UNIT – 1
Topic: JAVA Environment
Unit-01/Lecture-01

The Creation of Java
Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called "Oak," but was renamed "Java" in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

Somewhat surprisingly, the original impetus for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls. As you can probably guess, many different types of CPUs are used as controllers. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier—and more cost-efficient—solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.

Features of Java[RGPV/Dec2014 (3)]

There are following features of Java.

Simple
Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

Object-Oriented
Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to

objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance nonobjects.

Robust

The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. Many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using.

Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or "file not found," and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should—be managed by your program.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was “write once; run anywhere, any time, forever.” To a great extent, this goal was accomplished.

Interpreted and High Performance

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports *Remote Method Invocation (RMI).* This feature enables a program to invoke methods across a network.

Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

Java Editions

Sun uses a peculiar naming scheme to differentiate different versions of Java. First, Sun generates an abstract specification that defines what Java is. This is known as the *platform*. Major specification changes would require a change in platform. Then, a particular version of Java might target a different type of application (for example, a desktop computer or a handheld computer). These different types are known as *editions*. Finally, each specific implementation of an edition on the platform has a version number and is known as a *Java SDK* (Software Development Kit; formerly known as the Java Development Kit or JDK).

Sun has three editions of Java for a given platform or version (the current platform is Java 2). The editions for the current platform are:

- *J2ME (Micro Edition)*— Used to create programs that run on small handheld devices, such as phones, PDAs (personal digital assistants), and appliances.

- *J2SE (Standard Edition)*— Used primarily to create programs for desktop computers or for any computer too large for J2ME and too small for J2EE.

- *J2EE (Enterprise Edition)*— Used to create very large programs that run on servers managing heavy traffic and complicated transactions. These programs are the backbone of many online services, such as banking, e-commerce, and B2B (business-to-business) trading systems.

Java SDK
The SDK is a development environment for building programs using the Java programming language; the SDK includes everything you need to develop and test programs. The tools include command-line programs (which were used, incidentally, to develop the samples for this book). Although these tools do not provide a graphical user interface (GUI), using them is a good way to learn the Java language.

The SDK provides many tools, the three most important of which are:

- **The compiler**— The compiler converts the human-readable source file into platform-independent code that a JVM interprets. This code is called *bytecode* .

- **The runtime system**— The SDK includes a JVM that allows you to run Java programs and test your programs. The runtime system also includes a command-line debugger that you can use to monitor your program's execution.

- **The source code—** Sun provides quite a bit of source code for the Java libraries that form part of the JVM. You shouldn't change this code directly. Thanks to object orientation, however, you can modify these classes by making new classes that extend the existing ones. Examining the source code is often helpful in understanding how a class works.

If we have the source code for a Java program, and we want to run that program, we will need both a *compiler* and an *interpreter.* What does the Java compiler do, and what does the Java interpreter do?

The answer is, the Java compiler translates Java programs into a language called Java bytecode. Although bytecode is similar to machine language, it is not the machine language of any actual computer. A Java interpreter is used to run the compiled Java bytecode program. (Each type of computer needs its own Java bytecode interpreter, but all these interpreters interpret the same bytecode language.) **[RGPV/Dec2011(8)]**

SDK Contents
The SDK provides you with several tools that you'd expect to receive from a language vendor, along with a few additional tools that help with the overall development effort. The basic components include the compiler (javac.exe under Windows), the runtime engine (java.exe), and the debugger (jdb.exe). The SDK provides a few other tools that you probably won't use as often:

- *javadoc*— Generates HTML documentation from special comments in your files.
- *appletviewer*— Runs and debugs applets (small programs that run in other programs).
- *jar*— Manages Java archives (collections of files similar to a Zip file or a compressed tar
- archive).
- *native2ascii*— Used to convert files that contain native-encoded characters into UTF format.
- *keytool, jarsigner, policytool*— Provide security tools.
- The SDK also has tools that handle network programming, but you won't need these for a while yet.

The Java Virtual Machine [RGPV/Dec 2009(10)]
Java is the first truly useful portable language. The JVM architecture offers you several advantages: cross-platform portability, size, and security.

Cross-Platform Portability [RGPV/Dec 2010(8)]
The JVM provides cross-platform portability. You write code for the JVM, not for the operating system (OS). Because all JVMs look the same to Java programs, you have to write only one version of your program, and it will work on all JVMs. The JVM interprets the byte-code and carries out the program's operations in a way that is compatible with the current hardware architecture and operating system.

Size
The second interesting side effect of using JVM architecture is the small size of its compiled code. Most of the functionality is buried in the JVM, so the compiled code that runs on top of it doesn't need to be loaded with large libraries. Of course, the JVM is, among other things, a large library, but it is shared among all Java programs. That means that a Java program can be quite small— at least, the part of the program that is uniquely yours. All Java programs share the large JVM, but presumably it is already on the target machine. This is especially important when users are downloading programs over the Internet, for example. Of course, if users' computers don't have a JVM, they'll have a large download for installing the JVM on their machines first. After the JVM installs, the users won't have to worry about installing again.

Security
Java has been designed to protect users from malicious programs. Programs from an untrusted source (for example, the Internet) execute in a restricted environment (known as a *sandbox*). The JVM can then prevent those programs from causing mischief. For example, a Java applet (a small program that runs on a Web page) usually can't access local files or open network connections to arbitrary computers. These restrictions prevent a Web page from erasing your critical files or sending threatening email from your computer.

Java Is a Strongly Typed Language [RGPV/Dec2013 (7)]
It is important to state at the outset that Java is a strongly typed language. Indeed, part of Java's safety and robustness comes from this fact. Let's see what this means. First, every variable has a type, every expression has a type, and every type is strictly defined. Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercions or conversions of conflicting types as in some languages. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	Discuss the features of java language.	Dec2014	3
Q.1	Why Java is called a Strongly-Typed Language.	Dec2013	7
Q.2	If you have the source code for a Java program, and you want to run that program, you will need both a compiler and an interpreter. What does the Java compiler do, and what does the Java interpreter do?	Dec2011	8
Q-3.	What is Java Virtual Machine? What is its role in Java Programming Environment? Explain.	Dec 2009	10
Q-4.	What are the advantages of Plateform Independent Language? Also Explain how Java is Plateform independent?	Dec 2010	8

Unit-01

Topic: Primitive Data Types

Unit-01/Lecture-02

Primitive Data Types in Java [RGPV/Dec 2013(7)]

Integers

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers.

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

Floating point

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision.

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

float

The type **float** specifies a *single-precision* value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision. For example, **float** can be useful when representing dollars and cents. Here are some example **float** variable declarations:

float hightemp, lowtemp;

double

Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, **double** is the best choice.

Here is a short program that uses **double** variables to compute the area of a circle:

```
// Compute the area of a circle.

class Area {
public static void main(String args[]) {
double pi, r, a;
r = 10.8; // radius of circle
pi = 3.1416; // pi, approximately
a = pi * r * r; // compute area
System.out.println("Area of circle is " + a);
}
}
```

Characters

In Java, the data type used to store characters is **char**. However, C/C++ programmers beware: **char** in Java is not the same as **char** in C or C++. In C/C++, **char** is 8 bits wide. This is *not* the case in Java. Instead, Java uses Unicode to represent characters. *Unicode* defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. For this purpose, it requires 16 bits. Thus, in Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536.

Here is a program that demonstrates **char** variables:

```
// Demonstrate char data type.
class CharDemo {
public static void main(String args[]) {
char ch1, ch2;
ch1 = 88; // code for X
ch2 = 'Y';
System.out.print("ch1 and ch2: ");
System.out.println(ch1 + " " + ch2);
}
}
```

This program displays the following output:

```
ch1 and ch2: X Y
```

Booleans

Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**. This is the type returned by all relational operators, as in the case of **a < b**. **boolean** is also the type *required* by the conditional expressions that govern the control statements such as **if** and **for**.

Here is a program that demonstrates the **boolean** type:

```
// Demonstrate boolean values.

class BoolTest {
public static void main(String args[]) {
boolean b;
b = false;
System.out.println("b is " + b);
b = true;
System.out.println("b is " + b);
// a boolean value can control the if statement
if(b) System.out.println("This is executed.");
b = false;
if(b) System.out.println("This is not executed.");
// outcome of a relational operator is a boolean value
System.out.println("10 > 9 is " + (10 > 9));
}
}
```

The output generated by this program is shown here:

```
b is false
b is true
This is executed.
10 > 9 is true.
```

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	With the help of your own example, discuss the primitive data types of Java Language.	Dec.2013	7

Unit-01
Topic: Java Applets and Applications
Unit-01/Lecture 3

Java Applets and Application

An applet is a Java™ program designed to be included in an HTML Web document. You can write your Java applet and include it in an HTML page, much in the same way an image is included. When you use a Java-enabled browser to view an HTML page that contains an applet, the applet's code is transferred to your system and is run by the browser's Java virtual machine.

The HTML document contains tags, which specify the name of the Java applet and its Uniform Resource Locator (URL). The URL is the location at which the applet bytecodes reside on the Internet. When an HTML document containing a Java applet tag is displayed, a Java-enabled Web browser downloads the Java bytecodes from the Internet and uses the Java virtual machine to process the code from within the Web document. These Java applets are what enable Web pages to contain animated graphics or interactive content.

You can also write a Java application that does not require the use of a Web browser.

Applications are stand-alone programs that do not require the use of a browser. Java applications run by starting the Java interpreter from the command line and by specifying the file that contains the compiled application. Applications usually reside on the system on which they are deployed. Applications access resources on the system, and are restricted by the Java security model.

Advantages and Disadvantages of Object-Oriented Programming (OOP)

This reading discusses a advantages and disadvantages of object - oriented programming,
Which is a well – adopted programming style that uses interacting objects to model and
solve complex programming tasks. Two examples of popular object - oriented programming languages are Java and C++. Some other well - known object - oriented programming languages include Objective C, Perl, Python, Javascript, Simula, Modula, Ada, Smalltalk, and the Common Lisp Object Standard.

Some of the advantages of object - oriented programming include: [RGPV/Dec 2013(8)]

1. Improved software - development productivity: Object-oriented programming is modular, as it provides s separation of duties in object - based program development. It is also extensible, as objects can be extended to include new attributes and behaviors. Objects can also be reused within an across applications. Because of these three factors–modularity, extensibility, and reusability–object-oriented programming provides improved software - development productivity over traditional procedure-based programming techniques.

2. Improved software maintainability: For the reasons mentioned above, object -oriented software is also easier to maintain. Since the design is modular, part of the system can be updated in case of issues without a need to make large scale changes.

3. Faster development: Reuse enables faster development. Object - oriented programming languages come with rich libraries of objects, and code developed during projects is also reusable in future projects.

4. Lower cost of development: The reuse of software also lowers the cost of development. Typically, more effort is put in to the object - oriented analysis and design, which lowers the overall cost of development.

5. Higher quality software: Faster development of software and lower cost of development allows more time and resources to be used in the verification of the software. Although quality is dependent upon the experience of the teams object - oriented programming tends to result in higher - quality software.

Some of the disadvantages of object.
-
oriented programming include:

1. Steep learning curve: The thought process involved in object - oriented programming may not be natural for some people, and it can take time to get used to it. It is complex to create programs based on interaction of objects. Some of the key programming techniques, such as inheritance and polymorphism, can be challenging to comprehend initially.

2. Larger program size: Object - oriented programs typically involve more lines of code than procedural programs.

Java is Object Oriented Programming Language [RGPV/ Dec 2013(8)

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever. You, as the user, have only one method of affecting this complex encapsulation: by moving the gear-shift lever. You can't affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a well-defined (indeed, unique) interface to the transmission. Further, what occurs inside the transmission does not affect objects outside the transmission. For example, shifting gears does not turn on the headlights! Because an automatic transmission is encapsulated, dozens of car manufacturers can implement one in any way they please. However, from the driver's point of view, they all work the same. This same idea can be applied to programming.

The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects. In Java, the basis of encapsulation is the class. Although the class will be examined in great detail later in this book, the following brief discussion will be helpful now. A *class* defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as *instances of a class.* Thus, a class is a logical construct; an object has physical reality. When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called *members* of the class. Specifically, the data defined by the class are referred to as *member variables* or *instance variables.* The code that operates on that data is referred to as *member methods* or just *methods.* (If you are familiar with C/C++, it may help to know that what a Java programmer calls a *method,* a C/C++ programmer calls a *function.*) In properly written Java programs, the methods define how the member variables can be used. This means that the behavior and interface of a class are defined by the methods that operate on its instance data.

Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked private or public. The *public* interface of a class represents everything that external users of the class need to know, or may know. The *private* methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable. Since the private members of a class may only be accessed by other parts of your program through the class' public methods, you can ensure that no improper actions take place. Of course, this means that the public interface should be carefully designed not to expose too much of the inner workings of a class (see Figure 2-1).

Inheritance

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. As mentioned earlier, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification *dog,* which in turn is part of the *mammal* class, which is under the larger class *animal.* Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general a ttributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. Let's take a closer look at this process.

Most people naturally view the world as made up of objects that are related to each other in a hierarchical way, such as animals, mammals, and dogs. If you wanted to describe animals in an abstract way, you would say they have some attributes, such as size, intelligence, and type of skeletal system. Animals also have certain behavioral aspects; they eat, breathe, and sleep. This description of attributes and behavior is the *class* definition for animals. If you wanted to describe a more specific class of animals, such as mammals, they would have more specific attributes, such as type of teeth, and mammary glands. This is known as a *subclass* of animals, where animals are referred to as mammals' *superclass.* Since mammals are simply more precisely specified animals, they *inherit* all of the attributes from animals. Adeeply inherited subclass inherits all of the attributes from each of its ancestors in the *class hierarchy.*

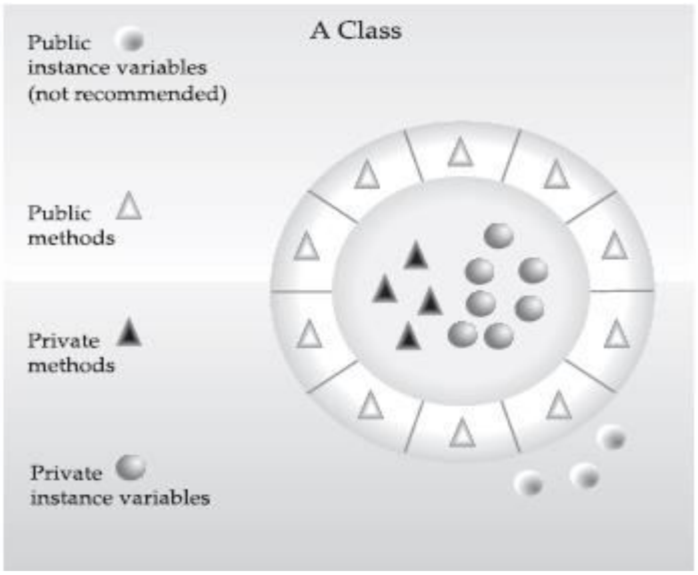

A Class
Public instance variables (not recommended)
Public methods
Private methods
Private instance variables

Fig. 1.3: Encapsulation: public methods can be used to protect private data

Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes *plus* any that it adds as part of its specialization (see Figure 2-2). This is a key concept that lets object-oriented programs grow in complexity linearly rather than geometrically. A new subclass inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.

Polymorphism

Polymorphism (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature FIGURE 2-2 Labrador inherits the encapsulation of all its superclasses of the situation.

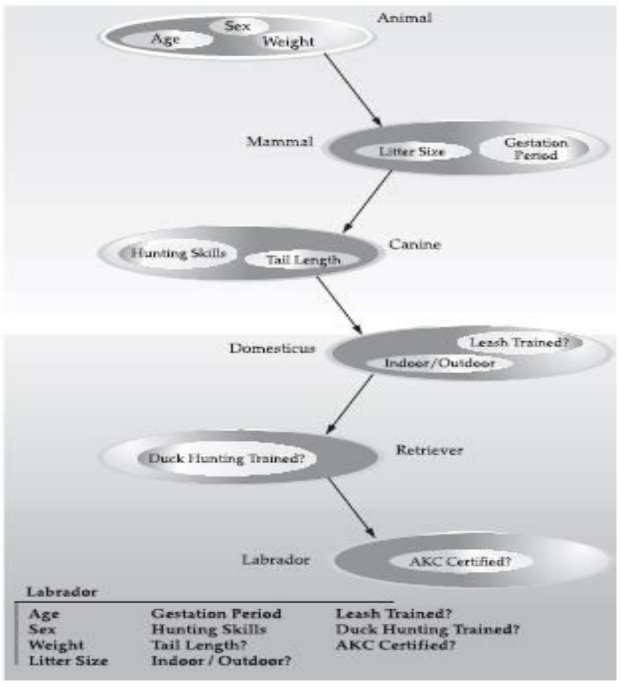


Fig. 1.4: Labrador inherits the encapsulation of all its superclasses

Consider a stack (which is a last-in, first-out list). You might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs. In a non–object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in Java you can specify a general set of stack routines
that all share the same names.

More generally, the concept of polymorphism is often expressed by the phrase "one interface, multiple methods." This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a *general class of action.* It is the compiler's job to select the *specific action* (that is, method) as it applies to each situation. You, the programmer, do not need to make this selection manually. You need only remember and utilize the general interface.

Extending the dog analogy, a dog's sense of smell is polymorphic. If the dog smells a cat,

it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog's nose! This same general concept can be implemented in Java as it applies to methods within a Java program.

S.NO	**RGPV QUESTIONS**	**Year**	**Marks**
Q.1	Explain with example. How you could achieve multiple inheritances in java?	Dec2014	7
Q.2	What are the advantages of Object Oriented Programming?	Dec.2013	8
Q.3	What is Inheritance? Differentiate Subclass and Superclass. Write a program, which inherits the class box (l *b *h) and make a subclass with fourth component weight (w).	Dec.2012	7

Unit-01

Topic: Object-Oriented Concepts with Java

Unit-01/Lecture-04

Overloading Methods[RGPV June 2011 (12), Dec 2014(7)]

We can have same name for more than one method. The number of arguments or the types of arguments is to be different for creating two or more methods with the same name. The return types of the methods can be different as long as the method signature is different. The example below shows overloading of a method called as add. The first add() takes integers arguments and returns the sum as a floating point number. The second add() takes floating point numbers as arguments and return the sum as a floating point number. The third add() takfes strings as arguments, converts them to integer and then returns sum as floating point number.

```
class OverloadMethod{
        static float add(int x, int y){ return (x+y)};
        static float add(float t1, float t2){return(t1+t2);}
        static float add(String s1, String s2)
        { float sum; sum=Integer.parseInt(s1)+Integer.parseInt(s2)}

        public static void main(String args[])
        {
                int x=10,y=20;
                float m=5.5f
                float n=10.5f;
                String s1=”25”, s2=”35”;
                System.out.println(add(x,y));
                System.out.println(add(x,y));
                System.out.println(add(x,y));

        }
}
```

Method Overloading[RGPV June 2011 (12), Dec 2014(7)]

When an object’s method is called, java looks for the method definition in the object’s class. If it can not find then it checks one level up in the hierarchy of classes. Consider the case when the same method name is used in both the subclass and superclass with the same signature(same number of arguments with same type). Here when method is called, method defined in the subclass is invoked. The method defined in the super class is overridden. It is now hidden for the objects of the subclass. If the method defined in the superclass has to be used, then the super keyword can be used along with the name of the method. In the example given below, the method display() and this.display() will invoke method display() of the subclass. The call super.display() will invoke the method display() of the super class. This program uses the super class coded as the program superclass.java.

```
Class superclass
{
	void addnum(int x, int y)
	{
		Int sum;
		sum=x+y;
		System.out.println("Sum of Numbers="+sum);
	}
	void display()
	{
		System.out.println("I am from superclass");
	}
}

Class override extends Superclass
{
	void access()
	{
		System.out.println("Displays from a different place");
		//display method of subclass
		display();
		//display method of superclass
		super.display();
		//display method of subclass
		this.display();
	}
	void display()
	{
		System.out.println("I am from subclass");
	}

	public static void main(String args[])
	{
		override s1 = new override();
		s1.access();
	}}
```

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	Write a Java Program to implement Method overloading and Method Overriding.	June 2011	12

Unit-01

Topic: Abstract Classes and Methods

Unit-01/Lecture-05

Abstract Classes and Methods [RGPV/June 2011(8)]

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.

Abstract Method

We can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

abstract *type name(parameter-list)*;

As you can see, no method body is present. Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract**. Here is a simple example of a class with an abstract method, followed by a class which implements that method:

```
// A Simple demonstration of abstract.
abstract class A {
abstract void callme();
// concrete methods are still allowed in abstract classes
void callmetoo() {
System.out.println("This is a concrete method.");
}
}
class B extends A {
```

```
void callme() {
System.out.println("B's implementation of callme.");
}
}
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}
```

Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class **A** implements a concrete method called **callmetoo( )**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object. You will see this feature put to use in the next example.

Using an abstract class, you can improve the **Figure** class shown earlier. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the program declares **area( )** as abstract inside **Figure**. This, of course, means that all classes derived from **Figure** must override **area( )**.

```
// Using abstract methods and classes.
abstract class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
// area is now an abstract method
abstract double area();
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
// override area for rectangle
double area() {
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
// override area for right triangle
double area() {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}
class AbstractAreas {
public static void main(String args[]) {
// Figure f = new Figure(10, 10); // illegal now
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref; // this is OK, no object is created
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
}
}
```

As the comment inside **main( )** indicates, it is no longer possible to declare objects of type **Figure**, since it is now abstract. And, all subclasses of **Figure** must override **area( )**. To prove this to yourself, try creating a subclass that does not override **area( )**. You will receive a compile-time error.

INTERFACES [RGPV/Dec 2012(8)]

Using the keyword **interface**, you can fully abstract a class' interface from its implementation. That is, using **interface**, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces that don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces. To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and non-extensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

Defining an Interface

An interface is defined much like a class. This is the general form of an interface:

```
access interface name {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
type final-varnameN = value;
}
```

When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface. *name* is the name of the interface, and can be any valid identifier. Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods. Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly **public**.

Here is an example of an interface definition. It declares a simple interface that contains one method called **callback( )** that takes a single integer parameter.

```
interface callback {
void callback(int param);
}
```

Implementing Interfaces
Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the **implements** clause looks like this:

class *classname* [extends *superclass*] [implements *interface* [,*interface...*]] {
// class-body
}

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

Here is a small example class that implements the **Callback** interface shown earlier.

```
class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("callback called with " + p);
}
}
```

Notice that **callback( )** is declared using the **public** access specifier.

```
class TestIface {
public static void main(String args[]) {
Callback c = new Client();
c.callback(42);
}
}
```

The output of this program is shown here:

```
callback called with 42
```

Applying Interfaces
To understand the power of interfaces, let's look at a more practical example. In earlier chapters, you developed a class called **Stack** that implemented a simple fixed-size stack. However, there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be "growable." The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods **push( )** and **pop( )** define the interface to the stack independently of

the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics. Let's look at two examples.

First, here is the interface that defines an integer stack. Put this in a file called **IntStack.java**. This interface will be used by both stack implementations.

```
// Define an integer stack interface.
interface IntStack {
void push(int item); // store an item
int pop(); // retrieve an item
}
```

The following program creates a class called **FixedStack** that implements a fixed-length version of an integer stack:

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
private int stck[];
private int tos;
// allocate and initialize stack
FixedStack(int size) {
stck = new int[size];
tos = -1;
}
// Push an item onto the stack
public void push(int item) {
if(tos==stck.length-1) // use length member
System.out.println("Stack is full.");
else
stck[++tos] = item;
}
// Pop an item from the stack
public int pop() {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
}
class IFTest {
public static void main(String args[]) {
FixedStack mystack1 = new FixedStack(5);
FixedStack mystack2 = new FixedStack(8);
// push some numbers onto the stack
for(int i=0; i<5; i++) mystack1.push(i);
for(int i=0; i<8; i++) mystack2.push(i);
// pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<5; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<8; i++)
System.out.println(mystack2.pop());
}
}
```

Following is another implementation of **IntStack** that creates a dynamic stack by use of the same **interface** definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.

```
// Implement a "growable" stack.
class DynStack implements IntStack {
private int stck[];
private int tos;
// allocate and initialize stack
DynStack(int size) {
stck = new int[size];
tos = -1;
}
// Push an item onto the stack
public void push(int item) {
// if stack is full, allocate a larger stack
if(tos==stck.length-1) {
int temp[] = new int[stck.length * 2]; // double size
for(int i=0; i<stck.length; i++) temp[i] = stck[i];
stck = temp;
stck[++tos] = item;
}
else
stck[++tos] = item;
}
// Pop an item from the stack
public int pop() {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
}
class IFTest2 {
public static void main(String args[]) {
DynStack mystack1 = new DynStack(5);
DynStack mystack2 = new DynStack(8);
// these loops cause each stack to grow
for(int i=0; i<12; i++) mystack1.push(i);
for(int i=0; i<20; i++) mystack2.push(i);
System.out.println("Stack in mystack1:");
for(int i=0; i<12; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<20; i++)
System.out.println(mystack2.pop());
}
}
```

Variables in Interfaces

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class (that is, when you “implement” the interface), all of those variable names will be in scope as constants. (This is similar to using a header file in C/C++ to create a large number of **#defined** constants or **const** declarations.) If an interface contains no methods, then any class that includes such an interface doesn’t actually implement anything.

It is as if that class were importing the constant fields into the class name space as **final** variables. The next example uses this technique to implement an automated “decision maker”:

```
import java.util.Random;
interface SharedConstants {
int NO = 0;
int YES = 1;
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5;
}
class Question implements SharedConstants {
Random rand = new Random();
```

```
int ask() {
int prob = (int) (100 * rand.nextDouble());
if (prob < 30)
return NO; // 30%
else if (prob < 60)
return YES; // 30%
else if (prob < 75)
return LATER; // 15%
else if (prob < 98)
return SOON; // 13%
else
return NEVER; // 2%
}
}
class AskMe implements SharedConstants {
static void answer(int result) {
switch(result) {
case NO:
System.out.println("No");
break;
case YES:
System.out.println("Yes");
break;
case MAYBE:
System.out.println("Maybe");
break;
case LATER:
System.out.println("Later");
break;
case SOON:
System.out.println("Soon");
break;
case NEVER:
System.out.println("Never");
break;
}
}

public static void main(String args[]) {
Question q = new Question();
answer(q.ask());
answer(q.ask());
answer(q.ask());
answer(q.ask());
}
}
```

Notice that this program makes use of one of Java's standard classes: **Random**. This class provides pseudorandom numbers. It contains several methods that allow you to obtain random numbers in the form required by your program. In this example, the method **nextDouble( )** is used. It returns random numbers in the range 0.0 to 1.0. In this sample program, the two classes, **Question** and **AskMe**, both implement the **SharedConstants** interface where **NO**, **YES**, **MAYBE**, **SOON**, **LATER**, and **NEVER** are defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly. Here is the output of a sample run of this program. Note that the results are different each time it is run.

```
Later
Soon
No
Yes
```

Extended Interfaces

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:

```
// One interface can extend another.
interface A {
void meth1();
void meth2();
}
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
void meth3();
}
// This class must implement all of A and B
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().");
}
}

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

As an experiment, you might want to try removing the implementation for **meth1( )** in **MyClass**. This will cause a compile-time error. As stated earlier, any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

S.NO	RGPV QUESTIONS	Year	Marks
.1	What is Polymorphism? What are the types of Polymorphism? Compare method overloading and method overriding with suitable example.	Dec 2014	7
Q.1	Explain the difference between Abstract class and Interface.	June 2011	8
Q.2	Describe the aspect of Polymorphism by the keword “Interface”. Explain the syntax for defining the Interface	Dec-2012	12

Unit-01
Topic: Packages
Unit-01/Lecture-06

Packages [RGPV/ June 2011(8)]

A Package is a collection of classes and interfaces of similar nature. For example, java.io package contains classes and interfaces for various kinds of input and output. Package defines a boundary to see how classes and interfaces interact with one another. Therefore, it also acts as a mode of protection. Java language programs automatically import all classes in the java.lang package. Package gives us the following advantages. They help to avoid conflict in naming classes. Classes, methods and variable can be protected in a better way.

To import classes from a package, import command is used.

```
import java.io.*;
import pack.subpack.Myclass;
```

In the first case all public classes in the package java.io are available. In the second case only class having the name pack.subpack.myclass is available for use.

Defining a Package
To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name. (This is why you haven't had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time,
you will define a package for your code.

This is the general form of the **package** statement:
package *pkg*;

Here, *pkg* is the name of the package. For example, the following statement creates a package called **MyPackage**.

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly.

Finding Packages and Classpath

As just explained, packages are mirrored by directories. This raises an important question: How does the Java run-time system know where to look for packages that you create? The answer has three parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes. For example, consider the following package specification: package MyPack

In order for a program to find **MyPack,** one of three things must be true. Either the program can be executed from a directory immediately above **MyPack**, or the **CLASSPATH** must be set to include the path to **MyPack**, or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**. When the second two options are used, the class path *must not* include **MyPack**, itself. It must simply specify the *path to* **MyPack**. For example, in a Windows environment, if the path to **MyPack** is

C:\MyPrograms\Java\MyPack
Then the class path to **MyPack** is

C:\MyPrograms\Java

The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the **.class** files into the appropriate directories, and then execute the programs from the development directory. This is the approach used in the following example.

A Short Package Example

```
// A simple package
package MyPack;
class Balance {
String name;
double bal;
Balance(String n, double b) {
name = n;
bal = b;
}

void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

Call this file **AccountBalance.java** and put it in a directory called **MyPack**. Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then, try executing the **AccountBalance** class, using the following command line:

```
java MyPack.AccountBalance
```

Remember, you will need to be in the directory above **MyPack** when you execute this command.
(Alternatively, you can use one of the other two options described in the preceding section to specify the path **MyPack**.)

As explained, **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line:

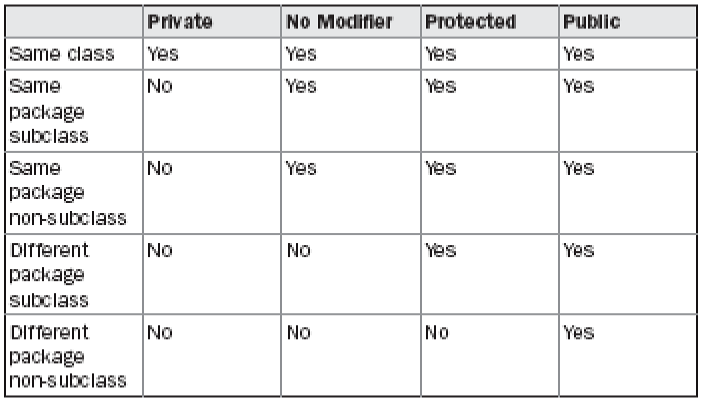
```
java AccountBalance
```

AccountBalance must be qualified with its package name.

Access Protection

The source for the first package defines three classes: **Protection**, **Derived**, and **SamePackage**. The first class defines four **int** variables in each of the legal protection modes. The variable **n** is declared with the default protection, **n_pri** is **private**, **n_pro** is **protected**, and **n_pub** is **public**. Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out. Before each of these lines is a comment listing the places from which this level of protection would allow access.

The second class, **Derived**, is a subclass of **Protection** in the same package, **p1**. This grants **Derived** access to every variable in **Protection** except for **n_pri**, the **private** one. The third class, **SamePackage**, is not a subclass of **Protection**, but is in the same package and also has access to all but **n_pri**.



	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Example

This is file **Protection.java**:

```
package p1;
public class Protection {
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
public int n_pub = 4;
public Protection() {
System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file **Derived.java**:

```
package p1;
class Derived extends Protection {
Derived() {
System.out.println("derived constructor");
System.out.println("n = " + n);
// class only
// System.out.println("n_pri = "4 + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file **SamePackage.java**:

```
package p1;
class SamePackage {
SamePackage() {
Protection p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

Following is the source code for the other package, **p2**. The two classes defined in **p2** cover the other two conditions that are affected by access control. The first class, **Protection2**, is a subclass of **p1.Protection**. This grants access to all of **p1.Protection**'s variables except for **n_pri** (because it is **private**) and **n**, the variable declared with the default protection. Remember, the default only allows access from within the class or the package, not extra-package subclasses. Finally, the class **OtherPackage** has access to only one variable, **n_pub**, which was declared **public**.

This is file **Protection2.java**:

```
package p2;
class Protection2 extends p1.Protection {
Protection2() {
System.out.println("derived other package constructor");
// class or package only
// System.out.println("n = " + n);
// class only
// System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file **OtherPackage.java**:

```
package p2;
class OtherPackage {
```

```
OtherPackage() {
p1.Protection p = new p1.Protection();
System.out.println("other package constructor");
// class or package only
// System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
// class, subclass or package only
// System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

If you wish to try these two packages, here are two test files you can use. The one for package **p1** is shown here:

```
// Demo package p1.
package p1;
// Instantiate the various classes in p1.
public class Demo {
public static void main(String args[]) {
Protection ob1 = new Protection();
Derived ob2 = new Derived();
SamePackage ob3 = new SamePackage();
}
}
```

The test file for **p2** is shown next:

```
// Demo package p2.
package p2;
// Instantiate the various classes in p2.
public class Demo {
public static void main(String args[]) {
Protection2 ob1 = new Protection2();
OtherPackage ob2 = new OtherPackage();
}
}
```

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	What is Package? Explain the Packages with example and how to import packages	Dec 2013	8

Unit 1

Topic: Constructors

Unit 1/Lecture 7

Constructor [RGPV/June 2011(12)]
A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes. Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

You can rework the **Box** example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace **setDim( )** with a constructor. Let's begin by defining a simple constructor that simply sets the dimensions of each box to the same values. This version is shown here:

```
/* Here, Box uses a constructor to initialize the
dimensions of a box.
*/
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box() {
System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

As you can see, both **mybox1** and **mybox2** were initialized by the **Box( )** constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10,both **mybox1** and **mybox2** will have the same volume. The **println( )** statement inside **Box( )** is for the sake of illustration only. Most constructors will not display anything. They will simply initialize an object. Before moving on, let's reexamine the **new** operator. As you know, when you allocate an object, you use the following general form:

class-var = new *classname*( );

Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

Box mybox1 = new Box();

new Box( ) is calling the **Box( )** constructor. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class.

Copy Constructor [RGPV/Dec-2012(8)]

Like C++, Java also supports copy constructor. But, unlike C++, Java doesn't create a default copy constructor if you don't write your own.

Following is an example Java program that shows a simple use of copy constructor.

```
// filename: Main.java
```

```
class Complex {

    private double re, im;

    // A normal parametrized constructor
    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    // copy constructor
    Complex(Complex c) {
        System.out.println("Copy constructor called");
        re = c.re;
        im = c.im;
    }

    // Overriding the toString of Object class
    @Override
    public String toString() {
        return "(" + re + " + " + im + "i)";
    }
}

public class Main {

    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);

        // Following involves a copy constructor call
        Complex c2 = new Complex(c1);

        // Note that following doesn't involve a copy constructor call as
        // non-primitive variables are just references.
        Complex c3 = c2;

        System.out.println(c2); // toString() of c2 is called here
    }
}
```

Output:

```
Copy constructor called
(10.0 + 15.0i)
```

Now try the following Java program:

```
// filename: Main.java

class Complex {

   private double re, im;

   public Complex(double re, double im) {
      this.re = re;
      this.im = im;
   }
}

public class Main {

   public static void main(String[] args) {
      Complex c1 = new Complex(10, 15);
      Complex c2 = new Complex(c1);  // compiler error here  }}
```

S.NO	**RGPV QUESTION**	**YEAR**	**MARKS**
Q.1	Explain Constructor Overloading with Example. Also, describe the usage of super keyword.	JUNE 2011	8
Q.2	Explain what are the constructors? Discuss Copy Constructor with example.	DEC 2012	8

Unit 1

Topic: The Keyword This

Unit 1/Lecture-8

The this keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted. To better understand what **this** refers to, consider the following version of **Box( )**:

```
// A redundant use of this.
Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
```

This version of **Box( )** operates exactly like the earlier version. The use of **this** is redundant, but perfectly correct. Inside **Box( )**, **this** will always refer to the invoking object. While it is redundant in this case, **this** is useful in other contexts, one of which is explained in the next section.

The finalize() Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization.* By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize( )** method on the object. The **finalize( )** method has this general form:

```
protected void finalize( )
{
```

```
// finalization code here
}
```

Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class. It is important to understand that **finalize( )** is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize( )** will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize( )** for normal program operation.

Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection.* It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program.

It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

References:

Book	Authr	Priority
Java Programming	Herbertt Schield	1
Java	E Balaguruswamy	2
Java	Khalid Mugal	3