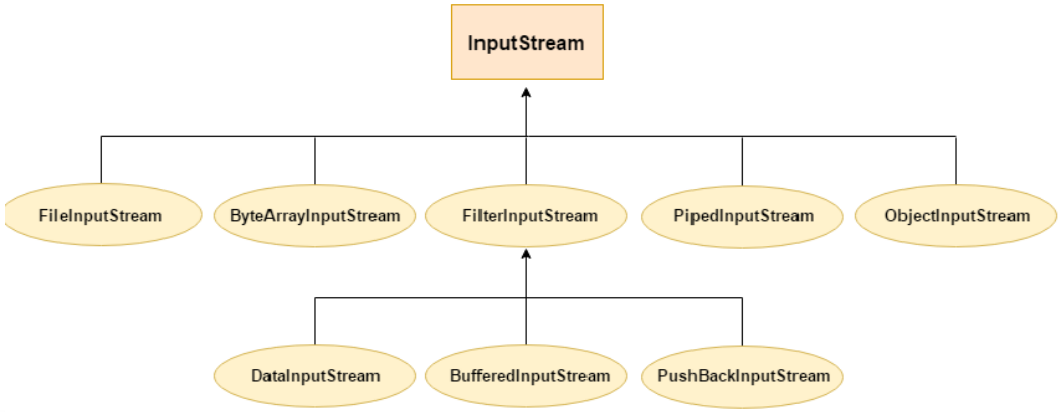
Streams – byte streams

Input Stream Class

- Input Stream class is an abstract class. It is the superclass of all classes representing an input stream of bytes. Useful methods

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

InputStream Hierarchy



Streams – Character streams

- **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode.
- There are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**.
- Internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

FileReader Class

The FileReader class is used to create a reader which reads the input from file. This class includes the following constructors.

- FileReader(string filepath)
  - Here, path defines the input file which is directly assigned to filereader object.
- FileReader(fileObj)
- Here file obj is the file object created for file class. The file class can create an object and read the path of the file.

Streams – Character streams

- **Methods in Reader class.**

Method	Description
void close()	This method will close the input source. An attempt made for reading generates IOException
void mark()	This method marks the specified number of characters by placing a mark to enable them for reading
int read()	This method reads a single character from the keyword. It returns the integer representation of it. A -1 is returned if it encounters end of stream
int read(char buffer[])	This method reds the specified number of characters into the buffer. It returns the number of character that are read and also returns integer value-1 if it encounters end of characters
boolean ready()	This method returns true, if the next input value is not in waiting state. If it is in waiting state, it returns false.

Streams – Character streams

FileWriter Class

The FileWriter class is used to write the content into the file. This class includes the following constructors.

- FileWriter (string filepath)
- FileWriter (string file path, boolean append)
- FileWriter (file fileobj)
- FileWriter (file fileobj, boolean append)

Here file path is the full path of the file and file obj is the object created for file class. When the appended boolean variable is true, then the output is appended at end of the file.

Streams – Character streams

- **Methods in Writer class.**

Method	Description
void close()	This method will close the file while writing. An attempt made for writing generates IOException.
void flush()	This method flushes the output that has to be sent and contained in the buffer.
void write(int ch)	This method writes a single character on output stream. Here parameter is declared as int and there is no need to cast it to char.
void write(char buffer[])	This method writes an array containing bytes to the output sream
void write(string str)	This method writes a string on to the calling output stream.
void write(string str, int offset, int numchars))	This method writes a subrange of characters specified by numchars from buffer to output stream. It begins writing from offset.

Text Input / Output

The text input and output operations can be performed using two classes. They are as follows.

1. Reader class (2) Writer class

(1) Reader class:

- Reader class can be used to read text as input from keyboard. While taking input, System.in method can be used which belongs to byte streams.
- This method can be defined in BufferedReader class. However, it is necessary to convert byte streams into text input by using InputStreamReader class.
- The constructor which can be used to convert byte stream into text input is as follows.

Syntax: InputStreamReader objbr= new InputStreamReader(System.in);
BufferedReader br= new BufferedReader(objbr);
(or)
BufferedReader objbr = new BufferedReader(new
InputStreamReader(System.in));

Text Input / Output

(2) Writer class

- Writer class can be used to write the output in the form ot text. While displaying output, System.out method can be used which belongs to byte streams.
- This method can be defined in BufferedWriter class. However, it is necessary to convert byte streams into text output by using OutputStreamWriter class.
- The constructor which can be used to convert byte stream into text output is as shown below.

Syntax: OutputStreamWriter objbw= new OutputStreamWriter(System.out);
BufferedWriter bw= new BufferedWriter(objbw);
(or)
BufferedWriter objbw = new BufferedWriter (new
OutputStreamWriter(System.out));

Binary input/output

The binary input and output operations are performed on binary values of priimi tive data types using two classes. They are,

1. DataInputStream class
2. DataOutputStream class

(1) DataInputStream class

- This class provides an opportunity to read input data in binary format, that can be implemented through DataInputinterface. It can be use the object of DataInputStream class, which can read the input values from different data types. The constructor of this class can be declared as follows,

Syntax: DataInputStream (InputStream obj Istream)

Binary input/output

- **Methods in this class is as follows**

Method	Description
boolean readBoolean()	It reads a boolean value as input and converts to binary value
byte readByte()	It reads a byte value and converts it into binary format
char readChar()	This method reads a character value and converts it into binary value
double readDouble	This method reads a double value and converts it into binary value
float readFloat()	This method reads a floating point value
int readInt()	This method reads an integer value
long readLong()	This method can read long data type value
short readShort()	This method reads short type value and converts it into binary value

Binary input/output

(2) DataOutputStream class

- This class can be used to write the values on output streams only in binary format. That can be implemented through DataOutputInterface. The methods than can provides the output are defined from this interface.

Syntax: DataOutputStream (OutputStream objstream)

Note: The DataInputStream and DataOutPutStream classes can throw IOException if an exception occurs. These classes can perform input and output operations only on binary values of data types.

Binary input/output

- **Methods in this class is as follows**

Method	Description
void writeBoolean()	This method writes boolean value on the output stream
void writeByte (int val)	It displays low-ordered byte value which can be specified in the variable val
void writeChar (int val)	This method writes a character value of double data type on the output stream
void writeDouble(double val)	This method can display the value of double data type on the output stream
void writeFloat (float val)	This method writes floating point value as output
void writeInt (int val)	This method writes integer value as output
void writeLong (long val)	This method writes long data value as output
void writeShort (int val)	This method can write the value which can be declared as short on to the output.

File Management

1. File class can be manage files by encapsulating the functionality of file system present in the computer.
2. This class can return the information such as last modified, removing and remaining of file. Stream classes can be managed with the contents of file on disk.

The various constructors that can be used by file class are as follows,

(a) File obffile = new File (“example.txt”);
(b) File obffile = new File (string path, string name);
(c) File obffile = new File (file dir, string name);
(d) File obffile = new File (file.separator+ “temp”);

File Management

- **There are different methods of file class that can be use dto manage a file.**

Method	Description
boolean canRead()	This method explains that whether the file can be readable or not.
boolean canWrite()	This method determines that the current file can be writable or not.
boolean canExecute()	This method can be used to determine whether the current file can be executed or not.
File get CannocialFile()	This method can return a file object which can hold the canonical path name of the file.
File getParentFile()	This method returns a file object which contains the name of the parent of this file directory.
File[] listfiles()	This method returns the file object for respective files and of file class. These objects can be displayed directories in array.

File Management

- **There are different methods of file class that can be use dto manage a file.**

Method	Description
boolean isFile()	This method determines whether the specified file is represents a file or not. If it is a file, it return true otherwise returns false.
boolean isDirectory()	This method returns true if the specified file represent a directory otherwise, it returns false.
boolean exist()	This method checks whether the specific file is existed or not. If existed, it return true otherwise returns false.
boolean mkdir()	This method will create a new directory by giving the argument in file objec as name of the directory. If the directory is created, this method returns true, otherwise returns false.

Java Database Connectivity (JDBC)

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database.

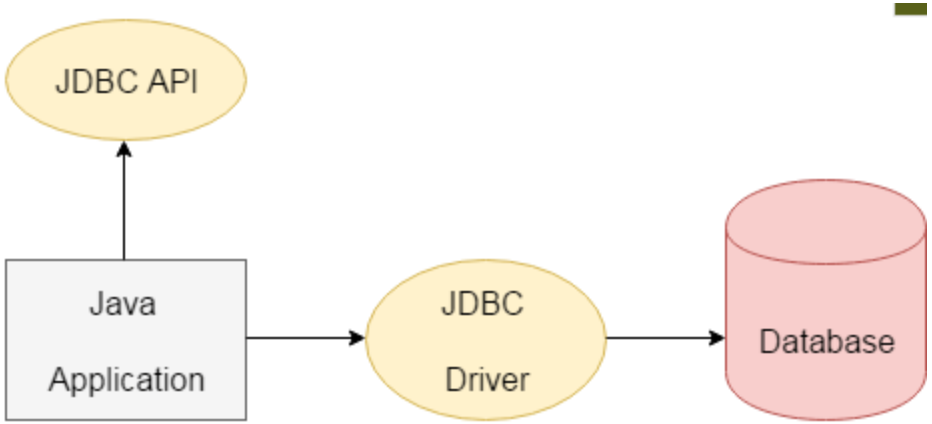
There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver,
- Native Driver,
- Network Protocol Driver, and
- Thin Driver

- We can use JDBC API to access tabular data stored in any relational database.
- With the help of JDBC API, we can save, update, delete and fetch data from the database.
- It is like Open Database Connectivity (ODBC) provided by Microsoft.

Java Database Connectivity (JDBC)

The **java.sql** package contains classes and interfaces for JDBC API.



Popular interfaces

1. Driver interface
2. Connection interface
3. Statement interface
4. PreparedStatement interface
5. CallableStatement interface
6. ResultSet interface
7. ResultSetMetaData interface
8. DatabaseMetaData interface
9. RowSet interface

What is API

API (Application programming interface) is a document that contains a description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc.

Java Database Connectivity (JDBC)

Why Should We Use JDBC

- Before JDBC, ODBC API was the database API to connect and execute the query with the database.
- But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured).
- That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

We can use JDBC API to handle database using Java program and can perform the following activities:

1.Connect to the database
2.Execute queries and update statements to the database
3.Retrieve the result received from the database.

Java Database Connectivity (JDBC)

Required Steps:

- There are following steps required to create a new Database using JDBC application:
- **Import the packages .** Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.
- **Register the JDBC driver .** Requires that you initialize a driver so you can open a communications channel with the database.
- **Open a connection .** Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with database server.
- To create a new database, you need not to give any database name while preparing database URL as mentioned in the below example.
- **Execute a query .** Requires using an object of type Statement for building and submitting an SQL statement to the database.
- **Clean up the environment .** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

Java Database Connectivity (JDBC)

JDBC Driver is a software component that enables java application to interact with the database.

There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

1) JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

- Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

Java Database Connectivity (JDBC)

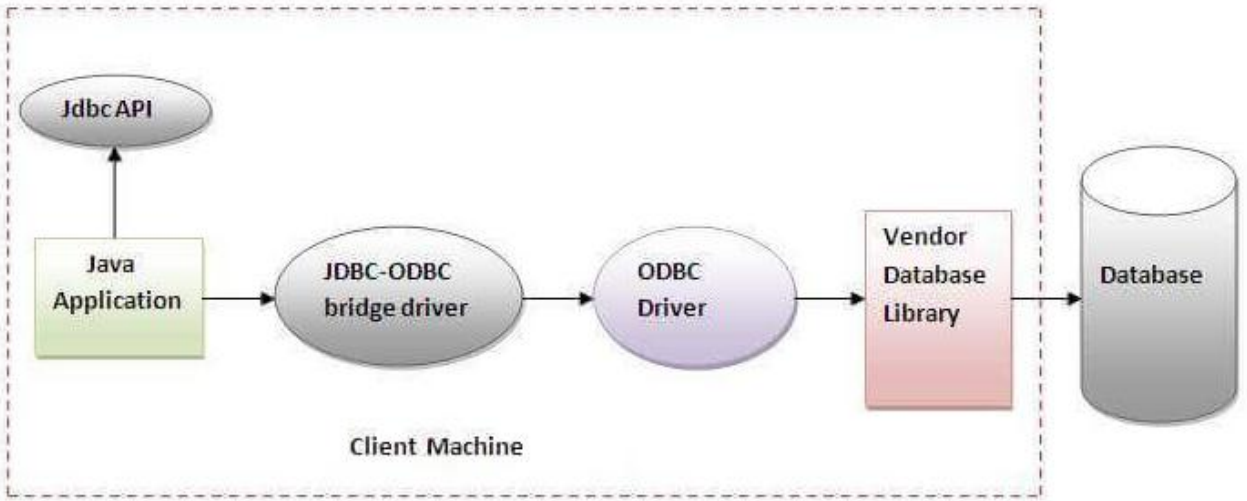


Figure- JDBC-ODBC Bridge Driver

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

2. Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

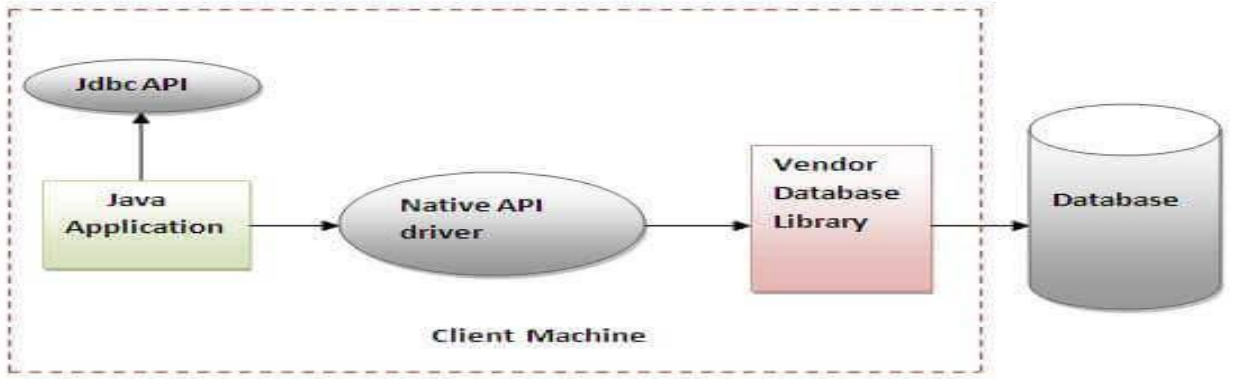


Figure- Native API Driver

Advantages:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantages:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

3.Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

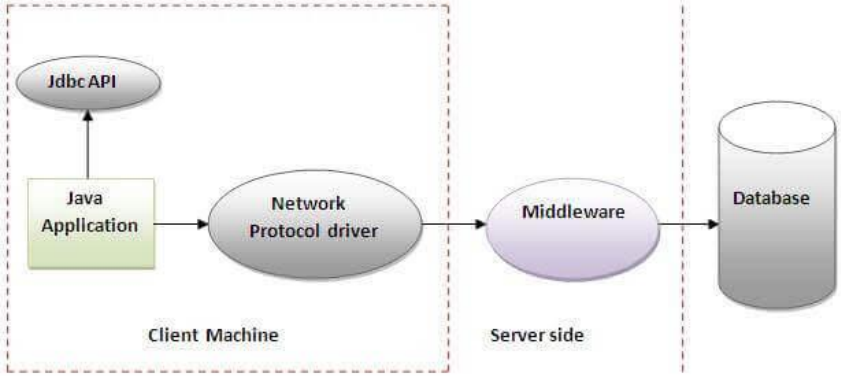


Figure- Network Protocol Driver

Advantages:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

4.Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

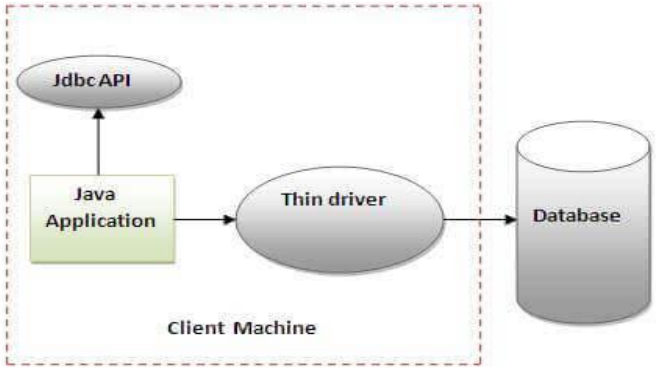


Figure- Thin Driver

Advantages:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantages:

- Drivers depend on the Database.

Java Database Connectivity (JDBC)

Which Driver should be used?

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver Type is 4.
- If your Java application is accessing multiple types of databases at the same time, Type 3 is the preferred driver.
- Type 2 drivers are useful in situations where a type 3 or Type 4 driver is not available yet for your database.
- The Type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

Java Database Connectivity (JDBC)

There are 5 steps to connect any java application with the database using JDBC.

These steps are as follows:

1. Register the Driver class
2. Create connection
3. Create statement
4. Execute queries
5. Close connection

Java Database Connectivity (JDBC)

1) Register the driver class

- The **forName()** method of Class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax of forName() method

- **public static void** forName(String className)**throws** ClassNotFoundException

Example to register the OracleDriver class

- Here, Java program is loading oracle driver to establish database connection.
- Class.forName("oracle.jdbc.driver.OracleDriver");

Java Database Connectivity (JDBC)

2) Create the connection object

- The **getConnection()** method of DriverManager class is used to establish connection with the database.

Syntax of getConnection() method

1) **public static** Connection getConnection(String url)**throws** SQLException

2) **public static** Connection getConnection(String url,String name,String password) **throws** SQLException

Example to establish connection with the Oracle database

- Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","password");

3) Create the Statement object

- The createStatement() method of Connection interface is used to create statement. The object of
- statement is responsible to execute queries with the database.

Syntax of createStatement() method

- **public** Statement createStatement()**throws** SQLException

Example to create the statement object

- Statement stmt=con.createStatement();

Java Database Connectivity (JDBC)

4) Execute the query

- The executeQuery() method of Statement interface is used to execute queries to the database.
- This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax of executeQuery() method

- **public** ResultSet executeQuery(String sql)**throws** SQLException

Example to execute query

- ResultSet rs=stmt.executeQuery("select * from emp");
- **while**(rs.next()){
- System.out.println(rs.getInt(1)+" "+rs.getString(2));
- }

Java Database Connectivity (JDBC)

5) Close the connection object

- By closing connection object statement and ResultSet will be closed automatically.
- The close() method of Connection interface is used to close the connection.

Syntax of close() method

- **public void** close()**throws** SQLException

Example to close connection

- con.close();

In general, to process any SQL statement with JDBC, you follow these steps:

- Establishing a connection.
- Create a statement.
- Execute the query.
- Process the result set object
- Close the connection.