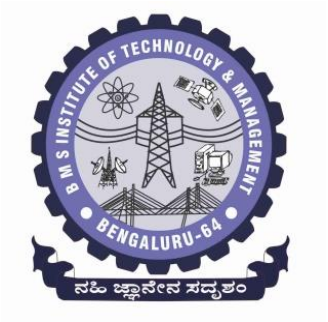
B M S INSTITUTE OF TECHNOLOGY & MANAGEMENT

YELAHANKA, BENGALURU – 560064.



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MODULEWISE NOTES OF

Advanced Java & J2EE -15CS553

[As per Choice Based Credit System (CBCS) scheme]

(Effective from the academic year 2017 -2018)

SEMESTER – V

Prepared by,

Mr. Muneshwara M S

Asst. Prof, Dept. of CSE

VISION AND MISSION OF THE CS&E DEPARTMENT

Vision

To develop technical professionals acquainted with recent trends and technologies of computer science to serve as valuable resource for the nation/society.

Mission:

Facilitating and exposing the students to various learning opportunities through dedicated academic teaching, guidance and monitoring.

VISION AND MISSION OF THE INSTITUTE

Vision

To emerge as one of the finest technical institutions of higher learning, to develop engineering professionals who are technically competent, ethical and environment friendly for betterment of the society.

Mission

Accomplish stimulating learning environment through high quality academic instruction, innovation and industry-institute interface

MODULE -1

Enumerations, Autoboxing, and Annotations(Metadata)

Enumerations

- Enumerations included in JDK 5. An enumeration is a list of named constants. It is similar to final variables.
- **Enum in java** is a data type that contains fixed set of constants.
- An enumeration defines a class type in Java. By making enumerations into classes, so it can have constructors, methods, and instance variables.
- An enumeration is created using the enum keyword.
  Ex:
  enum Apple { Jonathan, GoldenDel, RedDel, Winesap, Cortland }
- The identifiers Jonathan, GoldenDel, and so on, are called enumeration constants.
- Each is implicitly declared as a public, static final member of Apple.
- Enumeration variable can be created like other primitive variable. It does not use the new for creating object.

  Ex:Apple ap;

  Ap is of type Apple, the only values that it can be assigned (or can contain) are those defined by the enumeration. For example, this assigns:

  ap = Apple.RedDel;

  Example Code-1

```
enum Apple { Jonathan, GoldenDel, RedDel, Winesap, Cortland }
class EnumDemo
{
        public static void main(String args[])
        {
                Apple ap;
                ap = Apple.RedDel;
                System.out.println("Value of ap: " + ap);
              // Value of ap: RedDel ap = Apple.GoldenDel;

                if(ap == Apple.GoldenDel)

               System.out.println("ap contains GoldenDel.\n");
               // ap contains GoldenDel.
               switch(ap)
               {
                 case Jonathan: System.out.println("Jonathan is red."); break;
                 case GoldenDel: System.out.println("Golden Delicious is
                                        yellow.");// Golden Delicious is yellow
                                     break;
```

```
                    case RedDel:System.out.println("Red Delicious is red."); break;
                    case Winesap: System.out.println("Winesap is red."); break;
                    case Cortland: System.out.println("Cortland is red."); break;

                }

            }

    }
```

The values( ) and valueOf( ) Methods

All enumerations automatically contain two predefined methods: **values( ) and valueOf( ).**

Their general forms are shown here:

public static enum-type[ ] values( )
public static enum-type valueOf(String str)

The **values( )** method returns an array that contains a list of the enumeration constants.
The **valueOf( )** method returns the enumeration constant whose value corresponds to the string passed in str.

Example Code-2:

```
enum Season { WINTER, SPRING, SUMMER, FALL }
 class EnumExample1
{
        public static void main(String[] args)
         {
            for (Season s : Season.values())
                System.out.println(s);
           Season s = Season.valueOf("WINTER");
          System.out.println("S contains " + s);
        }
}
```

Example Code-3

```
class EnumExample5
{
  enum Day
  {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDA Y,
   SATURDAY
  }

public static void main(String args[])
{
     Day day=Day.MONDAY;
```

```
        switch(day)
        {
                case SUNDAY:  System.out.println("sunday"); break;
                case MONDAY: System.out.println("monday"); break;
               default: System.out.println("other day");
        }
    }
}
```

Class Type Enumeration

```
enum Apple
{
    Jonathan(10),    GoldenDel(9),    RedDel(12),    Winesap(15),
    Cortland(8);
    private int price;
    Apple(int p)
    { price = p; }
    int getPrice()
     { return price; }

}

class EnumDemo3
 {
    public static void main(String args[])
    {
            Apple ap;
        System.out.println("Winesap costs " + Apple.Winesap.getPrice() + " cents.\n");
        System.out.println("All apple prices:");
    for(Apple a : Apple.values())
            System.out.println(a + " costs " + a.getPrice() + " cents.");

    }

}
```

The Class type enumeration contains three things

The first is the instance variable price, which is used to hold the price of each variety of apple.

The second is the Apple constructor, which is passed the price of an apple.

The third is the method getPrice( ), which returns the value of price.

When the variable ap is declared in main( ), the constructor for Apple is called once for each constant that is specified. the arguments to the constructor are specified, by putting them inside parentheses after each constant, as shown here:

Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

These values are passed to the parameter of Apple(),which then assigns this value to price. The constructor is called once for each constant.

Because each enumeration constant has its own copy of price, you can obtain the price of a specified type of apple by calling getPrice().

For example, in main() the price of a Winesap is obtained by the following call:Apple.Winesap.getPrice()

Enum Super class

All enumerations automatically inherit one: java.lang.Enum.

Enum class defines several methods that are available for use by all enumerations.

ordinal( )

To obtain a value that indicates an enumeration constant's position in the list of constants. This is called its ordinal value, and it is retrieved by calling the ordinal( ) method, shown here:

final int ordinal( )

It returns the ordinal value of the invoking constant. Ordinal values begin at zero. Thus, in the Apple enumeration, Jonathan has an ordinal value of zero, GoldenDel has an ordinal value of 1, RedDel has an ordinal value of 2, and so on.

compareTo( )

To compare the ordinal value of two constants of the same enumeration by using the compareTo( ) method. It has this general form:

final int compareTo(enum-type e)

equals()

equals method is overridden method from Object class, it is used to compare the enumeration constant. Which returns true if both constants are same.

Program to demonstrate the use of ordinal(), compareTo(), equals()

```
enum Apple { Jonathan, GoldenDel, RedDel, Winesap, Cortland }

class EnumDemo4
{
public static void main(String args[])
{
Apple ap, ap2, ap3;
System.out.println("Here are all apple constants" + " and their ordinal values: ");
 for(Apple a : Apple.values())
            System.out.println(a + " " + a.ordinal());
 ap = Apple.RedDel;
ap2 = Apple.GoldenDel;
ap3 = Apple.RedDel;
System.out.println();
```

```
if(ap.compareTo(ap2) < 0)
    System.out.println(ap + " comes before " + ap2);

if(ap.compareTo(ap2) > 0)
    System.out.println(ap2 + " comes before " + ap);

if(ap.compareTo(ap3) == 0)
    System.out.println(ap + " equals " + ap3);

System.out.println();

if(ap.equals(ap2))
    System.out.println("Error!");

if(ap.equals(ap3))
     System.out.println(ap + " equals " + ap3);

if(ap == ap3)
    System.out.println(ap + " == " + ap3);
}
}
```

Wrappers Classes

Java uses primitive types such as int or double, to hold the basic data types supported by the language.

The primitive types are not part of the object hierarchy, and they do not inherit Object.

Despite the performance benefit offered by the primitive types, there are times when you will need an object representation.

Many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types.

To handle the above situation, Java provides type wrappers, which are classes that encapsulate a primitive type within an object.

The type wrappers are

Double, Float, Long, Integer, Short, Byte, Character, and Boolean. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

Character:

Character is a 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 here:

char charValue( )

Boolean:

Boolean is a wrapper around boolean values. It defines these constructors:

Boolean(boolean boolValue)
Boolean(String boolString)

In the first version, boolValue must be either true or false.

In the second version, if boolString contains the string "true" (in uppercase or lowercase), then the new Boolean object will be true. Otherwise, it will be false.

To obtain a boolean value from a Boolean object, use booleanValue( ), shown here:
boolean booleanValue( )

It returns the boolean equivalent of the invoking object.

Integer Wrapper class example code:

Integer(int num)
Integer(String str)

```
class Wrap
{
 public static void main(String args[])
 {
   Integer iOb = new Integer(100);
   int i = iOb.intValue();
   System.out.println(i + " " + iOb);
 }
}
```

This program wraps the integer value100 inside an Integer object called iOb.

The program then obtains this value by calling intValue() and stores the result in i. The process of encapsulating a value within an object is called boxing.

Thus, in the program, this line boxes the value 100 into an Integer: Integer iOb = new Integer(100);

The process of extracting a value from a type wrapper is called unboxing.
The program unboxes the value in iOb with this statement:

int i = iOb.intValue();

AutoBoxing

Auto boxing is the process by which a primitive type is automatically encapsulated(boxed) into its equivalent type wrapper

whenever an object of that type is needed. There is no need to explicitly construct an object.

Integer iOb = 100; // autobox an int

Auto-unboxing

Auto- unboxing is the process by which the value of a boxed object is automatically extracted from a type wrapper when it is assigned to primitive type value is needed.

There is no need to call a method such as intValue( ). **int i = iOb; // auto-unbox**

Example Program:

```
class AutoBoxUnBox
{
 public static void main(String args[])
 {
 Integer iOb = 100; // autobox an int
int i = iOb; // auto-unbox
System.out.println(i + " " + iOb); // displays 100 100
}
}
```

Explain auto boxing and auto unboxing during method call

```
class AutoBox2
{
static int m(Integer v)
{
 return v ; }
public static void main(String args[])
{
Integer iOb = m(100);
System.out.println(iOb);// 100
}
}
```

In the program, notice that m( ) specifies an Integer parameter and returns an int result.

Inside main( ), m( ) is passed the value 100.

Because m( ) is expecting an Integer, this value is automatically boxed.

Then, m( ) returns the int equivalent of its argument. This causes v to be auto-unboxed.

Next, this int value is assigned to iOb in main( ), which causes the int return value to be autoboxed.

Explain auto boxing and unboxing during expression evaluation

Autoboxing/Unboxing Occurs in Expressions autoboxing and unboxing take place whenever a conversion into an object or from an object is required.

This applies to expressions. Within an expression, a numeric object is automatically unboxed.

The outcome of the expression is reboxed, if necessary. For example, consider the following program:

```
class AutoBox3
{
 public static void main(String args[])
{
Integer iOb, iOb2; int i;
iOb = 100;
System.out.println("Original value of iOb: " + iOb);
 ++iOb; // auto unbox and rebox
System.out.println("After ++iOb: " + iOb);
iOb2 = iOb + (iOb / 3);
 System.out.println("iOb2 after expression: " + iOb2);
i = iOb + (iOb / 3); // auto unbox and rebox
System.out.println("i after expression: " + i);
}
}
```

++iOb;

This causes the value in iOb to be incremented. It works like this: iOb is unboxed, the value is incremented, and the result is reboxed.

Auto-unboxing also allows you to mix different types of numeric objects in an expression. Once the values are unboxed,the standard type promotions and conversions are applied.For example, the following program is perfectly valid:

```
Integer iOb = 100; Double dOb = 98.6;
dOb = dOb + iOb; // type promoted to double
System.out.println("dOb after expression: " + dOb);
Integer iOb = 2;
switch(iOb)
{
    Case1: System.out.println("one"); break;
    case 2: System.out.println("two"); break;
    default: System.out.println("error");
}
```

When the switch expression is evaluated, iOb is unboxed and its int value is obtained.

As the examples in the program show, because of autoboxing/unboxing, using numeric objects in an expression is both intuitive and easy.

Autoboxing/unboxing a Boolean and Character.

```
class AutoBox5
 {
     public static void main(String args[])
     {
            Boolean b = true; // auto boxing boolean
            if(b)
                  System.out.println("b is true");// auto unboxed when used in
            conditional expression Character ch = 'x'; // box a char
            char ch2 = ch; // unbox a char
            System.out.println("ch2 is " + ch2);

     }
     }
```

The output is shown here:
b is true ch2 is x

Annotations

Annotations (Metadata) Beginning with JDK 5, a new facility was added to Java that enables you to embed supplemental information into a source file.

This information, called an annotation, does not change the actions of a program. Thus, an annotation leaves the semantics of a program unchanged.

However this information can be used by various tools during both development and deployment.

For example, an annotation might be processed by a source-code generator. The term metadata is also used to refer to this feature, but the term annotation is the most descriptive and more commonly used.

An annotation is created through a mechanism based on the interface. Let's begin with an example.

Here is the declaration for an annotation called MyAnno:

```
@interface MyAnno
{
        String str();
         int val();
}
```

@ that precedes the keyword interface.
This tells the compiler that an annotation type is being declared.

Next, notice the **two members str( ) and val( ).**

All annotations consist solely of method declarations. However, you don't provide bodies for these methods. Instead, Java implements these methods. Moreover, the methods act much like fields.

An annotation cannot include an extends clause.

```
MyAnno(str = "Annotation Example", val = 100)
public static void myMeth()
{ // ...
```

Notice that no parentheses follow str in this assignment.

What is retention policy ? Explain the use of retention tag.

- A retention policy determines at what point an annotation is discarded.
- Java defines three such policies, which are encapsulated within the java.lang.annotation.
- RetentionPolicy enumeration.

They are SOURCE, CLASS, and RUNTIME.

- An annotation with a retention policy of SOURCE is retained only in the source file and is discarded during compilation.
- An annotation with a retention policy of CLASS is stored in the .class file during compilation. However, it is not available through the JVM during run time.
- An annotation with a retention policy of RUNTIME is stored in the .class file during compilation and is available through the JVM during run time.

A retention policy is specified for an annotation by using one of Java's built-in annotations: @Retention.

@Retention(retention-policy)

Here, retention-policy must be one of the previously discussed enumeration constants.
If no retention policy is specified for an annotation, then the default policy of CLASS is used.
The following version of MyAnno uses @Retention to specify the

RUNTIME retention policy.

Thus, MyAnno will be available to the JVM during program execution.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno
{
        String str();
        int val();
}
```

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// An annotation type declaration.
 @Retention(RetentionPolicy.RUNTIME)
@interface MyAnno { String str(); int val(); }

class Meta
{
        // Annotate a method.
        @MyAnno(str = "Annotation Example", val = 100)

     public static void myMeth()
     {
        Meta ob = new Meta();
         try {
                // First, get a Class object that represents // this class.
                Class c = ob.getClass();

               // Now, get a Method object that represents // this method.
                  Method m = c.getMethod("myMeth");
                // Next, get the annotation for this class.
                MyAnno anno = m.getAnnotation(MyAnno.class);
                System.out.println(anno.str() + " " + anno.val());
            }
         catch (NoSuchMethodException exc)
        {
                System.out.println("Method Not Found."); }
        }

     public static void main(String args[])
        {
                 myMeth();
        }

}
```

The output from the program is shown here:

Annotation Example 100

This program uses reflection as described to obtain and display the values of str and val in the MyAnnoannotation associated with myMeth()in the Metaclass.

MyAnno anno = m.getAnnotation(MyAnno.class);

notice the expression MyAnno.class. This expression evaluates to a Class object of type MyAnno, the annotation.

This construct is called a class literal. You can use this type of expression whenever a Class object of a known class is needed.

However, to obtain a method that has parameters, you must specify class objects representing the types of those parameters as arguments to getMethod( ). For example, here is a slightly different version of the preceding program:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno { String str(); int val(); }

class Meta
 {
    // myMeth now has two arguments.
   @MyAnno(str = "Two Parameters", val = 19)
    public static void myMeth(String str, int i)
    {
            Meta ob = new Meta();
            Try
             {
                    Class c = ob.getClass();
                // Here, the parameter types are specified.
               Method m = c.getMethod("myMeth", String.class, int.class);
                MyAnno anno = m.getAnnotation(MyAnno.class);
                 System.out.println(anno.str() + " " + anno.val());
            }
            catch (NoSuchMethodException exc)
            {
                    System.out.println("Method Not Found.");
             }
  }

public static void main(String args[])
 {
      myMeth("test", 10); }

}
```

The output from this version is shown here:

Two Parameters 19

myMeth( ) takes a String and an int parameter.

To obtain information about this method, getMethod( ) must be called as shown here:

Method m = c.getMethod("myMeth", String.class, int.class);

Here, the Class objects representing String and int are passed as additional arguments.
Obtaining All Annotations

You can obtain all annotations that have RUNTIME retention that are associated with an item by calling getAnnotations( ) on that item.

It has this general form:

Annotation[ ] getAnnotations( )

It returns an array of the annotations.

getAnnotations( )

It can be called on objects of type Class, Method, Constructor, and Field. Here is another reflection example that shows how to obtain all annotations associated with a class and with a method.

It declares two annotations.It then uses those annotations to annotate a class and a method.

Example code:

```
import java.lang.annotation.*; import java.lang.reflect.*;
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno
{
     String str(); int val();
}
@Retention(RetentionPolicy.RUNTIME)
@interface What
 {
    String description();
}
@What(description = "An annotation test class")
 @MyAnno(str = "Meta2", val = 99)
  class Meta2
 {
    @What(description = "An annotation test method")
    @MyAnno(str = "Testing", val = 100)
    public static void myMeth() { Meta2 ob = new Meta2();
    try
    {
            Annotation annos[] = ob.getClass().getAnnotations();
// Display all annotations for Meta2.
// System.out.println("All annotations for Meta2:");
// for(Annotation a : annos)
// System.out.println(a);
// System.out.println();
// Display all annotations for myMeth.

 Method m = ob.getClass( ).getMethod("myMeth");
 annos = m.getAnnotations();
System.out.println("All annotations for myMeth:");
 for(Annotation a : annos)
    System.out.println(a);
```

```
} catch (NoSuchMethodException exc) { System.out.println("Method Not Found."); }

}

public static void main(String args[]) { myMeth(); }

}
```

The output is shown here:

```
All annotations for Meta2:

@What(description=An annotation test class)

@MyAnno(str=Meta2, val=99)

All annotations for myMeth:

@What(description=An annotation test method)

@MyAnno(str=Testing, val=100)
```

The program uses getAnnotations( ) to obtain an array of all annotations associated with the Meta2 class and with the myMeth( ) method. As explained, getAnnotations( ) returns an array of Annotation objects.

Recall that Annotation is a super-interface of all annotation interfaces and that it overrides toString( ) in Object.

Thus, when a reference to an Annotation is output, its toString( ) method is called to generate a string that describes the annotation, as the preceding output shows.

The AnnotatedElement Interface

The methods declared in AnnotatedElement Interface

1. getAnnotation( ) --- It can be invoked with method, class. It return the used annotation.

2. getAnnotations( ) --- It can be invoked with method, class. It return the used annotations.

3. getDeclaredAnnotations( ) -- It returns all non-inherited annotations present in the invoking object.

4. isAnnotationPresent( ), which has this general form:

   It returns true if the annotation specified by annoType is associated with the invoking object. It returns false otherwise.

Default Values in annotation

You can give annotation members default values that will be used if no value is specified when the annotation is applied.

A default value is specified by adding a default clause to a member's declaration. It has this general form:

type member( ) default value;

// An annotation type declaration that includes defaults.

@interface MyAnno { String str() default "Testing"; int val() default 9000; }

@MyAnno() // both str and val default

@MyAnno(str = "some string") // val defaults

@MyAnno(val = 100) // str defaults

@MyAnno(str = "Testing", val = 100) // no defaults

Example:

```
import java.lang.annotation.*;
import java.lang.reflect.*;
 @Retention(RetentionPolicy.RUNTIME)
```

```
@interface MyAnno { String str() default "Testing"; int val() default 9000;
} class Meta3 {

 @MyAnno()

public static void myMeth() { Meta3 ob = new
Meta3(); try { Class c = ob.getClass();

Method m = c.getMethod("myMeth");

MyAnno anno =
m.getAnnotation(MyAnno.class);
System.out.println(anno.str() + " " + anno.val()); }
catch (NoSuchMethodException exc)
 {

System.out.println("Method Not Found."); }

}

public static void main(String args[]) { myMeth(); }

}
```

Output:

Testing 9000

Marker Annotations

A marker annotation is a special kind of annotation that contains no members.

Its sole purpose is to mark a declaration. Thus, its presence as an annotation is sufficient.

The best way to determine if a marker annotation is present is to use the method isAnnotationPresent( ), which is a defined by the AnnotatedElement interface.

Here is an example that uses a marker annotation.

Because a marker interface contains no members, simply determining whether it is present or absent is sufficient.

```
import java.lang.annotation.*;
import java.lang.reflect.*;
```

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }
class Marker {
@MyMarker
public static void myMeth() { Marker ob = new Marker();
try { Method m = ob.getClass().getMethod("myMeth");
if(m.isAnnotationPresent(MyMarker.class))
System.out.println("MyMarker is present.");
} catch (NoSuchMethodException exc)
{ System.out.println("Method Not Found."); }
}
public static void main(String args[]) { myMeth(); }
}
```

Output

```
MyMarker is present.
public static void main(String args[]) { myMeth(); }
}
```

Built in Annotations

Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Built-In Java Annotations used in java code

- @Override
- @SuppressWarnings
- @Deprecated

Built-In Java Annotations used in other annotations

- @Target
- @Retention
- @Inherited
- @Documented

@Override

@Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.Sometimes, we does the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden.

Example:

```
class Animal{

void eatSomething()
{System.out.println("eating something");}

}

class Dog extends
Animal{ @Override

void eatsomething()
{

System.out.println("eating foods");
}//Compile time error }
```

@SuppressWarnings

annotation: is used to suppress warnings issued by the compiler.

If you remove the @SuppressWarnings("unchecked") annotation, it will show warning at compile time because we are using non-generic collection.

```
import java.util.*;

class TestAnnotation2{

@SuppressWarnings("unchecked")
```

```
public static void main(String args[]){
ArrayList list=new ArrayList();
list.add("sonoo");
}}
```

@Deprecated

@Deprecated annoation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions.

So, it is better not to use such methods.

```
class A{
void m(){System.out.println("hello m");}
@Deprecated
void n(){System.out.println("hello n");}
}
 class TestAnnotation3{
    public static void main(String args[]){
    A a=new A();
    a.n();
    }}
```

Error message: Test.java uses or overrides a deprecated API.

@Inherited

is a marker annotation that can be used only on another annotation declaration. Furthermore, it affects only annotations that will be used on class declarations. @Inherited causes the annotation for a superclass to be inherited by a subclass.

```
@Inherited
public @interface MyCustomAnnotation {
}

@MyCustomAnnotation
public class
MyParentClass {

  ...
}
```

```
public class MyChildClass extends MyParentClass {

  ...
}
```

Here the class MyParentClass is using annotation @MyCustomAnnotation which is marked with @inherited annotation. It means the sub class MyChildClass inherits the @MyCustomAnnotation.

@Documented

@Documented annotation indicates that elements using this annotation should be documented by JavaDoc.

```
@Documented

public @interface MyCustomAnnotation {
  //Annotation body
}

@MyCustomAnnotatio
n public class MyClass
{
    //Class body
}
```

While generating the javadoc for class MyClass, the annotation @MyCustomAnnotation would be included in that

@Target

It specifies where we can use the annotation.

For example: In the below code, we have defined the target type as METHOD which means the below annotation can only be used on methods.

```
import
java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target({ElementType.METHOD})
public @interface MyCustomAnnotation {

}

public class MyClass {
  @MyCustomAnnotatio
  n public void
  myMethod()
  {

     //Doing something
  }
}
```

If you do not define any Target type that means annotation can be applied to any element.

@Retention is mention in earlier topic.

Module 2-The collections and Framework

1. Explain brief about collection frame work.

- The Java Collections Framework standardizes the way in which groups of

  objects are handled by your programs.

  The framework had to be high-performance.

  The implementations for the fundamental collections(dynamic arrays, linked lists, trees, and hash tables) are highly efficient.

  The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.

  Extending and/or adapting a collection had to be easy.

  Mechanisms were added that allow the integration of standard arrays into the Collections Framework.

  Algorithms are another important part of the collection mechanism.

  Algorithms operate on collections and are defined as static methods within the Collections class.

  An iterator offers a general-purpose, standardized way of accessing the elements within a collection, one at a time. Thus, an iterator provides a means of enumerating the contents of a collection.

  Because each collection implements Iterator, the elements of any collection class can be accessed through the methods defined by Iterator.

\} **What are the recent changes to collection framework?**

Recently, the Collections Framework underwent a fundamental change that significantly increased its power and streamlined its use. The changes were the addition of generics, autoboxing/unboxing, and the for-each style for loop.

Generics

The addition of generics caused a significant change to the Collections Framework because the entire Collections Framework has been reengineered for it. All collections are now generic, and many of the methods that operate on collections take generic type

parameters Generics add the one feature that collections had been missing: type safety.

Prior to generics, all collections stored Object references, which meant that any collection could store any type of object. Thus, it was possible to accidentally store in compatible type s in a collection.

Doing so could result in run-time type mismatch errors. With generics, it is possible to explicitly state the type of data being stored, and run-time type mismatch errors can be avoided.

Autoboxing/unboxing

Autoboxing facilitates the Use of Primitive Types.

Autoboxing/unboxing facilitates the storing of primitive types in collections.

As you will see, a collection can store only references, not primitive values. In the past, if you wanted to store a primitive value, such as an int, in a collection, you had to manually box it into its type wrapper.

When the value was retrieved, it needed to be manually unboxed (by using an explicit cast) into its proper primitive type.

Because of autoboxing/unboxing, Java can automatically perform the proper boxing and unboxing needed when storing or retrieving primitive types. There is no need to manually perform these operations.

The For-Each Style for Loop

collection can be cycled through by use of the for-each style for loop.

Earlier it was done with Iteratable interface. For each loop is easier than the earlier iterator.

3. **List the Collection Interfaces?**

- The Collections Framework defines several interfaces. This section provides an overview of each interface. Collection enables you to work with groups of objects; it is at the top of the collections hierarchy.

- Deque extends Queue to handle a double-ended queue.

- List extends Collection to handle sequences
- NavigableSet extends SortedSet to handle retrieval of elements based on closest-match searches.
- Queue extends Collection to handle special types of lists in which elements are removed only from the head.
- Set extends Collection to handle sets, which must contain unique elements.
- SortedSet extends Set to handle sorted sets.

4. Give the syntax of collection interface. Explain the methods present in collection interface.

interface Collection<E>

E specifies the type of objects that the collection
Collection extends the Iterable interface.

Iterating through the list cane be done through the iteratable interface.
Methods in collection interface

add

boolean add(E obj)

adds obj to the invoking collection.

Returns true if obj was added to the collection.

Returns false if obj is already a member of the collection and the collection does not allow duplicates.

addAll

boolean addAll(Collection<? extends E> c)

Adds all the elements of c to the invoking collection.

Returns true if the operation succeeded

(i.e., the elements were added). Otherwise, returns false.

clear

void clear( )

Removes all elements from the invoking collection.

contains

boolean contains(Object obj)

Returns true if obj is an element of the invoking collection.

Otherwise, returns false.

containsAll

boolean containsAll(Collection<?> c)

Returns true if the invoking collection contains all elements of c.

Otherwise, returns false.

equals

boolean equals(Object obj)

Returns true if the invoking collection and obj are equal.

Otherwise, returns false.

hashCode

int hashCode( ) Returns the hash code for the invoking collection.

isEmpty

boolean isEmpty( )

Returns true if the invoking collection is empty.

Otherwise, returns false.

iterator

Iterator<E> iterator( ) Returns an iterator for the invoking collection.

remove

boolean remove(Object obj)

Removes one instance of obj from the invoking collection.

Returns true if the element was removed. Otherwise, returns false.

removeAll

boolean removeAll(Collection<?> c)

Removes all elements of c from the invoking collection.

Returns true if the collection changed (i.e., elements were removed).

Otherwise, returns false.

retainAll

boolean retainAll(Collection<?> c)

Removes all elements from the invoking collection except those in c.

Returns true if the collection changed (i.e., elements were removed).

Otherwise, returns false.

size

int size( ) Returns the number of elements held in the invoking collection.

toArray

Object[ ] toArray( )

Returns an array that contains all the elements stored in the invoking collection.

The array elements are copies of the collection elements.

The array elements are copies of the collection elements.

If the size of array equals the number of elements, these are returned in array.

5. Explain the methods present in List interface.

List interface extends collection interface. It includes new method. Which are given below.

void add(int index , E obj)

Inserts obj into the invoking list at the index passed in index.

Any pre existing elements at or beyond the point of insertion are shifted up.

boolean addAll(int index , Collection<? extends E> c)

Inserts all elements of C into the invoking list at the index passed in

index . Any pre existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.

E get(int index)

Returns the object stored at the specified index within the invoking collection.

int indexOf(Object obj)

Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, –1 is returned.

int lastIndexOf(Object obj)

Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, –1 is returned.

ListIterator<E> listIterator( )

Returns an iterator to the start of the invoking list.

ListIterator<E> listIterator(int index)

Returns an iterator to the invoking list that begins at the specified index.

E remove(int index)

Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.

E set(int index , E obj)

Assigns obj to the location specified by index within the invoking list.

List<E> subList(int start , int end)

Returns a list that includes elements from start to end –1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

6. Explain Set Interface and set method:

The Set interface defines a set.

It extends Collection and declares the behaviour of a collection that does not allow duplicate elements.

Therefore, the add() method returns false if an attempt.

Set is a generic interface that has this declaration:

interface Set<E>

Here, E specifies the type of objects that the set will hold.

The SortedSet Interface

The SortedSet interface extends Set and declares the behavior of a set sorted in ascending order.

SortedSet is a generic interface that has this declaration: interface SortedSet<E> Here, E specifies the type of objects that the set will hold.

In addition to those methods defined by Set, the SortedSet interface declares the methods.

Comparator<? super E> comparator( )

Returns the invoking sorted set's comparator.
If the natural ordering is used for this set, null is returned.

E first( )

Returns the first element in the invoking sorted set.

SortedSet<E> headSet(E end)

Returns a SortedSet containing those elements less than end that are contained in the invoking sorted set.

Elements in the returned sorted set are also referenced by the invoking sorted set.

E last( )

Returns the last element in the invoking sorted set.

SortedSet<E> subSet(E start , E end)

Returns a SortedSet that includes those elements between start and end– 1. Elements in the returned collection are also referenced by the invoking object.

SortedSet<E> tailSet(E start)

Returns a SortedSet that contains those elements greater than or equal to start that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

Several methods throw a NoSuchElementException when no items are contained in the invoking set.

A ClassCastException is thrown when an object is in compatible with the elements in a set.

A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the set.

An IllegalArgumentException is thrown if an invalid argument is used.

7. NavigableSet Interface and method

The NavigableSet interface extends SortedSet and declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values.

NavigableSet is a generic interface that has this declaration:

interface NavigableSet<E>

Here, E specifies the type of objects that the set will hold.

NavigableSet adds the following

E ceiling(E obj)

Searches the set for the smallest element

If such an element is found, it is returned. Otherwise, null is returned. Iterator<E> descendingIterator( )

Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator.

NavigableSet<E> descendingSet( )

Returns a NavigableSet that is the reverse of the invoking set. The resulting set is backed by the invoking set.

E floor(E obj)

Searches the set for the largest element e such that e <= obj . If such an element is found, it is returned. Otherwise, null is returned.

NavigableSet<E> headSet(E upperBound , boolean incl)

Returns a NavigableSet that includes all elements from the invoking set that are less than upperBound . If incl is true, then an element equal to upperBound is included. The resulting set is backed by the invoking set.

E higher(E obj) Searches the set for the largest element e such that e > obj . If such an element is found, it is returned. Otherwise, null is returned.

E lower(E obj)

Searches the set for the largest element e such that e < obj . If such an element is found, it is returned. Otherwise, null is returned.

E pollFirst( )

Returns the first element, removing the element in the process. Because the set is sorted, this is the element with the least value. null is returned if the set is empty.

E pollLast( )

Returns the last element, removing the element in the process. Because the set is sorted, this is the element with the greatest value. null is returned if the set is empty.

NavigableSet<E> subSet(E lowerBound , boolean lowIncl , E upperBound , boolean highIncl)

Returns a NavigableSet that includes all elements from the invoking set that are greater than lowerBound and less than upperBound .

If lowIncl is true, then an element equal to lowerBound is included.

If highIncl is true, then an element equal to upperBound is included.

The resulting set is backed by the invoking set.

NavigableSet<E> tailSet(E lowerBound , boolean incl)

Returns a NavigableSet that includes all elements from the invoking set that are greater than lowerBound . If incl is true, then an element equal to lowerBound is included. The resulting set is backed by the invoking set

8. The Queue Interface and methods

The Queue interface extends Collection and declares the behaviour of a queue, which is often a first-in, first-out list. However, there are types of queues in which the ordering is based upon other criteria. Queue is a generic interface that has this declaration:

interface Queue<E>

E element( )

Returns the element at the head of the queue. The element is not removed. It throws NoSuchElementException if the queue is empty.

boolean offer(E obj)

Attempts to add obj to the queue. Returns true if obj was added and false otherwise.

E peek( )

Returns the element at the head of the queue. It returns null if the queue is empty. The element is not removed.

E poll( )

Returns the element at the head of the queue, removing the element in the process. It returns null if the queue is empty.

E remove( )

Removes the element at the head of the queue, returning the element in the process. It throws NoSuchElementException if the queue is empty.

9. <u>Deque interface</u>

It extends Queue and declares the behavior of a double-ended queue.

Double-ended queues can function as standard, first-in, first-out queues or as last-in, first-out stacks.

Deque is a generic interface that has this declaration:

interface Deque<E>

void addFirst(E obj)

Adds obj to the head of the deque. Throws an IllegalStateException if a capacity-restricted deque is out of space.

void addLast(E obj)

Adds obj to the tail of the deque. Throws an IllegalStateException if a capacity-restricted deque is out of space.

Iterator<E> descendingIterator( )

Returns an iterator that moves from the tail to the head of the deque. In other words, it returns a reverse iterator.

E getFirst( )

Returns the first element in the deque. The object is not removed from the deque. It throws NoSuchElementException if the deque is empty.

E getLast( )

Returns the last element in the deque.

The object is not removed from the deque. It throws NoSuchElementException if the deque is empty.

boolean offerFirst(E obj)

Attempts to add obj to the head of the deque. Returns true if obj was added and false otherwise. Therefore, this method returns false when an attempt is made to add obj to a full, capacity-restricted deque.

boolean offerLast(E obj)

Attempts to add obj to the tail of the deque. Returns true if obj was added and false otherwise.

E peekFirst( )

Returns the element at the head of the deque. It returns null if the deque is empty. The object is not removed.

E peekLast( )

Returns the element at the tail of the deque. It returns null if the deque is empty. The object is not removed.

E pollFirst( )

Returns the element at the head of the deque, removing the element in the process. It returns null if the deque is empty.

E pollLast( ) Returns the element at the tail of the deque, removing the element in the process. It returns null if the deque is empty.

E pop( ) Returns the element at the head of the deque, removing it in the process. It throws NoSuchElementException if the deque is empty.

void push(E obj) Adds obj to the head of the deque. Throws an IllegalStateException if a capacity-restricted deque is out of space.

E removeFirst( ) Returns the element at the head of the deque, removing the element in the process. It throws NoSuchElementException if the deque is empty.

boolean removeFirstOccurrence(Object obj)

Removes the first occurrence of obj from the deque. Returns true if successful and false if the deque did not contain obj .

E removeLast( )

Returns the element at the tail of the deque, removing the element in the process. It throws NoSuchElementException if the deque is empty.

boolean removeLastOccurrence(Object obj)

Removes the last occurrence of obj from the deque. Returns true if successful and false if the deque did not contain

10. <u>The Collection Classes with example code</u>

AbstractCollection

Implements most of the Collection interface.

AbstractList

Extends AbstractCollection and implements most of the List interface. Queue interface.

AbstractSequentialList

Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.

AbstractSet

Extends AbstractCollection and implements most of the Set interface.
EnumSet

Extends AbstractSet for use with enum elements.
HashSet

Extends AbstractSet for use with a hash table.
LinkedHashSet

Extends HashSet to allow insertion-order iterations.
PriorityQueue

Extends AbstractQueue to support a priority-based queue. TreeSet Implements a set stored in a tree. Extends AbstractSet. LinkedList Implements a linked list by extending AbstractSequentialList. ArrayList Implements a dynamic array by extending AbstractList. ArrayDeque Implements a dynamic double-ended queue by extending AbstractCollection and implementing the Deque interface.

ArrayList

ArrayList class extends **AbstractList** and implements the **List** interface

ArrayList is a generic class that has this declaration:

class ArrayList<E>

ArrayList has the constructors shown here:
ArrayList( )

constructor builds an empty array list
ArrayList(Collection<? extends E> *c*)

builds an array list that is initialized with the elements of the collection *c*.

ArrayList(int *capacity*)

builds an array list that has the specified initial *capacity*. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

```
class ArrayListDemo {

public static void main(String args[]) {
ArrayList<String> al = new ArrayList<String>();
System.out.println("Initial size of al: " + al.size());
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");

al.add("F");
al.add(1, "A2");

System.out.println("Size of al after additions: " +
al.size());

System.out.println("Contents of al: " + al);
al.remove("F");

al.remove(2);

System.out.println("Size of al after deletions: " +
al.size());

}}
```

Converting ArrayList to Array

```
class ArrayListToArray {

public static void main(String args[]) {
ArrayList<Integer> al = new ArrayList<Integer>();
al.add(1);
al.add(2);
al.add(3);
al.add(4);

System.out.println("Contents of al: " + al);
Integer ia[] = new Integer[al.size()];
```

```
        ia = al.toArray(ia); int
        sum = 0;

        for(int i : ia) sum += i;
        System.out.println("Sum is: " + sum);

        }
}
```

LinkedList

The **LinkedList** class extends **AbstractSequentialList** and implements the **List**, **Deque**, and

Queue interfaces.

It provides a linked-list data structure. **LinkedList** is a generic class that has this declaration:

```
class LinkedList<E>
```

Here, **E** specifies the type of objects that the list will hold. **LinkedList** has the two constructors

LinkedList( )
LinkedList(Collection<? extends E> *c*)
The first constructor builds an empty linked list.

The second constructor builds a linked list that is initialized with the elements of the collection *c*.

Example code:

```
import java.util.*;

class LinkedListDemo {




public static void main(String args[]) {
LinkedList<String> ll = new LinkedList<String>();
ll.add("F");

ll.add("B");
ll.add("D");
ll.add("E");
ll.add("C");
ll.addLast("Z");
```

```
ll.addFirst("A");
ll.add(1, "A2");

System.out.println("Original contents of ll: " + ll);
ll.remove("F");

ll.remove(2);

System.out.println("Contents of ll after deletion: "+ ll);
ll.removeFirst();

ll.removeLast();

System.out.println("ll after deleting first and last: "+ ll);
String val = ll.get(2);

ll.set(2, val + " Changed");
System.out.println("ll after change: " + ll);

}
}
```

HashSet

HashSet extends **AbstractSet** and implements the **Set** interface. It creates a collection that uses a hash table for storage.

Java HashSet class is used to create a collection that uses a hash table for storage.

It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.

HashSet is a generic class that has this declaration:
class HashSet<E>

Here, **E** specifies the type of objects that the set will hold.

Constructor
HashSet( )

HashSet(Collection<? extends E> *c*)

HashSet(int *capacity*)

HashSet(int *capacity*, float *fillRatio*)

Example: import
java.util.*;
class HashSetDemo {

public static void main(String args[]) {
HashSet<String> hs = new HashSet<String>();

hs.add("B");
hs.add("A");
hs.add("D");
hs.add("E");
hs.add("C");
hs.add("F");
System.out.println(hs);
}
}
output

[D, A, F, C, B, E]
LinkedHashSet

LinkedHashSet class is a Hash table and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

- Contains unique elements only like HashSet.
- Provides all optional set operations, and permits null elements.
- Maintains insertion order.

The **LinkedHashSet** class extends **HashSet** and adds no members of its own. It is a generic class that has this declaration:

class LinkedHashSet<E>
Here, **E** specifies the type of objects that the set will hold.

LinkedHashSet maintains a linked list of the entries in the set, in the order in which they were inserted.

This allows insertion-order iteration over the set.

That is, when cycling through a **LinkedHashSet** using an iterator, the elements will be returned in the order in which they were inserted.

This is also the order in which they are contained in the string returned by **toString( )** when called on a **LinkedHashSet** object.

To see the effect of **LinkedHashSet**, try substituting **LinkedHashSet** for **HashSet** in the preceding program. The output will be

[B, A, D, E, C, F]

which is the order in which the elements were inserted.

TreeSet

TreeSet extends **AbstractSet** and implements the **NavigableSet** interface. It creates a collection that uses a tree for storage. Objects are stored in sorted, ascending order. Access and retrieval times are quite fast, which makes **TreeSet** an

excellent choice when storing large amounts of sorted information that must be found quickly.

TreeSet is a generic class that has this declaration:

class TreeSet<E>

Here, **E** specifies the type of objects that the set will hold.

TreeSet has the following constructors:

TreeSet( )
TreeSet(Collection<? extends E> *c*)
TreeSet(Comparator<? super E> *comp*)
TreeSet(SortedSet<E> *ss*)

Example

```
import java.util.*; class
TreeSetDemo {

public static void main(String args[]) {
TreeSet<String> ts = new TreeSet<String>();
ts.add("C");
ts.add("A");
ts.add("B");

ts.add("E");
ts.add("F");
```

```
ts.add("D");
System.out.println(ts);
}
}
```

The output from this program is shown here: [A, B, C,
D, E, F]

PriorityQueue

PriorityQueue extends **AbstractQueue** and implements the **Queue** interface. It creates a queue that is prioritized based on the queue's comparator.

PriorityQueue is a generic class that has this declaration: class PriorityQueue<E>

Here, **E** specifies the type of objects stored in the queue. **PriorityQueue**s are dynamic, growing as necessary. **PriorityQueue** defines the six constructors shown here: PriorityQueue( )

PriorityQueue(int *capacity*)

PriorityQueue(int *capacity*, Comparator<? super E> *comp*)
PriorityQueue(Collection<? extends E> *c*)
PriorityQueue(PriorityQueue<? extends E> *c*)
PriorityQueue(SortedSet<? extends E> *c*)

ArrayDeque

Java SE 6 added the **ArrayDeque** class, which extends **AbstractCollection** and implements the **Deque** interface.

It adds no methods of its own.

ArrayDeque creates a dynamic array and has no capacity restrictions. **ArrayDeque** is a generic class that has this declaration:

class ArrayDeque<E>

Here, **E** specifies the type of objects stored in the collection. **ArrayDeque** defines the following constructors: ArrayDeque( )

ArrayDeque(int *size*) ArrayDeque(Collection<?
extends E> *c*)

Example:

```
import java.util.*;
class ArrayDequeDemo {

public static void main(String args[]) { ArrayDeque<String>
adq = new ArrayDeque<String>(); adq.push("A");
adq.push("B");
adq.push("D");
adq.push("E");

adq.push("F");      System.out.print("Popping
the stack: "); while(adq.peek() != null)
System.out.print(adq.pop() + " ");
System.out.println();

}
}
```

The output is shown here: Popping
the stack: F E D B A

Accessing a collection Via an Iterator:

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an **iterator( )** method that returns an iterator to the start of the collection.

By using this iterator object, you can access each element in the collection. Element at a time. In general, to use an iterator to cycle through the contents of a collection, follow these steps:

1. Obtain an iterator to the start of the collection by calling the collection's **iterator( )** method.

2. Set up a loop that makes a call to **hasNext( )**. Have the loop iterate as long as **hasNext( )** returns **true**.

3. Within the loop, obtain each element by calling **next( )**.

4. For collections that implement **List**, you can also obtain an iterator by calling **listIterator( )**.

5.As explained, a list iterator gives you the ability to access the collection in either the forward or backward direction and lets you modify an element.

6. Otherwise, **ListIterator** is used just like **Iterator**.

```
import java.util.*; class
IteratorDemo {

public static void main(String args[]) {

ArrayList<String> al = new ArrayList<String>();
al.add("C");

al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");

System.out.print("Original contents of al: ");

Iterator<String> itr = al.iterator(); while(itr.hasNext()) {

String element = itr.next(); System.out.print(element + " ");

}
System.out.println();

ListIterator<String> litr = al.listIterator(); while(litr.hasNext()) {

String element = litr.next(); litr.set(element + "+");

}

System.out.print("Modified contents of al: "); itr = al.iterator();

while(itr.hasNext()) { String element = itr.next();

System.out.print(element + " ");
}

System.out.println(); System.out.print("Modified list backwards: ");
while(litr.hasPrevious()) {

String element = litr.previous(); System.out.print(element + " ");

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

Output:

Original contents of al: C A E B D F Modified contents of al: C+ A+ E+ B+ D+ F+

Modified list backwards: F+ D+ B+ E+ A+ C+

For Each loop for iterating through collection: import java.util.*;

class ForEachDemo {

public static void main(String args[]) { ArrayList<Integer> vals = new ArrayList<Integer>();
vals.add(1);
vals.add(2);
vals.add(3);
vals.add(4);
vals.add(5);

System.out.print("Original contents of vals: "); for(int v : vals)
System.out.print(v + " "); System.out.println();

int sum = 0; for(int v : vals) sum += v;
System.out.println("Sum of values: " + sum);

}
}

Output:

Original contents of vals: 1 2 3 4 5 Sum of values: 15

Storing User Defined Classes in Collections:

collections are not limited to the storage of built-in objects.

The power of collections is that they can store any type of object, including objects of classes that you create.

User defined objects stored in **LinkedList** to store mailing addresses:

import java.util.*; class Address { private
String name; private String street; private String
city; private String state; private String code;

Address(String n, String s, String c, String st, String cd) {

name = n; street = s; city = c; state =
st; code = cd;

```
}

public String toString() {

return name + "\n" + street + "\n" + city + " " + state + " " +
code;
}}
class MailList {

public static void main(String args[]) { LinkedList<Address> ml = new
LinkedList<Address>(); ml.add(new Address("J.W. West", "11 Oak Ave",
"Urbana", "IL", "61801"));

ml.add(new Address("Ralph Baker", "1142 Maple Lane", "Mahomet", "IL",
"61853"));

ml.add(new Address("Tom Carlton", "867 Elm St", "Champaign", "IL",
"61820"));

for(Address element : ml) System.out.println(element + "\n");
System.out.println();
}
}
```

The output from the program is shown here:

```
J.W. West

11 Oak Ave Urbana
IL 61801 Ralph
Baker 1142 Maple
Lane

Mahomet IL 61853
Tom Carlton

867 Elm St Champaign
IL 61820
```

Random Access Interface:

RandomAccess interface contains no members.

However, by implementing this interface, a collection signals that it supports efficient random access to its elements.

By checking for the **RandomAccess** interface, client code can determine at run time whether a collection is suitable for certain types of random access operations—especially as they apply to

large collections.

RandomAccess is implemented by **ArrayList** and by the legacy **Vector** class, among others.

Working With Maps:

A *map* is an object that stores associations between keys and values, or *key/value pairs.*

Keys and values are objects. Keys must be unique, but the values may be duplicated.

Some maps can accept a **null** key and **null** values, others cannot.

There is one key point about maps that is important to mention at the outset: they don't implement the **Iterable** interface. This means that you *cannot* cycle through a map using a for-each style **for** loop. Furthermore, you can't obtain an iterator to a map.

However, as you will soon see, you can obtain a collection-view of a map, which does allow the use of either the **for** loop or an iterator.

The Map Interfaces

Because the map interfaces define the character and nature of maps, this discussion of maps begins with them.

The following interfaces support maps:
The Map Interface

The **Map** interface maps unique keys to values. A *key* is an object that you use to retrieve a value at a later date. Given a key and a value, you can store the value in a **Map** object. After the value is stored, you can retrieve it by using its key.

Map is generic: interface
Map<K, V>
Here, **K** specifies the type of keys, and **V** specifies the type of values.
The methods declared by **Map.**

Several methods

throw a **ClassCastException** when an object is incompatible with the elements in a map.

A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** is not allowed in the map.

An **UnsupportedOperationException** is thrown when an attempt is made to change an unmodifiable map.

An **IllegalArgumentException** is thrown if an invalid argument is used.

Maps revolve around two basic operations: **get( )** and **put( )**. To put a value into a map, use **put( )**, specifying the key and the value.

To obtain a value, call **get( )**, passing the key as an argument. The value is returned.

maps are not, themselves, collections because they do not implement the **Collection** interface. However, you can obtain a collection-view of a map. To do this, you can use the **entrySet( )** method. It returns a **Set** that contains the elements in the map.

To obtain a collection-view of the keys, use **keySet( )**.

To get a collection-view of the values, use **values( )**.

Collection-views are the means by which maps are integrated into the larger Collections Framework.

SortedMap

The **SortedMap** interface extends **Map**. It ensures that the entries are maintained in ascending

order based on the keys.

SortedMap is generic and is declared as shown here:

interface SortedMap<K, V>

K specifies the type of keys,

V specifies the type of values.

Several methods throw a **NoSuchElementException** when no items are in the invoking map. A **ClassCastException** is thrown when an object is incompatible with the elements in a map. A **NullPointerException** is thrown if an attempt is made to use a **null** object when **null** is not allowed in the map.

An **IllegalArgumentException** is thrown if an invalid argument is used.
Sorted maps allow very efficient manipulations of *submaps*

To obtain a submap, use **headMap( )**, **tailMap( )**, or **subMap( )**.

To get the first key in the set, call **firstKey( )**.

To get the last key, use **lastKey( )**.

NavigableMap Interface

The **NavigableMap** interface was added by Java SE 6.

It extends **SortedMap** and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys. **NavigableMap** is a generic interface that has this declaration:
interface NavigableMap<K,V>

Here, **K** specifies the type of the keys, and **V** specifies the type of the values associated with the keys.

Several methods throw a **ClassCastException** when an object is incompatible with the keys in the map.

A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** keys are not allowed in the set.

An **IllegalArgumentException** is thrown if an invalid argument is used.
equal to start.

Map.Entry Interface

The **Map.Entry** interface enables you to work with a map entry. Recall that the **entrySet( )** method declared by the **Map** interface returns a **Set** containing the map entries.

Each of these set elements is a **Map.Entry** object. **Map.Entry** is generic and is declared like this:

interface Map.Entry<K, V> Here, **K** specifies the type of keys, and **V** specifies the type of values.

the methods declared by **Map.Entry**.

Map Classes

Several classes provide implementations of the map interfaces. The classes that can be used for maps are summarized here:

HashMap:

The **HashMap** class extends **AbstractMap** and implements the **Map** interface. It uses a hash table to store the map.

This allows the execution time of **get( )** and **put( )** to remain constant even for large sets. **HashMap** is a generic class that has this declaration:

class HashMap<K, V>
Here, **K** specifies the type of keys, and **V** specifies the type of values.

The following constructors are defined:
HashMap( )

HashMap(Map<? extends K, ? extends V> *m*)
HashMap(int *capacity*)

HashMap(int *capacity*, float *fillRatio*)

The first form constructs a default hash map. The second form initializes the hash map by using the elements of *m.* The third form initializes the capacity of the hash map to *capacity.* The fourth form initializes both the capacity and fill ratio of the hash map by using its arguments.

The meaning of capacity and fill ratio is the same as for **HashSet**, described earlier. The default capacity is 16.

The default fill ratio is 0.75.

HashMap implements **Map** and extends **AbstractMap**. It does not add any methods of its own.

import java.util.*; class
HashMapDemo {

public static void main(String args[]) {

```
HashMap<String, Double> hm = new HashMap<String, Double>();
hm.put("John Doe", new Double(3434.34));

hm.put("Tom Smith", new Double(123.22)); hm.put("Jane
Baker", new Double(1378.00)); hm.put("Tod Hall", new
Double(99.22)); hm.put("Ralph Smith", new Double(-
19.08)); Set<Map.Entry<String, Double>> set =
hm.entrySet(); for(Map.Entry<String, Double> me : set) {
System.out.print(me.getKey() + ": ");
System.out.println(me.getValue());

}
System.out.println();

double balance = hm.get("John Doe"); hm.put("John
Doe", balance + 1000); System.out.println("John
Doe's new balance: " + hm.get("John Doe"));

}
}
```

Output from this program is shown here (the precise order may vary):
Ralph Smith: -19.08

Tom Smith: 123.22

John Doe: 3434.34 Tod
Hall: 99.22 Jane Baker:
1378.0
John Doe’s new balance: 4434.34
The program begins by creating a hash map and then adds the mapping of names to balances. Next, the contents of the map are displayed by using a set-view, obtained by calling **entrySet( )**. The keys and values are displayed by calling the **getKey( )** and **getValue( )** methods

that are defined by **Map.Entry**. Pay close attention to how the deposit is made into John

Doe’s

account. The **put( )** method automatically replaces any preexisting value that is associated with the specified key with the new value. Thus, after John Doe’s account is updated, the hash map will still contain just one “John Doe” account.

TreeMap

The **TreeMap** class extends **AbstractMap** and implements the **NavigableMap** interface. It creates maps stored in a tree structure.

A **TreeMap** provides an efficient means of storing key/value pairs in sorted order and allows

rapid retrieval. You should note that, unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order.

TreeMap is a generic class that has this declaration:
class TreeMap<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The following **TreeMap** constructors are defined:
TreeMap( )

TreeMap(Comparator<? super K> *comp*)
TreeMap(Map<? extends K, ? extends V> *m*)
TreeMap(SortedMap<K, ? extends V> *sm*)

The first form constructs an empty tree map that will be sorted by using the natural order of its keys. The second form constructs an empty tree-based map that will be sorted by using the **Comparator** *comp*. (Comparators are discussed later in this chapter.) The third form initializes

a tree map with the entries from *m*, which will be sorted by using the natural order of the keys. The fourth form initializes a tree map with the entries from *sm*, which will be sorted in the same order as *sm*.

TreeMap has no methods beyond those specified by the **NavigableMap** interface and the **AbstractMap** class.

The following program reworks the preceding example so that it uses **TreeMap**:

```
import java.util.*;

class TreeMapDemo {

public static void main(String args[]) { //
Create a tree map.

TreeMap<String, Double> tm = new TreeMap<String, Double>(); //
Put elements to the map.

tm.put("John Doe", new Double(3434.34));
tm.put("Tom Smith", new Double(123.22));
tm.put("Jane Baker", new Double(1378.00));
tm.put("Tod Hall", new Double(99.22));
tm.put("Ralph Smith", new Double(-19.08));
// Get a set of the entries.
```

```
Set<Map.Entry<String, Double>> set = tm.entrySet(); //
Display the elements.

for(Map.Entry<String, Double> me : set) {
System.out.print(me.getKey() + ": ");
System.out.println(me.getValue());

}
System.out.println();

double balance = tm.get("John Doe"); tm.put("John
Doe", balance + 1000); System.out.println("John
Doe's new balance: " + tm.get("John Doe"));

}
}
```

The following is the output from this program: Jane Baker: 1378.0

John Doe: 3434.34

Ralph Smith: -19.08 Todd Hall: 99.22 Tom Smith: 123.22
John Doe's current balance: 4434.34

TreeMap sorts the keys.

However, in this case, they are sorted by first name instead of last name.

You can alter this behavior by specifying a comparator when the map is created, as described shortly.

LinkedHashMap

LinkedHashMap extends **HashMap**.

It maintains a linked list of the entries in the map, in the

1order in which they were inserted. This allows insertion-order iteration over the map. That is, when iterating through a collection-view of a **LinkedHashMap**, the elements will be returned in the order in which they were inserted.

LinkedHashMap that returns its elements in the order in which they were last accessed.

LinkedHashMap is a generic class that has this declaration:

class LinkedHashMap<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values.

LinkedHashMap defines the following constructors:

LinkedHashMap( )
LinkedHashMap(Map<? extends K, ? extends V> *m*)

LinkedHashMap(int *capacity*)
LinkedHashMap(int *capacity*, float *fillRatio*)
LinkedHashMap(int *capacity*, float *fillRatio*, boolean *Order*)
The first form constructs a default **LinkedHashMap**.

The second form initializes the **LinkedHashMap** with the elements from *m.* The third form initializes the capacity. The fourth form initializes both capacity and fill ratio. The meaning of capacity and fill ratio are the same as for **HashMap**. The default capactiy is 16. The default ratio is 0.75. The last form allows you to specify whether the elements will be stored in the linked list by insertion order, or by order of last access.

IdentityHashMap

IdentityHashMap extends **AbstractMap** and implements the **Map** interface.

It is similar to **HashMap** except that it uses reference equality when comparing elements. **IdentityHashMap** is a generic class that has this declaration:

class IdentityHashMap<K, V>

Here, **K** specifies the type of key, and **V** specifies the type of value. The API documentation explicitly states that **IdentityHashMap** is not for general use.

The EnumMap Class

EnumMap extends **AbstractMap** and implements **Map**. It is specifically for use with keys of an **enum** type. It is a generic class that has this declaration:

class EnumMap<K extends Enum<K>, V>

Here, **K** specifies the type of key, and **V** specifies the type of value. Notice that **K** must extend **Enum<K>**, which enforces the requirement that the keys must be of an **enum** type.

EnumMap defines the following constructors:

EnumMap(Class<K> *kType*)
EnumMap(Map<K, ? extends V> *m*)
EnumMap(EnumMap<K, ? extends V> *em*)

The first constructor creates an empty **EnumMap** of type *kType*. The second creates an **EnumMap** map that contains the same entries as *m*. The third creates an **EnumMap** initialized with the values in *em.*

Comparator interface

Comparator is a generic interface that has this declaration:
interface Comparator<T>

Here, **T** specifies the type of objects being compared.

The **Comparator** interface defines two methods: **compare( )** and **equals( )**. The **compare( )** method, shown here, compares two elements for order:

int compare(T *obj1*, T *obj2*)
obj1 and *obj2* are the objects to be compared.

This method returns zero if the objects are equal.

It returns a positive value if *obj1* is greater than *obj2*. Otherwise, a negative value is returned. **ClassCastException** if the types of the objects are not compatible for comparison.

By overriding **compare( )**, you can alter the way that objects are ordered.
For example, to sort in reverse order, you can create a comparator that reverses the outcome of a comparison. The **equals( )** method, shown here, tests whether an object equals the invoking comparator:

boolean equals(Object *obj*)

Here, *obj* is the object to be tested for equality. The method returns **true** if *obj* and the invoking object are both **Comparator** objects and use the same ordering. Otherwise, it returns **false**.

```
import java.util.*;

class MyComp implements Comparator<String> {
public int compare(String a, String b) {
```

```
String aStr, bStr;
aStr = a;

bStr = b;
return bStr.compareTo(aStr);
}
}
class CompDemo {

public static void main(String args[]) {

TreeSet<String> ts = new TreeSet<String>(new MyComp());
ts.add("C");

ts.add("A");
ts.add("B");
ts.add("E");
ts.add("F");

ts.add("D");       for(String
element : ts)

System.out.print(element + " ");
System.out.println();

}}
```

Output:
F E D C B A

The Collection Algorithms:

Collections Framework defines several algorithms that can be applied to collections and maps.

algorithms are defined as static methods within the **Collections** class. static
<T> Boolean addAll(Collection <? super T> c, T ... elements)

Inserts the elements specified by elements into the

collection specified by c. Returns true if the elements were added and false otherwise.

static <T> Queue<T> asLifoQueue(Deque<T> c)
Returns a last-in, first-out view of c.

static <T>int binarySearch(List<? extends T> list, T value, Comparator<? super T> c)
Searches for value in list ordered according to c. Returns the position of value

in list, or a negative value if value is not found.

static <T> int binarySearch(List<? extends Comparable<? super T>> list,T value)

Searches for value in list. The list must be sorted. Returns the position of value in list, or a negative value if value is not found.

static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> t) Returns a run-time type-safe view of a collection. An attempt to insert an incompatible element will cause a ClassCastException.

static <E> List<E> checkedList(List<E> c, Class<E> t)

Returns a run-time type-safe view of a List. An attempt to insert an incompatible element will cause a ClassCastException.

static <K, V> Map<K, V> checkedMap(Map<K, V> c, Class<K> keyT, Class<V> valueT)

Returns a run-time type-safe view of a Map.

An attempt to insert an incompatible element will cause a ClassCastException.

static <E> List<E> checkedSet(Set<E> c, Class<E> t)
Returns a run-time type-safe view of a Set. An

attempt to insert an incompatible element will cause a ClassCastException.

static int frequency(Collection<?> c, Object obj)

Counts the number of occurrences of obj in c and returns the result.
static int indexOfSubList(List<?> list, List<?> subList)

Searches list for the first occurrence of subList.

Returns the index of the first match, or –1 if no match is found.
static int lastIndexOfSubList(List<?> list, List<?> subList)

Searches list for the last occurrence of subList.
Returns the index of the last match, or –1 if no match is found.

```
import java.util.*;
class AlgorithmsDemo {

public static void main(String args[]) {
LinkedList<Integer> ll = new LinkedList<Integer>();
```

```
ll.add(-8);

ll.add(20);
ll.add(-20);
ll.add(8);

Comparator<Integer> r = Collections.reverseOrder();
Collections.sort(ll, r);

System.out.print("List sorted in reverse: ");
for(int i : ll)

System.out.print(i+ " ");
System.out.println();
Collections.shuffle(ll);
System.out.print("List shuffled: ");
for(int i : ll)

System.out.print(i + " ");
System.out.println();

System.out.println("Minimum: " + Collections.min(ll));
System.out.println("Maximum: " + Collections.max(ll));
}
}
```

Output:

List sorted in reverse: 20 8 -8 -20 List
shuffled: 20 -20 8 -8 Minimum: -20
Maximum: 20

Notice that **min( )** and **max( )** operate on the list after it has been shuffled. Neither requires a sorted list for its operation.

Why Generic Collections?

As mentioned at the start of this chapter, the entire Collections Framework was refitted for generics when JDK 5 was released.

Furthermore, the Collections Framework is arguably the single most important use of generics in the Java API.

The reason for this is that generics add type safety to the Collections Framework. Before moving on, it is worth taking some time to examine in detail the significance of this improvement.

```
import java.util.*;
class OldStyle {

public static void main(String args[]) {
ArrayList list = new ArrayList();
list.add("one");

list.add("two");
list.add("three");
list.add("four");

Iterator itr = list.iterator();
while(itr.hasNext()) {

String str = (String) itr.next(); // explicit cast needed here.
System.out.println(str + " is " + str.length() + " chars long.");

}
}
}
```

Prior to generics, all collections stored references of type **Object**.

This allowed any type of reference to be stored in the collection.

The preceding program uses this feature to store references to objects of type **String** in **list**, but any type of reference could have been stored. Unfortunately, the fact that a pre-generics collection stored **Object** references could easily lead to errors.

First, it required that you, rather than the compiler, ensure that only objects of

the proper type be stored in a specific collection. For example, in the preceding example, **list** is clearly intended to store **String**s, but there is nothing that actually prevents another type of reference from being added to the collection
For example, the compiler will find nothing wrong with this line of code:
list.add(new Integer(100));

Because **list** stores **Object** references, it can store a reference to **Integer** as well as it can store a reference to **String**.

However, if you intended **list** to hold only strings, then the preceding statement would corrupt the collection. Again, the compiler had no way to know that the preceding statement is invalid.

The second problem with pre-generics collections is that when you retrieve a reference from the collection, you must manually cast that reference into the proper type.

This is why the preceding program casts the reference returned by **next( )** into **String**. Prior to generics, collections simply stored **Object** references. Thus, the cast was necessary when

retrieving

objects from a collection.

Aside from the inconvenience of always having to cast a retrieved reference into its proper type, this lack of type safety often led to a rather serious, but surprisingly easy-to-create, error. Because **Object** can be cast into any type of object, it was possible to cast a reference obtained from a collection into the *wrong type.* For example, if the following statement were added to the preceding example, it would still compile without error, but generate a run-time exception when executed:

Integer i = (Integer) itr.next();

• Ensures that only references to objects of the proper type can actually be stored in a collection. Thus, a collection will always contain references of a known type.

• Eliminates the need to cast a reference retrieved from a collection. Instead, a reference retrieved from a collection is automatically cast into the proper type. This prevents run-time errors due to invalid casts and avoids an entire category of errors.

The Legacy Classes and Interfaces

As explained at the start of this chapter, early versions of **java.util** did not include the Collections Framework. Instead, it defined several classes and an interface that provided an ad hoc method of storing objects.

When collections were added (by J2SE 1.2), several of the original classes were reengineered to support the collection interfaces.

Thus, they are fully compatible with the framework. While no classes have actually been deprecated, one has been rendered obsolete.

Of course, where a collection duplicates the functionality of a legacy class,you will usually want to use the collection for new code. In general, the legacy classes are supported because there is still code that uses them.

One other point: none of the collection classes are synchronized, but all the legacy classes are synchronized.
This distinction may be important in some situations. Of course, you can

easily synchronize collections, too, by using one of the algorithms provided by **Collections**.

The legacy classes defined by **java.util** are shown here:

Dictionary
Hashtable
Properties

Stack
Vector

There is one legacy interface called **Enumeration**.

The following sections examine **Enumeration** and each of the legacy classes, in turn. The Enumeration Interface

The **Enumeration** interface defines the methods by which you can *enumerate* (obtain one at a time) the elements in a collection of objects. This legacy interface has been superseded by **Iterator**.

interface Enumeration<E>
where **E** specifies the type of element being enumerated.

Vector

Vector implements a dynamic array. It is similar to **ArrayList**, but with two differences: **Vector** is synchronized, and it contains many legacy methods that are not part of the Collections.

Vector is declared like this:
class Vector<E>

Here, **E** specifies the type of element that will be stored.

Here are the **Vector** constructors:
Vector( )
Vector(int *size*)
Vector(int *size*, int *incr*)
Vector(Collection<? extends E> *c*)

- The first form creates a default vector, which has an initial size of 10.
- The second form creates a vector whose initial capacity is specified by *size.*
- The third form creates a vector whose initial capacity is specified by *size* and whose increment is specified by *incr.*

The increment specifies the number of elements to allocate each time that a vector is resized upward.

The fourth form creates a vector that contains the elements of collection *c.*

Stack

Stack is a subclass of **Vector** that implements a standard last-in, first-out stack. **Stack** only defines the default constructor, which creates an empty stack. With the release of JDK 5, **Stack**

was retrofitted for generics and is declared as shown here:

class Stack<E>
Here, **E** specifies the type of element stored in the stack.
Stack includes all the methods defined by **Vector.**

Dictionary

Dictionary is an abstract class that represents a key/value storage repository and operates much like **Map**.

Given a key and value, you can store the value in a **Dictionary** object. Once

the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs.

Although not currently deprecated, **Dictionary** is

classified as obsolete, because it is fully superseded by **Map**. However, **Dictionary** is still in use and thus is fully discussed here.

class Dictionary<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values. The abstract methods defined by **Dictionary** are listed in Table 17-17.

Hashtable

Hashtable was part of the original **java.util** and is a concrete implementation of a Dictionary.

HashMap, **Hashtable** stores key/value pairs in a hash table. However, neither keys

nor values can be **null**. When using a **Hashtable**, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

Hashtable was made generic by JDK 5.
It is declared like this: class Hashtable<K, V>

Hashtable( )
Hashtable(int *size*)
Hashtable(int *size*, float *fillRatio*)
Hashtable(Map<? extends K, ? extends V> *m*)

The first version is the default constructor.

The second version creates a hash table that has an
initial size specified by *size*.

The third version creates a hash table that has an initial size specified by *size* and a fill ratio specified by *fillRatio.* This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hashtable multiplied by its fill ratio, the hash table is expanded. If you do not specify a fill ratio,

then 0.75 is used.

Finally, the fourth version creates a hash table that is initialized with the

elements in *m.* The capacity of the hash table is set to twice the number of elements in *m.* The default load factor of 0.75 is used.

Properties

Properties is a subclass of **Hashtable**. It is used to maintain lists of values in which the key is a **String** and the value is also a **String**.

The **Properties** class is used by many other Java classes. For example, it is the type of object returned by **System.getProperties( )** when obtaining environmental values.

Although the **Properties** class, itself, is not generic, several of its methods are.
Properties defines the following instance variable:

Properties defaults;

This variable holds a default property list associated with a **Properties** object. **Properties** defines these constructors:

Properties( )
Properties(Properties *propDefault*)

Module 3

Syllabus -String Handling :

The String Constructors, String Length, Special String Operations, String Literals, String Concatenation, String Concatenation with Other Data Types, String Conversion and toString( ) Character Extraction, charAt( ), getChars( ), getBytes( ) toCharArray(), String Comparison, equals( ) and equalsIgnoreCase( ), regionMatches( ) startsWith( ) and endsWith( ), equals( ) Versus == , compareTo( ) Searching Strings, Modifying a String, substring( ), concat( ), replace( ), trim( ), Data Conversion Using valueOf( ), Changing the Case of Characters Within a String, Additional String Methods, StringBuffer , StringBuffer Constructors, length( ) and capacity( ), ensureCapacity( ), setLength( ), charAt( ) and setCharAt( ), getChars( ),append( ), insert( ), reverse( ), delete( ) and deleteCharAt( ), replace( ), substring( ), Additional StringBuffer Methods, StringBuilder.

1. **What are the different types of String Constructors available in Java?**

The String class supports several constructors. a.
To create an empty String

the default constructor is used.
Ex: String s = new String();

will create an instance of String with no characters in it.

- To create a String initialized by an array of characters, use the constructor shown here:

String(char chars[ ])

ex: char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
This constructor initializes s with the string “abc”.

\} To specify a subrange of a character array as an initializer using the following constructor:

String(char chars[ ], int startIndex, int numChars)

Here, startIndex specifies the index at which the subrange begins, and numChars specifies the number of characters to use. Here is an example:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
```

This initializes s with the characters cde.

\{ To construct a String object that contains the same character sequence as another String object using this constructor:

String(String strObj)

Here, strObj is a String object.

```
class MakeString

{ public static void main(String args[]) {
char c[] = {'J', 'a', 'v', 'a'};

String s1 = new String(c);




 String s2 = new String(s1);
System.out.println(s1);
System.out.println(s2);
}

}
```

The output from this program is as follows: Java

Java

As you can see, s1 and s2 contain the same string.

- To Construct string using byte array:

Even though Java's char type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set.

Because 8-bit ASCII strings are common, the String class provides constructors that initialize a string when given a byte array.

Ex: String(byte asciiChars[ ])

String(byte asciiChars[ ], int startIndex, int numChars)

The following program illustrates these constructors: class SubStringCons

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

{

byte ascii[] = {65, 66, 67, 68, 69, 70 };
String s1 = new String(ascii);
System.out.println(s1);

String s2 = new String(ascii, 2, 3);
System.out.println(s2);

}

}
```

This program generates the following output:
ABCDEF

CDE

- To construct a String from a StringBuffer by using the constructor shown here:

  Ex: String(StringBuffer strBufObj)

- Constructing string using Unicode character set and is shown here:

  String(int codePoints[ ], int startIndex, int numChars)

  codePoints is an array that contains Unicode code points.

h. Constructing string that supports the new StringBuilder class. Ex : String(StringBuilder strBuildObj)

Note:

String can be constructed by using string literals.

String s1=”Hello World”

String concatenation can be done using + operator. With other data type also.

String Length

- The length of a string is the number of characters that it contains. To obtain this value, call the length( ) method,

- Syntax:

  int length( )

\{ Example

char chars[] = { 'a', 'b', 'c' }; String s = new String(chars);
System.out.println(s.length());// 3

toString()

// Every class implements toString( ) because it is defined by Object. However, the default Implementation of toString() is sufficient.

// For most important classes that you create, will want to override toString() and provide your own string representations.

String toString( )

// To implement toString( ), simply return a String object that contains the human-readable string that appropriately describes an object of your class.

// By overriding toString() for classes that you create, you allow them to be fully integrated into Java's programming environment. For example, they can be used in print() and println() statements and in concatenation expressions.

// The following program demonstrates this by overriding

toString( ) for the Box class:

```
class Box

{

double width; double height; double depth;
 Box(double w, double h, double d)

{ width = w; height = h; depth = d; }

public String toString()
```

```
{ return "Dimensions are " + width + " by " + depth + " by " + height + "."; }

}

class toStringDemo {

public static void main(String args[])

{

 Box b = new Box(10, 12, 14);
String s = "Box b: " + b;
System.out.println(b);

System.out.println(s);

}

}
```

The output of this program is shown here:
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0

Character Extraction

The String class provides a number of ways in which characters can be extracted from a String object. String object can not be indexed as if they were a character array, many of the String methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

A. charAt( )

1. description:

To extract a single character from a String, you can refer directly to an individual character via the charAt( ) method.

2. Syntax

char charAt(int where)

Here, where is the index of the character that you want to obtain.
charAt( ) returns the character at the specified location.

3. example,

```
char ch;
```

ch = "abc".charAt(1); assigns
the value “b” to ch.

B. getChars( )

5. to extract more than one character at a time, you can use the getChars() method.

6. Syntax

   void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)

Here, sourceStart specifies the index of the beginning of the substring, sourceEnd specifies an index that is one past the end of the desired The array that will receive the characters is specified by target. The index within target at which the substring will be copied is passed in targetStart.

```
\}  class getCharsDemo {

    public static void main(String args[])

    { String s = "This is a demo of the getChars method."; int
    start = 10;

     int end = 14;

     char buf[] = new char[end - start];
      s.getChars(start, end, buf, 0);
     System.out.println(buf);

   }

 }
```

Here is the output of this program:
demo

C. getBytes( )

1. This method is called getBytes( ), and it uses the default character-to-byte conversions provided by the platform.

   Syntax:

   byte[ ] getBytes( )

Other forms of getBytes( ) are also available.

2. getBytes( ) is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters. For example,

   most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

D. toCharArray( )

If you want to convert all the characters in a String object into a character array, the easiest way is to call toCharArray( ).

It returns an array of characters for the entire string.

It has this general form:
char[ ] toCharArray( )

2. <u>String Comparison:</u>

The String class includes several methods that compare strings or substrings within strings.

equals( )

To compare two strings for equality, use equals( ). It has this general form:

```
boolean equals(Object str)
```

Here, str is the String object being compared with the invoking String object. It returns true if the strings contain the same characters in the same order, and false otherwise. The comparison is case-sensitive.

A. equalsIgnoreCase( )

To perform a comparison that ignores case differences, call equalsIgnoreCase( ). When it compares two strings, it considers A-Z to be the same as a-z.

It has this general form:

```
boolean equalsIgnoreCase(String str)
```

Here, str is the String object being compared with the invoking String object. It, too, returns true if the strings contain the same characters in the same order, and

false otherwise.

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {

 public static void main(String args[]) {
String s1 = "Hello";

String s2 = "Hello"; String
s3 = "Good-bye"; String s4
= "HELLO";

System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
System.out.println(s1 + " equals " + s3 + " -> " + s1.equals(s3));
System.out.println(s1 + " equals " + s4 + " -> " + s1.equals(s4));
System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
s1.equalsIgnoreCase(s4)); } }
```

The output from the program is shown here:

Hello equals Hello -> true

Hello equals Good-bye -> false

Hello equals HELLO -> false

Hello equalsIgnoreCase HELLO -> true

B. regionMatches( )

1. The regionMatches() method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons.

2. Syntax:

boolean regionMatches(int startIndex, String str2, int str2StartIndex, int numChars)

boolean regionMatches(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars)

3. For both versions, startIndex specifies the index at which the region begins within the invoking String object.

The String being compared is specified by str2. The index at which the

comparison will start within str2 is specified by str2 StartIndex. The length of the substring being compared is passed in numChars.

4. In the second version, if ignoreCase is true, the case of the characters is ignored. Otherwise, case is significant.

C. startsWith( ) and endsWith( )

1. The startsWith( ) method determines whether a given String begins with a specified string.

2. endsWith( ) determines whether the String in question ends with a specified string.

3. Syntax

```
boolean startsWith(String str)
boolean endsWith(String str)
```

Here, str is the String being tested.

If the string matches, true is returned. Otherwise, false is returned.

For example,

"Foobar".endsWith("bar")

"Foobar".startsWith("Foo") are both true.

4. A second form of startsWith( ), shown here, lets you specify a starting point:

```
boolean startsWith(String str, int startIndex)
```

Here, startIndex specifies the index into the invoking string at which point the search will begin. For example,

"Foobar".startsWith("bar", 3)
returns true.

D. **equals( ) Versus ==**

It is important to understand that the equals( ) method and the == operator perform two different operations.

the equals( ) method compares the characters inside a String object.

The == operator compares two object references to see whether they refer to the same instance.

```
class EqualsNotEqualTo {

public static void main(String args[]) {
String s1 = "Hello";

String s2 = new String(s1);

System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
}

}
```

E. **compareTo( )**

1. Sorting applications, you need to know which is less than, equal to, or greater than the next.

2. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The String method compareTo() serves this purpose.
3. It has this general form:

```
int compareTo(String str)
```

Here, str is the String being compared with the invoking String. The result of the comparison is returned and is interpreted,

4. Less than zero when invoking string is less than str.

5. Greater than zero when invoking string is greater than str.

6. Zero The two strings are equal.

```
// A bubble sort for Strings.
class SortString

{ static String arr[] = { "Now", "is", "the", "time", "for", "all", "good", "men",
"to", "come", "to", "the", "aid", "of", "their", "country" };

 public static void main(String args[]) {
 for(int j = 0; j < arr.length; j++)
```

```
{ for(int i = j + 1; i < arr.length; i++) {
 if(arr[i].compareTo(arr[j]) < 0)

 { String t = arr[j];

arr[j]  =  arr[i];
 arr[i] = t;

}

} System.out.println(arr[j]);

}

}

}
```

The output of this program is the list of words:

Now aid all come country for good is men of the the their time to to

As you can see

7. Ignore case differences when comparing two strings, use compareToIgnoreCase( ), This method returns the same results as compareTo(), except that case differences are ignored.

5. <u>Searching String</u>

A. indexOf( ) and lastIndexOf( )

1. indexOf( ) Searches for the first occurrence of a character or substring.

2. lastIndexOf( ) Searches for the last occurrence of a character or substring.

3. These two methods are overloaded in several different ways

4. return the index at which the character or substring was found, or

   –1 on failure.

5. To search for the first occurrence of a character, int indexOf(int ch)

6. To search for the last occurrence of a character,

   int lastIndexOf(int ch) Here, ch is the character being sought

7. To search for the first or last occurrence of a substring, use int indexOf(String str) int lastIndexOf(String str) Here, str specifies the substring.

6. You can specify a starting point for the search using these forms: int indexOf(int ch, int startIndex)

   int lastIndexOf(int ch, int startIndex)

7. int indexOf(String str, int startIndex) int lastIndexOf(String str, int startIndex) Here, startIndex specifies the index at which point the search begins.

8. For indexOf( ), the search runs from startIndex to the end of the string. For lastIndexOf( ), the search runs from startIndex to zero. The following example shows how to use the various index methods to search inside of Strings:

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {

 public static void main(String args[])

{ String s = "Now is the time for all good men " + "to come to the aid of
their country.";

System.out.println(s); System.out.println("indexOf(t) =
" + s.indexOf('t'));

System.out.println("lastIndexOf(t) = " + s.lastIndexOf('t'));
System.out.println("indexOf(the) = " + s.indexOf("the"));
System.out.println("lastIndexOf(the) = " + s.lastIndexOf("the"));
System.out.println("indexOf(t, 10) = " + s.indexOf('t', 10));
System.out.println("lastIndexOf(t, 60) = " + s.lastIndexOf('t', 60));
System.out.println("indexOf(the, 10) = " + s.indexOf("the", 10));
System.out.println("lastIndexOf(the, 60) = " + s.lastIndexOf("the",
60));
}

}
```

Output

Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7

```
lastIndexOf(t)        =       65
indexOf(the)          =        7
lastIndexOf(the)      =       55
indexOf(t,     10)    =       11
lastIndexOf(t,   60)  =   55
indexOf(the,   10)    =   44
lastIndexOf(the, 60) = 55
```

6. **Modifying a String**

String objects are immutable, whenever you want to modify a String, you must either copy it into a StringBuffer or StringBuilder, or use one of the following String methods, which will construct a new copy of the string with your modifications complete.

1. You can extract a substring using substring( ). It has two forms. The first is String substring(int startIndex)

2. Here, startIndex specifies the index at which the substring will begin. This form returns a copy of the substring that begins at startIndex and runs to the end of the invoking string.

3. The second form of substring( ) allows you to specify both the beginning and ending index of the substring:

   String substring(int startIndex, int endIndex)

   Here, startIndex specifies the beginning index, and endIndex specifies the stopping point.

4. The string returned contains all the characters from the beginning index, up to, but not including, the ending index. The following program uses substring( ) to replace all instances of one substring with another within a string:

```
// Substring replacement.
class StringReplace {

public static void main(String args[]) {
String org = "This is a test. This is, too.";
String search = "is";

String sub = "was";
String result = ""; int i;

do                              {
```

```
System.out.println(org);  i =
org.indexOf(search);

 if(i  !=  -1)  {  result  =  org.substring(0,  i);
result = result + sub;

result = result + org.substring(i + search.length()); org
 = result;

} } while(i != -1);

}

}
```

The output from this program is shown here:
This is a test. This is, too.

Thwas is a test. This is, too. Thwas was a test. This is, too. Thwas was a test. Thwas is, too. Thwas was a test. Thwas was, too.

B. concat( )

1. concatenate two strings using concat( )

String concat(String str)

2. This method creates a new object that contains the invoking string with the contents of str appended to the end.

3. concat( ) performs the same function as +.

4. String s1 = "one";

   String s2 = s1.concat("two");

C. replace( )

1. The replace( ) method has two forms.

2. The first replaces all occurrences of one character in the invoking string with another character.

Syntax:

String replace(char original, char replacement) Here, original specifies the character to be replaced by the character specified by replacement. The resulting string is returned.

Example

String s = "Hello".replace('l', 'w'); puts
the string “Hewwo” into s.

The second form of replace( ) replaces one character sequence with another. It has this general form:

String replace(CharSequence original, CharSequence replacement)

D. trim( )

The trim() method returns a copy of the invoking string from which any leading and trailing whitespace has been removed.

Syntax: String
trim( ) Example:
String s = " Hello World ".trim();

This puts the string “Hello World” into s.

The trim( ) method is quite useful when you process user commands.

```
// Using trim() to process commands.
import java.io.*;

class UseTrim

{ public static void main(String args[]) throws IOException {
BufferedReader br = new BufferedReader(new
nputStreamReader(System.in));

String str;

System.out.println("Enter 'stop' to quit.");
System.out.println("Enter State: ");

do { str = br.readLine();
 str = str.trim();
if(str.equals("Illinois"))

System.out.println("Capital is Springfield."); else
if(str.equals("Missouri"))
System.out.println("Capital is Jefferson City."); else
```

```
if(str.equals("California"))
System.out.println("Capital is Sacramento."); else
if(str.equals("Washington"))
System.out.println("Capital is Olympia."); // ... }
while(!str.equals("stop"));

}

}
```

5. Data Conversion

1. The valueOf( ) method converts data from its internal format into a human-readable form.

2. It is a static method that is overloaded within String for all of Java's built-in types so that each type can be converted properly into a string.

3. valueOf( ) is also overloaded for type Object, so an object of any class type you create can also be used as an argument

Syntax:

```
static String valueOf(double num)
static String valueOf(long num) static
String valueOf(Object ob) static
String valueOf(char chars[ ])
```

4. valueOf( ) is called when a string representation of some other type of data is needed. example, during concatenation operations.

5. Any object that you pass to valueOf( ) will return the result of a call to the object's toString( ) method.

6. There is a special version of valueOf() that allows you tospecify a subset of a char array.

Syntax:

```
static String valueOf(char chars[ ], int startIndex, int numChars)
```

7. Here, chars is the array that holds the characters, startIndex is the index into the array of characters at which the desired substring begins, and numChars specifies the length of the substring.

6. Changing Case of Characters

A. toLowerCase( )

1. converts all the characters in a string from uppercase to lowercase.

2. This method return a String object that contains the lowercase equivalent of the invoking String.

3. Non alphabetical characters, such as digits, are unaffected.

Syntax

String toLowerCase( )

B. toUpperCase()

1. converts all the characters in a string from lowercase to uppercase.

2. This method return a String object that contains the uppercase equivalent of the invoking String.

3. Non alphabetical characters, such as digits, are unaffected.

Syntax

String toUpperCase( )

```
class ChangeCase {

 public static void main(String args[]) {
String s = "This is a test.";
System.out.println("Original: " + s); String
upper = s.toUpperCase();

String lower = s.toLowerCase();
System.out.println("Uppercase: " + upper);
System.out.println("Lowercase: " + lower);
}

}
```

Output:

```
Original: This is a test. Uppercase:
 THIS IS A TEST. Lowercase: this
 is a test.
```

StringBuffer

StringBuffer is a peer class of String that provides much of the functionality of strings. As you know, String represents fixed-length, immutable character sequences.

StringBuffer represents growable and writeable character sequences.

StringBuffer may have characters and substrings inserted in the middle or appended to the end. StringBuffer will automatically grow to make room for such additions and often has more characters pre allocated than are actually needed, to allow room for growth.

StringBuffer Constructors

StringBuffer defines these four constructors:

StringBuffer( )

StringBuffer(int size)

StringBuffer(String str)

StringBuffer(CharSequence chars)

a. The default constructor (the one with no parameters) reserves room for 16 characters without reallocation.

b. The second version accepts an integer argument that explicitly sets the size of the buffer.

c. The third version accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

d. StringBuffer allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time.

A. length( ) and capacity( )

a. The current length of a StringBuffer can be found via the length( ) method, while the

total allocated capacity can be found through the capacity( ) method.
Syntax

int length( ) int

capacity( )

b. Example:

```
class StringBufferDemo
{
  public static void main(String args[])
  {
     StringBuffer sb = new StringBuffer("Hello");
    System.out.println("buffer = " + sb);
   System.out.println("length = " + sb.length());
   System.out.println("capacity = " + sb.capacity());
}}
```

Output

buffer = Hello

length = 5

capacity = 21

B. ensureCapacity( )

a. If you want to pre allocate room for a certain number of characters after a StringBuffer has been constructed, you can use ensureCapacity( ) to set the size of the buffer.

b. This is useful if you know in advance that you will be appending a large number of small strings to a StringBuffer.

Syntax

void ensureCapacity(int capacity)

Here, capacity specifies the size of the buffer.

C. setLength( )

a. To set the length of the buffer with in a StringBufferobject,

Syntax:

void setLength(int len)

Here, len specifies the length of the buffer. This value must be nonnegative.

When you increase the size of the buffer, null characters are added to the end of the existing buffer.

If you call setLength( ) with a value less than the current value returned by length(), then the characters stored beyond the new length will be lost.

D. charAt( ) and setCharAt( )

a. The value of a single character can be obtained from a StringBuffer via the charAt()method. You can set the value of a character within a StringBuffer using setCharAt( ).

b. Syntax

   char charAt(int where)

   void setCharAt(int where, char ch)

c. For charAt( ), where specifies the index of the character being obtained.

d. For setCharAt( ), where specifies the index of the character being set, and ch specifies the new value of that character.

```
// Demonstrate charAt() and setCharAt().

class setCharAtDemo {

public static void main(String args[])

{ StringBuffer sb = new StringBuffer("Hello");

System.out.println("buffer before = " + sb);

System.out.println("charAt(1) before = " + sb.charAt(1));

sb.setCharAt(1, 'i');

sb.setLength(2);
```

```
System.out.println("buffer after = " + sb);

System.out.println("charAt(1) after = " + sb.charAt(1)); } }
```

Output

```
buffer before = Hello

charAt(1) before = e

 buffer after = Hi

 charAt(1) after = i
```

E. getChars( )

a. To copy a substring of a StringBuffer into an array, use the getChars( ) method.

Syntax

Syntax

```
void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
```

Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring.

b. This means that the substring contains the characters from sourceStart through sourceEnd–1.

c. The array that will receive the characters is specified by target.

The index within target which the substring will be copied is passed in targetStart.

d. Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

F. append( )

1. The append() method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object. It has several overloaded versions. Here are a few of its forms:

StringBuffer append(String str)
StringBuffer append(int num)

StringBuffer append(Object obj)

2. The result is appended to the current StringBuffer object.

3. The buffer itself is returned by each version of append( ).

4. This allows subsequent calls to be chained together, as shown in the following example:

```
class appendDemo {

 public static void main(String args[]) {
 String s; int a = 42;

  StringBuffer sb = new StringBuffer(40);

  s = sb.append("a = ").append(a).append("!").toString();
  System.out.println(s);
 }

}
```

Output

a = 42!

G. **insert( )**

1. The insert() method inserts one string in to another.

2. It is overloaded to accept values of all the simple types, plus Strings, Objects, and CharSequences.

3. Like append(),it calls String.valueOf() to obtain the string representation of the value it is called with.

4. This string is then inserted into the invoking StringBuffer object.

5. These are a few of its forms:

```
StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)
```

Here, index specifies the index at which point the string will be inserted into the invoking StringBuffer object.

6. The following sample program inserts “like” between “I” and “Java”:

```
class insertDemo { public static void main(String args[]) {
  StringBuffer sb = new StringBuffer("I Java!"); sb.insert(2,
  "like ");
  System.out.println(sb);

 }

}
```

7. Output

I like Java!

H. reverse( )

You can reverse the characters within a StringBuffer object using reverse( ), shown here:
StringBuffer reverse( )

This method returns the reversed object on which it was called. The following program demonstrates reverse( )

```
class ReverseDemo {

public static void main(String args[])

{ StringBuffer s = new StringBuffer("abcdef");
System.out.println(s);

s.reverse();

System.out.println(s);

}

}
```

Output abcdef
fedcba

I. delete( ) and deleteCharAt( )

You can delete characters within a StringBuffer by using the methods delete( ) and deleteCharAt( ).

Syntax:

StringBuffer delete(int startIndex, int endIndex)
StringBuffer deleteCharAt(int loc)

The delete( ) method deletes a sequence of characters from the invoking object.

Here, startIndex specifies the index of the first character to remove, and endIndex specifies an index one past the last character to remove.

Thus, the substring deleted runs from startIndex to endIndex–1. The resulting StringBuffer object is returned.

The deleteCharAt( ) method deletes the character at the index specified by loc. It returns the resulting StringBuffer object.

```
// Demonstrate delete() and deleteCharAt()

class deleteDemo { public static void main(String args[])

{ StringBuffer sb = new StringBuffer("This is a test.");

sb.delete(4, 7);

System.out.println("After delete: " + sb);

sb.deleteCharAt(0);

System.out.println("After deleteCharAt: " + sb);

}

}
```

Output

After delete: This a test.

After deleteCharAt: his a test.

J. **replace( )**

a. You can replace one set of characters with another set inside a StringBuffer object by

calling replace( ).

b. Syntax

StringBuffer replace(int startIndex, int endIndex, String str)

The substring being replaced is specified by the indexes startIndex and endIndex.

c. Thus, the substringatstartIndexthroughendIndex–1 is replaced.The replacement string is passed in str.

The resulting StringBuffer object is returned.

```
class replaceDemo {

public static void main(String args[])

{ StringBuffer sb = new StringBuffer("This is a test.");
sb.replace(5, 7, "was");

System.out.println("After replace: " + sb);

}
}
```

Here is the output:

After replace: This was a test.

K. **substring( )**

1. It has the following two forms:
   Syntax

   String substring(int startIndex)

   String substring(int startIndex, int endIndex)

2. The first form returns the substring that starts at startIndex and runs to the end of the invoking StringBuffer object.

3.The second form returns the substring that starts at startIndex and runs through endIndex–1. These methods work just like those defined for String that were described earlier.

Difference between StringBuffer and StringBuilder.

1. J2SE 5 adds a new string class to Java's already powerful string handling capabilities. This new class is called StringBuilder.

2. It is identical to StringBuffer except for one important difference: it is not synchronized, which means that it is not thread-safe.

3. The advantage of StringBuilder is faster performance. However, in cases in which you are using multithreading, you must use StringBuffer rather than StringBuilder.

Additional Methods in String which was included in Java 5

1. **int codePointAt(int i)**

Returns the Unicode code point at the location specified by i.

2. **int codePointBefore(int i)**

   Returns the Unicode code point at the location that precedes that specified by i.

3. **int codePointCount(int start , int end)**

   Returns the number of code points in the portion of the invoking String that are between start and end– 1.

4. **boolean contains(CharSequence str)**

   Returns true if the invoking object contains the string specified by str . Returns false, otherwise.

5. **boolean contentEquals(CharSequence str)**

   Returns true if the invoking string contains the same string as str. Otherwise, returns false.

6. **boolean contentEquals(StringBuffer str)**

   Returns true if the invoking string contains the same string as str. Otherwise, returns false.

7. **static String format(String fmtstr , Object ... args)**
   Returns a string formatted as specified by fmtstr.

8. **static String format(Locale loc , String fmtstr , Object ... args)**

   Returns a string formatted as specified by fmtstr.

9. **boolean matches(string regExp)**

Returns true if the invoking string matches the regular expression passed in regExp. Otherwise, returns false.

10. int offsetByCodePoints(int start , int num)

Returns the index with the invoking string that is num code points beyond the starting index specified by start.

11. **String replaceFirst(String regExp , String newStr)**

Returns a string in which the first substring that matches the regular expression specified by regExp is replaced by newStr.

12. **String replaceAll(String regExp , String newStr)**

Returns a string in which all substrings that match the regular expression specified by regExp are replaced by newStr

13. **String[ ] split(String regExp)**

Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in regExp.

14. **String[ ] split(String regExp , int max)**

Decomposes the invoking string into parts and returns an array that contains the

result. Each part is delimited by the regular expression passed in regExp. The number of pieces is specified by max. If max is negative, then the invoking string is fully decomposed. Otherwise, if max contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If max is zero, the invoking string is fully decomposed.

15. **CharSequence subSequence(int startIndex , int stopIndex)**

Returns a substring of the invoking string, beginning at startIndex and stopping at stopIndex . This method is required by the CharSequence interface, which is now implemented by String.

Additional Methods in StringBuffer which was included in Java 5

StringBuffer appendCodePoint(int ch)

Appends a Unicode code point to the end of the invoking object. A reference to the object is returned.

int codePointAt(int i)

Returns the Unicode code point at the location specified by i.

int codePointBefore(int i)

Returns the Unicode code point at the location that precedes that specified by i.

int codePointCount(int start , int end)

Returns the number of code points in the portion of the invoking String that are between start and end– 1.

int indexOf(String str)

Searches the invoking StringBuffer for the first occurrence of str. Returns the index of the match, or –1 if no match is found.

int indexOf(String str , int startIndex)

Searches the invoking StringBuffer for the first occurrence of str, beginning at startIndex. Returns the index of the match, or –1 if no match is found.

int lastIndexOf(String str)

Searches the invoking StringBuffer for the last occurrence of str. Returns the index of the match, or –1 if no match is found.

int lastIndexOf(String str , int startIndex)

Searches the invoking StringBuffer for the last occurrence of str, beginning at startIndex. Returns the index of the match, or –1 if no match is found.

Module- 4 Servlet

Introdution to servlet

Servlet is small program that execute on the server side of a web connection. Just as applet extend the functionality of web browser the applet extend the functionality of web server.

In order to understand the advantages of servlet, you must have basic understanding of how web browser communicates with the web server.

Consider a request for static page. A user enters a URL into browser. The browser generates http request to a specific file. The file is returned by http response. Web server map this particular request for this purpose. The http header in the response indicates the content. Source of web page as MIME type of text/html.

1. What are the Advantage of Servlet Over "Traditional" CGI?

Java servlet is more efficient, easier to use, more powerful, more portable, and cheaper than traditional CGI and than many alternative CGI-like technologies. (More importantly, servlet developers get paid more than Perl programmers :-).

- **Efficient.** With traditional CGI, a new process is started for each HTTP request. If the CGI program does a relatively fast operation, the overhead of starting the process can dominate the execution time. With servlets, the Java Virtual Machine stays up, and each request is handled by a lightweight Java thread, not a heavyweight operating system process. Similarly, in traditional CGI, if there are *N* simultaneous request to the same CGI program, then the code for the CGI program is loaded into memory N times. With servlets, however, there are *N* threads but only a single copy of the servlet class.

- **Convenient.** Hey, you already know Java. Why learn Perl too? Besides the convenience of being able to use a familiar language, servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, and many other such utilities.

Powerful. Java servlets let you easily do several things that are difficult or impossible with regular CGI. For one thing, servlets can talk directly to the Web server (regular CGI programs can't). This simplifies operations that need to look up images and other data stored in standard places. Servlets can also share data among each other, making useful things like database connection pools easy to implement. They can also maintain

information from request to request, simplifying things like session tracking and caching

of previous computations.

\} **Portable.** Servlets are written in Java and follow a well-standardized API. Consequently,

servlets written for, say I-Planet Enterprise Server can run virtually unchanged on Apache, Microsoft IIS, or Web Star. Servlets are supported directly or via a plug in on almost every major Web server.

\} **Inexpensive.** There are a number of free or very inexpensive Web servers available that are good for "personal" use or low-volume Web sites. However, with the major exception of Apache, which is free, most commercial-quality Web servers are relatively expensive. Nevertheless, once you have a Web server, no matter the cost of that server, adding servlet support to it (if it doesn't come preconfigured to support servlets) is generally free or cheap.

2. What is servlet? What are the phases of servlet life cycle? Give an example.

Servlets are small programs that execute on the server side of a web connection. Just as applet extend the functionality of web browser the applet extend the functionality of web server.

Servlet class is loaded.

Servlet class is loaded when first request to web container.

servlet instance is created:

Web container creates the instance of servlet class only once.

init method is invoked:

It class the init method when it loads the instance. It is used to intialise servlet.

Syntax of init method is

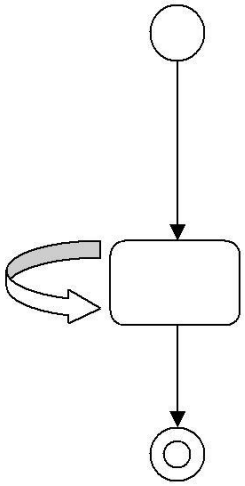
public void init(ServletConfig config) throws ServletException

Service method is invoked:

Web container calls service method each time when request for the servlet is received. If servlet is not initialized it calls init then it calls the service method. Syntax of service method is as follows

- *public void service(Servlet request, ServletResponse response) throws ServletException, IOException*
-
- Destroy method is invoked.
-
- The web container calls the destroy method before it removes the servlet from service. It gives servlet an opportunity to clean up memory, resources etc. Servlet destroy method has following syntax.
-

public void destroy().



Ready

There are three states of servlet new, ready, end. It is in new state when servlet is created. The servlet instance is created when it is in new state. After invoking the init () method servlet comes to ready state. In ready state servlet invokes destroy method it comes to end state.

3. Explain about deployment descriptor

Deployment descriptor is a file located in the WEB-INF directory that controls the behavior of a java servlet and java server pages. The file is called the web.xml file and contains the header, DOCTYPE, and web app element. The web app element should contain a servlet element with three elements. These are servlet name, servlet class, and init-param.

The servlet name elements contain the name used to access the java servlet. The servlet class is the name of the java servlet class. init-param is the name of an initialization parameter that is used when request is made to the java servlet.

Example file:

<?xml version=”1.0” encoding-“ISO-8859=1”?>…..XML header

< !DOCTYPE web-app PUBLIC “~//Sun Microsystems, Inc.?? DTD Web

Application2.2//EN”> ..doctype

```
<web-app>

        <servlet>                    <servlet-
        name>MyJavaservlet</servlet-name>

        <servlet-class>myPackage.MyJavaservletClass</servlet-class>
        <init-param><param-name>parameter1</param-name>
                <param-value>735</param-value>

        </init-param>
        </servlet>

</web-app>
```

\{ **How to read data from client in servlet?**

A client uses either the GET or POST method to pass information to a java servlet. Depending on the method used by the client either doGet() or doPost() method is called in servlet.

Data sent by a client is read into java servlet by calling getParameter() method of HttpservletRequest() object that instantiated in the argument list of doGet() method and doPost() method.

getParameter() requires one argument, which is the name of parameter that contains the data sent by the client. getParameter() returns the String object.

String object contains the value assigned by the client. An empty string object is returned when it does not assign a value to the parameter. Also a null is retuned when parameter is not returned in the client.

getParameterValues() used to return the array of string objects.

Example code

Html code that calls a servlet:

```
<FORM ACTION=”/servlet/myservlets.js2”>

Enter Email Address :< INPUT TYPE=”TEXT” NAME=”email”>
<INPUT TYPE=”SUBMIT”>
```

```
            </FORM>
        import          java.io.*;
        import javax.servlet.*;

        import javax.servlet.http.*;
        public class js2 extends Httpservlet {

public void doGet(HttpservletRequset request,HttpservletResponse response)
        throws servletException , IOException {

        //String                                email;
        //Email=request.getParameter("email");
        Respose.setContentType("text/html");

        PrinterWriter   pw=response.getWriter();
        pw.println("<HTML>\n" +

        "HEAD><TITLE> Java Servlet</TITLE></HEAD>\n" +
        "<BODY>\n"+

        //"<p>MY Email Address :" +email +"</p>\n" +
        <h1> My First Servlet

        "</BODY>\n"  +
        </HTML>");

 }
}
```

5. How to read HTTP Request Headers?

A request from client contains two components these are implicit data, such as email address explicit data at HTTP request header. Servlet can read these request headers to process the data component of the request.

Example of HTTP header:
Accept: image.jpg, image.gif,*/*
Accept- Encoding: Zip
Cookie: CustNum-12345
Host:www.mywebsite.com
Referer: http://www.mywebsite.com/index.html

The uses of HTTP header:
Accept: Identifies the mail extension

Accept-Charset : Identifies the character set that can be used by browser.
Cookie returns the cookies to server.

Host: contains host portal.

Referrer: Contains the URL of the web page that is currently displayed in the browser.

A java servlet can read an HTTP request header by calling the getHeader() method of the HttpservletRequest object. getHeader() requires one argument which is the name of the http request header.

getHeader()

6. How to send data to client and writing the HTTP Response Header?

A java servlet responds to a client's request by reading client data and HTTP request headers, and then processing information based on the nature of the request.

For example, a client request for information about merchandise in an online product catalog requires the java servlet to search the product database to retrieve product information and then format the product information into a web page, which is returned to client.

There are two ways in which java servlet replies to client request. These are sent by sending information to the response stream and sending information in http response header. The http response header is similar to the http request header.

Explicit data are sent by creating an instance of the PrintWriter object and then using println() method to transmit the information to the client.

Implcit data example: HTTP/1.1 200 OK
Content-Type:text/plain

My Response

Java servlet can write to the HTTP response header by calling setStatus() method requires one argument which is an integer that represent the status code.

Response.setStatus(100);

7. Explain about Cookies in servlet.

Cookies are text files stored on the client computer and they are kept for various information tracking purpose. Java Servlets transparently supports HTTP cookies.

There are three steps involved in identifying returning users:

- Server script sends a set of cookies to the browser. For example name, age, or identification number etc.
- Browser stores this information on local machine for future use.
- When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user.

Setting Cookies with Servlet:

Setting cookies with servlet involves three steps:

(1) Creating a Cookie object: You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

Cookie cookie = new Cookie("key","value");
(2) Setting the maximum age: You use setMaxAge to specify how long (in seconds) the cookie should be valid. Following would set up a cookie for 24 hours.

cookie.setMaxAge(60*60*24);

(3) Sending the Cookie into the HTTP response headers: You use **response.addCookie** to add cookies in the HTTP response header as follows:

response.addCookie(cookie);

Writing Cookie

```
import java.io.*;
import
javax.servlet.*;

import javax.servlet.http.*;

public class HelloForm extends HttpServlet {
 public void doGet(HttpServletRequest
 request,

            HttpServletResponse response)
        throws ServletException, IOException

 {

   Cookie myCookie = new Cookie("user id",
   123); myCookie.setMaxAge(60*60);
   response.addCookie(myCookie );
   response.setContentType("text/html");
```

```
    PrintWriter out = response.getWriter();

        out.println( "<html>\n" +

            "<head><title>" + My Cookie + "</title></head>\n" +

         "<nody>\n" +

         "<h1>+ My Cookie +"<h1>\n" +
         "<p> Cookie Written + </p>\n"
         + "</body></HTML>");

}

 }
```

Reading Cookies with Servlet:

To read cookies, you need to create an array of *javax.servlet.http.Cookie* objects by calling the **getCookies( )** method of *HttpServletRequest*. Then cycle through the array, and use getName() and getValue() methods to access each cookie and associated value.

Example:Let us read cookies which we have set in previous example:

```
import java.io.*;import javax.servlet.*;import javax.servlet.http.*;
 public class ReadCookies extends HttpServlet {

 public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException

  {

     Cookie cookie;

         Cookie[]  cookies;

     cookies        =        request.getCookies();
     response.setContentType("text/html");
     PrintWriter  out  =  response.getWriter();
     String title = "Reading Cookies Example";
     out.println( "<html>\n" +

           "<head><title>" + title + "</title></head>\n" );
               if( cookies != null ){

          out.println("<h2>      Found      Cookies      Name      and
           Value</h2>"); for (int i = 0; i < cookies.length; i++){
```

```
            cookie = cookies[i];

           out.print("Name : " + cookie.getName( ) + ", ");
        out.print("Value: " + cookie.getValue( )+" <br/>");

}
   }else{   out.println( "<h2>No cookies founds</h2>");

   }     out.println("</body>");     out.println("</html>");

 }}
```

8. Explain Session Tracking:

1. A session is created each time a client requests service from a java servlet. The java. The java servlet processes the request and response accordingly, after which the session is terminated. Many times the same client follows with another request to the same client follows with another request to the same java servlet, java servlet requires information regarding the previous session to process request.

2. However , HTTP is stateless protocol, meaning that there is not hold over from the previous sessions.

3. Java servlet is capable of tracking sessions by using Httpsession API. It determines if the request is a continuation from an existing session or new session.

4. A java servlet calls a getSession() method of HttpservletRequset object, which returns a session object if it is a new session. The getSession() method requires one argument which is Boolean true. Returns session object.

   Syntax :

   HttpSession s1=request.getSession(true);

JSP program

A jsp is java server page is server side program that is similar in design and functionality to a java servlet.

A JSP is called by a client to provide web services, the nature of which depends on client application.

A jsp is simpler to create than a java servlet because a jsp is written in HTML rather than with the java programming language. . There are three methods that are automatically called

when jsp is requested and when jsp terminates normally. These are the jspInit() method , the jspDestroy()m ethod and service() method.

A jspInit() is identical ti init() method of java servlet. It is called when first time jsp is

called.

A jspDestroy() is identical to destroy() method of servlet. The destroy() method is automatically called when jsp erminates normally.It is not called when jsp terminates abruptly. It is used for placing clean up codes.

1. Explain JSP tags(repeated question)

A jsp tag consists of a combination of HTML tags and JSP tags. JSP tags define java code that is to be executed before the output of jsp program is sent to the browser.

A jsp tag begin with a <%, which is followed by java code , and wnds with %>,

There ia an XML version of jsp tag <jap:TagId></jsp:TagId>

A jsp tags are embedded into the HTML component of a jsp program and are processed by Jsp virtual engine such as Tomcat.

Java code associated with jsp tag are executed and sent to browser.

There are five types of jsp tags :

Comment tag :A comment tag opens with <%-- and close with --%> and is follwed by a comment that usually describes the functionality of statements that follow a comment tag.

Declaration statement tags: A declartion statement tag opens with <%! And is followed by declaration statements that define the variables, object, and methods that are avilabe to other component of jsp program.

Directive tags: A directive tag opens with <%@ and commands the jsp virtual engine to perform a specific task, such as importing java package required by objects and methods used in a declaration statement. The directive tag closes with %> . There are commonly used in directives import, include , and taglib. The import tag is used to import java packages into the jsp program. Include is used for importing file. Taglib is used for including file.

Example:

<%@ page import=”import java.sql.*” ; %>

<%@ include file=”keogh\books.html” %>

<%@ taglib url=”myTags.tld” ; %>

Expression tags: An expression tag opens with <%= and is used for an expression statement whose result page replaces the expression tag when the jsp virtual engine resolves JSP tags. An expression tag closes with %>

Scriptlet tag: A sciptlet tag opens with <% and contains commonly used java control statements and loops. And Scriptlet tag closes with %>

2. How variables and objcts declared in JSP program?

You can declare java variables and objects that are used in a JSP program by using the same codin technique used to declare them in java. JSP declaration statements must appear as jsp tag

Ex:

```
<html>
<head>
    <title> Jsp Programming </title>
<.head>
<body>
    <%! Int age=29; %><p> Your age is : <%=age%> </p>
</body>
</html>
```

3. How method are declared and used in jsp programs?

Methods are defined same way as it is defined in jsp program, except these are placed in JSP tag.methods are declared in JSP decalration tag. The jsp calls method in side the expression tag.

Example:

<html>

```
<head>

        <title>                    Jsp
programming</title> </head>

<body>

<%! int add(int n1, int n2)

        {

        int      c;
        c=a+b;
        return c;
        }

%>

<p> Addition of two numbers : <%= add(45,46)%> </p>
</body></html>
```

5. Explain the control statements of JSP with example program:

One of the most powerful features avilable in JSP is the ability to change the flow of the program to truly create dynamic content for a web based on conditions received from the browser.

Ther are two control statements used to change the flow of program are “if “ and “switch” statement , both of which are also used to direct the flow of a java program.

Ex:

```
<html>

<head>

        <title> JSP Programming </title>
</head>

<body>

        <%! int grade=26; %>
        </body>

<% if(grade >69) { %>
```

```
            <p> You Passed !</p>
        <% } else { %>

        <p> Better Luck Next Time</p>
        <% } %>

        </body>

    </html>
```

6. Looping Statement of JSP

Jsp loops are nearly identical to loops that you use in your java program except you can repeat the html tags

There are three kind of jsp loop that are commonly used in jsp program. Ex: for loop , while loop , do while .

Loop plays an important role in JSP database program. The following program is example for "FOR LOOP".

```
<html><head><title>For Loop Example</title></head>

<body>
 <%

   for (int i = 0; i < 10;i++) {

 %>

<p> Hello World</p>

 <% }  %> </body>

</html>
```

// **Explain Requset String generated by browser. how to read a request string in jsp?**

A browser generate requst string whenever the submit button is selected. The user requests the string consists of URL and the query the string.

Example of request string: http://www.jimkeogh.com/jsp/?fname=" Bob" & lname ="Smith"

Your jsp programneeds to parse the query string to extract the values of fields

that are to be processed by your program. You can parse the query string by using the methods of the request object.

getParameter(Name) method used to parse a value of a specific field that are to be processed by your program

code to process the requset string

```
<%! String FirstName =requst.getParameter(fname);
    String LastName =requst.getParameter(lname); %>
```

5. Copying from multivalued field such as selection list field can be tricky multivalued fields are handled by using getParameterValues()

6. Other than requset string url has protocols, port no, the host name

7. **Write the JSP program to create and read cookie called "EMPID" and that has value "AN2536".**

Cookie is small piece of information created b a JSP program that is stored on the clien's hard dik by the browser. Cookie isare used to store various kind of information, such as user preference. The cookies are created by using Cookie class.

Create cookie:

```
<html>
<head>

<title>                creating
cookie</title> </head>

<body>
<%! String MyCookieName="EMPID";
      String UserValue="AN2536";
%>
</body>
</html>
```

Reading Cookie:

```
<html>
<head>

<title>reading cookie </title>
</head>

<body>
<% String myCookieName="EMPID";
```

```
String  myCookieValue;
String CName, CValue;
int found=0;

Cookie[] cookies=request.getCoookies(); for(
        int     i=0;i<cookies.length;i++)     {
        CName=          cookies[i].getName();
        CValue          =cookies[i].getValue();
        If(myCookieName.equals(CName))  {
        found=1;

        myCookieValue=Cvalue; } }
        If(found== 1) { %>

        <p> Cookie Name = < %= CName %> </p>
        <p> Cookie Value = < %= CValue %> </p>
        <% } %> </body></html>
```

8. Explain steps to configure tomcat.

i. Jsp program programs are executed by a JSP virtual machine that run on a web server.

ii. We can download and install JSP virtual machine.

iii. Installation Steps

Connect to Jakarta.apache.org.
Select down load

Select Binaries to display the binary Download Page.
Create a folder from the root directory called tomcat.
Download latest release.
Unzip Jakarta-tomcat.zip.

The extraction process creates the following folder in the tomcat directory: bin, conf, doc, lib, src, and webapps

Modify the batch file , which is located in the \tomcat\bin folder. Change the JAVA_HOME variable is assigned the pathe where JDK is installed on your computer.

Open dos window and type \tomcat\bin\tomcat to start Tomcat.

Open your browser. Enter http://localhost:8080.
Tomcat home page is displayed on the screen verifying that Tomcat is

running.

9. Explain how session objects are created.

A JSP database system ia able to share information among JSP programs within a session by using a session object. Each time a session is created , a unique ID is assigned to the session and stored as acookie.

A unique ID enables JSP program to track multiple session session simultaneously while maintaining data integrity of each session. The session ID is used to prevent the intermingling of each session.

Create session Object:

```
<html> <head><title> Jsp Session</title></head>

<body>

<% ! String AtName=”Product”;

    String AtValue =”1234”;

    Session.setAttributes(AtName, AtValue);

%></body></html>
```

In session object we can store information about purchases as session attributes can be retrived and modified each time the jsp program runs. setAttributes() used for creating attributes.

Read Session Object:

getAttributeNames() methos returns names of all the attribbutes as Enumeration , the attributes are processsed.

```
<html> <head><title> Jsp Session</title></head>

<body><% !

    Enumaration purchases=session.getAttributeNames();

    String AtName=(String) attributeNames.nextElement();

    String AtValu=(String) session.getAttribute(AtName); %>

    <p> Attirbute Name <%= AtName %> </p>
```

```
<p> Attribute Value <% = Atvalue %> </p>

<% } %> %></body></html>
```

Module -5

The Concept of JDBC:

- Java was not considered industrial strength programming language since java was unable to access the DBMS.
- Each dbms has its own way to access the data storage. low level code required to access oracle data storage need to be rewritten to access db2.
- JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases
- JDBC drivers has to do the following

  Open connection between DBMS and J2EE environment.

  Translate low level equivalents of sql statements sent by J2EE component into messages that can be processed by the DBMS.

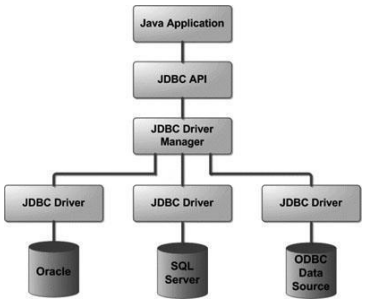
  Return the data that conforms to JDBC specifications to the JDBC driver

  Return the error messages that conforms to JDBC specifications to the JDBC driver

  Provides transaction management routines that conforms to JDBC specifications to the JDBC driver

  Close connection between the DBMS and the J2EE component.

JDBC Architecture



June 2012 (Briefly discuss the various JDBC driver types 10 M)

JDBC Driver Types
Type 1 driver JDBC to ODBC Driver

1. It is also called JDBC/ODBC Bridge , developed by MicroSoft.
2. It receives messages from a J2EE component that conforms to the JDBC specifications
3. Then it translates into the messages understood by the DBMS.
4. This is DBMS independent database program that is ODBC open database connectivity.

Type 2 JAVA / Native Code Driver

5. Generates platform specific code that is code understood by platform specific code only understood by specific databases.
6. Manufacturer of DBMS provides both java/ Native code driver.
7. Using this provides lost of portability of code.
8. It won't work for another DBMS manufacturer

Type 3 JDBC Driver

- Most commonly used JDBC driver.
- Coverts SQL queries into JDBC Formatted statements.
- Then JDBC Formatted statements are translated into the format required by the DBMS.
- Referred as Java protocol

Type 4 JDBC Driver

9. Referred as Type 4 database protocol
10. SQL statements are transferred into the format required by the DBMS.
11. This is the fastest communication protocol.

JDBC Packages

JDBC API contains two packages. First package is called java.sql, second package is called javax.sql which extends java.sql for advanced JDBC features.

Explain the various steps of the JDBC process with code snippets.

1. Loading the JDBC driver
    - The jdbc driver must be loaded before the J2EE compnet can be connected to

the database.

- Driver is loaded by calling the method and passing it the name of driver

Class.forName("sun:jdbc.odbc.JdbcOdbcDriver");

7. Connecting to the DBMS.

Once the driver is loaded , J2EE component must connect to the DBMS using DriverManager.getConnection() method.

It is highest class in hierarchy and is responsible for managing driver information.
It takes three arguments URL, User, Password

It returns connection interface that is used through out the process to reference a database

String url="jdbc:odbc:JdbcOdbcDriver";
String userId="jim"
String password="Keogh";

Statement DatRequest;
Private Connection db;

try{
Class.forName("sun:jdbc.odbc.JdbcOdbcDriver");
Db=DriverManager.getConnection(url,userId,password);
}

8. Creating and Executing a statement.

The next step after the JDBC is loaded and connection is successfully made with a particular database managed by the dbms, is to end a particular query to the DBMS for processing.

SQL query consists series of SQL command that direct DBMS to do something example Return rows.

Connect.createStatement() method is used to create a statement Object.

The statement object is then used to execute a query and return result object that contain response from the DBMS

Statement DataRequest;
ResultSet Results;
try {

```
                    String query="select * from Customers";

                    DataRequest=Database.createStatement();
                    Results= DataRequests.executeQuery(query);
            }
```

9. Processing data returned by the DBMS

java.sql.ResultSet object is assigned the result received from the DBMS after the query is processed.

java.sql.ResultSet contain method to interct with data that is returned by the DBMS to the J2EE Component.

```
Results= DataRequests.executeQuery(query);
do

{
Fname=Results.getString(Fname)
}
While(Results.next())
```

In the above code it return result from the query and executes the query. And getString is used to process the String retrived from the database.

5. Terminating the connection with the DBMS.
   To terminate the connection Database.close() method is used.

With proper syntax, explain three types of getConnection() method.

1 After the JDBC driver is successfully loaded and registered, the J2EE component must connect to the database. The database must be associated with the JDBC driver.

2 The datasource that JDBC component will connect to is identified using the URL format. The URL consists of three format.

- These are jdbc which indicate jdbc protocol is used to read the URL.
- <subprotocol> which is JDBC driver name.
- <subname> which is the name of database.

3 Connection to the database is achieved by using one of three getConnection()

methods. It returns connection object otherwise returns SQLException

4 Three getConnection() method

- getConnection(String url)
- getConnection(String url, String pass, String user)
- getConnection(String url, Properties prop)

5 getConnection(String url)

- Sometimes the DBMS grant access to a database to anyone that time J2EE component uses getConnection(url) method is used.

```
String url=" jdbc:odbc:JdbcOdbcDriver ";
try{

Class.forName("sun:jdbc.odbc.JdbcOdbcDriver");
Db=DriverManager.getConnection(url);
}
```

6 getConnection(String url, String pass, String user)

- Database has limited access to the database to authorized user and require J2EE to supply user id and password with request access to the database. The this method is used.

```
try{
Class.forName("sun:jdbc.odbc.JdbcOdbcDriver");
Db=DriverManager.getConnection(url,userId,password);
}
```

- **getConnection(String url, Properties prop)**

  There might be occasions when a DBMS require information besides userid and password before DBMS grant access to the database.

- This additional information is called properties and that must be associated with Properties object.
- The property is stored in text file. And then loaded by load method of Properties class.

```
Connection db;

Properties  props=new  Properties();
```

```
try {

 FileInputStream inputfile=new FileInputStream("text.txt");
Prop.load(inputfile);
}
```

Write short notes on Timeout:

4. Competition to use the same database is a common occurrence in the J2EE environment and can lead to performance degradation of J2EE application

5. Database may not connect immediately delayed response because database may not available.

6. Rather than delayed waiting timeJ2EE component can stop connection. After some time. This time can bet set with the following method:

   DriverManager.setLoginTimeout(int sec).

7. **DriverManager.getLoginTimeout(int sec)** return the current timeout in seconds.

Explain Connection Pool

B Client needs frequent that needs to frequently interact with database must either open connection and leave open connection during processing or open or close and reconnect each time.

C Leaving the connection may open might prevent another client from accessing the database when DBS have limited no of connections. Connecting and reconnecting is time consuming.

D The release of JDBC 2.1 Standard extension API introduced concept on connection pooling

E A connection pool is a collection of database connection that are opened and loaded into memory so these connection can be reused with out reconnecting to the database.

F DataSource interface to connect to the connection pool. connection pool is implemented in application server.

G There are two types of connection to the database 1.) Logical 2.) Physical
H The following is code to connect to connection pool.

```
Context ctext= new IntialContext()
DataSource pool =(DataSource) ctext.lookup("java:comp/env/jdbc/pool");
```

Connection db=pool.getConnection();

Briefly explain the Statement object. Write program to execute a SQL statement.(10)

1. Statement object executes query immediately with out precompiling.

2. The statement object contains the **excuteQuery()** method , which accept query as argument then query is transmitted for processing.It returns ResultSet as object.

Example Program

```
String url="jdbc:odbc:JdbcOdbcDriver";
String userId="jim"
String password="Keogh";
Statement DatRequest;
Private Connection db;
ResultSet rs;

// code to load driver

//code to connect to the
 database try{

String query="SELECT * FROM Customers;

DatRequest=Db.createStaement();
rs=DatRequest.executeQuery(query);// return result set
object
}catch(SQLException  err)

{
System.err.println("Error");
System.exit(1);
}
```

3. Another method is used when DML and DDL operations are used for processing query is executeUpdate(). This returns no of rows as integer.

```
try{

String query="UPDATE Customer set PAID='Y' where BALANCE ='0';
DatRequest=Db.createStaement();

int n=DatRequest.executeUpdate(query);// returns no of rows updated
}catch(SQLException err)
{
```

```
System.err.println("Error");
System.exit(1);

}
```

Briefly explain the prepared statement object. Write program to call a stored procedure.(10)

1. A SQL query must be compiled before DBMS processes the query. Query is precompiled and executed using Prepared statements.

2. Question mark is placed as the value of the customer number. The value will be inserted into the precompiled query later in the code.

3. Setxxx() is used to replace the question mark with the value passed to the setxxx() method . xxx represents data type of the field.

   Example if it is string then setString() is used.

4. It takes two arguments on is position of question mark and other is value to the filed.

5. This is referred as late binding.

```
String url="jdbc:odbc:JdbcOdbcDriver";
String userId="jim"
String password="Keogh";
ResultSet rs;

// code to load driver

//code to connect to the database
 try{

String query="SELECT * FROM Customers where cno=?";
PreparedStatement pstatement=db.preparedStatement(query);
pstatement.setString( 1,"123"); // 1 represents first place holder, 123 is value
rs= pstatement.executeQuery();


}catch(SQLException err)
{
System.err.println("Error");
System.exit(1);
}
```

Briefly explain the callable statement object. Write program to call a stored procedure.(10)

1. The callableStatement object is used to call a stored procedure from with in J2EE object. A stored procedure is block of code and is identified by unique name. the code can bewritten in Transact-C ,PL/SQL.

2. Stored procedure is executed by invoking by the name of procedure.

3. The callableStatement uses three types of parameter when calling stored procedure. The parameters are IN ,OUT,INOUT.

4. IN parameter contains data that needs to be passed to the stored procedure whose value is assigned using setxxx() method.
5. OUT parameter contains value returned by stored procedure.the OUT parameter should be registers by using registerOutParameter() method and then later retrieved by the J2EE component using getxxx() method.

6. INOUT parameter is used to both pass information to the stored procedure and retrieve the information from the procedure.

7. Suppose, you need to execute the following Oracle stored procedure:

   CREATE OR REPLACE PROCEDURE getEmpName

   (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS

   BEGIN

   SELECT first INTO EMP_FIRST FROM Employees WHERE ID = EMP_ID;

   END;

8. The following code snippets is used

   ```
   CallableStatement cstmt = null;

       try {   String SQL = "{call getEmpName (?, ?)}";

               cstmt = conn.prepareCall (SQL);catch (SQLException e) { }
   ```

9. Using CallableStatement objects is much like using PreparedStatement objects. You must bind values to all parameters before executing the statement, or you will

receive an SQLException.

10. If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.

11. When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type to the data type the stored procedure is expected to return.

12. Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

ResultSet

1. ResultSet object contain the methods that are used to copy data from ResultSet into java collection object or variable for further processing.

8. Data in the ResultSet is logically organized into the virtual table for further processing. Result set along with row and column it also contains meta data.

9. ResultSet uses virtual cursor to point to a row of the table.

10. J2EE component should use the virtual cursor to each row and the use other methods of the ResultSet to object to interact with the data stored in column of the row.

11. The virtual cursor is positioned above the first row of data when the ResultSet is returned by executeQuery () method.

12. The virtual cursor is moved to the first row with help of next() method of ResultSet

13. Once virtual cursor is positioned getxxx() is used to return the data. Data type of data is represents by xxx. It should match with column data type.

14. getString(fname) …..fname is column name.

15. setString(1)…….. in this 1 indicates first column selected by query.

```
stmt                    =
conn.createStatement();
String sql;

sql = "SELECT id, first, last, age FROM
```

```
Employees"; ResultSet rs = stmt.executeQuery(sql);
while(rs.next()){

  int id = rs.getInt("id");/ /
  rs.getInt(1); int age =
  rs.getInt("age");

  String first = rs.getString("first");

  String last = rs.getString("last");

  System.out.print("ID: " + id);

  System.out.print(", Age: " + age);
  System.out.print(", First: " + first);

  System.out.println(", Last: " + last);

}
```

Explain the with an example Scrollable Result Set (6 Marks)

9. Until the release of JDBC 2.1 API , the virtual cursor can move only in forward directions. But today the virtual cursor can be positioned at a specific row.

10. There are six methods to position the cursor at specific location in addition to next() in scrollable result set. firs() ,last(), absolute(), relative(), previous(), and getRow().

11. first() …… position at first row.

12. last()……position at last row.

13. previous()…….position at previous row.

14. absolute()……. To the row specified in the absolute function

15. relative()…… move relative to current row. Positive and negative no can be given.

    Ex. relative(-4) … 4 position backward direction.

8. getRow() …… returns the no of current row.

9. There are three constants can be passed to the createStatement()

10. Default is TYPE_FORWARD_ONLY. Otherwise three constant can be passed to the

create statement 1.) TYPE_SCROLL_INSENSITIVE

2.) TYPE_SCROLL_SENSITIVE

11. TYPE_SCROLL makes cursor to move both direction. INSENSITIVE makes changes made by J2EE component will not reflect. SENSITIVE means changes by J2EE will reflect in the result set.

    Example code.

    String sql=" select * from emp";

    DR=Db.createStatement(TYPE_SCROLL_INSENSITIVE);
    RS= DR.executeQuery(sql);

12. Now we can use all the methods of ResultSet.

Explain the with an example updatable Result Set.

1. Rows contained in the result set is updatable similar to how rows in the table can be updated. This is possible by sending CONCUR_UPDATABLE.

2. There are three ways in which result set can be changed. These are updating row , deleting a row, inserting a new row.

3. **Update ResultSet**

B. Once the executeQuery() method of the statement object returns a result set. updatexxx() method is used to change the value of column in the current row of result set.

C. It requires two parameters, position of the column in query. Second parameter is value

D. updateRow() method is called after all the updatexxx() methods are called.

Example:

```
try{
        String query= "select Fname, Lname from Customers
        where Fname= 'Mary' and Lanme='Smith';

        DataRequest= Db.
        createStatement(ResultSet.CONCUR_UPDATABLE);

        Rs= DataRequest.executeQuery(query);
```

```
                    Rs.updateString("LastName","Smith");
                    Rs.updateRow();
                }
```

5. **Delete row in result set**

   By using absolute method positioning the virtual cursor and calling deleteRow(int n) n is the number of rows to be deleted.

   Rs.deleteRow(0) current row is deleted.

6. **Insert Row in result set**

   Once the executeQuery() method of the statement object returns a result set. updatexxx() method is used to insert the new row of result set.

   It requires two parameters, position of the column in query. Second parameter is value

   insertRow() method is called after all the updatexxx() methods are called.

```
            try{
                String query= "select Fname, Lname from Customers
                where Fname= 'Mary' and Lanme='Smith';

                DataRequest= Db.
                createStatement(ResultSet.CONCUR_UPDATABLE);

                Rs= DataRequest.executeQuery(query);

                Rs.updateString(1 ,"Jon");
                Rs.updateString(2 ,"Smith");
                Rs.insertRow();
            }
```

6. Whatever the changes making will affect only in the result set not in the table. To update in the table have to execute the DML(update, insert, delete) statements.

Explain the Transaction processing with example

1. A transaction may consists of a set of SQL statements, each of which must be successfully completed for the transaction to be completed. If one fails SQL statements successfully completed must be rolled back.

2. Transaction is not completed until the J2EE component calls the commit() method of the connection object. All SQL statements executed prior to the call to commit() method can be rolled back.

3. Commit() method was automatically called in the program. DBMS has set AutoCommit feature.

4. If the J2EE component is processing a transaction then it has to deactivate the auto commit() option false.

```
try {

        DataBase.setAutoCommit(false)

        String query="UPDATE Customer set Street ='5 main Street' "+
              "WHERE FirstName ='Bob' ";

        DR=          DataBase.createStatement();
        DataRequest=DataBase.createStatement();
DataRequest.executeUpdate(query1);

DataBase.commit();

}
```

6. Transaction can also be rolled back. When not happened. Db.rollback().

7. A transaction may consists of many tasks , some of which no need to roll back . in such situation we can create a savepoints, in between transactions. It was introduced in JDBC 3.0. save points are created and then passed as parameters to rollback() methods.
8. releseSavepint() is used to remove the savepoint from the transaction.

9. Savepoint s1=DataBase.setSavePoint("sp1");to create the savepoint.

10. Database.rollback(sp1); to rollback the transaction.

Batch Execution of transaction

10. Another way to combine sql statements into a single into a single transaction and then execute the entire transaction .

11. To do this the addBatch() method of statement object. The addBatch() method receives a SQL statement as a parameter and places the SQL statement in the batch.

12. executeBatch() method is called to execute the entire batch at the same time. It returns an array that contains no of SQL statement that execute successfully.

```
String query1="UPDATE Customers SET street =' 5 th Main'" +
        "Where Fname='BoB' ";

String query2="UPDATE Customers SET street =' 10 th Main'" +
        "Where Fname='Tom' ";

Statement DR=DB.createStatement();
 DR.addBatch(query1);

 DR.addBatch(query2);

 int [] updated= DR.executeBatch();
```

Write notes on metadata interface Metadata

1. Metadata is data about data. MetaData is accessed by using the DatabaseMetaData interface.
2. This interface is used to return the meta data information about database.
3. Meta data is retrieved by using getMetaData() method of connection object.

   Database metadata

   The method used to retrieve meta data informations are

4. getDatabaseProductNAme()…returns the product name of database.
5. getUserNAme() returns the usernamr()
6. getURL() returns the URL of the databse.
7. getSchemas() returns all the schema name
8. getPrimaryKey() returns primary key
9. getTables() returns names of tables in the database

   ResultSet Metadata

   ResultSetMetaData rm=Result.getMeatData()

   The method used to retrieve meta data information about result set are

10. getColumnCount() returns the number of columns contained in result set

Data types of Sql used in setXXX() and getXXX() methods.

SQL	JDBC/Java
VARCHAR	java.lang.String
CHAR	java.lang.String
LONGVARCHAR	java.lang.String
BIT	Boolean
NUMERIC	java.math.BigDecimal
TINYINT	Byte
SMALLINT	Short
INTEGER	Int
BIGINT	Long
REAL	Float
FLOAT	Float
DOUBLE	Double
VARBINARY	byte[ ]
BINARY	byte[ ]
DATE	java.sql.Date
TIME	java.sql.Time

TIMESTAMP	java.sql.Timestamp
CLOB	java.sql.Clob
BLOB	java.sql.Blob
ARRAY	java.sql.Array
REF	java.sql.Ref

Exceptions handling with jdbc

Exception handling allows you to handle exceptional conditions such as program-defined errors in a controlled fashion.

1. When an exception condition occurs, an exception is thrown. The term thrown means that current program execution stops, and control is redirected to the nearest applicable catch clause. If no applicable catch clause exists, then the program's execution ends.

2. JDBC Exception handling is very similar to Java Excpetion handling but for JDBC.

3. There are three kind of exception thrown by jdbc methods.

4. SQLException ,SQLWarnings, DataTruncation
   SQLException

5. The most common exception you'll deal with is **java.sql.SQLException which result in SQL sysntax errors.**

6. getNextException() method returns details about the error.

7. getErrorCode() method retrieves vendor specific error codes.

   SQLWarnings

8. it throws warnings related to connection from DBMS. getWarnings() method

   of connecction object retrieves t **warnings.** getNextWarnings() **returns** subsequent warnings.

DataTruncation

9. Whenever data is lost due to truncation of the data value , a truncation exception is thrown.

Differentiate between a Statement and a PreparedStatement.

• A standard Statement is used for creating a Java representation for a literal SQL statement and for executing it on the database.

• A PreparedStatement is a precompiled Statement.

A Statement has to verify its metadata in the database every time
4. But ,the prepared statement has to verify its metadata in the database only once.

5. If we execute the SQL statement, it will go to the STATEMENT.

6. But, if we want to execute a single SQL statement for the multiple number of times, it'll go to the PreparedStatement.

Explain the classes, interface , methods available in java.sql.* package.

Java.sql.package include classes and interface to perform almost all JDBC operation such as creating and executing SQL queries

C. java.sql.BLOB --------provide support to BLOB SQL data type.

D. java.sql.Connection----- creates connection with specific data type

Methods in Connection
setSavePoint()
rollback()
commit()

setAutoCommit()

3. java.sql.CallableStatement-------- Executes stored procedures

   Methods in CallableStatement
   execute()
   registerOutParameter()
4. java.sql.CLOB ---------- support for CLOB data type.

5. java.sql.Date---------- support for Date SQL type.

6. Java.sql.Driver -----create instance of driver with the DriverManager

7. java.sql.DriverManager---- manages the data base driver

   getConnection()
   setLoginTimeout()
   getLoginTimeout()

8. java.sql.PreparedStatement—create parameterized query
   executeQuery()

   executeUpdate()

9. java.sql.ResultSet---------- it is interface to access result row by row
   rs.next()

   rs.last()
   rs.first()
10. java.sql.Savepoint------- Specify savepoint in transaction.

11. java.sql.SQLException--------- Encapsulates JDBC related exception.

12. java.sql.Statement…….. interface used to execute SQL statement.

13. java.sql.DataBaseMetaData….. returns mata data