MARUDHAR KESARI JAIN COLLEGE FOR WOMEN VANIYAMBADI

PG AND RESEARCH DEPARTMENT OF COMPUTER SCIENCE

CLASS: II BSC COMPUTERSCIENCE

SUBJECT CODE : FCS 31

SUBJECT NAME: PROGRAMMING IN JAVA

SYLLABUS

UNIT-I

Declaration and Access control:Identifiers and Keywords:Oracle's Java Code convention
Define Classes:Import Statements and the java API-Static Import statements .Use interfaces Declaring an interface Constants.Declare Class members:Access modifiers-Non access modifires-Constructor Declaration-Variable Declarations.Declare and Use Enums:Declaring Enums.Object orientation:Encapsulation-Inheritance and Polymorphism-Polymorphism over riding/Over Loading:Over ridden Methods-Overloaded methods

UNIT - I

INTRODUCTION TO JAVA PROGRAMMING

1.1 INTRODUCTION TO JAVA PROGRAMMING

JAVA was developed by James Gosling at Sun **Microsystems** in the year **1991**, later acquired by Oracle Corporation. Java allows writing, compiling, and debugging programming easy to the programmer. It helps to create reusable code and modular programs.

Java is a class-based, object-oriented programming language. A general-purpose programming language made for developers to **write once run anywhere** that is compiled Java code can run on all platforms(operating systems) that support Java. Java applications are compiled to byte code that can run on any Java Virtual Machine.

Java Terminology

Java Virtual Machine(JVM): This is generally referred to as JVM. There are three execution phases of a program. They are written, compile and run the program.

- ✯ Writing a program is done by a java programmer.
- ✯ The compilation is done by the **JAVAC** compiler which is a primary Java compiler included in the Java development kit (JDK). It takes Java program as input and generates bytecode **(.class file)** as output.
- ✯ In the Running phase of a program, **JVM** executes the bytecode generated by the compiler.

The function of Java Virtual Machine is to execute the bytecode produced by the **compiler**. Every Operating System has a different JVM but the output they produce after the execution of bytecode is the same across all the operating systems. This is why Java is known as a **platform-independent language.**

Bytecode: Javac compiler of JDK compiles the java source code into bytecode so that it can be executed by JVM. It is saved as **.class** file by the compiler.

Java Development Kit(JDK): It is a complete Java development kit that includes everything including compiler, **Java Runtime Environment (JRE)**, java **debuggers**, java docs, etc. For the program to execute in java, we need to install JDK on our computer in order to create, compile and run the java program.

Java Runtime Environment (JRE): JDK includes JRE. JRE installation on computers allows the java program to run, but cannot compile it. JRE includes a **browser, JVM, applet supports, and plugins.**

Garbage Collector: In Java, programmers cannot delete the objects. To delete or recollect that memory JVM has a program called Garbage Collector. Garbage Collectors can recollect the objects that are not referenced. So, Java itself handling memory management. Garbage cannot recover the memory of objects being referenced.

ClassPath: The classpath is the file path where the java runtime and Java compiler look for **.class** files to load. By default, JDK provides many libraries. If you want to include external libraries they should be added to the classpath.

Primary/Main Features of Java

Platform Independent: Compiler converts source code to bytecode and then the JVM executes the bytecode generated by the compiler. This bytecode can run on any platform be it Windows,

Linux, macOS which means if we compile a program on Windows, then we can run it on Linux and vice versa. Each operating system has a different JVM, but the output produced by all the OS is the same after the execution of bytecode. That is why we call java a platform-independent language.

Object-Oriented Programming Language: Organizing the program in the terms of collection of objects is a way of object-oriented programming, each of which represents an instance of the class.

The four main concepts of Object-Oriented programming are:

- ✯ Abstraction
- ✯ Encapsulation
- ✯ Inheritance
- ✯ Polymorphism

Simple: Java is one of the simple languages as it does not have complex features like pointers, operator overloading, multiple inheritances, explicit memory allocation.

Robust: Java language is robust that means reliable. It is developed in such a way that it puts a lot of effort into checking errors as early as possible, that is why the java compiler is able to detect even, those errors that are not easy to detect by another programming language. The main features of java that make it robust are garbage collection, Exception Handling, and memory allocation.

Secure: In java, we don't have pointers, and so we cannot access out-of-bound arrays i.e it shows ***ArrayIndexOutOfBoundsException*** if we try to do so. That is why several security flaws like stack corruption or buffer overflow is impossible to exploit in Java.

Distributed: To create distributed applications using the java programming language. Remote Method Invocation and Enterprise

Java Beans are used for creating distributed applications in java. The java programs can be easily distributed on one or more systems that are connected to each other through an internet connection.

Multithreading: Java supports multithreading. It is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.

Portable: Java code written on one machine can be run on another machine. The platform-independent feature of java in which its platform-independent bytecode can be taken to any platform for execution makes java portable.

Basic Java Program Example

Sample.java

```
import java.io.*;
class Sample
{
    public static void main(String[] args)
    {
      System.out.println("Hello World");
    }
}
```

Compile

Javac Sample.java

Run

Java Sample

Output

Hello World

Explanation:

Comments: Comments are used for explaining code and are used in a similar manner in Java or C or C++. Compilers ignore the comment entries and do not execute them. Comments can be of a single line or multiple lines.

Single line Comments:

Syntax

// Single line comment

Multi-line comments:

Syntax

/* Multi line comments*/

import java.io.*: This means all the classes of *io* package can be imported. *java.io* package provides a set of input and output streams for reading and writing data to files or other input or output sources.

Class: The class contains the data and methods to be used in the program. Methods define the behavior of the class. Class **sample** has only one method Main in JAVA.

static void main(): static keyword tells us that this method is accessible without instantiating the class.

void: keywords tell that this method will not return anything. The main() method is the entry point of our application.

System.in: This is the **standard input stream** that is used to read characters from the keyboard or any other standard input device.

System.out: This is the **standard output stream** that is used to produce the result of a program on an output device like the computer screen.

println(): This method in Java is also used to display text on the console. It prints the text on the console and the cursor moves to the start of the next line at the console. The next printing takes place from the next line.

1.2 DECLARATIONS AND ACCESS CONTROL

Java Refresher

Class - A template that describes the kinds of *state* and *behavior* that objects of its type support.

Object - At runtime, when the Java Virtual Machine (JVM) encounters the *new* keyword, it will use the appropriate class to make an object that is an instance of that class.

State (instance variables) - Each object (instance of a class) will have its own unique set of instance variables as defined in the class

Behavior (methods) - A programmer creates a class also creates methods for that class. Methods are where the class logic is stored and where the real work gets done.

Declarations And Access Control

Source File Declaration

A Java file can have only one public class. If one source file contains a public class, the java filename should be the public class name. Also, a Java source file can only have one package statement and unlimited import statements.

Identifiers Declaration

Identifiers in Java can begin with a letter, an underscore, or a currency character. Other types of naming are not allowed. They can be of any length. Only in the case of JavaBeans methods, they must be named using camelCase, and counting on the methods purpose,

must start with set, get, is, add, or remove. In Java, we have variables, methods, classes, packages, and interfaces as identifiers.

Local Variables Declaration

The scope of local variables will be only within the given method or class. These variables should be initialized during declaration. Access modifiers cannot be applied to local variables.

Example

```
public static void main(String[] args)
{
    String helloMessage;
                //local variable helloMessage
    helloMessage = "Hello, World!";
    System.out.println(helloMessage);
}
```

Instance Variables Declaration

Instance variables are values that can be defined inside the class but outside the methods and begin to live when the class is defined.

Here,unlike local variables, we don't need to assign initial values..

- ✯ It can be defined as public, private and protected.
- ✯ It can be defined as a final.
- ✯ It cannot be defined as abstract and static.

Example

```
class Page
{
  public String pageName;  // instance variable
                              with public access
  private int pageNumber;  // instance variable
                              with private access
}
```

Classes and interfaces Declarations

The first letter should be capitalized, and if several words are linked together to form the name, the first letter of the inner words should be uppercase (a format that's sometimes called "CamelCase"). For classes, the names should typically be nouns.

Example for Classes

Dog

Account

PrintWriter

Example for interfaces

Runnable

Serializable

Methods Declaration

The first letter should be lowercase, and then normal CamelCase rules should be used. In addition, the names should typically be verb-noun pairs.

Examples

getBalance

doCalculation

setCustomerName

Constants Declaration

Java constants are created by marking variables static and final. They should be named using uppercase letters with underscore characters as separators.

Example

MIN_HEIGHT

MAX_HEIGHT

Constructors Declaration

Constructor is used to create new objects. Constructors can take arguments, including methods and variable arguments. They must have the same name as the name of the class in which it is defined. They can be defined as public, protected, private. Static cannot be defined because they have a responsibility to create objects. Since they cannot be overridden, they cannot be defined as final and abstract. When the constructor is overloaded, the compiler does not define the default constructor, so we have to define it. The constructor creation order is from bottom to top in the inheritance tree.

Static Declaration

It allows invoking the variable and method that it defines without the need for any object. Abstract and static cannot be defined together, because the method presented as static can be called without creating objects and by giving parameters directly. The abstract is called to override a method.

ENUM Declaration

It is a structure that allows a variable to be constrained to be predefined by one value. With Enum's getValues method. This is the most elegant way to define and use constants in our Java program.

Class Modifiers (non-access) Declaration

- ✯ Classes can be defined as final, abstract or strictfp.
- ✯ Classes cannot be defined as both final and abstract.
- ✯ Sub classes of the final classes cannot be created.

- ★ Instances of abstract classes are not created.
- ★ Even if there is one abstract method in a class, this class should also be defined as abstract.
- ★ The abstract class can contain both the non-abstract method and abstract method, or it may not contain any abstract method.
- ★ All abstract methods should be overridden by the first concrete (non-abstract) class that extends the abstract class.

Class Access Modifiers Declaration

Access modifiers stay an important part of a declaration that can be accessed outside the class or package in which it is made. Access modifiers enable you to decide whether a declaration is limited to a particular class, a class including its subclasses, a package, or if it is freely accessible. Java language has three access modifiers: public, protected and private.

- ★ **public:** Enables a class or interface to be located outside of its package. It also permits a variable, method, or constructor to be located anywhere its class may be accessed.
- ★ **protected:** Enables a variable, method, or constructor to be accessed by classes or interfaces of the same package or by subclasses of the class in which it is declared.
- ★ **private:** Prevents a variable, method, or constructor from being accessed outside of the class in which it is declared.

1.2.1 Identifiers

Identifiers in Java are symbolic names used for identification. They can be a class name, variable name, method name, package name, constant name. There are some reserved words that cannot be used as an identifier. For every identifier there are some conventions that should be used before declaring them.

Legal identifiers

- ★ Legal identifiers The compiler uses to determine whether a name is legal.
- ★ Legal identifiers must be composed of only Unicode characters, numbers, currency symbols, and connecting characters (such as underscores).
- ★ Identifiers must start with a letter, a currency character ($), or a connecting character such as the underscore (_). Identifiers cannot start with a digit.
- ★ After the first character, identifiers can contain any combination of letters, currency characters, connecting characters, or numbers.
- ★ There is no limit to the number of characters an identifier can contain.
- ★ Can't use a Java keyword as an identifier.
- ★ Identifiers in Java are case-sensitive; foo and FOO are two different identifiers.

Examples

```
int _a;

int $c;

int _2_w;

int _$;

int this_is_a_very_detailed_name_for_an_identifier;
```

Illegal Identifiers

- ★ We cannot include a space in an identifier

- ★ The identifier should not begin with numbers
- ★ The plus (+) symbol cannot be used
- ★ Hyphen symbol is not allowed
- ★ Ampersand symbol is not allowed
- ★ We cannot use an apostrophe symbol in an identifier

Examples

int java books

int java+

int e#

int .f

int 7g

int –a

1.2.2 Keywords

- ★ Java has a set of built-in keywords. These keywords must not be used as identifiers. Java reserved keywords are predefined words, which are reserved for any functionality or meaning.
- ★ We cannot use these keywords as our identifier names, such as class name or method name. These keywords are used by the syntax of Java for some functionality.
- ★ If we use a reserved word as our variable name, it will throw an error.
- ★ In Java, every reserved word has a unique meaning and functionality.

Complete List of Java Keywords

abstract	else	long	super
boolean	extends	native	switch
break	final	new	synchronized
byte	finally	package	this
case	float	private	throw
catch	for	protected	throws
char	goto	public	transient
class	if	return	try
const	implements	short	void
continue	import	static	volatile
default	instanceof	istrictfp	while
do	int		assert
double	interface		enum

1.2.3 Oracle's Java Code Conventions

Oracle's recommendations for naming classes, variables, and methods. Oracle estimates that over the lifetime of a standard piece of code, 20 percent of the effort will go into the original creation and testing of the code, and 80 percent of the effort will go into the subsequent maintenance and enhancement of the code.

The only standards that are followed as much as possible in the real exam are the naming standards. Here are the naming standards that Oracle recommends.

1.3 DEFINE CLASSES

A class is a user defined prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. ***Modifiers:*** A class can be public or has default access.
2. ***class keyword:*** class keyword is used to create a class.
3. ***Class name:*** The name should begin with an initial letter.
4. ***Superclass:*** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
5. ***Interfaces:*** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
6. ***Body:*** The class body surrounded by braces { }.

Constructors are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

Before class declarations, the rules associated with declaring classes, import statements, and package statements in a source file.

Syntax

```
public class <class name>
{
//Instance Variable Declarations
//Method Declarations
}
```

Source File Declaration Rules

- There can be only one public class per source code file.
- Comments can appear at the beginning or end of any line in the source code file.
- If there is a public class in a file, the name of the file must match the name of the public class. For example, a class declared as public class Dog { } must be in a source code file named Dog. java.
- If the class is part of a package, the package statement must be the first line in the source code file, before any import statements that may be present.
- If there are import statements, they must go between the package statement (if there is one) and the class declaration. If there isn't a package statement, then the import statement(s) must be the first line(s) in the source code file.
- If there are no package or import statements, the class declaration must be the first line in the source code file.
- Import and package statements apply to all classes within a source code file.
- In other words, there's no way to declare multiple classes in a file and have them in different packages or use different imports.
- A file can have more than one nonpublic class.
- Files with no public classes can have a name that does not match any of the classes in the file.

1.16 **Compiling with javac and interpret with java command**

The javac command is used to invoke Java compiler. The java command is used to invoke the Java Virtual Machine (JVM).

```
public class MyClass
{
    public static void main(String[] args)
    {
    System.out.println(args[0] + " " + args[1]);
    }
}
javac MyClass.java
java MyClass
```

Using public static void main(String[ ] args) main() is the method that the JVM uses to start execution of a Java program

The following are all legal declarations for the main():

```
static public void main(String[] args)
public static void main(String... x)
static public void main(String bang_a_gong[])
```

1.3.1 Import Statements and the Java API

Import is a keyword in java language used to import the predefined properties of java API into current working java program.

Syntax

import packagename;

Java API is a collection of package, package is a container which is collection of predefined classes and interfaces

The Java API has thousands of classes and the Java community has written the rest.

Example :

```
public class ArrayList
{
    public static void main(String[] args)
    {
      System.out.println("fake ArrayList class");
    }
}
```

This is a perfectly legal class, but as it turns out, one of the most commonly used classes in the Java API is also named ArrayList, can also use the ArrayList class from the API. The API version's actual name is java.util.ArrayList(below example). That's its fully qualified name.

```
public class MyClass
{
    public static void main(String[] args)
    {
      java.util.ArrayList<String> a = new java.util.
      ArrayList<String>();
    }
}
```

In this program, interpret the import statement as saying Java API there is a package called 'util', and in that package is a class called 'ArrayList'.

```
import java.util.ArrayList;
public class MyClass
{
    public static void main(String[] args)
    {
```

```
    ArrayList<String> a = new ArrayList<String>();
    }
}
```

To use a few different classes from the java.util package: ArrayList and TreeSet, can add a wildcard character (*) to your import statement

```
import java.util.*;
public class MyClass
{
    public static void main(String[] args)
    {
      ArrayList<String> a = new ArrayList<String>();
      TreeSet<String> t = new TreeSet<String>();
    }
}
```

1.3.2 Static Import Statements

Classes will contain static members. Static class members can exist in the classes in the Java API. In most of time many classes are having Static members in it. Static members are extensively used in java API. In java 5, static import feature allow programmer to use directly any static members of the class and no need to invoke with its class name.

Syntax

import static package.class name.static member name;

Should be "import static", not "static import".

Before static imports

```
public class TestStatic
{
    public static void main(String[] args)
    {
      System.out.println(Integer.MAX_VALUE);
```

```
        System.out.println(Integer.toHexString(42));
    }
}
```

After static imports

```
import static java.lang.System.out;
import static java.lang.Integer.*;
public class TestStaticImport
{
    public static void main(String[] args)
    {
        out.println(MAX_VALUE);
        out.println(toHexString(42));
    }
}
```

Output

2147483647

2a

The syntax MUST be import static followed by the fully qualified name of the static member we want to import.

Rules for using static imports:

- ★ Only can do a static import on static object references, constants (they are static and final), and static methods.
- ★ Must say import static; you can't say static import.
- ★ Can do ambiguously named static members. For instance, if you do a static import for both the Integer class and the Long class, referring to MAX_VALUE will cause a compiler error, since both Integer and Long have a MAX_VALUE constant, and Java won't know which MAX_VALUE you're referring to.

Advantage of static import

Greater keystroke-reduction capabilities and less coding (save typing).

Disadvantage of Static import

Many people argue this makes code less readable. If two class have the same for its static members then we will get ambiguity.

1.4 USE INTERFACES

1.4.1 Declaring an Interface

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

Syntax

interface <interface_name>

{

// declare constant fields

// declare methods that abstract

}

Example

```
interface Player
{
    final int id = 10;
    int move();
}
```

Interfaces can be implemented by any class, from any inheritance tree. For example, both a Ball and a Tire to have bounce behavior, but Ball and Tire do not share any inheritance relationship. Ball extends

Toy while Tire extends only java.lang.Object. But by making both Ball and Tire implement Bounceable. Like an abstract class, an interface defines abstract methods that take the following form:

abstract void bounce();

An abstract class can define both abstract and nonabstract methods, an interface can have only abstract methods.

The relationship between interfaces and classes

What you declare :

```
interface Bounceable
void bounce( );

void setBounceFactor(int bf);
```

What the compiler sees:

```
interface Bounceable
public abstract void bounce( );

public abstract void setBounceFactor(int bf);
```

What the implementing class must do. All interface methods must be implemented, and must be marked public.

```
Class Tire implements Bounceable
public void bounce( )
{
// statements
}
public void setBounceFactor(int bf)
{
// statements
}
```

- All interface methods are implicitly public and abstract. In other words, you do not need to actually type the public or abstract modifiers in the method declaration, but the method is still always public and abstract.
- All variables defined in an interface must be public, static, and final in other words, interfaces can declare only constants, not instance variables.
- Interface methods must not be static.
- Because interface methods are abstract, they cannot be marked final, strictfp, or native.
- An interface can extend one or more other interfaces.
- An interface cannot extend anything but another interface.
- An interface cannot implement another interface or class.
- An interface must be declared with the keyword interface.
- Interface types can be used polymorphically.

The following is a legal interface declaration:

```
public abstract interface Rollable { }
public interface Rollable { }
```

- Typing in the abstract modifier is considered redundant, interfaces are implicitly abstract whether you type abstract or not.
- The public modifier is required if you want the interface to have public rather than default access.

For example, the following five method declarations, if declared within their own interfaces, are legal and identical.

```
void bounce();
```

public void bounce();

abstract void bounce();

public abstract void bounce();

abstract public void bounce();

The following interface method declarations won't compile:

final void bounce(); // final and abstract can never be used together, and abstract is implied

static void bounce(); // interfaces define instance methods

private void bounce(); // interface methods are always public

protected void bounce();

Example

```
import java.io.*;
interface In1
{
    final int a = 10;     // public, static and final
    void display();       // public and abstract
}
// A class that implements the interface.
class TestClass imple   ts In1
{
    public void display() // Implementing the capa-
      bilities of interface.
    {
      System.out.println("Java Programming");
    }
    public static void main (String[] args)
    {
      TestClass t = new TestClass();
      t.display();
```

```
        System.out.println(a);
    }
}
```

Output

Java Programming

10

1.4.2 Declaring Interface Constants

★ By placing, the constants in the interface, any class that implements the interface has direct access to the constants, just as if the class had inherited them.

★ Interface constants must always be public static final. There are no different from any other publicly accessible constants, so they obviously must be declared public,static, and final.

★ Interface constants are defined in an interface, they do not have to be declared as public, static, or final. They must be public, static, and final, but you don't actually have to declare them that way.

Example

```
interface Foo
{
    int BAR = 42;  // constant variables
    void go();     // Abstract method
}
class Zap implements Foo
{
    public void go()
    {
        BAR = 27;
    }
```

}

Define constants, but without explicitly using the required modifiers. For example, the following are all identical:

public int x = 1; // Looks non-static and non-final,

int x = 1; // Looks default, non-final, non-static,

static int x = 1; // Doesn't show final or public

final int x = 1; // Doesn't show static or public

public static int x = 1; // Doesn't show final

public final int x = 1; // Doesn't show static

static final int x = 1 // Doesn't show public

public static final int x = 1; // what you get implicitly

1.5 DECLARE CLASS MEMBERS

Methods and instance (nonlocal) variables are collectively known as members, can modify a member with both access and non-access modifiers.

Class Declaration

```
class MyClass
{
    //Intance Variables;
    //Methods;
}
```

Class Access

Code from one class (class A) has access to another class (class B), it means class A can do

★ Create an instance of class B.

- ★ Extend class B (in other words, become a subclass of class B).
- ★ Access certain methods and variables within class B, depending on the access control of those methods and variables.

Here, access means visibility.

Class Modifiers

There are two modifiers used when declaring classes, In the access modifiers, a class can use just two of the four access control levels (default or public), members can use all four.

- ★ Access modifiers (public, protected, private, default)
- ★ Non-access modifiers (including strictfp, final, and abstract)

You need to understand two different access issues:

- ★ Whether method code in one class can access a member of another class
- ★ Whether a subclass can inherit a member of its superclass

Access occurs when a method in one class tries to access a method or a variable of another class, using the dot operator (.) to invoke a method or retrieve a variable.

Example

```
class Zoo
{
    public String coolMethod()
    {
      return "Wow baby";
    }
}
class Moo
{
```

```
    public void useAZoo()
    {
      Zoo z = new Zoo();
      System.out.println("A Zoo says, " + z.cool-
      Method());
    }
}
```

1.5.1 Access modifiers (public, protected, private, default)

Default Access

A class with default access has no modifier preceding it in the declaration. If default access as package-level access, because a class with default access can be seen only by classes within the same package.

For example, if class A and class B are in different packages, and class A has default access, class B won't be able to create an instance of class A or even declare a variable or return type of class A. In fact, class B has to pretend that class A doesn't even exist, or the compiler will complain.

Example

First source file :

```
package cert;
class Beverage
{
  // statements
}
```

Second source file:

```
package exam.stuff;
import cert.Beverage;
```

```
class Tea extends Beverage
{
  //statements
}
```

In the above example, superclass (Beverage) is in a different package from the subclass (Tea). The import statement at the top of the Tea file is trying to import the Beverage class. The Beverage file compiles fine, but when try to compile the Tea file, we get something like this:

Can't access class cert.Beverage. Class or interface must be public, in same package, or an accessible member class.

Tea won't compile because its superclass, Beverage, has default access and is in a different package. In this case, put both classes in the same package, or declare Beverage as public.

Public Access

A class declaration with the public keyword gives all classes from all packages access to the public class. In other words, all classes in the Java Universe (JU) have access to a public class. To add the keyword public in front of the superclass (Beverage) declaration, as follows:

Example

```
package cert;
public class Beverage
{
  // statements
}
```

This changes the Beverage class so it will be visible to all classes in all packages. The class can now be instantiated from all other classes, and any class is now free to subclass (extend from) it unless, that is, the class is also marked with the nonaccess modifier final.

```
package book;
import cert.*; // Import all classes in the cert
    package
class Goo
{
    public static void main(String[] args)
    {
      Sludge o = new Sludge();
      o.testIt();
    }
}
```

Now look at the second file:

```
package cert;
public class Sludge
{
    public void testIt()
    {
      System.out.println("sludge");
    }
}
```

Goo and Sludge are in different packages. However, Goo can invoke the method in Sludge without problems, because both the Sludge class and its testIt() method are marked public.

Private Members

Members marked private can't be accessed by code in any class other than the class in which the private member was declared.

Example :

```
package cert;
public class Roo
{
```

```
    private String doRooThings()
    {
      return "fun";
    }
}
```

The doRooThings() method is private, so no other class can use it. If try to invoke the method from any other class,output will be

package notcert;

```
import cert.Roo;
class UseARoo
{
    public void testIt()
    {
      Roo r = new Roo();
      System.out.println(r.doRooThings());
    }
}
```

If try to compile UseARoo, get a compiler error something like this:

cannot find symbol

symbol : method doRooThings()

1.5.2 Non-access Class Modifiers(strictfp, final and abstract)

- ★ To modify a class declaration using the keyword final, abstract, or strictfp.
- ★ These modifiers are in addition to whatever access control is on the class, for example, declare a class as both public and final.
- ★ Can't always mix with nonaccess modifiers like mark a class as both final and abstract.
- ★ Use strictfp in combination with final.

Final Classes

In a class declaration, the final keyword means the class cannot be subclassed. Many classes in the Java core libraries are final. For example, the String class cannot be subclassed.

Example

```
package cert;
public final class Beverage
{
   public void importantMethod() { }
}
   package exam.stuff;
   import cert.Beverage;
   class Tea extends Beverage
   {
     //statements
   }
```

Output

Can't subclass final classes: class

cert.Beverage class Tea extends Beverage{

1 error

Final Methods

The final keyword prevents a method from being overridden in a subclass, and is often used to enforce the API functionality of a method.

For example, the Thread class has a method called isAlive() that checks whether a thread is still active. If extend the Thread class, there is really no way that you can correctly implement this method so,this method made it final.

Example

```
class SuperClass
{
public final void showSample()
{
    System.out.println("One thing.");
}
}
```

It's legal to extend SuperClass, since the class is not marked final, but we cannot override the final method showSample().

Example

```
class SubClass extends SuperClass
{
    public void showSample()
    {
    System.out.println("Another thing.");
    }
}
```

Output

javac FinalTest.java

FinalTest.java:5: The method void showSample() declared in class SubClass cannot override the final method of the same signature declared in class SuperClass. Final methods cannot be overridden.

public void showSample() { }

1 error

Final Arguments

Method arguments are the variable declarations that appear in between the parentheses in a method declaration. A typical method declaration with multiple arguments looks like this:

public Record getRecord(int fileNumber, int recNumber) {}

Method arguments are essentially the same as local variables. In the preceding example, the variables fileNumber and recNumber will both follow all the rules applied to local variables. This means they can also have the modifier final:

public Record getRecord(int fileNumber, final int recNumber) {}

In this example, the variable recNumber is declared as final, which of course means it can't be modified within the method. In this case, "modified" means reassigning a new value to the variable. In other words, a final argument must keep the same value that the parameter had when it was passed into the method.

Abstract Classes

An abstract class can never be instantiated.

Example

```
abstract class Car
{
    private double price;
    private String model;
    private String year;
    public abstract void goFast();
    public abstract void goUpHill();
    public abstract void impressNeighbors();
}
```

The preceding code will compile fine. However, if try to instantiate a Car in another body of code,will get a compiler error something like this:

AnotherClass.java:7: class Car is an abstract class. It can't be instantiated.

Car x = new Car();

1 error

Note: that the methods marked abstract end in a semicolon rather than curly braces.

Abstract Method

An abstract method is a method that's been declared (as abstract) but not implemented. In other words, the method contains no functional code.

It is illegal to have even a single abstract method in a class that is not explicitly declared abstract, following illegal class:

```
public class IllegalClass
{
    public abstract void doIt();
}
```

The preceding class will produce the following error if you try to compile it:

IllegalClass.java:1: class IllegalClass must be declared abstract.

It does not define void doIt() from class IllegalClass.

public class IllegalClass{

1 error

If have an abstract class with no abstract methods. The following example will compile fine:

```
public abstract class LegalClass
    {
      void goodMethod()
    {
    // lots of real implementation code here
    }
}
```

In the preceding example, goodMethod() is not abstract. Three different things tell, it's not an abstract method:

- ✯ The method is not marked abstract.
- ✯ The method declaration includes curly braces, as opposed to ending in a semicolon. In other words, the method has a method body.
- ✯ The method might provide actual implementation code inside the curly braces.

Any class that extends an abstract class must implement all abstract methods of the superclass, unless the subclass is also abstract.

Rules

- ✯ The first concrete subclass of an abstract class must implement all abstract methods of the superclass.
- ✯ Concrete just means nonabstract, so if you have an abstract class extending another abstract class, the abstract subclass doesn't need to provide implementations for the inherited abstract methods.

The following example demonstrates an inheritance tree with two abstract classes and one concrete class:

```
public abstract class Vehicle
{
      private String type;
      public abstract void goUpHill();
      public String getType()
      {
       return type;
      }
}
public abstract class Car extends Vehicle
```

```
{
    public abstract void goUpHill(); // Still ab-
      stract
    public void doCarThings()
    {
      //statements
    }
}
public class Mini extends Car {
public void goUpHill() {
// Mini-specific going uphill code
}
```

Determining Access to Class Members

Visibility	Public	Protected	Default	Private
From the same Class	Yes	Yes	Yes	Yes
From any class in same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, through inheritance	No	No
From any non-subclass outside package	Yes	No	No	No

1.5.3 Constructor Declarations

In Java, objects are constructed. Every time you make a new object, at least one constructor is invoked. Every class has a constructor, although if do not create one explicitly, the compiler will build one.

Example

```
class Foo
{
    protected Foo() { } // this is Foo's constructor
    protected void Foo() { } // this is a badly
                             named, but legal, method
}
```

Constructor declarations can however have all of the normal access modifiers, and they can take arguments (including var-args), just like methods constructors is that they must have the same name as the class in which they are declared. Constructors can't be marked static (they are after all associated with object instantiation), and they can't be marked final or abstract (because they can't be overridden).

Legal and illegal constructor declarations:

```
class Foo2
{
// legal constructors
      Foo2() { }
      private Foo2(byte b) { }
      Foo2(int x) { }
      Foo2(int x, int... y) { }
// illegal constructors
      void Foo2() { } // it's a method, not a con-
      structor
      Foo() { } // not a method or a constructor
      Foo2(short s); // looks like an abstract method
      static Foo2(float f) { } // can't be static
```

```
    final Foo2(long x) { } // can't be final
    abstract Foo2(char c) { } // can't be abstract
    Foo2(int... x, int t) { } // bad var-arg syntax
}
```

1.5.4 Variable Declarations

There are two types of variables in Java:

Primitives A primitive can be one of eight types: char, boolean, byte, short, int, long, double, or float. Once a primitive has been declared, its primitive type can never change, although in most cases its value can change.

Reference variables A reference variable is used to refer to (or access) an object. A reference variable is declared to be of a specific type, and that type can never be changed. A reference variable can be used to refer to any object of the declared type or of a subtype of the declared type (a compatible type).

Declaring Primitives and Primitive Ranges

Primitive variables can be declared as class variables (statics), instance variables, method parameters, or local variables primitive variable declarations. Declaring a primitive variable causes the computer to reserve some memory where the value assigned to the variable can be stored. The size of the storage container reserved depends on type of the primitive.

Example

```
byte b;

boolean myBooleanPrimitive;

int x, y, z; // declare three int primitives
```

Java has eight different primitive variables. These are:

- boolean (a truth value: either true or false),
- byte (a byte containing 8 bits, between the values -128 and 127),
- char (a 16-bit value representing a single character),
- short (a 16-bit value that represents a small integer, between the values -32768 and 32767),
- int (a 32-bit value that represents a medium-sized integer, between the values -231 and 231-1),
- long (a 64-bit value that represents a large integer, between values -263 and 263-1),
- float (a floating-point number that uses 32 bits), and double (a floating-point number that uses 64 bits).

Declaring Reference Variables

All of the variables provided by Java (other than the eight primitive variables mentioned above) are reference type. A programmer is also free to create their own variable types by defining new classes. Any object instanced from a class is a reference variable. Reference variables can be declared as static variables, instance variables, method parameters, or local variables. Can declare one or more reference variables, of the same type, in a single line.

```
Object o;

Dog myNewDogReferenceVariable;

String s1, s2, s3;
```

Difference between primitive and reference variables:

- Primitives (usually numbers) are immutable. The internal state of reference variables, on the other hand, can typically be

mutated which means that the value of a primitive variable is stored directly in the variable, whereas the value of a reference variable is a reference to the variable's data, i.e., its internal state.

- ★ Arithmetic operations, such as addition, subtraction, and multiplication can be used with primitive variables; these operations do not change the original values of the variables. Arithmetic operations create new values that can be stored in variables as needed. Conversely, the values of reference variables cannot be changed by these arithmetic expressions.

Instance Variables

Instance variables are defined inside the class, but outside of any method, and are initialized only when the class is instantiated. Instance variables are the fields that belong to each unique object. For example, the following code defines fields (instance variables) for the name, title, and manager for employee objects:

Example

```
class Employee
{
  private String name;
  private String title,
  private String manager;
}
```

Comparison of modifiers on variables Vs. methods

Local Variables	(Non-local) Variables		Methods	
Final	final	public	Final	public
	protected	private	protected	private
	static	transient	static	abstract
	volatile		synchronized	strictfp
			native	

Local (Automatic/Stack/Method) Variables

- ★ A local variable is a variable declared within a method. That means the variable is not just initialized within the method, but also declared within the method.
- ★ The local variable starts its life inside the method, it's also destroyed when the method has completed.
- ★ Local variables are always on the stack, not the heap.
- ★ Local variable declarations can't use most of the modifiers that can be applied to instance variables, such as public, transient, volatile, abstract, or static but local variables can be marked final.

Example

```
class TestServer
{
    public void logIn()
  {
    int count = 10;
  }
    public void doSomething(int i)
```

```
    {
      count = i; //Can't access count outside method
      logIn()
    }
}
```

It is possible to declare a local variable with the same name as an instance variable. It's known as shadowing, as the following code demonstrates:

```
class TestServer
{
    int count = 9; // Declare an instance variable
      named count
    public void logIn()
    {
      int count = 10; // Declare a local variable
      named count
      System.out.println("local variable count is "
      + count);
    }
    public void count()
    {
      System.out.println("instance variable count is
      " + count);
    }
    public static void main(String[] args)
    {
      new TestServer().logIn();
      new TestServer().count();
    }
}
```

Output

local variable count is 10

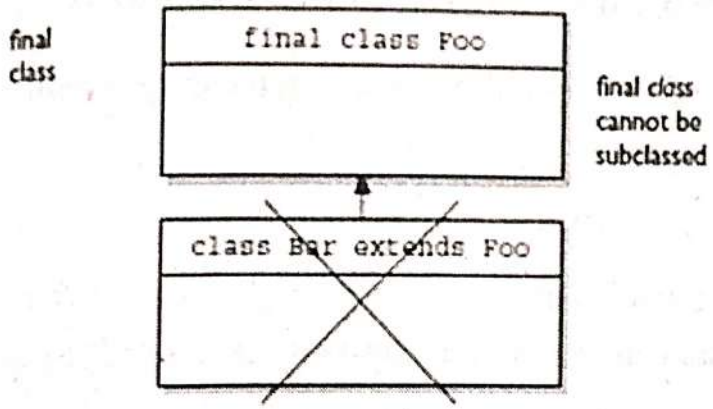
instance variable count is 9

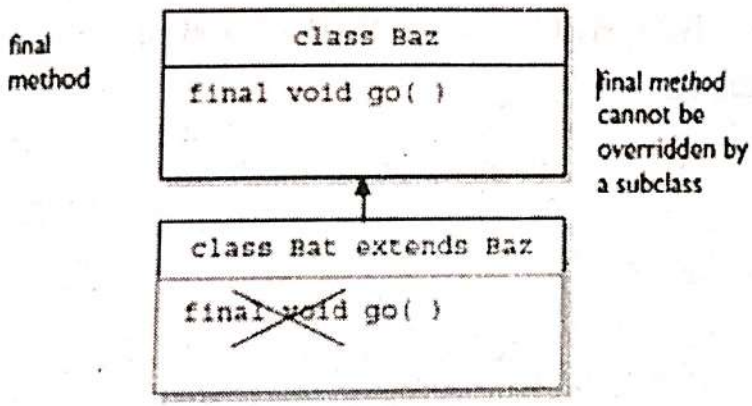
Final Variables

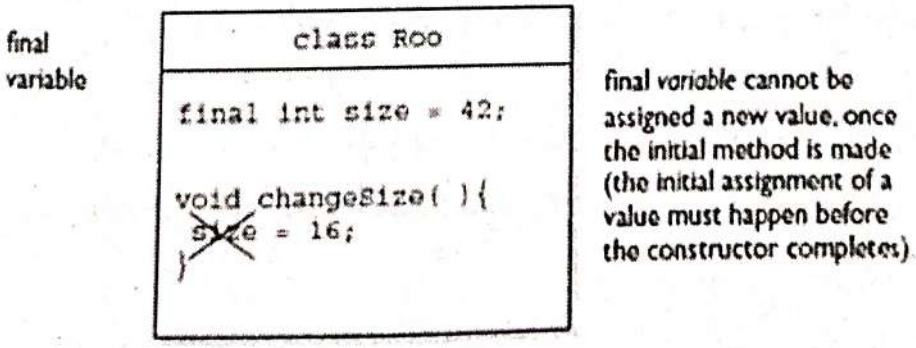
Declaring a variable with the final keyword makes it impossible to reassign that variable once it has been initialized with an explicit value. For primitives, this means that once the variable is assigned a value, the value can't be altered

eg: final int a=10;

Effect of final on variables, methods, and classes







1.6 DECLARE AND USE ENUMS

Declaring enums

Using enums can help reduce the bugs in the code. For instance, in coffee shop application might want to restrict the CoffeeSize selections to BIG, HUGE, and OVERWHELMING. With the following simple declaration, you can guarantee that the compiler will stop you from assigning anything to a CoffeeSize except BIG, HUGE, or OVERWHELMING:

enum CoffeeSize { BIG, HUGE, OVERWHELMING };

only way to get a CoffeeSize will be with a statement something like this:

CoffeeSize cs = CoffeeSize.BIG;

It's not required that enum constants be in all caps, but borrowing from the Oracle code convention that constants are named in caps.

An example declaring an enum outside a class:

enum CoffeeSize { BIG, HUGE, OVERWHELMING } // this cannot be private or protected

```
class Coffee
{
    CoffeeSize size;
}
public class CoffeeTest1
{
    public static void main(String[] args)
    {
      Coffee drink = new Coffee();
      drink.size = CoffeeSize.BIG; // enum outside
      class
    }
}
```

An example of declaring an enum inside a class:

```
class Coffee2
{
    enum CoffeeSize {BIG, HUGE, OVERWHELMING }
    CoffeeSize size;
}
public class CoffeeTest2
{
    public static void main(String[] args)
    {
      Coffee2 drink = new Coffee2();
      drink.size = Coffee2.CoffeeSize.BIG; // en-
      closing class name required
    }
```

1.7 OBJECT ORIENTATION

1.7.1 Encapsulation

- ★ Java are multi-paradigm high-level programming languages that means they support both OOP and procedural programming.
- ★ Encapsulation is defined as wrapping up of data and information under a single unit.
- ★ In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulates them.
- ★ This is a programming style where implementation details are hidden. It reduces software development complexity greatly.
- ★ With Encapsulation, only methods are exposed. The programmer does not have to worry about implementation details but is only concerned with the operations.

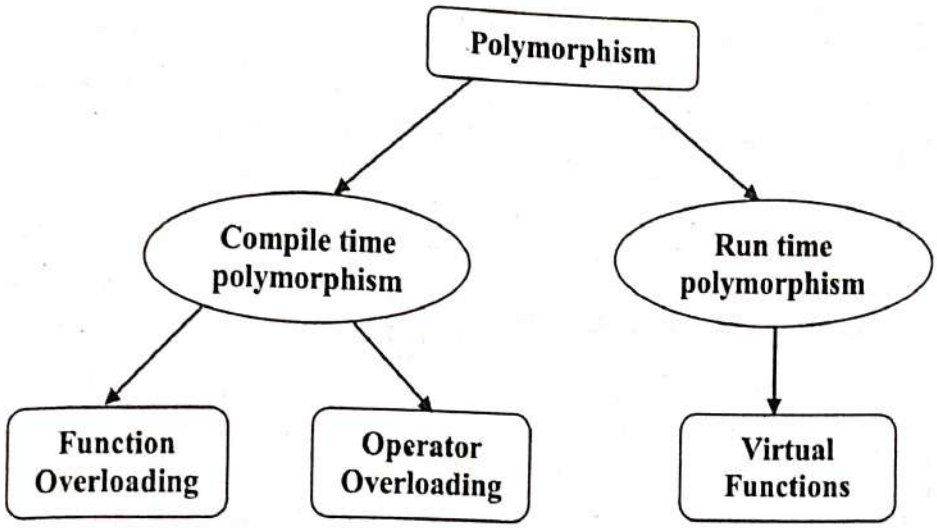
- ★ For example, if a developer wants to use a dynamic link library to display date and time, he does not have to worry about the codes in the date and time class rather he would simply use the data and time class by using public variables to call it up
- ★ In essence encapsulation is achieved in Java by creating Private variables to define hidden classes in and then using public variables to call them up for use.
- ★ With this approach, a class can be updated or maintained without worrying about the methods using them. Once the class is changed, it automatically updates the methods accordingly. Encapsulation also ensures that your data is hidden from external modification.
- ★ Encapsulation is also known as Data-Hidden.
- ★ Encapsulation can be viewed as a shield that protects data from getting accessed by outside code.

1.7.2 Inheritance

- ★ Inheritance (OOP) is when an object or class is based on another object or class, using the same implementation specifying implementation to maintain the same behavior.
- ★ Objects can interact with one another by using the properties of each block or extending the functionalities of a block through inheritance.
- ★ Inheritance ensures that codes are reused. There are millions of Java libraries that a programmer can use through inheritance. The properties of a class can be inherited and extended by other classes or functions.
- ★ There are two types of classes. One is the Parent or base class, and the other is the child class which can inherit the properties of the parent class.

★ A real-world example of inheritance is a mother and child. The child may inherit attributes such as height, Voice patters, color. The mother can reproduce other children with the same attributes as well

1.7.3 Polymorphism



Polymorphism means existing in many forms. Variables, functions, and objects can exist in multiple forms in Java. There are two types of polymorphism which are run time polymorphism and compile-time polymorphism. Run time can take a different form while the application is running and compile-time can take a different form during compilation.

An excellent example of Polymorphism in Object-oriented programing is a cursor behavior. A cursor may take different forms like an arrow, a line, cross, or other shapes depending on the behavior of the user or the program mode.

1.8 OVERRIDING / OVERLOADING

1.8.1 Overloaded Methods

★ Method Overloading is a ***Compile time polymorphism.***

★ In method overloading, more than one method shares the same method name with different signature in the class.

★ In method overloading, return type can or can not be be same, but we must have to change the parameter because in java, cannot achieve the method overloading by changing only the return type of the method.

Example

```
class MethodOverloadingEx
{
   static int add(int a, int b)
    {
      return a+b;
    }
   static int add(int a, int b, int c)
    {
      return a+b+c;
    }
   public static void main(String args[])
    {
       System.out.println(add(4, 6));
       System.out.println(add(4, 6, 7));
     }
}
```

Output

//add with two parameter method runs

10

//add with three parameter method runs

17

1.8.2 Overridden Methods

★ Method Overriding is a ***Run time polymorphism.***

★ In method overriding, derived class provides the specific implementation of the method that is already provided by the base class or parent class.

☆ In method overriding, return type must be same or co-variant (return type may vary in same direction as the derived class).

Example

```
class Animal
{
      void eat()
    {
      System.out.println("eating.");}
    }
class Dog extends Animal
{
    void eat()
    {
      System.out.println("Dog is eating.");}
    }
class MethodOverridingEx
{
    public static void main(String args[])
    {
          Dog d1=new Dog();
          Animal a1=new Animal();
          d1.eat();
          a1.eat();
    }
}
```

Output

// Derived class method eat() runs

Dog is eating

// Base class method eat() runs

eating

A method eat() has overridden in the derived class name **Dog** that is already provided by the base class name Animal. When create

the instance of class Dog and call the eat() method, that only derived class eat() method run instead of base class method eat() and create the instance of class **Animal** and call the eat() method, that only base class eat() method run instead of derived class method eat(). So, that in method overriding, method is bound to the instances on the run time which is decided by the **JVM**. So, it is called **Run time polymorphism**

Difference between Method Overloading and Method Overriding in Java

S.No	Method Overloading	Method Overriding
1.	Method overloading is a compile time polymorphism.	Method overriding is a run time polymorphism.
2.	It help to rise the readability of the program.	While it is used to grant the specific implementation of the method which is already provided by its parent class or super class.
3.	It is occur within the class.	While it is performed in two classes with inheritance relationship.
4.	Method overloading may or may not require inheritance.	While method overriding always needs inheritance.
5.	In this, methods must have same name and different signature.	While in this, methods must have same name and same signature.
6.	In method overloading, return type can or can not be be same, but we must have to change the parameter.	While in this, return type must be same or co-variant.