UNIT 2: LANGUAGE FEATURES

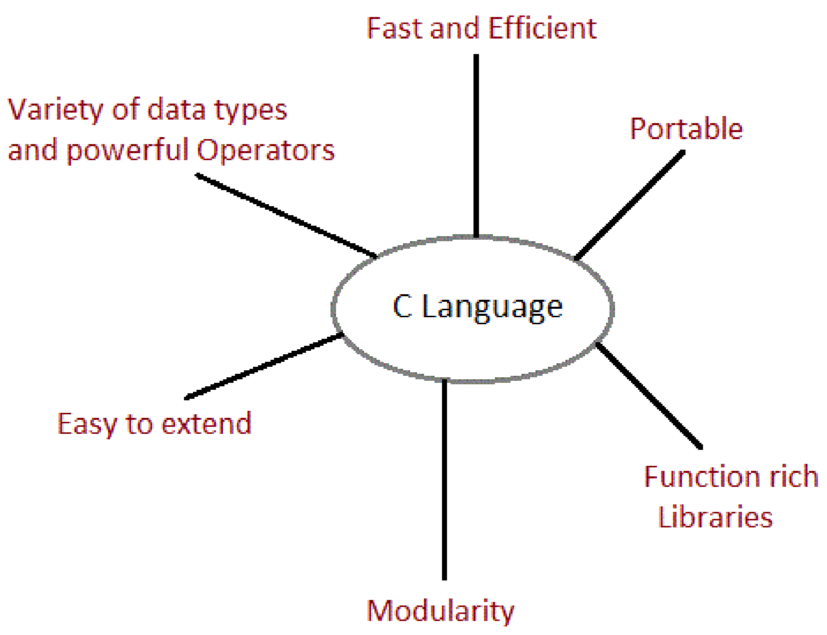
Unit Structure

2.0 Learning Objectives

After learning this unit, you will be able to:

- Static Keyword
- Abstract Classes, Interfaces
- Simple type Wrappers
- Converting Numbers to and from Strings
- Packages

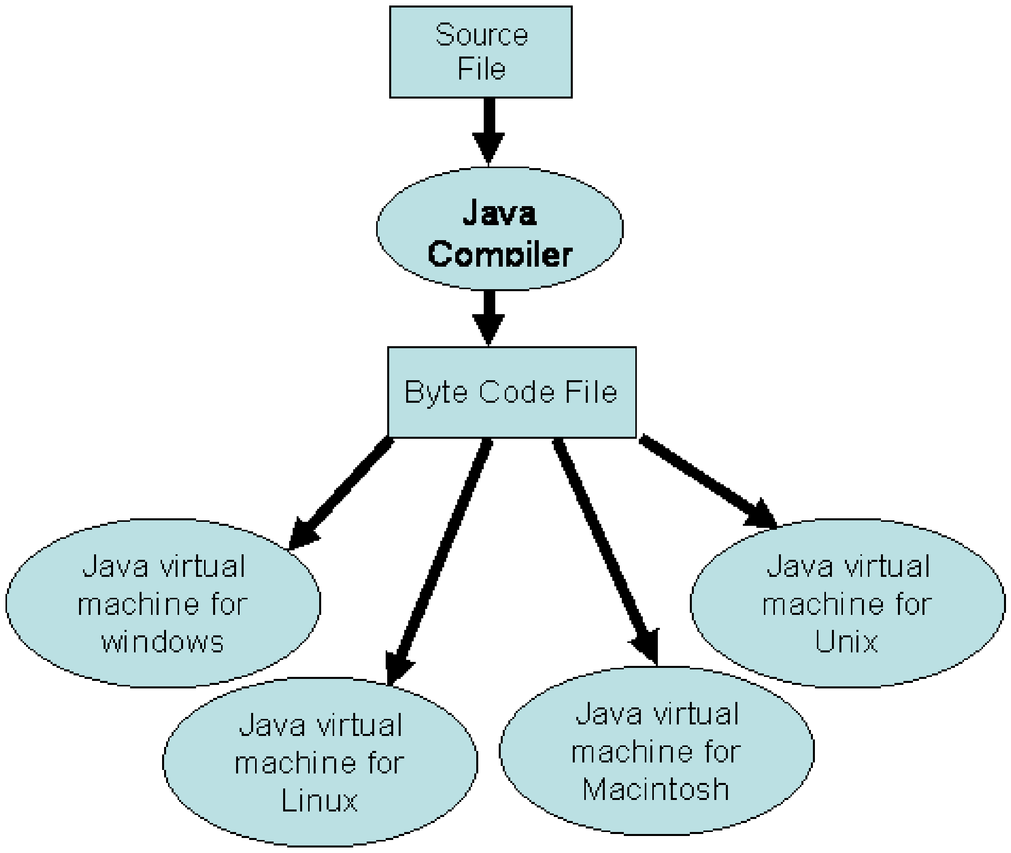
2.1 Introduction



The development of Java has been a compilation of the best points of various programming languages such as C and C++. Java therefore utilizes algorithms and methodologies that are already proven. The Java environment automatically tackles tasks which are prone to errors such as pointers and memory management rather than the programmer taking the initiative.

Since Java is primarily a derivative of C++ that most programmers are conversant with, it implies that Java has a familiar feel rendering it easy to use. The Java language supports many high-performance features such as multithreading, just-in-time compiling and native code usage.

2.2 Static Keyword


Source File
Java Compiler
Byte Code File
Java virtual machine for windows
Java virtual machine for Linux
Java virtual machine for Macintosh
Java virtual machine for Unix

When an object is created or, primitive type variable or method is called, the memory for that object, variable or method is set aside.

The different objects, variables and methods occupy different areas of memory when created/called. In some cases, we would like to have multiple objects, variables or methods which occupy the same area of memory (in effect just having the one instance of that variable or method). The above can be achieved by using the static keyword; it is possible to have static methods and variables.

In Java, global variables are not allowed. In order to do the same, the instance variable in the class can be declared static. The effect of doing this is that when we create multiple objects of that class, every object shares the same instance variable that was declared to be static.

To make an instance variable static, we simply precede the declaration with the static keyword:

public static intInstanceVariable = 0

In effect, what we are really doing is saying that this instance variable, no matter how many objects are created should always reside in the same memory location regardless of the object. This then stimulates a ‘global variable’ of sorts.

We usually make a variable declared to be final, static as well since it makes sense to only have the one instance of a constant. The static instance variables are also called as class variables.

Outside of the class in which they are defined, static methods and variables can be used independently of any object. In order to do so, you only need to specify the name of the class followed by the dot operator.

Check your progress 1

1. Explain how to make an instance variable static.
2. What are static instance variables also called as?

....

....

....

....

....

....

....

....

....

....

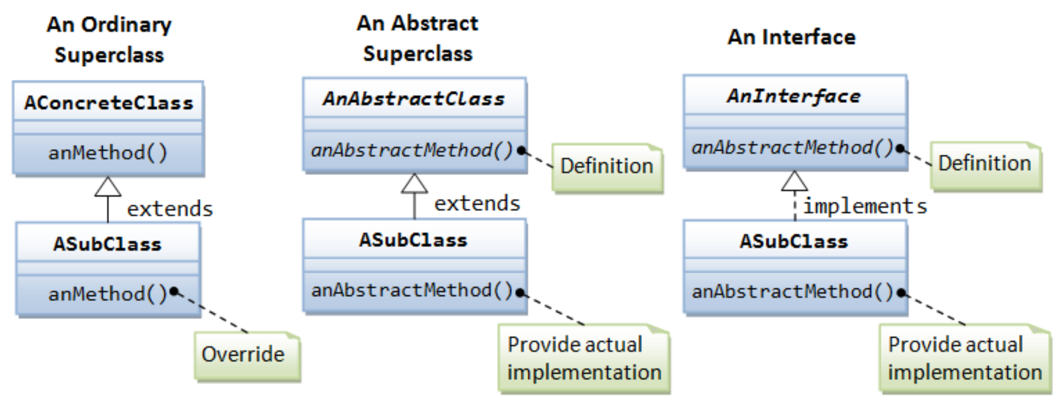
....

....

....

....

2.3 Using Abstract Classes



There are often situations where you want to determine a superclass, which without providing a complete implementation of every method declares the structure of an abstraction. That is, many a times you'll 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.

The abstract keyword can be used with:

1. A class
2. A method

Abstract Method

In a method declaration, abstract indicates that the implementation will be in subclass. Since these methods do not have an implementation specified in the superclass they are sometimes cited as subclasser responsibility. Hence, a subclass cannot use the version defined in the superclass, it must override them. To declare an abstract method, use this general form:

abstract type name (parameter-list);

No method body is present as specified above.

Abstract Class:

A class that is declared abstract is defined as an abstract class. The class need not necessarily include abstract methods and can be subclassed. Abstract classes cannot be instantiated.

In order to declare a class as abstract, you have to use the abstract keyword before the class keyword at the beginning of the class declaration. There can be no objects of the abstract class, that is, an abstract class cannot be directly instantiated with the new operator.

Any derived class that does not implement all abstract methods of its superclass must be declared abstract. Let us take an Example: to understand this concept in more detail.

In the given program, the class Figure is declared as abstract because we don't want objects of this class to be created. Instead, this class should be subclassed. Notice that the method area ( ) is also abstract because we cannot define it in the Figure class. It is defined in the subclass –Rect

```
abstract class Figure
{
protected double dim1, dim2
figure (double dim1, double dim2)
{
this.dim1 =dim1
this.dim2 =dim2
}
abstract double area ( )          //abstract method
}
classRect extends Figure
{
Rect (double l, double d)
{
super (l, d)
}
double area ( )
{
return dim1 * dim2;
}
}

public class AbstractDemo
{
public static void main (String args [ ])
{
Rect r = new rect (15.2, 25.5)
System.out.println (“The area=” + r.area ( ))
}
}
```

Check your progress 2

1. Where is the abstract keyword used?
2. Write the general form of abstract method.

....

....

....

....

....

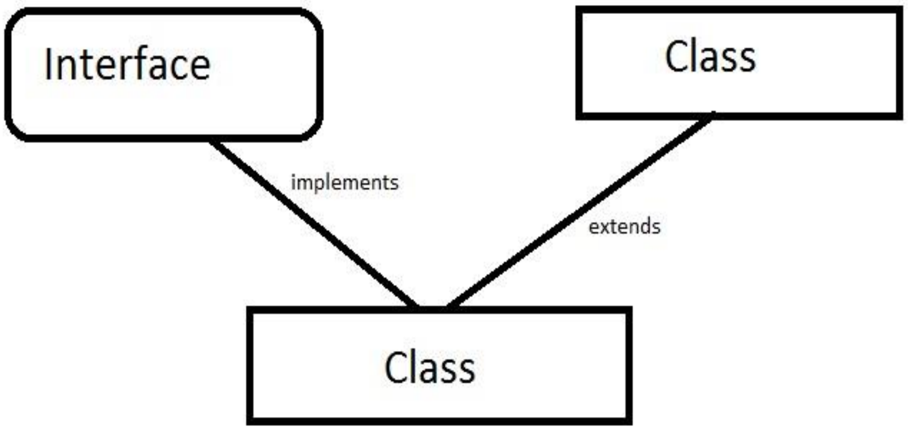
....

....

....

....

2.4 Interfaces



“A collection of abstract methods is an interface. Thus, be inheriting the abstract methods of an interface a class implements an interface. “

“An interface is not a class. They are two different concepts but writing an interface is similar to a class. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements.”

“Every method of the interface is defined in the class unless the class implementing the interface is abstract.”

An interface is similar to a class in the following ways:

- The interface contains various methods.
- The name of the interface matches the name of the file and it is written with a .java extension.
- The bytecode of an interface appears in a .class file.
- An interface appears in packages and the bytecode file it corresponds to must appear in a directory structure matching its name.

However, an interface is different from a class in several ways, including:

- You cannot instantiate an interface.
- Constructors do not constitute an interface.
- All of the methods in an interface are abstract.
- An interface can only contain fields that are declared both static and final and it cannot contain instance fields.

- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interfaces:

The interface keyword is used to declare an interface.

Encapsulation is defined as a barrier protecting and preventing the code and data from being randomly accessed by other code outside the class. The access is tightly controlled by an interface.

The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code. With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

Example:

Let us look at an Example: that depicts encapsulation:

```
/* File name : NameOfInterface.java */
import java.lang.*
//Any number of import statements
public interface NameOfInterface
{
  //Any number of final, static fields
  //Any number of abstract method declarations\
}
```

Interfaces have the following properties:

- While declaring an interface you do not need to use the abstract keyword since the interface is implicitly abstract.
- The abstract keyword is not needed as each method in an interface is implicitly abstract.
- Methods in an interface are implicitly public.

Implementing Interfaces:

The process of a class implementing an interface can be seen as the class signing a contract, complying to carry out certain behaviors of the interface. In case a class fails to carry out these behaviors, the class must declare itself abstract.

In a class the implements keyword is used to implement the interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

“When you define overriding methods in interfaces, the following rules are to be followed:

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.

- When overriding methods, you must maintain the signature of the interface method and also the same return type or subtype.
- Interface methods do not have to be implanted if in case an implementation class itself is abstract.
- While implementing interfaces, there are several rules:
    - A class can implement more than one interface at a time.
    - A class can extend only one class but implement many interfaces.
    - An interface itself can extend another interface. An interface cannot extend another interface.”

Extending Interfaces:

Just as a class can extend another class, an interface can extend another interface as well. The extends keyword is used to extend an interface and the child interface inherits the methods of the parent interface.

“The following Sports interface is extended by Hockey and Football interfaces.

```
//Filename: Sports.java

public interface Sports
```

```
{
   public void setHomeTeam(String name)
   public void setVisitingTeam(String name)
}


//Filename: Football.java
public interface Football extends Sports
{
   public void homeTeamScored(int points)
   public void visitingTeamScored(int points)
   public void endOfQuarter(int quarter)
}
//Filename: Hockey.java
public interface Hockey extends Sports
{
   public void homeGoalScored()
   public void visitingGoalScored()
   public void endOfPeriod(int period)
   public void overtimePeriod(intot)
}
```

The Hockey interface has four methods but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.”

Check your progress 3

1. Explain in what ways is an interface similar to a class.
2. Write the rules for implementing interfaces.

....

....

....

....

....

....

....

....

2.5 Simple type Wrappers

Java uses simple types, such as int and char, for performance reasons. These data types are passed by value to methods and cannot be passed directly by reference, hence not making them a part of the object hierarchy. Also, there is no way for two methods to refer to the same instance of an int. At times, you'll need to create an object representation for one of these simple types. To address this need, Java provides classes that correspond to each of the simple types.

These classes are commonly referred to as type wrappers since they encapsulate the simple types in a class.

Number:

"A superclass is defined by the abstract class Number that implements the classes that wrap the numeric type's byte, short, int, long, float and double. Number possesses abstract methods that return the value of the object in each of the different number formats." That is, doubleValue ( ) returns the value as a double, floatValue ( ) returns the value as a float and so on. These methods are shown here:

bytebyteValue( )

doubledoubleValue( )

floatfloatValue( )

intintValue( )

longlongValue( )

shortshortValue( )

The values returned by these methods can be rounded. Number wrapper class has six concrete subclasses that hold explicit values of each numeric type: Double, Float, Byte, Short, Integer and Long.

Double and Float:

"Double and Float are wrappers for floating-point values of type double and float respectively. The constructors of float are given below:

Float (double num)

Float (float num)

Float (String str) throws NumberFormatException

The Float objects can be constructed with values of type float or double. They can also be constructed from the string representation of a floating-point number.

Whereas, the constructors for Double are shown below:

Double (double num)

Double (String str) throws NumberFormatException

Double objects can be constructed with a double value or a string containing a floating-point value."

The given Example: creates two Double objects, one by using a double value and the other by passing a string that can be parsed as a double.

```
classDoubleDemo
{
public static void main (String args [ ])
{
```

```
double d1 = new Double (3.14159);
double d2=new Double (“314159E-5”)
System.out.println (d1 + “= “ + d2 + “ -> “ + c1.equals(d2))
}
}
```

The output of this program is given below:

```
3.14159 = 3.14159 ->true
```

As shown in the output, both constructors created identical Double instances as shown by the equals ( ) method returning true.

Byte, Short, Integer and Long:

The Byte, Short, Integer and Long classes are wrappers for byte, short, int and long integer types respectively. Their constructors are shown here:

```
Byte (byte num)
Byte (String str) throws NumberFormatException
Short(short num)
Short(String str) throws NumberFormatException
Integer (intnum)
Integer (String str) throws NumberFormatException
Long(long num)
Long (String str) throws NumberFormatException
```

These objects can be constructed from numeric values or from strings that contain valid whole number values.

Character:

Character is a simple wrapper around a char. The constructor for Character is

```
Character (char ch)
```

Here, ch specifies the character that will be wrapped by the character object being created.To obtain the char value contained in a Character object, call charValue( ), shown below:

charcharValue( )

The above statement returns a character. Character includes several static methods that categorize characters and alter their case. The given Example: demonstrates several of these methods.

```
public class IsDemo {

  public static void main(String[] args) {

    char a[] = {'a','b','5','?','A',' '}

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

      if(Character.isDigit(a[i]))
        System.out.println(a[i] + "is a digit ")
      if(Character.isLetter(a[i]))
        System.out.println(a[i] + "is a letter ")
      if(Character.isWhitespace(a[i]))
        System.out.println(a[i] + "is a White Space ")
      if(Character.isLowerCase(a[i]))
        System.out.println(a[i] + "is a lower case ")
      if(Character.isLowerCase(a[i]))
        System.out.println(a[i] + "is a upper case ")
    }

  }
}
```

Boolean:

Boolean is a very thin wrapper around boolean values, which is useful mostly when you want to pass a boolean variable by reference. It contains the constants TRUE and FALSE which define true and false Boolean objects. Boolean also defines the TYPE field, which is the Class object for boolean. Boolean defines these constructors:

Boolean (booleanboolValue)

Boolean (String boolString)

In the fist version, boolValue must be either true or false. In the second version, if boolString contains the string “true” (upper or lowercase), then the new Boolean object will be true. Otherwise, it will be false.

VOID

The Void class has one field, TYPE which holds a reference to the Class object for type void. You do not create instances of this class

Check your progress 4

1. Write a note on abstract class number.
2. Explain Boolean wrapper.

....

....

....

....

....

....

....

....

....

....

....

....

....

....

2.6 Converting Numbers to and from Strings

Java provides an easy way to convert numbers into string. The Byte, Short, Integer and Long classes provide the parseByte( ), parseShort( ), parseInt( ) and parseLong( ) methods, respectively. These methods return the byte, short, int or long equivalent of the numeric string with which they are called.

The given below program demonstrates the use of parseInt( ). It finds the sum of a list of integers entered by the user. It reads the integers using readLine ( ) and uses parseInt( ) to convert these strings into their int equivalents.

/*Program to convert an integer into binary, hexadecimal and octal */

```
Class StringConversions

{

public static void main (String args[ ])

{

intnum=19648

System.out.println (num + “ in binary” + Integer.toBinaryString(num))

System.out.println (num + “in octal” + Integer.toOctalString (num))

System.out.println (num + “in hexadecimal” + Integer.toHexString (num))

}

}
```

The output of the above program is as follows:

```
19648 in binary: 100110011000000

19648 in octal: 46300

19648 in hexadecimal: 4cc0.
```

Check your progress 5

1. Explain how you can convert numbers into string.
2. Write a program to demonstrate the use of parseInt ().

....

....

....

....

....

....

....

....

....

....

....

....

2.7 Packages

Java uses packages to avoid naming conflicts, to ease the searching and usage of interfaces, classes, annotations and enumerations and to control access.

Packages are a collection or group of related types of (classes, interfaces, enumerations and annotations) providing access protection and name space management.

Some of the existing packages in Java are:

- java.lang - bundles the fundamental classes
- java.io - classes for input , output functions are bundled in this package

Programmers can bundle up a group of classes/interfaces in order to define their own packages. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, annotations are related.

There are to be no conflicts with names in various other packages since a package creates a new namespace. With the help of packages, providing access control and locating related classes can be done with ease.

Creating a package:

You have to select a name for the package and put a package statement with that very name at the top of every source file that contains the classes, interfaces, enumerations and annotation types that you want to include in the package when you are creating it.

The first line in the source file must be the package statement. Each source file can have only one package statement which shall apply to all types in the file.

The class, interfaces, enumerations and annotation types are put into an unnamed package if a package statement is not used.

Example:

Let us look at an Example: that creates a package called animals. It is a common practice to use lowercased names of packages to avoid any conflicts with the names of classes, interfaces.

Put an interface in the package *animals*:

```
/* File name : Animal.java */
package animals


interface Animal {
  public void eat()
  public void travel()
}
```

Now put an implementation in the same package *animals*:

```
package animals;


/* File name : MammalInt.java */
public class MammalInt implements Animal{


   public void eat(){
System.out.println("Mammal eats")
   }


   public void travel(){
System.out.println("Mammal travels")
   }


   public intnoOfLegs(){
      return 0
   }


   public static void main(String args[]){
MammalInt m = new MammalInt()
m.eat()
      m.travel()
   }
}
```

The import Keyword:

If a class wants to use another class in the same package, the package name does not need to be used. Classes in the same package find each other without any special syntax.

Example:

Here a class named Boss is added to the payroll package that already contains Employee. The Boss can then refer to the Employee class without using the payroll prefix, as demonstrated by the following Boss class.

```
package payroll;


public class Boss
{
public void payEmployee(Employee e)
{
e.mailCheck()
}
}
```

- The package can be imported using the import keyword and the wild card (*) character. For Example:,

  import payroll.*

- The class itself can be imported using the import keyword. For Example:

  import payroll.Employee;

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

The Directory Structure of Packages:

When a class is placed in a package, the following results are concluded:

- As stated in the previous section, the name of the package becomes a part of that of the classes' name.
- The name of the package must match the directory structure where the corresponding bytecode resides.

Check your progress 6

1. Name the existing packages in Java.
2. What are the results when a class is placed in a package?.

....

....

....

....

....

....

....

....

....

....

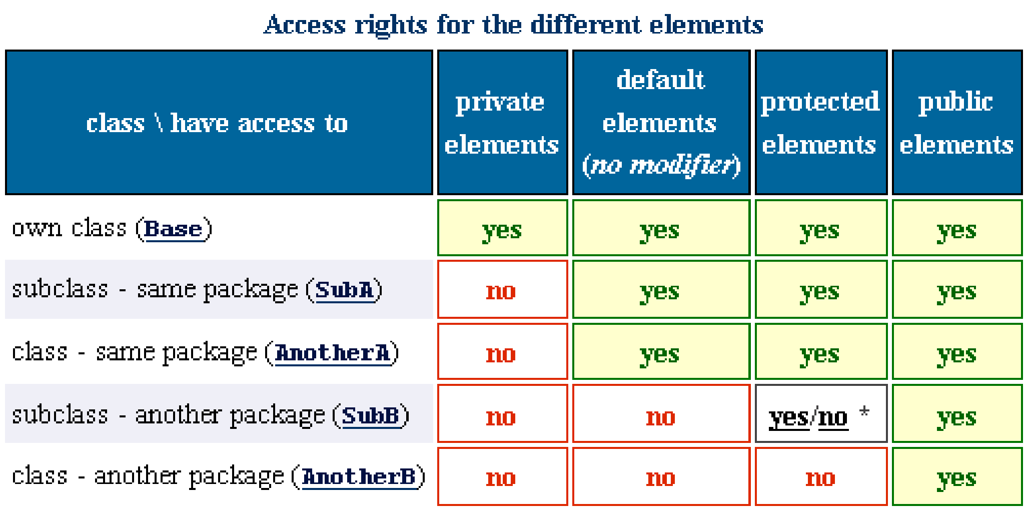
....

....

2.8 Access Protection

Packages add another dimension to access control, they act as containers for classes, other subordinate packages, data and code. The class is Java's smallest unit of abstraction, due to the interplay between classes and packages, Java addresses four categories of visibility for class members, which are mentioned below:

- Subclasses in the same package.
- Non-subclasses in the same package.
- Subclasses in different packages.
- Classes that are neither in the same package nor subclasses.



Access rights for the different elements

class \ have access to	private elements	default elements (*no modifier*)	protected elements	public elements
own class (**Base**)	yes	yes	yes	yes
subclass - same package (**SubA**)	no	yes	yes	yes
class - same package (**AnotherA**)	no	yes	yes	yes
subclass - another package (**SubB**)	no	no	yes/no *	yes
class - another package (**AnotherB**)	no	no	no	yes

Table 6.1: Class Member Access

The three access specifies, private, public and protected, provide a variety of ways to produce the many levels of access required by these categories.

Anything declared public can be accessed from anywhere, whereas anything declared private cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package, which is the default access.

An element is declared protected if you want to allow an element to be seen outside your current package but only to classes that subclass your class directly.

Table 6.1 is applicable only to members of classes. A class has only two possible access levels: default and public.

Let us consider an Example: to illustrate the above concepts:

```
packagepackageA;


public class Base {
public String publicStr = "publicString"
protected String protectedStr = "protectedString"
   String defaultStr = "defaultString"
private String privateStr = "privateString"
```

```
public void print() {
System.out.println("packageA.Base has access to")
System.out.println("    " + publicStr)
System.out.println("    " + protectedStr)
System.out.println("    " + defaultStr)
System.out.println("    " + privateStr)

    Base b = new Base(); // -- other Base instance
System.out.println("    b." + b.publicStr)
System.out.println("    b." + b.protectedStr)
System.out.println("    b." + b.defaultStr)
System.out.println("    b." + b.privateStr)
    }
}
---------------------------------------------------------------------------
packagepackageB
importpackageA.Base
public class SubB extends Base {
public void print() {
System.out.println("packageB.SubB has access to")
System.out.println("    " + publicStr + " (inherited from Base)")
      // -- protectedStr is inherited element -> accessible
System.out.println("    " + protectedStr + " (inherited from Base)")
      // -- not accessible
      // System.out.println(defaultStr)
      // System.out.println(privateStr)
```

```
        Base b = new Base(); // -- other Base instance
System.out.println("    b." + b.publicStr)
        // -- protected element, which belongs to other object -> not accessible
        // System.out.println(b.protectedStr)

        // -- not accessible
        // System.out.println(b.defaultStr)
        // System.out.println(b.privateStr)
    }
}
--------------------------------------------------------------------------------
import packageA.*
import packageB.*

// -- testing class
public class TestProtection {
public static void main(String[] args) {
        // -- all classes are public, so class TestProtection
        // -- has access to all of them
new Base().print()
newSubA().print()
newAnotherA().print()
newSubB().print()
newAnotherB().print()
    }
}
```

Types of Variables:

There are three kinds of variables in Java:

1. Local variables
2. Instance variables
3. Class/static variables

Local variables:

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

Instance variables:

- Instance variables are declared in a class but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object’s state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally it is recommended to make these variables private (access

level).However, visibility for subclasses can be given for these variables with the use of access modifiers.

- Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods and different class (when instance variables are given accessibility) they should be called using the fully qualified name. Object Reference. Variable Name.

Class/static variables:

- Class variables also known as static variables are declared with the static keyword in a class but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
- Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initialiser blocks.
- Static variables can be accessed by calling with the class name. Class Name. Variable Name.
- When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

Note: If the variables are accessed from an outside class, the constant should be accessed as Employee. Department

Check your progress 7

1. List the four categories of visibility for class members.
2. Write a note on instance variables?

....

....

....

....

....

....

....

....

....

....

....

2.9 Let Us Sum Up

When an object is created or, primitive type variable or method is called, the memory for that object, variable or method is set aside.

The different objects, variables and methods occupy different areas of memory when created/called. In some cases, we would like to have multiple objects, variables or methods which occupy the same area of memory (in effect just having the one instance of that variable or method). The above can be achieved by using the static keyword; it is possible to have static methods and variables.

In Java, global variables are not allowed. In order to do the same, the instance variable in the class can be declared static. The effect of doing this is that when we create multiple objects of that class, every object shares the same instance variable that was declared to be static.

Sometimes there are situations in which you will want to define a superclass, which declares the structure of a given abstraction without providing a complete implementation of every method. That is, many a times you'll 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. The abstract keyword can be used with: a) A class, b) A method

An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface. An interface is not a class. Writing an interface is similar to writing a class but they are two different concepts. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements. Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

There is also learning about an interface is similar to a class in the several ways: However, an interface is different from a class in several ways. Further we learned about Declaring Interfaces, in this the interface keyword is used to declare an interface. Here is a simple Example: to declare an interface. Next thing which we understood is encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Access to the data and code is tightly controlled by an interface.

In this second Unit we have also learned about a superclass which is defined by the abstract class Number that implements the classes that wrap the numeric type's byte, short, int, long, float and double. Number possesses abstract methods that return the value of the object in each of the different number formats. That is, doubleValue ( ) returns the value as a double, floatValue ( ) returns the value as a float and so on. Let us recollect DOUBLE AND FLOAT. Double and Float are wrappers for floating-point values of type double and float respectively. The constructors of float are Float (double num) ,Float (float num) and Float (String str) throws NumberFormatException. The Float objects can be constructed with values of type float or double. They can also be constructed from the string representation of a floating-point number. Whereas, the constructors for

Double are shown as Double (double num), Double (String str) throws Number Format Exception. Double objects can be constructed with a double value or a string containing a floating-point value.

Java provides an easy way to convert numbers into string. The Byte, Short, Integer and Long classes provide the parseByte( ), parseShort( ), parseInt( ) and parseLong( ) methods, respectively. These methods return the byte, short, int or long equivalent of the numeric string with which they are called. There is also learning related to Packages are used in Java in-order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier. Packages add another dimension to access control, they act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction, due to the interplay between classes and packages, Java addresses four categories of visibility for class members, which are 1) Subclasses in the same package. 2) Non-subclasses in the same package. 3) Subclasses in different packages. 4) Classes that are neither in the same package nor subclasses.

2.10 Suggested Answer for Check Your Progress

Check your progress 1

Answers: See Section 2.2

Check your progress 2

Answers: See Section 2.3

Check your progress 3

Answers: See Section 2.4

Check your progress 4

Answers: See Section 2.5

Check your progress 5

Answers: See Section 2.6

Check your progress 6

Answers: See Section 2.7

Check your progress 7

Answers: See Section 2.8

2.11 Glossary

1. **Interface -** An interface is a collection of abstract methods.
2. **Instance variables -** are declared variables in a class but outside a method, constructor or any block.
3. **Local variables -** Local variables are declared in methods, constructors, or blocks.

2.12 Assignment

Write a note on Java technology.

2.13 Activities

Write any two programs to show the use of Interfaces

2.14 Case Study

Explain Java programming environment with the help of diagram

2.15 Further Reading

1. Core Java 2, 2 volumes, Cay S. Horstmann, Gary Cornell, The Sun Microsystems Press, 1999, Indian reprint 2000.
2. Java 2, the Complete Reference, Patrick Naughton and Herbert Schildt, Tata McGraw Hill, 1999.
3. Programming with Java, Ed. 2, E. Balagurusamy, Tata McGraw Hill, 1998, reprint, 2000.
4. The Java Tutorial, Ed. 2, 2 volumes, Mary Campione and Kathy Walrath, Addison Wesley Longmans, 1998.
5. The Java Language Specification, Ed. 2, James Gosling, Bill Joy, Guy Steele & Gilad Bracha, (Ed. 1 1996, Ed. 2 2000), Sun Microsystems, 2000.
6. Using Java 2, Joseph L. Weber, Prentice Hall, Eastern Economy Edition, 2000