

**ADVANCED JAVA and J2EE****Subject Code: 15CS553****Hours/Week: 3****Total Hours: 40****IA Marks: 20****Exam Hours: 3****Exam Marks: 80**

<b>MODULE-I</b>	<b>8 Hours</b>
<b>Enumerations, Autoboxing and Annotations(metadata):</b> Enumerations, Enumeration fundamentals, the values() and valueOf() Methods, java enumerations are class types, enumerations Inherits Enum, example, type wrappers, Autoboxing, Autoboxing and Methods, Autoboxing/Unboxing occurs in Expressions, Autoboxing/Unboxing, Boolean and character values, Autoboxing/Unboxing helps prevent errors, A word of Warning. Annotations, Annotation basics, specifying retention policy, Obtaining Annotations at runtime by use of reflection, Annotated element Interface, Using Default values, Marker Annotations, Single Member annotations, Built-In annotations.	
<b>MODULE-II</b>	<b>8 Hours</b>
<b>The collections and Framework:</b> Collections Overview, Recent Changes to Collections, The Collection Interfaces, The Collection Classes, Accessing a collection Via an Iterator, Storing User Defined Classes in Collections, The Random Access Interface, Working With Maps, Comparators, The Collection Algorithms, Why Generic Collections?, The legacy Classes and Interfaces, Parting Thoughts on Collections.	
<b>MODULE-III</b>	<b>8 Hours</b>
<b>String Handling :</b> 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	
<b>MODULE-IV</b>	<b>8 Hours</b>
<b>Servlet:</b> Background; The Life Cycle of a Servlet; Using Tomcat for Servlet Development; A simple Servlet; The Servlet API; The Javax.servlet Package; Reading Servlet Parameter; The Javax.servlet.http package; Handling HTTP Requests and Responses; Using Cookies; Session Tracking. <b>Java Server Pages (JSP):</b> JSP, JSP Tags, Tomcat, Request String, User Sessions, Cookies, Session Objects	
<b>MODULE-V</b>	<b>8 Hours</b>

**The Concept of JDBC;** JDBC Driver Types; JDBC Packages; A Brief Overview of the JDBC process; Database Connection; Associating the JDBC/ODBC Bridge with the Database; Statement Objects; ResultSet; Transaction Processing; Metadata, Data types; Exceptions.

### Reference / Text Book Details

Sl.No.	Title of Book	Author	Publication	Edition
1	<b>JAVA The Complete Reference</b>	Herbert Schildt	Tata McGraw Hill	7 <sup>th</sup> /9 <sup>th</sup>
2	<b>J2EE The Complete Reference</b>	Jim Keogh	Tata McGraw Hill	2007
3	<b>Introduction to JAVA Programming</b>	Y. Daniel Liang	Pearson Education, 2007	7 <sup>th</sup>
4	<b>The J2EE Tutorial</b>	Stephanie Bodoff	Pearson Education, 2004	2 <sup>nd</sup>
5	<b>Advanced JAVA Programming</b>	Uttam K Roy	Oxford University Press	2015

**Table of Contents**

<b>SL No</b>	<b>Module Description</b>	<b>Page No</b>
1	<b>Module 1 – Enumeration, AutoBoxing and Annotations</b>	<b>1-23</b>
2	<b>Module 2 – Collection Framework</b>	<b>24-57</b>
3	<b>Module 3 – String Methods</b>	<b>58-119</b>
4	<b>Module 4 – Servlets and JSP</b>	<b>120-149</b>
5	<b>Module 5 – JDBC Methods</b>	<b>150-166</b>

## Module – 1

# Enumerations, Autoboxing and Annotations

### Enumerations

Enumerations was added to Java language in JDK5. **Enumeration** means a list of named constant. In Java, enumeration defines a class type. An Enumeration can have constructors, methods and instance variables. It is created using **enum** keyword. Each enumeration constant is *public*, *static* and *final* by default. Even though enumeration defines a class type and have constructors, you do not instantiate an **enum** using **new**. Enumeration variables are used and declared in much a same way as you do a primitive variable.

---

### How to Define and Use an Enumeration

1. An enumeration can be defined simply by creating a list of enum variable. Let us take an example for list of Subject variable, with different subjects in the list.

```
enum Subject      //Enumeration defined
{
    Java, Cpp, C, Dbms
}
```

2. Identifiers Java, Cpp, C and Dbms are called **enumeration constants**. These are public, static and final by default.
3. Variables of Enumeration can be defined directly without any **new** keyword.

```
Subject sub;
```

4. Variables of Enumeration type can have only enumeration constants as value. We define an enum variable as `enum_variable = enum_type.enum_constant;`

```
sub = Subject.Java;
```

5. Two enumeration constants can be compared for equality by using the `==` relational operator.

#### Example:

```
if(sub == Subject.Java) {
    ...
}
```

---

**Example of Enumeration**

```
enum WeekDays
{ sun, mon, tues, wed, thurs, fri, sat }

class Test
{
public static void main(String args[])
{
WeekDays wk; //wk is an enumeration variable of type WeekDays
wk = WeekDays.sun; //wk can be assigned only the constants defined under
//enum type Weekdays
System.out.println("Today is "+wk);
}
}

Output :
Today is sun
```

The enum can be defined within or outside the class because it is similar to a class.

```
enum Season { WINTER, SPRING, SUMMER, FALL }
class EnumExample2{
public static void main(String[] args) {
Season s=Season.WINTER;
System.out.println(s);
}}
```

**Output:**WINTER

```
class EnumExample3{
enum Season { WINTER, SPRING, SUMMER, FALL;}//semicolon; is optional here
public static void main(String[] args) {
Season s=Season.WINTER;//enum type is required to access WINTER
System.out.println(s);
}}
```

**Output:**WINTER

**Example of Enumeration using switch statement**

```
class EnumExample5{
enum Day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
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");
}
}

```

Output:monday

### **Values( ) and ValueOf( ) method**

All the enumerations has predefined methods **values()** and **valueOf()**. values() method returns an array of enum-type containing all the enumeration constants in it. Its general form is,

**public static enum-type[ ] values()**

valueOf() method is used to return the enumeration constant whose value is equal to the string passed in as argument while calling this method. It's general form is,

**public static enum-type valueOf (String str)**

### **Example of enumeration using values() and valueOf() methods:**

```

enum Restaurants {
dominos, kfc, pizzahut, paninos, burgerking
}
class Test {
public static void main(String args[])
{
Restaurants r;
System.out.println("All constants of enum type Restaurants are:");
Restaurants rArray[] = Restaurants.values(); //returns an array of constants of type Restaurants
for(Restaurants a : rArray) //using foreach loop
System.out.println(a);

```

```

r = Restaurants.valueOf("dominos");
System.out.println("I AM " + r);
}
}

```

#### **Output:**

All constants of enum type Restaurants are:  
dominos

kfc  
pizzahut  
paninos  
burgerking  
I AM dominos

Instead of creating array we can directly obtain all values.

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

**Output:**WINTER  
SPRING  
SUMMER  
FALL

---

### Points to remember about Enumerations

1. Enumerations are of class type, and have all the capabilities that a Java class has.
  2. Enumerations can have Constructors, instance Variables, methods and can even implement Interfaces.
  3. Enumerations are not instantiated using **new** keyword.
  4. All Enumerations by default inherit **java.lang.Enum** class.
  5. As a class can only extend **one** parent in Java, so an enum cannot extend anything else.
  6. enum may implement many interfaces but cannot extend any class because it internally extends Enum class
- 

**Java Enum are class type and can contain Constructor, instance variable and Method & implement interface.**

```
enum Student {
    John(11), Jonny(10), Sam(13), Viraaj(9);
    private int age; //variable defined in enum Student
    int getage() { return age; } //method defined in enum Student
    Student(int age) //constructor defined in enum Student
    {
        this.age= age;
    }
}
```

---

```
class EnumDemo
{
    public static void main( String args[] )
    {
        Student S; //is enum variable or enum object. Constructor is called when
                    //each enum object is created.

        System.out.println("Age of Viraaj is " +Student.Viraaj.getage()+"years");

        //display all student and ages
        System.out.println("All students age is:");
        for (Student a : Student.values())
            System.out.println(a +" age is " +a.getage());

    }
}
```

**{Output :**

Age of Viraaj is 9 years

All students age is:  
John age is 11  
Jonny age is 10  
Sam age is 13  
Viraaj age is 9

In this example as soon as we declare an enum variable(*Student S*), the constructor is called, and it initializes age for every enumeration constant with values specified with them in parenthesis.

Each enum constant(John, Jonny...) has its own copy of value(age...)

Student.Viraaj.getage() returns age of Viraaj.

**Java Enum containing overloaded constructor**

```
class EnumExamp{
    enum Season{
        WINTER(5), SPRING(10), SUMMER(15), FALL;

        private int value;
        Season(int v){
            value=v;
        }
        //default constructor initializes value to -1
        Season(){
            value= -1;
        }
}
```

```

}

public static void main(String args[]){
    //printing enum constant and its value
    System.out.println("Season is "+ Season.SUMMER+ " value is "+ Season.SUMMER.value);
//printing all enum constant and its value
for (Season s : Season.values())
System.out.println(s+" "+s.value);

}} output
Season is SUMMER value is 15
WINTER 5
SPRING 10
SUMMER 15
FALL -1

```

### **Enumerations Inherits Enum**

All enumerations automatically inherit **java.lang.Enum**. The **Enum** class defines several methods such as **ordinal()**, **compareTo()**, **equals()** and so on, that are available for use by all enumerations. You can 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. This method returns the ordinal value of the invoking constant. Ordinal values begin at '**0**'.

#### **final int ordinal()**

```

//example using ordinal() method
enum Season { WINTER, SPRING, SUMMER, FALL }
class EnumExample2{
public static void main(String[] args) {
Season s=Season.WINTER;
System.out.println(s.ordinal());
}}

```

Output:

0

#### **//example using **compareTo()** method**

You can compare the ordinal value of two contents of the same enumeration by using the **compareTo()** method. This method returns a negative integer, zero, or a positive integer as this object ordinal value less than, equal to, or greater than the specified object ordinal value.

```

enum Tutorials {
topic1, topic2, topic3;
}

public class EnumDemo {

```

```
public static void main(String args[]) {  
  
    Tutorials t1, t2, t3;  
  
    t1 = Tutorials.topic1;  
    t2 = Tutorials.topic2;  
    t3 = Tutorials.topic3;  
  
    if(t1.compareTo(t2) > 0) {  
        System.out.println(t2 + " completed before " + t1);  
    }  
  
    if(t1.compareTo(t2) < 0) {  
        System.out.println(t1 + " completed before " + t2);  
    }  
  
    if(t1.compareTo(t3) == 0) {  
        System.out.println(t1 + " completed with " + t3);  
    }  
}  
topic1 completed before topic2
```

You can compare an enumeration constant with any other object by using **equal( )**, which overrides the **equals( )** method defined by **Object**. Although **equals( )** can compare an enumeration constant to **any other object**, those two objects will only be equal if they both refer to the same constant, within the same enumeration.

```
enum Tutorials {  
    topic1, topic2, topic3;  
}  
  
public class EnumDemo {  
  
    public static void main(String args[]) {  
  
        Tutorials t1, t2, t3;  
  
        t1 = Tutorials.topic1;  
        t2 = Tutorials.topic2;  
        t3 = Tutorials.topic1;  
  
        if(t1.equals(t2)) {  
            System.out.println("Error");  
        }  
  
        if (t1.equals(t3)) {  
            System.out.println("Success");  
        }  
    }  
}
```

```
        System.out.println(t1 + " Equals " + t3);
    }
}
}
topic1 Equals topic1
```

**We can compare 2 enumerations references for equality using ==( operator).**

```
enum Apple {
    shimla, ooty, wood, green, red
}

public class EnumDemo4567 {
    public static void main(String args[])
    {
        Apple ap, ap2, ap3;

        // Obtain all ordinal values using ordinal().
        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.wood;
        ap2 = Apple.ooty;
        ap3 = Apple.wood;

        System.out.println();

        // Demonstrate compareTo() and equals()
        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);  
}  
}  
}
```

Here are all apple constants and their ordinal values:

```
shimla 0  
ooty 1  
wood 2  
green 3  
red 4
```

```
ooty comes before wood  
wood equals wood
```

```
wood equals wood  
wood == wood
```

## Type Wrapper

Java uses primitive data types such as int, double, float etc. to hold the basic data types.

Eg. Int a =10;

Float f=24.7;

Char ch='c';

Despite the performance benefits offered by the primitive data types, there are situations when you will need an object representation of the primitive data type. For example, many data structures in Java operate on objects. So you cannot use primitive data types with those data structures. To handle such type of situations, Java provides **type Wrappers** which provide classes that encapsulate a primitive type within an object.

### Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as **ArrayList** and **Vector**, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

**Character** : It encapsulates primitive type char within object.

Character (char ch) //constructor for Character class

ch-specifies character that will be wrapped by Character object being created.

To obtain primitive char value contained in **Character** object call

char charValue()

**Boolean** : It encapsulates primitive type boolean within object.

Boolean (boolean b) //constructor for Boolean class

To obtain primitive bool value contained in **Boolean** object call  
boolean booleanValue()

Likewise for below wrapper classes

**Numeric type wrappers** : It is the most commonly used type wrapper.

Byte	Short	Integer	Long	Float	Double
------	-------	---------	------	-------	--------

Primitive	Wrapper Class	Constructor Argument	Methods to get primitive values
boolean	Boolean	Boolean (boolean b) or String	booleanValue()
byte	Byte	Byte (byte b) or String	byte Value()
char	Character	Character (char ch)	char Value()
int	Integer	Integer(int a) or String	int Value()
float	Float	Float(float f) or double or String	float Value()
double	Double	Double (double d) or String	double Value()
long	Long	Long (long l) or String	long Value()
short	Short	Short (short) or String	short Value()

Following example shows constructors in wrapper classes.

```
public class WrapperClasses
{
    public static void main(String[] args)
    {
        Byte B1 = new Byte((byte) 10); //Constructor which takes byte value as an argument
        Byte B2 = new Byte("10"); //Constructor which takes String as an argument

        //Byte B3 = new Byte("abc"); //Run Time Error : NumberFormatException

        //Because, String abc can not be parse-able to byte

        Short S1 = new Short((short) 20); //Constructor which takes short value as an argument
        Short S2 = new Short("10"); //Constructor which takes String as an argument

        Integer I1 = new Integer(30); //Constructor which takes int value as an argument
        Integer I2 = new Integer("30"); //Constructor which takes String as an argument

        Long L1 = new Long(40); //Constructor which takes long value as an argument
        Long L2 = new Long("40"); //Constructor which takes String as an argument

        Float F1 = new Float(12.2f); //Constructor which takes float value as an argument
        Float F2 = new Float("15.6"); //Constructor which takes String as an argument
```

```

Float F3 = new Float(15.6d); //Constructor which takes double value as an argument

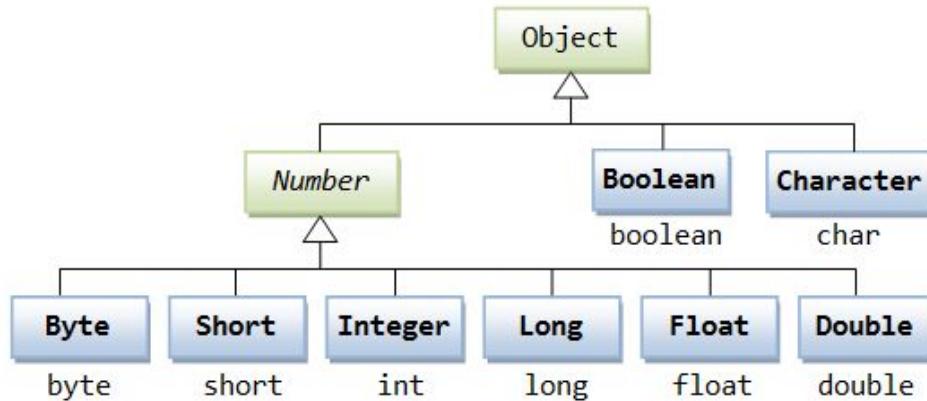
Double D1 = new Double(17.8d); //Constructor which takes double value as an argument
Double D2 = new Double("17.8"); //Constructor which takes String as an argument

Boolean BLN1 = new Boolean(false); //Constructor which takes boolean value as an argument
Boolean BLN2 = new Boolean("true"); //Constructor which takes String as an argument

Character C1 = new Character('D'); //Constructor which takes char value as an argument
Character C2 = new Character("abc"); //Compile time error : String abc can not be converted to
character
}
}

```

### Type Wrapper Hierarchy



**Boxing** : Process of converting primitive type to corresponding wrapper.

Eg.     Integer i = new Integer(10);  
       Integer j = 20;

**UnBoxing** : Process of extracting value for type wrapper.

int a = i.intValue(i);

### Autoboxing and Unboxing

- Autoboxing and Unboxing features was added in Java5.
- **Autoboxing** is a process by which primitive type is automatically encapsulated(boxed) into its equivalent type wrapper
- **Auto-Unboxing** is a process by which the value of an object is automatically extracted from a type Wrapper class.

### Benefits of Autoboxing / Unboxing

1. Autoboxing / Unboxing lets us use primitive types and Wrapper class objects interchangeably.

2. We don't have to perform Explicit **typecasting**.
3. It helps prevent errors, but may lead to unexpected results sometimes. Hence must be used with care.
4. Auto-unboxing also allows you to mix different types of numeric objects in an expression. When the values are unboxed, the standard type conversions can be applied.

### Simple Example of Autoboxing in java:

```
class BoxingExample1{
    public static void main(String args[]){
        int a=50;
        Integer a2=new Integer(a);//Boxing

        Integer a3=5;//Boxing

        System.out.println(a2+" "+a3);
    }
}
Output:50 5
```

### Simple Example of Unboxing in java:

```
class UnboxingExample1 {
    public static void main(String args[]){
        Integer i=new Integer(50);
        int a=i;

        System.out.println(a);
    }
}
Output:50
```

### Autoboxing / Unboxing in Expressions

Whenever we use object of Wrapper class in an expression, automatic unboxing and boxing is done by JVM.

```
Integer iOb;
iOb = 100;      //Autoboxing of int
++iOb;
```

When we perform increment operation on Integer object, it is first unboxed, then incremented and then again reboxed into Integer type object.

This will happen always, when we will use Wrapper class objects in expressions or conditions etc.

**Example 2**

```
class Test {
    public static void main(String args[]) {
        Integer i = 35;
        Double d = 33.3;
        d = d + i;
        System.out.println("Value of d is " + d);
    }
}
Output:
Value of d is 68.3
```

**Note:** When the statement **d = d + i;** was executed, i was auto-unboxed into int, d was auto-unboxed into double, addition was performed and then finally, auto-boxing of d was done into Double type Wrapper class.

**Autoboxing / Unboxing in Methods**

```
class Boxing1{
    static void m(int i)
        {System.out.println("int");}
    public static void main(String args[ ]){
        Integer s=30;
        m(s);
    }
}
Output:int
```

**Autoboxing / Unboxing Boolean**

```
class UnboxingExample2{
    public static void main(String args[ ]){
        Integer i=new Integer(50);
        if(i<100){          //unboxing internally
            System.out.println(i);
        }
    }
}
Output:50
```

**Autoboxing / Unboxing Boolean and character values**

```
// Autoboxing/unboxing a Boolean and Character.
class AutoBox5 {
```

---

```

public static void main(String args[]) {
    // Autobox/unbox a boolean.
    Boolean b = true;
    // Below, b is auto-unboxed when used in
    // a conditional expression, such as an if.
    if(b) System.out.println("b is true");
    // Autobox/unbox a char.
    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

**Autoboxing / Unboxing helps preventing errors**

```

// An error produced by manual unboxing.
class UnboxingError {
    public static void main(String args[]) {
        Integer iOb = 1000; // autobox the value 1000
        int i = iOb.byteValue(); // manually unbox as byte !!!
        System.out.println(i); // does not display 1000 !
    }
}

```

**Annotations (Metadata)**

[Java Annotations](#) allow us to add metadata information into our source code, Annotations were added to the java from JDK 5.

Annotations, 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.

- Annotations start with '@'.
- Annotations do not change action of a compiled program.
- Annotations help to associate *metadata* (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.
- Annotations are not pure comments as they can change the way a program is treated by compiler.

**Annotations basics**

An annotation always starts with the symbol @ followed by the annotation name. The symbol @ indicates to the compiler that this is an annotation.

**Where we can use annotations?**

Annotations can be applied to the classes, interfaces, methods and fields.

## Built-In Java Annotations

There are **7** built-in annotations in java. Some annotations are applied to java code and some to other annotations.

### Built-In Java Annotations used in java code imported from java.lang

- `@Override`
- `@SuppressWarnings`
- `@Deprecated`

### Built-In Java Annotations used in other annotations

4 Annotations imported from `java.lang.annotation`

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

1. **`@Override`** It is a marker annotation that can be used only on methods. A method annotated with **`@Override`** must override a method from a superclass. If it doesn't, a compile-time error will result . It is used to ensure that a superclass method is actually overridden, and not simply overloaded.

```
class Base
{
    public void Display()
    {
        System.out.println("Base display()");
    }
    public static void main(String args[])
    {
        Base t1 = new Derived();
        t1.Display();
    }
}
class Derived extends Base
{
    @Override
    public void Display()
    {
        System.out.println("Derived display()");
    }
}
```

**Output:**  
Derived display()

## 2. **@SuppressWarnings**

It is used to inform the compiler to suppress specified compiler warnings. The warnings to suppress are specified by name, in string form. This type of annotation can be applied to any type of declaration.

Java groups warnings under two categories. They are : **deprecation** and **unchecked**. Any unchecked warning is generated when a legacy code interfaces with a code that uses generics.

```
class DeprecatedTest
{
    @Deprecated
    public void Display()
    {
        System.out.println("Deprecatedtest display()");
    }
}

public class SuppressWarningTest
{
    // If we comment below annotation, program generates
    // warning
    @SuppressWarnings({"checked", "deprecation"})
    public static void main(String args[])
    {
        DeprecatedTest d1 = new DeprecatedTest();
        d1.Display();
    }
}

Output:
Deprecatedtest display()
```

3. **@Deprecated** It is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form. The Javadoc [@deprecated tag](#) should be used when an element has been deprecated.

```
public class DeprecatedTest
{
    @Deprecated
    public void Display()
    {
        System.out.println("Deprecatedtest display()");
    }
}

public static void main(String args[])
{
    DeprecatedTest d1 = new DeprecatedTest();
```

```
    d1.Display();  
}  
}
```

**Output:**

Deprecatedtest display()

## Types of Annotation

There are 3 categories of Annotations:-

### 1. Marker Annotations:

The only purpose is to mark a declaration. These annotations contain no members and do not consist any data. Thus, its presence as an annotation is sufficient. Since, marker interface contains no members, simply determining whether it is present or absent is sufficient. `@Override`, `@Deprecated` is an example of Marker Annotation.

Example: - `@TestAnnotation()`

### 2. Single value Annotations:

These annotations contain only one member and allow a shorthand form of specifying the value of the member. We only need to specify the value for that member when the annotation is applied and don't need to specify the name of the member. However in order to use this shorthand, the name of the member must be **value**.

Example: - `@TestAnnotation("testing")`;

### 3. Multivalue Annotations:

These annotations consist of multiple data members/ name, value, pairs.

Example:- `@TestAnnotation(owner="Umesh", value="Tutor")`

## Java Custom Annotation

**Java Custom annotations** or Java User-defined annotations are easy to create and use. The `@interface` element is used to declare an annotation. For example:

```
@interface MyAnnotation { }
```

Here, MyAnnotation is the custom annotation name.

## Points to remember for java custom annotation signature

There are few points that should be remembered by the programmer.

1. Annotations are created by using `@interface`, followed by annotation name as shown in the below example.
2. An annotation can have elements as well. They look like methods. For example in the below code, we have four elements. We should not provide implementation for these elements.

- 
3. Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
  4. Method should not have any parameter.
  5. We should attach @ just before interface keyword to define annotation.
  6. It may assign a default value to the method.
- 

### Declaring custom Annotation

```
@interface MyAnnotation{
    int value1( ) default 1;
    String value2( ) default "";
    String value3( ) default "xyz";
}
```

### How to apply custom Annotation

```
@MyAnnotation(value1=10,value2="Umesh",value3="SJBIT")
```

4. **@Documented** It is a marker interface that tells a tool that an annotation is to be documented. Annotations are not included by Javadoc comments. Use of @Documented annotation in the code enables tools like Javadoc to process it and include the annotation type information in the generated document.

```
java.lang.annotation.Documented
@Documented
public @interface MyCustomAnnotation {
    //Annotation body
}
```

```
@MyCustomAnnotation
public class MyClass {
    //Class body
}
```

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

### 5. @Inherited

The @Inherited annotation signals that a custom annotation used in a class should be inherited by all of its sub classes. For example:

```
java.lang.annotation.Inherited
```

---

```

@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.

## 6. @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.

The java.lang.annotation.ElementType enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc.

Element Types	Where the annotation can be applied
TYPE	class, interface or enumeration
FIELD	fields
METHOD	methods
CONSTRUCTOR	constructors
LOCAL_VARIABLE	local variables
ANNOTATION_TYPE	annotation type
PARAMETER	parameter

```

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

@Target({ElementType.METHOD})
// u can also target multiple elements
//{@Target({ ElementType.FIELD, ElementType.METHOD})
public @interface MyCustomAnnotation {

}

public class MyClass {
    @MyCustomAnnotation

```

```
public void myMethod()
{
    //Doing something
}
```

## 7. **@Retention**

It indicates how long annotations with the annotated type are to be retained.

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
@interface MyCustomAnnotation {

}
```

Here we have used RetentionPolicy.RUNTIME. There are two other options as well. Lets see what do they mean:

RetentionPolicy.RUNTIME: The annotation should be available at runtime, for inspection via java reflection.

RetentionPolicy.CLASS: The annotation would be in the .class file but it would not be available at runtime.

RetentionPolicy.SOURCE: The annotation would be available in the source code of the program, it would neither be in the .class file nor be available at the runtime.

### Complete in one example

```
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Documented
@Target(ElementType.METHOD)
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface MyCustomAnnotation{
    int studentAge() default 18;
    String studentName();
    String stuAddress();
    String stuStream() default "CSE";
}

@MyCustomAnnotation( studentName="umesh", stuAddress="India")
public class MyClass {
    ...
}
```

}

### Obtaining annotation at runtime using reflection

Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.

- The required classes for reflection are provided under `java.lang.reflect` package.

Reflection can be used to get information about –

- **Class** The `getClass()` method is used to get the name of the class to which an object belongs.
- **Constructors** The `getConstructors()` method is used to get the public constructors of the class to which an object belongs.
- **Methods** The `getMethods()` method is used to get the public methods of the class to which an objects belongs.

```
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();
// Obtain the annotation for this method
// and display the values of the members.

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);
// Finally, display the values.
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

### 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( )** can be called on objects of type **Class**, **Method**, **Constructor**, and **Field**.

```
// Show all annotations for a class and a method.
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 AnnotatedElement Interface

The methods **getAnnotation( )** and **getAnnotations( )** used by the preceding examples are defined by the **AnnotatedElement** interface, which is defined in **java.lang.reflect**. This interface supports reflection for annotations and is implemented by the classes Method, Field, Constructor, Class, and Package.

In addition to **getAnnotation( )** and **getAnnotations( )**, **AnnotatedElement** defines two other methods. The first is **getDeclaredAnnotations( )**, which has this general form:

**Annotation[ ] getDeclaredAnnotations( )**

It returns all non-inherited annotations present in the invoking object. The second is **isAnnotationPresent( )**, which has this general form:

**boolean isAnnotationPresent(Class annoType)**

It returns true if the annotation specified by **annoType** is associated with the **invokingobject**. It returns false otherwise.

## MODULE – 2

# COLLECTION FRAMEWORK

### Introduction to Collections

A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers). If you have used the Java programming language — or just about any other programming language — you are already familiar with collections.

### What Is a Collections Framework?

A *collections framework* is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Apart from the Java Collections Framework, the best-known examples of collections frameworks are the C++ Standard Template Library (STL) and Smalltalk's collection hierarchy. Historically, collections frameworks have been quite complex, which gave them a reputation for having a steep learning curve. We believe that the Java Collections Framework breaks with this tradition, as you will learn for yourself in this chapter.

### Benefits of the Java Collections Framework

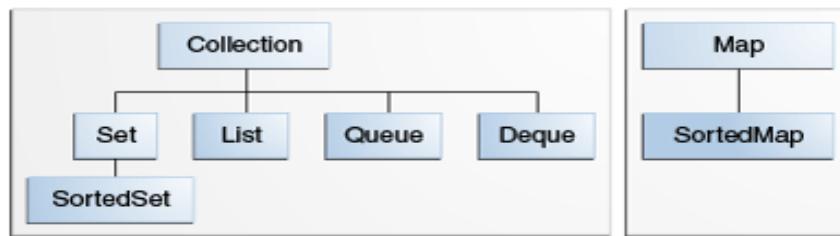
The Java Collections Framework provides the following benefits:

- **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.

- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.
- **Allows interoperability among unrelated APIs:** The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.
- **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.
- **Reduces effort to design new APIs:** This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
- **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

## Interfaces

The *core collection interfaces* encapsulate different types of collections, which are shown in the figure below. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the Java Collections Framework. As you can see in the following figure, the core collection interfaces form a hierarchy.



The core collection interfaces.

A `Set` is a special kind of `Collection`, a `SortedSet` is a special kind of `Set`, and so forth. Note also that the hierarchy consists of two distinct trees — a `Map` is not a true `Collection`.

Note that all the core collection interfaces are generic. For example, this is the declaration of the `Collection` interface.

```
public interface Collection<E>...
```

The `<E>` syntax tells you that the interface is generic. When you declare a `Collection` instance you can *and should* specify the type of object contained in the collection. Specifying the type allows the compiler to verify (at compile-time) that the type of object you put into the collection is correct, thus reducing errors at runtime. For information on generic types, see the [Generics \(Updated\)](#) lesson.

When you understand how to use these interfaces, you will know most of what there is to know about the Java Collections Framework. This chapter discusses general guidelines for effective use of the interfaces, including when to use which interface. You'll also learn programming idioms for each interface to help you get the most out of it.

To keep the number of core collection interfaces manageable, the Java platform doesn't provide separate interfaces for each variant of each collection type. (Such variants might include immutable, fixed-size, and append-only.) Instead, the modification operations in each interface are designated *optional* — a given implementation may elect not to support all operations. If an unsupported operation is invoked, a collection throws an [UnsupportedOperationException](#). Implementations are responsible for documenting which of the optional operations they support. All of the Java platform's general-purpose implementations support all of the optional operations.

The following list describes the core collection interfaces:

- `Collection` — the root of the collection hierarchy. A collection represents a group of objects known as its *elements*. The `Collection` interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as `Set` and `List`. Also see [The Collection Interface](#) section.
- `Set` — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine. See also [The Set Interface](#) section.
- `List` — an ordered collection (sometimes called a *sequence*). `Lists` can contain duplicate elements. The user of a `List` generally has precise control over where in the list each element is inserted and can access elements by their integer index (position). If you've used `vector`, you're familiar with the general flavor of `List`. Also see [The List Interface](#) section.
- `Queue` — a collection used to hold multiple elements prior to processing. Besides basic `Collection` operations, a `Queue` provides additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to `remove` or `poll`. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every `Queue` implementation must specify its ordering properties. Also see [The Queue Interface](#) section.

- `Deque` — a collection used to hold multiple elements prior to processing. Besides basic `Collection` operations, a `Deque` provides additional insertion, extraction, and inspection operations.

Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out). In a deque all new elements can be inserted, retrieved and removed at both ends. Also see [The Deque Interface](#) section.

- `Map` — an object that maps keys to values. A `Map` cannot contain duplicate keys; each key can map to at most one value. If you've used `Hashtable`, you're already familiar with the basics of `Map`. Also see [The Map Interface](#) section.

The last two core collection interfaces are merely sorted versions of `Set` and `Map`:

- `SortedSet` — a `Set` that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls. Also see [The SortedSet Interface](#) section.
- `SortedMap` — a `Map` that maintains its mappings in ascending key order. This is the `Map` analog of `SortedSet`. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories. Also see [The SortedMap Interface](#) section.

To understand how the sorted interfaces maintain the order of their elements, see the [Object Ordering](#) section.

## The Collection Interface

A `Collection` represents a group of objects known as its elements. The `Collection` interface is used to pass around collections of objects where maximum generality is desired. For example, by convention all general-purpose collection implementations have a constructor that takes a `Collection` argument. This constructor, known as a *conversion constructor*, initializes the new collection to contain all of the elements in the specified collection, whatever the given collection's subinterface or implementation type. In other words, it allows you to *convert* the collection's type.

Suppose, for example, that you have a `Collection<String> c`, which may be a `List`, a `Set`, or another kind of `Collection`. This idiom creates a new `ArrayList` (an implementation of the `List` interface), initially containing all the elements in `c`.

```
List<String> list = new ArrayList<String>(c);  
Or — if you are using JDK 7 or later — you can use the diamond operator:
```

```
List<String> list = new ArrayList<>(c);
```

The `Collection` interface contains methods that perform basic operations, such as `int size()`, `boolean isEmpty()`, `boolean contains(Object element)`, `boolean add(E element)`, `boolean remove(Object element)`, and `Iterator<E> iterator()`.

It also contains methods that operate on entire collections, such as `boolean containsAll(Collection<?> c)`, `boolean addAll(Collection<? extends E> c)`, `boolean removeAll(Collection<?> c)`, `boolean retainAll(Collection<?> c)`, and `void clear()`.

Additional methods for array operations (such as `Object[] toArray()` and `<T> T[] toArray(T[] a)`) exist as well.

In JDK 8 and later, the `Collection` interface also exposes methods `Stream<E> stream()` and `Stream<E> parallelStream()`, for obtaining sequential or parallel streams from the underlying collection. (See the lesson entitled [Aggregate Operations](#) for more information about using streams.)

The `Collection` interface does about what you'd expect given that a `Collection` represents a group of objects. It has methods that tell you how many elements are in the collection (`size`, `isEmpty`), methods that check whether a given object is in the collection (`contains`), methods that add and remove an element from the collection (`add`, `remove`), and methods that provide an iterator over the collection (`iterator`).

The `add` method is defined generally enough so that it makes sense for collections that allow duplicates as well as those that don't. It guarantees that the `Collection` will contain the specified element after the call completes, and returns `true` if the `Collection` changes as a result of the call. Similarly, the `remove` method is designed to remove a single instance of the specified element from the `Collection`, assuming that it contains the element to start with, and to return `true` if the `Collection` was modified as a result.

## Traversing Collections

There are three ways to traverse collections: (1) using aggregate operations (2) with the `for-each` construct and (3) by using `Iterators`.

## Aggregate Operations

In JDK 8 and later, the preferred method of iterating over a collection is to obtain a stream and perform aggregate operations on it. Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code. The following code sequentially iterates through a collection of shapes and prints out the red objects:

```
myShapesCollection.stream()
    .filter(e -> e.getColor() == Color.RED)
    .forEach(e -> System.out.println(e.getName()));
```

Likewise, you could easily request a parallel stream, which might make sense if the collection is large enough and your computer has enough cores:

```
myShapesCollection.parallelStream()
    .filter(e -> e.getColor() == Color.RED)
    .forEach(e -> System.out.println(e.getName()));
```

There are many different ways to collect data with this API. For example, you might want to convert the elements of a `Collection` to `String` objects, then join them, separated by commas:

```
String joined = elements.stream()
    .map(Object::toString)
    .collect(Collectors.joining(", "));
```

Or perhaps sum the salaries of all employees:

```
int total = employees.stream()
    .collect(Collectors.summingInt(Employee::getSalary));
```

These are but a few examples of what you can do with streams and aggregate operations. For more information and examples, see the lesson entitled [Aggregate Operations](#).

The Collections framework has always provided a number of so-called "bulk operations" as part of its API. These include methods that operate on entire collections, such as `containsAll`, `addAll`, `removeAll`, etc. Do not confuse those methods with the aggregate operations that were introduced in JDK 8. The key difference between the new aggregate operations and the existing bulk operations (`containsAll`, `addAll`, etc.) is that the old versions are all *mutative*, meaning that they all modify the underlying collection. In contrast, the new aggregate operations *do not* modify the underlying collection. When using the new aggregate operations and lambda expressions, you must take care to avoid mutation so as not to introduce problems in the future, should your code be run later from a parallel stream.

## for-each Construct

The `for-each` construct allows you to concisely traverse a collection or array using a `for` loop — see [The for Statement](#). The following code uses the `for-each` construct to print out each element of a collection on a separate line.

```
for (Object o : collection)
    System.out.println(o);
```

## Iterators

An [Iterator](#) is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an `Iterator` for a collection by calling its `iterator` method. The following is the `Iterator` interface.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

The `hasNext` method returns `true` if the iteration has more elements, and the `next` method returns the next element in the iteration. The `remove` method removes the last element that was returned by `next` from the underlying `Collection`. The `remove` method may be called only once per call to `next` and throws an exception if this rule is violated.

Note that `Iterator.remove` is the *only* safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.

Use `Iterator` instead of the `for-each` construct when you need to:

- Remove the current element. The `for-each` construct hides the iterator, so you cannot call `remove`. Therefore, the `for-each` construct is not usable for filtering.
- Iterate over multiple collections in parallel.

The following method shows you how to use an `Iterator` to filter an arbitrary `Collection` — that is, traverse the collection removing specific elements.

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

This simple piece of code is polymorphic, which means that it works for *any* `Collection` regardless of implementation. This example demonstrates how easy it is to write a polymorphic algorithm using the Java Collections Framework.

## Collection Interface Bulk Operations

*Bulk operations* perform an operation on an entire `Collection`. You could implement these shorthand operations using the basic operations, though in most cases such implementations would be less efficient. The following are the bulk operations:

- `containsAll` — returns `true` if the target `Collection` contains all of the elements in the specified `Collection`.
- `addAll` — adds all of the elements in the specified `Collection` to the target `Collection`.
- `removeAll` — removes from the target `Collection` all of its elements that are also contained in the specified `Collection`.
- `retainAll` — removes from the target `Collection` all its elements that are *not* also contained in the specified `Collection`. That is, it retains only those elements in the target `Collection` that are also contained in the specified `Collection`.
- `clear` — removes all elements from the `Collection`.

The `addAll`, `removeAll`, and `retainAll` methods all return `true` if the target `Collection` was modified in the process of executing the operation.

As a simple example of the power of bulk operations, consider the following idiom to remove *all* instances of a specified element, `e`, from a `Collection`, `c`.

```
c.removeAll(Collections.singleton(e));
```

More specifically, suppose you want to remove all of the `null` elements from a `Collection`.

```
c.removeAll(Collections.singleton(null));
```

This idiom uses `Collections.singleton`, which is a static factory method that returns an immutable `Set` containing only the specified element.

## Collection Interface Array Operations

The `toArray` methods are provided as a bridge between collections and older APIs that expect arrays on input. The array operations allow the contents of a `Collection` to be translated into an array. The simple form with no arguments creates a new array of `Object`. The more complex form allows the caller to provide an array or to choose the runtime type of the output array.

For example, suppose that `c` is a `Collection`. The following snippet dumps the contents of `c` into a newly allocated array of `Object` whose length is identical to the number of elements in `c`.

```
Object[] a = c.toArray();
```

Suppose that `c` is known to contain only strings (perhaps because `c` is of type `Collection<String>`). The following snippet dumps the contents of `c` into a newly allocated array of `String` whose length is identical to the number of elements in `c`.

```
String[] a = c.toArray(new String[0]);
```

## The Set Interface

---

A [Set](#) is a [Collection](#) that cannot contain duplicate elements. It models the mathematical set abstraction. The `Set` interface contains *only* methods inherited from `Collection` and adds the restriction that duplicate elements are prohibited. `Set` also adds a stronger contract on the behavior of the `equals` and `hashCode` operations, allowing `Set` instances to be compared meaningfully even if their implementation types differ. Two `Set` instances are equal if they contain the same elements.

The Java platform contains three general-purpose `Set` implementations: `HashSet`, `TreeSet`, and `LinkedHashSet`. [HashSet](#), which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration. [TreeSet](#), which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than `HashSet`. [LinkedHashSet](#), which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order). `LinkedHashSet` spares its clients from the unspecified, generally chaotic ordering provided by `HashSet` at a cost that is only slightly higher.

Here's a simple but useful `Set` idiom. Suppose you have a `Collection`, `c`, and you want to create another `Collection` containing the same elements but with all duplicates eliminated. The following one-liner does the trick.

```
Collection<Type> noDups = new HashSet<Type>(c);
```

It works by creating a `Set` (which, by definition, cannot contain duplicates), initially containing all the elements in `c`. It uses the standard conversion constructor described in the [The Collection Interface](#) section.

Or, if using JDK 8 or later, you could easily collect into a `Set` using aggregate operations:

```
c.stream()
.collect(Collectors.toSet()); // no duplicates
```

Here's a slightly longer example that accumulates a `Collection` of names into a `TreeSet`:

```
Set<String> set = people.stream()
.map(Person::getName)
.collect(Collectors.toCollection(TreeSet::new));
```

And the following is a minor variant of the first idiom that preserves the order of the original collection while removing duplicate elements:

```
Collection<Type> noDups = new LinkedHashSet<Type>(c);
```

The following is a generic method that encapsulates the preceding idiom, returning a `Set` of the same generic type as the one passed.

```
public static <E> Set<E> removeDups(Collection<E> c) {
    return new LinkedHashSet<E>(c);
}
```

---

## Set Interface Basic Operations

The `size` operation returns the number of elements in the `Set` (its *cardinality*). The `isEmpty` method does exactly what you think it would. The `add` method adds the specified element to the `Set` if it is not already present and returns a boolean indicating whether the element was added. Similarly, the `remove` method removes the specified element from the `Set` if it is present and returns a boolean indicating whether the element was present. The `iterator` method returns an `Iterator` over the `Set`.

The following [program](#) prints out all distinct words in its argument list. Two versions of this program are provided. The first uses JDK 8 aggregate operations. The second uses the for-each construct.

Using JDK 8 Aggregate Operations:

```
import java.util.*;
import java.util.stream.*;

public class FindDups {
    public static void main(String[] args) {
        Set<String> distinctWords = Arrays.asList(args).stream()
            .collect(Collectors.toSet());
        System.out.println(distinctWords.size() +
            " distinct words: " +
            distinctWords);
    }
}
```

Using the for-each Construct:

```
import java.util.*;

public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            s.add(a);
        System.out.println(s.size() + " distinct words: " + s);
    }
}
```

Now run either version of the program.

```
java FindDups i came i saw i left
```

The following output is produced:

```
4 distinct words: [left, came, saw, i]
```

Note that the code always refers to the `Collection` by its interface type (`Set`) rather than by its implementation type. This is a *strongly* recommended programming practice because it gives you the flexibility to change implementations merely by changing the constructor. If either of the variables used to store a collection or the parameters used to pass it around are declared to be of the `Collection`'s implementation type rather than its interface type, *all* such variables and parameters must be changed in order to change its implementation type.

Furthermore, there's no guarantee that the resulting program will work. If the program uses any nonstandard operations present in the original implementation type but not in the new one, the program will fail. Referring to collections only by their interface prevents you from using any nonstandard operations.

The implementation type of the `Set` in the preceding example is `HashSet`, which makes no guarantees as to the order of the elements in the `Set`. If you want the program to print the word list in alphabetical order, merely change the `Set`'s implementation type from `HashSet` to `TreeSet`. Making this trivial one-line change causes the command line in the previous example to generate the following output.

```
java FindDups i came i saw i left  
4 distinct words: [came, i, left, saw]
```

## Set Interface Bulk Operations

Bulk operations are particularly well suited to `sets`; when applied, they perform standard set-algebraic operations. Suppose `s1` and `s2` are sets. Here's what bulk operations do:

- `s1.containsAll(s2)` — returns `true` if `s2` is a **subset** of `s1`. (`s2` is a subset of `s1` if set `s1` contains all of the elements in `s2`.)
- `s1.addAll(s2)` — transforms `s1` into the **union** of `s1` and `s2`. (The union of two sets is the set containing all of the elements contained in either set.)
- `s1.retainAll(s2)` — transforms `s1` into the intersection of `s1` and `s2`. (The intersection of two sets is the set containing only the elements common to both sets.)
- `s1.removeAll(s2)` — transforms `s1` into the (asymmetric) set difference of `s1` and `s2`. (For example, the set difference of `s1` minus `s2` is the set containing all of the elements found in `s1` but not in `s2`.)

To calculate the union, intersection, or set difference of two sets *nondestructively* (without modifying either set), the caller must copy one set before calling the appropriate bulk operation. The following are the resulting idioms.

```
Set<Type> union = new HashSet<Type>(s1);  
union.addAll(s2);  
  
Set<Type> intersection = new HashSet<Type>(s1);  
intersection.retainAll(s2);
```

---

```
Set<Type> difference = new HashSet<Type>(s1);
difference.removeAll(s2);
```

The implementation type of the result `set` in the preceding idioms is `HashSet`, which is, as already mentioned, the best all-around `Set` implementation in the Java platform. However, any general-purpose `Set` implementation could be substituted.

Let's revisit the `FindDups` program. Suppose you want to know which words in the argument list occur only once and which occur more than once, but you do not want any duplicates printed out repeatedly. This effect can be achieved by generating two sets — one containing every word in the argument list and the other containing only the duplicates. The words that occur only once are the set difference of these two sets, which we know how to compute. Here's how [the resulting program](#) looks.

```
import java.util.*;

public class FindDups2 {
    public static void main(String[] args) {
        Set<String> uniques = new HashSet<String>();
        Set<String> dups     = new HashSet<String>();

        for (String a : args)
            if (!uniques.add(a))
                dups.add(a);

        // Destructive set-difference
        uniques.removeAll(dups);

        System.out.println("Unique words: " + uniques);
        System.out.println("Duplicate words: " + dups);
    }
}
```

When run with the same argument list used earlier (`i came i saw i left`), the program yields the following output.

```
Unique words: [left, saw, came]
Duplicate words: [i]
```

A less common set-algebraic operation is the *symmetric set difference* — the set of elements contained in either of two specified sets but not in both. The following code calculates the symmetric set difference of two sets nondestructively.

```
Set<Type> symmetricDiff = new HashSet<Type>(s1);
symmetricDiff.addAll(s2);
Set<Type> tmp = new HashSet<Type>(s1);
tmp.retainAll(s2);
symmetricDiff.removeAll(tmp);
```

# The List Interface

A [List](#) is an ordered [Collection](#) (sometimes called a *sequence*). Lists may contain duplicate elements. In addition to the operations inherited from [Collection](#), the [List](#) interface includes operations for the following:

- Positional access — manipulates elements based on their numerical position in the list. This includes methods such as `get`, `set`, `add`, `addAll`, and `remove`.
- Search — searches for a specified object in the list and returns its numerical position. Search methods include `indexOf` and `lastIndexOf`.
- Iteration — extends [Iterator](#) semantics to take advantage of the list's sequential nature. The `listIterator` methods provide this behavior.
- Range-view — The `sublist` method performs arbitrary *range operations* on the list.

The Java platform contains two general-purpose [List](#) implementations. [ArrayList](#), which is usually the better-performing implementation, and [LinkedList](#) which offers better performance under certain circumstances.

## Collection Operations

The operations inherited from [Collection](#) all do about what you'd expect them to do, assuming you're already familiar with them. If you're not familiar with them from [Collection](#), now would be a good time to read [The Collection Interface](#) section. The `remove` operation always removes *the first* occurrence of the specified element from the list. The `add` and `addAll` operations always append the new element(s) to the *end* of the list. Thus, the following idiom concatenates one list to another.

```
list1.addAll(list2);
```

Here's a nondestructive form of this idiom, which produces a third [List](#) consisting of the second list appended to the first.

```
List<Type> list3 = new ArrayList<Type>(list1);
list3.addAll(list2);
```

Note that the idiom, in its nondestructive form, takes advantage of [ArrayList](#)'s standard conversion constructor.

And here's an example (JDK 8 and later) that aggregates some names into a [List](#):

```
List<String> list = people.stream()
.map(Person::getName)
.collect(Collectors.toList());
```

Like the [Set](#) interface, `List` strengthens the requirements on the `equals` and `hashCode` methods so that two `List` objects can be compared for logical equality without regard to their implementation classes. Two `List` objects are equal if they contain the same elements in the same order.

## Positional Access and Search Operations

The basic positional access operations are `get`, `set`, `add` and `remove`. (The `set` and `remove` operations return the old value that is being overwritten or removed.) Other operations (`indexOf` and `lastIndexOf`) return the first or last index of the specified element in the list.

The `addAll` operation inserts all the elements of the specified `Collection` starting at the specified position. The elements are inserted in the order they are returned by the specified `Collection`'s iterator. This call is the positional access analog of `Collection`'s `addAll` operation.

Here's a little method to swap two indexed values in a `List`.

```
public static <E> void swap(List<E> a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
```

Of course, there's one big difference. This is a polymorphic algorithm: It swaps two elements in any `List`, regardless of its implementation type. Here's another polymorphic algorithm that uses the preceding `swap` method.

```
public static void shuffle(List<?> list, Random rnd) {
    for (int i = list.size(); i > 1; i--)
        swap(list, i - 1, rnd.nextInt(i));
}
```

This algorithm, which is included in the Java platform's [Collections](#) class, randomly permutes the specified list using the specified source of randomness. It's a bit subtle: It runs up the list from the bottom, repeatedly swapping a randomly selected element into the current position. Unlike most naive attempts at shuffling, it's *fair* (all permutations occur with equal likelihood, assuming an unbiased source of randomness) and *fast* (requiring exactly `list.size()`-1 swaps). The following program uses this algorithm to print the words in its argument list in random order.

```
import java.util.*;

public class Shuffle {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        for (String a : args)
            list.add(a);
```

```

        Collections.shuffle(list, new Random());
        System.out.println(list);
    }
}

```

In fact, this program can be made even shorter and faster. The [Arrays](#) class has a static factory method called `asList`, which allows an array to be viewed as a `List`. This method does not copy the array. Changes in the `List` write through to the array and vice versa. The resulting `List` is not a general-purpose `List` implementation, because it doesn't implement the (optional) `add` and `remove` operations: Arrays are not resizable. Taking advantage of `Arrays.asList` and calling the library version of `shuffle`, which uses a default source of randomness, you get the following [tiny program](#) whose behavior is identical to the previous program.

```

import java.util.*;

public class Shuffle {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.shuffle(list);
        System.out.println(list);
    }
}

```

## Iterators

As you'd expect, the `Iterator` returned by `List`'s `iterator` operation returns the elements of the list in proper sequence. `List` also provides a richer iterator, called a `ListIterator`, which allows you to traverse the list in either direction, modify the list during iteration, and obtain the current position of the iterator.

The three methods that `ListIterator` inherits from `Iterator` (`hasNext`, `next`, and `remove`) do exactly the same thing in both interfaces. The `hasPrevious` and the `previous` operations are exact analogues of `hasNext` and `next`. The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor. The `previous` operation moves the cursor backward, whereas `next` moves it forward.

Here's the standard idiom for iterating backward through a list.

```

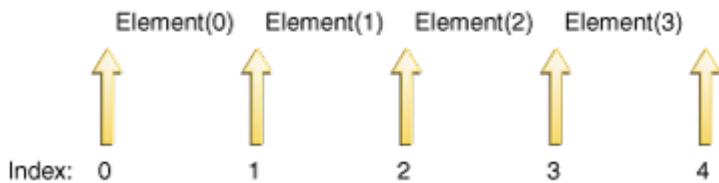
for      (ListIterator<Type>      it      =      list.listIterator(list.size());
it.hasPrevious(); ) {
    Type t = it.previous();
    ...
}

```

Note the argument to `listIterator` in the preceding idiom. The `List` interface has two forms of the `listIterator` method. The form with no arguments returns a `ListIterator` positioned at the beginning of the list; the form with an `int` argument returns a `ListIterator` positioned at the specified index. The index refers to the element that would be returned by an initial call to

`next`. An initial call to `previous` would return the element whose index was `index-1`. In a list of length `n`, there are `n+1` valid values for `index`, from `0` to `n`, inclusive.

Intuitively speaking, the cursor is always between two elements — the one that would be returned by a call to `previous` and the one that would be returned by a call to `next`. The `n+1` valid `index` values correspond to the `n+1` gaps between elements, from the gap before the first element to the gap after the last one. The following figure shows the five possible cursor positions in a list containing four elements.



The five possible cursor positions.

Calls to `next` and `previous` can be intermixed, but you have to be a bit careful. The first call to `previous` returns the same element as the last call to `next`. Similarly, the first call to `next` after a sequence of calls to `previous` returns the same element as the last call to `previous`.

It should come as no surprise that the `nextIndex` method returns the index of the element that would be returned by a subsequent call to `next`, and `previousIndex` returns the index of the element that would be returned by a subsequent call to `previous`. These calls are typically used either to report the position where something was found or to record the position of the `ListIterator` so that another `ListIterator` with identical position can be created.

It should also come as no surprise that the number returned by `nextIndex` is always one greater than the number returned by `previousIndex`. This implies the behavior of the two boundary cases: (1) a call to `previousIndex` when the cursor is before the initial element returns `-1` and (2) a call to `nextIndex` when the cursor is after the final element returns `list.size()`. To make all this concrete, the following is a possible implementation of `List.indexOf`.

```
public int indexOf(E e) {
    for (ListIterator<E> it = listIterator(); it.hasNext(); )
        if (e == null ? it.next() == null : e.equals(it.next()))
            return it.previousIndex();
    // Element not found
    return -1;
}
```

Note that the `indexOf` method returns `it.previousIndex()` even though it is traversing the list in the forward direction. The reason is that `it.nextIndex()` would return the index of the element we are about to examine, and we want to return the index of the element we just examined.

The `Iterator` interface provides the `remove` operation to remove the last element returned by `next` from the `Collection`. For `ListIterator`, this operation removes the last element returned by `next` or `previous`. The `ListIterator` interface provides two additional operations to modify the list — `set` and `add`. The `set` method overwrites the last element returned by `next` or `previous` with the specified element. The following polymorphic algorithm uses `set` to replace all occurrences of one specified value with another.

```
public static <E> void replace(List<E> list, E val, E newVal) {
    for (ListIterator<E> it = list.listIterator(); it.hasNext(); )
        if (val == null ? it.next() == null : val.equals(it.next()))
            it.set(newVal);
}
```

The only bit of trickiness in this example is the equality test between `val` and `it.next`. You need to special-case a `val` value of `null` to prevent a `NullPointerException`.

The `add` method inserts a new element into the list immediately before the current cursor position. This method is illustrated in the following polymorphic algorithm to replace all occurrences of a specified value with the sequence of values contained in the specified list.

```
public static <E>
void replace(List<E> list, E val, List<? extends E> newVals) {
    for (ListIterator<E> it = list.listIterator(); it.hasNext(); ) {
        if (val == null ? it.next() == null : val.equals(it.next())) {
            it.remove();
            for (E e : newVals)
                it.add(e);
        }
    }
}
```

## Range-View Operation

The range-view operation, `subList(int fromIndex, int toIndex)`, returns a `List` view of the portion of this list whose indices range from `fromIndex`, inclusive, to `toIndex`, exclusive. This *half-open range* mirrors the typical `for` loop.

```
for (int i = fromIndex; i < toIndex; i++) {
    ...
}
```

As the term *view* implies, the returned `List` is backed up by the `List` on which `subList` was called, so changes in the former are reflected in the latter.

This method eliminates the need for explicit range operations (of the sort that commonly exist for arrays). Any operation that expects a `List` can be used as a range operation by passing a `subList` view instead of a whole `List`. For example, the following idiom removes a range of elements from a `List`.

```
list.subList(fromIndex, toIndex).clear();
```

Similar idioms can be constructed to search for an element in a range.

```
int i = list.subList(fromIndex, toIndex).indexOf(o);
int j = list.subList(fromIndex, toIndex).lastIndexOf(o);
```

Note that the preceding idioms return the index of the found element in the `subList`, not the index in the backing `List`.

Any polymorphic algorithm that operates on a `List`, such as the `replace` and `shuffle` examples, works with the `List` returned by `subList`.

Here's a polymorphic algorithm whose implementation uses `subList` to deal a hand from a deck. That is, it returns a new `List` (the "hand") containing the specified number of elements taken from the end of the specified `List` (the "deck"). The elements returned in the hand are removed from the deck.

```
public static <E> List<E> dealHand(List<E> deck, int n) {
    int deckSize = deck.size();
    List<E> handView = deck.subList(deckSize - n, deckSize);
    List<E> hand = new ArrayList<E>(handView);
    handView.clear();
    return hand;
}
```

Note that this algorithm removes the hand from the *end* of the deck. For many common `List` implementations, such as `ArrayList`, the performance of removing elements from the end of the list is substantially better than that of removing elements from the beginning.

The following is [a program](#) that uses the `dealHand` method in combination with `Collections.shuffle` to generate hands from a normal 52-card deck. The program takes two command-line arguments: (1) the number of hands to deal and (2) the number of cards in each hand.

```
import java.util.*;

public class Deal {
    public static void main(String[] args) {
        if (args.length < 2) {
            System.out.println("Usage: Deal hands cards");
            return;
        }
        int numHands = Integer.parseInt(args[0]);
        int cardsPerHand = Integer.parseInt(args[1]);

        // Make a normal 52-card deck.
        String[] suit = new String[] {
            "spades", "hearts",
            "diamonds", "clubs"
        };
    }
}
```

```

String[] rank = new String[] {
    "ace", "2", "3", "4",
    "5", "6", "7", "8", "9", "10",
    "jack", "queen", "king"
};

List<String> deck = new ArrayList<String>();
for (int i = 0; i < suit.length; i++)
    for (int j = 0; j < rank.length; j++)
        deck.add(rank[j] + " of " + suit[i]);

// Shuffle the deck.
Collections.shuffle(deck);

if (numHands * cardsPerHand > deck.size()) {
    System.out.println("Not enough cards.");
    return;
}

for (int i = 0; i < numHands; i++)
    System.out.println(dealHand(deck, cardsPerHand));
}

public static <E> List<E> dealHand(List<E> deck, int n) {
    int deckSize = deck.size();
    List<E> handView = deck.subList(deckSize - n, deckSize);
    List<E> hand = new ArrayList<E>(handView);
    handView.clear();
    return hand;
}
}

```

Running the program produces output like the following.

```
% java Deal 4 5

[8 of hearts, jack of spades, 3 of spades, 4 of spades,
 king of diamonds]
[4 of diamonds, ace of clubs, 6 of clubs, jack of hearts,
 queen of hearts]
[7 of spades, 5 of spades, 2 of diamonds, queen of diamonds,
 9 of clubs]
[8 of spades, 6 of diamonds, ace of spades, 3 of hearts,
 ace of hearts]
```

Although the `subList` operation is extremely powerful, some care must be exercised when using it. The semantics of the `List` returned by `subList` become undefined if elements are added to or removed from the backing `List` in any way other than via the returned `List`. Thus, it's highly recommended that you use the `List` returned by `subList` only as a transient object — to perform one or a sequence of range operations on the backing `List`. The longer you use the `subList` instance, the greater the probability that you'll compromise it by modifying the backing `List` directly or through another `subList` object. Note that it is legal to modify a sublist of a sublist and to continue using the original sublist (though not concurrently).

## List Algorithms

Most polymorphic algorithms in the `Collections` class apply specifically to `List`. Having all these algorithms at your disposal makes it very easy to manipulate lists. Here's a summary of these algorithms, which are described in more detail in the [Algorithms](#) section.

- `sort` — sorts a `List` using a merge sort algorithm, which provides a fast, stable sort. (A *stable sort* is one that does not reorder equal elements.)
- `shuffle` — randomly permutes the elements in a `List`.
- `reverse` — reverses the order of the elements in a `List`.
- `rotate` — rotates all the elements in a `List` by a specified distance.
- `swap` — swaps the elements at specified positions in a `List`.
- `replaceAll` — replaces all occurrences of one specified value with another.
- `fill` — overwrites every element in a `List` with the specified value.
- `copy` — copies the source `List` into the destination `List`.
- `binarySearch` — searches for an element in an ordered `List` using the binary search algorithm.
- `indexOfSubList` — returns the index of the first sublist of one `List` that is equal to another.
- `lastIndexOfSubList` — returns the index of the last sublist of one `List` that is equal to another.

## The Queue Interface

- A `Queue` is a collection for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, removal, and inspection operations. The `Queue` interface follows.
- ```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```
- Each `Queue` method exists in two forms: (1) one throws an exception if the operation fails, and (2) the other returns a special value if the operation fails (either `null` or `false`, depending on the operation). The regular structure of the interface is illustrated in the following table.

**Queue Interface Structure**

| Type of Operation | Throws exception       | Returns special value |
|-------------------|------------------------|-----------------------|
| Insert            | <code>add(e)</code>    | <code>offer(e)</code> |
| Remove            | <code>remove()</code>  | <code>poll()</code>   |
| Examine           | <code>element()</code> | <code>peek()</code>   |

•

- Queues typically, but not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to their values — see the [Object Ordering](#) section for details). Whatever ordering is used, the head of the queue is the element that would be removed by a call to `remove` or `poll`. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every `Queue` implementation must specify its ordering properties.
- It is possible for a `Queue` implementation to restrict the number of elements that it holds; such queues are known as *bounded*. Some `Queue` implementations in `java.util.concurrent` are bounded, but the implementations in `java.util` are not.
- The `add` method, which `Queue` inherits from `Collection`, inserts an element unless it would violate the queue's capacity restrictions, in which case it throws `IllegalStateException`. The `offer` method, which is intended solely for use on bounded queues, differs from `add` only in that it indicates failure to insert an element by returning `false`.
- The `remove` and `poll` methods both remove and return the head of the queue. Exactly which element gets removed is a function of the queue's ordering policy. The `remove` and `poll` methods differ in their behavior only when the queue is empty. Under these circumstances, `remove` throws `NoSuchElementException`, while `poll` returns `null`.
- The `element` and `peek` methods return, but do not remove, the head of the queue. They differ from one another in precisely the same fashion as `remove` and `poll`: If the queue is empty, `element` throws `NoSuchElementException`, while `peek` returns `null`.
- `Queue` implementations generally do not allow insertion of `null` elements. The `LinkedList` implementation, which was retrofitted to implement `Queue`, is an exception. For historical reasons, it permits `null` elements, but you should refrain from taking advantage of this, because `null` is used as a special return value by the `poll` and `peek` methods.
- `Queue` implementations generally do not define element-based versions of the `equals` and `hashCode` methods but instead inherit the identity-based versions from `Object`.
- The `Queue` interface does not define the blocking queue methods, which are common in concurrent programming. These methods, which wait for elements to appear or for space to become available, are defined in the interface [java.util.concurrent.BlockingQueue](#), which extends `Queue`.
- In the following example program, a queue is used to implement a countdown timer. The queue is preloaded with all the integer values from a number specified on the command line to zero, in descending order. Then, the values are removed from the queue and printed at one-second intervals. The program is artificial in that it would be more natural to do the same thing without using a queue, but it illustrates the use of a queue to store elements prior to subsequent processing.

```
import java.util.*;  
  
public class Countdown {  
    public static void main(String[] args) throws InterruptedException {  
        int time = Integer.parseInt(args[0]);  
        Queue<Integer> queue = new LinkedList<Integer>();  
        for (int i = time; i >= 0; i--)  
            queue.add(i);  
        while (!queue.isEmpty())  
            System.out.println(queue.poll());  
        Thread.sleep(1000);  
    }  
}
```

```

        for (int i = time; i >= 0; i--)
            queue.add(i);

        while (!queue.isEmpty()) {
            System.out.println(queue.remove());
            Thread.sleep(1000);
        }
    }
}

```

In the following example, a priority queue is used to sort a collection of elements. Again this program is artificial in that there is no reason to use it in favor of the `sort` method provided in `Collections`, but it illustrates the behavior of priority queues.

```

static <E> List<E> heapSort(Collection<E> c) {
    Queue<E> queue = new PriorityQueue<E>(c);
    List<E> result = new ArrayList<E>();

    while (!queue.isEmpty())
        result.add(queue.remove());

    return result;
}

```

## The Map Interface

A [Map](#) is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. It models the mathematical *function* abstraction. The `Map` interface includes methods for basic operations (such as `put`, `get`, `remove`, `containsKey`, `containsValue`, `size`, and `empty`), bulk operations (such as `putAll` and `clear`), and collection views (such as `keySet`, `entrySet`, and `values`).

The Java platform contains three general-purpose `Map` implementations: [HashMap](#), [TreeMap](#), and [LinkedHashMap](#). Their behavior and performance are precisely analogous to `HashSet`, `TreeSet`, and `LinkedHashSet`, as described in [The Set Interface](#) section.

The remainder of this page discusses the `Map` interface in detail. But first, here are some more examples of collecting to `Maps` using JDK 8 aggregate operations. Modeling real-world objects is a common task in object-oriented programming, so it is reasonable to think that some programs might, for example, group employees by department:

```
// Group employees by department
Map<Department, List<Employee>> byDept = employees.stream()
.collect(Collectors.groupingBy(Employee::getDepartment));
```

Or compute the sum of all salaries by department:

```
// Compute sum of salaries by department
```

```
Map<Department, Integer> totalByDept = employees.stream()
.collect(Collectors.groupingBy(Employee::getDepartment,
Collectors.summingInt(Employee::getSalary)));
```

Or perhaps group students by passing or failing grades:

```
// Partition students into passing and failing
Map<Boolean, List<Student>> passingFailing = students.stream()
.collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD));
```

You could also group people by city:

```
// Classify Person objects by city
Map<String, List<Person>> peopleByCity
    = personStream.collect(Collectors.groupingBy(Person::getCity));
```

Or even cascade two collectors to classify people by state and city:

```
// Cascade Collectors
Map<String, Map<String, List<Person>>> peopleByStateAndCity
    = personStream.collect(Collectors.groupingBy(Person::getState,
    Collectors.groupingBy(Person::getCity)))
```

Again, these are but a few examples of how to use the new JDK 8 APIs. For in-depth coverage of lambda expressions and aggregate operations see the lesson entitled [Aggregate Operations](#).

## Map Interface Basic Operations

The basic operations of `Map` (`put`, `get`, `containsKey`, `containsValue`, `size`, and `isEmpty`) behave exactly like their counterparts in `Hashtable`. The [following program](#) generates a frequency table of the words found in its argument list. The frequency table maps each word to the number of times it occurs in the argument list.

```
import java.util.*;

public class Freq {
    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<String, Integer>();

        // Initialize frequency table from command line
        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }

        System.out.println(m.size() + " distinct words:");
        System.out.println(m);
    }
}
```

The only tricky thing about this program is the second argument of the `put` statement. That argument is a conditional expression that has the effect of setting the frequency to one if the word has never been seen before or one more than its current value if the word has already been seen. Try running this program with the command:

```
java Freq if it is to be it is up to me to delegate
```

The program yields the following output.

```
8 distinct words:  
{to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}
```

Suppose you'd prefer to see the frequency table in alphabetical order. All you have to do is change the implementation type of the `Map` from `HashMap` to `TreeMap`. Making this four-character change causes the program to generate the following output from the same command line.

```
8 distinct words:  
{be=1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1}
```

Similarly, you could make the program print the frequency table in the order the words first appear on the command line simply by changing the implementation type of the map to `LinkedHashMap`. Doing so results in the following output.

```
8 distinct words:  
{if=1, it=2, is=2, to=3, be=1, up=1, me=1, delegate=1}
```

This flexibility provides a potent illustration of the power of an interface-based framework.

Like the [Set](#) and [List](#) interfaces, `Map` strengthens the requirements on the `equals` and `hashCode` methods so that two `Map` objects can be compared for logical equality without regard to their implementation types. Two `Map` instances are equal if they represent the same key-value mappings.

By convention, all general-purpose `Map` implementations provide constructors that take a `Map` object and initialize the new `Map` to contain all the key-value mappings in the specified `Map`. This standard `Map` conversion constructor is entirely analogous to the standard `Collection` constructor: It allows the caller to create a `Map` of a desired implementation type that initially contains all of the mappings in another `Map`, regardless of the other `Map`'s implementation type. For example, suppose you have a `Map`, named `m`. The following one-liner creates a new `HashMap` initially containing all of the same key-value mappings as `m`.

```
Map<K, V> copy = new HashMap<K, V>(m);
```

## Map Interface Bulk Operations

The `clear` operation does exactly what you would think it could do: It removes all the mappings from the `Map`. The `putAll` operation is the `Map` analogue of the `Collection` interface's `addAll` operation. In addition to its obvious use of dumping one `Map` into another, it has a second, more subtle use. Suppose a `Map` is used to represent a collection of attribute-value pairs; the `putAll` operation, in combination with the `Map` conversion constructor, provides a neat way to implement attribute map creation with default values. The following is a static factory method that demonstrates this technique.

```
static <K, V> Map<K, V> newAttributeMap(Map<K, V> defaults, Map<K, V>
overrides) {
    Map<K, V> result = new HashMap<K, V>(defaults);
    result.putAll(overrides);
    return result;
}
```

## Collection Views

The `Collection` view methods allow a `Map` to be viewed as a `Collection` in these three ways:

- `keySet` — the `Set` of keys contained in the `Map`.
- `values` — The `Collection` of values contained in the `Map`. This `Collection` is not a `Set`, because multiple keys can map to the same value.
- `entrySet` — the `Set` of key-value pairs contained in the `Map`. The `Map` interface provides a small nested interface called `Map.Entry`, the type of the elements in this `Set`.

The `Collection` views provide the *only* means to iterate over a `Map`. This example illustrates the standard idiom for iterating over the keys in a `Map` with a `for-each` construct:

```
for (KeyType key : m.keySet())
    System.out.println(key);
```

and with an iterator:

```
// Filter a map based on some
// property of its keys.
for (Iterator<Type> it = m.keySet().iterator(); it.hasNext(); )
    if (it.next().isBogus())
        it.remove();
```

The idiom for iterating over values is analogous. Following is the idiom for iterating over key-value pairs.

```
for (Map.Entry<KeyType, ValType> e : m.entrySet())
    System.out.println(e.getKey() + ":" + e.getValue());
```

At first, many people worry that these idioms may be slow because the `Map` has to create a new `Collection` instance each time a `Collection` view operation is called. Rest easy: There's no

reason that a `Map` cannot always return the same object each time it is asked for a given `Collection` view. This is precisely what all the `Map` implementations in `java.util` do.

With all three `Collection` views, calling an `Iterator`'s `remove` operation removes the associated entry from the backing `Map`, assuming that the backing `Map` supports element removal to begin with. This is illustrated by the preceding filtering idiom.

With the `entrySet` view, it is also possible to change the value associated with a key by calling a `Map.Entry`'s `setValue` method during iteration (again, assuming the `Map` supports value modification to begin with). Note that these are the *only* safe ways to modify a `Map` during iteration; the behavior is unspecified if the underlying `Map` is modified in any other way while the iteration is in progress.

The `Collection` views support element removal in all its many forms — `remove`, `removeAll`, `retainAll`, and `clear` operations, as well as the `Iterator.remove` operation. (Yet again, this assumes that the backing `Map` supports element removal.)

The `Collection` views *do not* support element addition under any circumstances. It would make no sense for the `keySet` and `values` views, and it's unnecessary for the `entrySet` view, because the backing `Map`'s `put` and `putAll` methods provide the same functionality.

## Fancy Uses of Collection Views: Map Algebra

When applied to the `Collection` views, bulk operations (`containsAll`, `removeAll`, and `retainAll`) are surprisingly potent tools. For starters, suppose you want to know whether one `Map` is a submap of another — that is, whether the first `Map` contains all the key-value mappings in the second. The following idiom does the trick.

```
if (m1.entrySet().containsAll(m2.entrySet())) {
    ...
}
```

Along similar lines, suppose you want to know whether two `Map` objects contain mappings for all of the same keys.

```
if (m1.keySet().equals(m2.keySet())) {
    ...
}
```

Suppose you have a `Map` that represents a collection of attribute-value pairs, and two `Sets` representing required attributes and permissible attributes. (The permissible attributes include the required attributes.) The following snippet determines whether the attribute map conforms to these constraints and prints a detailed error message if it doesn't.

```
static <K, V> boolean validate(Map<K, V> attrMap, Set<K> requiredAttrs,
Set<K>permittedAttrs) {
    boolean valid = true;
```

```

Set<K> attrs = attrMap.keySet();

if (! attrs.containsAll(requiredAttrs)) {
    Set<K> missing = new HashSet<K>(requiredAttrs);
    missing.removeAll(attrs);
    System.out.println("Missing attributes: " + missing);
    valid = false;
}
if (! permittedAttrs.containsAll(attrs)) {
    Set<K> illegal = new HashSet<K>(attrs);
    illegal.removeAll(permittedAttrs);
    System.out.println("Illegal attributes: " + illegal);
    valid = false;
}
return valid;
}

```

Suppose you want to know all the keys common to two `Map` objects.

```

Set<KeyType>commonKeys = new HashSet<KeyType>(m1.keySet());
commonKeys.retainAll(m2.keySet());

```

A similar idiom gets you the common values.

All the idioms presented thus far have been nondestructive; that is, they don't modify the backing `Map`. Here are a few that do. Suppose you want to remove all of the key-value pairs that one `Map` has in common with another.

```
m1.entrySet().removeAll(m2.entrySet());
```

Suppose you want to remove from one `Map` all of the keys that have mappings in another.

```
m1.keySet().removeAll(m2.keySet());
```

What happens when you start mixing keys and values in the same bulk operation? Suppose you have a `Map`, `managers`, that maps each employee in a company to the employee's manager. We'll be deliberately vague about the types of the key and the value objects. It doesn't matter, as long as they're the same. Now suppose you want to know who all the "individual contributors" (or nonmanagers) are. The following snippet tells you exactly what you want to know.

```

Set<Employee>           individualContributors      =      new
HashSet<Employee>(managers.keySet());
individualContributors.removeAll(managers.values());

```

Suppose you want to fire all the employees who report directly to some manager, Simon.

```

Employee simon = ... ;
managers.values().removeAll(Collections.singleton(simon));

```

Note that this idiom makes use of `Collections.singleton`, a static factory method that returns an immutable `Set` with the single, specified element.

Once you've done this, you may have a bunch of employees whose managers no longer work for the company (if any of Simon's direct-reports were themselves managers). The following code will tell you which employees have managers who no longer works for the company.

```
Map<Employee, Employee> m = new HashMap<Employee, Employee>(managers);
m.values().removeAll(managers.keySet());
Set<Employee> slackers = m.keySet();
```

This example is a bit tricky. First, it makes a temporary copy of the `Map`, and it removes from the temporary copy all entries whose `(manager)` value is a key in the original `Map`. Remember that the original `Map` has an entry for each employee. Thus, the remaining entries in the temporary `Map` comprise all the entries from the original `Map` whose `(manager)` values are no longer employees. The keys in the temporary copy, then, represent precisely the employees that we're looking for.

There are many more idioms like the ones contained in this section, but it would be impractical and tedious to list them all. Once you get the hang of it, it's not that difficult to come up with the right one when you need it.

## Multimaps

A *mymap* is like a `Map` but it can map each key to multiple values. The Java Collections Framework doesn't include an interface for multimaps because they aren't used all that commonly. It's a fairly simple matter to use a `Map` whose values are `List` instances as a multimap. This technique is demonstrated in the next code example, which reads a word list containing one word per line (all lowercase) and prints out all the anagram groups that meet a size criterion. An *anagram group* is a bunch of words, all of which contain exactly the same letters but in a different order. The program takes two arguments on the command line: (1) the name of the dictionary file and (2) the minimum size of anagram group to print out. Anagram groups containing fewer words than the specified minimum are not printed.

There is a standard trick for finding anagram groups: For each word in the dictionary, alphabetize the letters in the word (that is, reorder the word's letters into alphabetical order) and put an entry into a multimap, mapping the alphabetized word to the original word. For example, the word *bad* causes an entry mapping *abd* into *bad* to be put into the multimap. A moment's reflection will show that all the words to which any given key maps form an anagram group. It's a simple matter to iterate over the keys in the multimap, printing out each anagram group that meets the size constraint.

[The following program](#) is a straightforward implementation of this technique.

```
import java.util.*;
import java.io.*;

public class Anagrams {
    public static void main(String[] args) {
        int minGroupSize = Integer.parseInt(args[1]);
    }
}
```

---

```

// Read words from file and put into a simulated multimap
Map<String, List<String>> m = new HashMap<String, List<String>>();

try {
    Scanner s = new Scanner(new File(args[0]));
    while (s.hasNext()) {
        String word = s.next();
        String alpha = alphabetize(word);
        List<String> l = m.get(alpha);
        if (l == null)
            m.put(alpha, l=new ArrayList<String>());
        l.add(word);
    }
} catch (IOException e) {
    System.err.println(e);
    System.exit(1);
}

// Print all permutation groups above size threshold
for (List<String> l : m.values())
    if (l.size() >= minGroupSize)
        System.out.println(l.size() + ":" + l);
}

private static String alphabetize(String s) {
    char[] a = s.toCharArray();
    Arrays.sort(a);
    return new String(a);
}
}

```

Running this program on a 173,000-word dictionary file with a minimum anagram group size of eight produces the following output.

```

9: [estrin, inerts, insert, inters, niters, nitres, sinter,
   triens, trines]
8: [lapse, leaps, pales, peals, pleas, salep, sepal, spale]
8: [aspers, parses, passer, prases, repass, spares, sparse,
   spears]
10: [least, setal, slate, stale, steal, stela, taela, tales,
    teals, tesla]
8: [enters, nester, renest, rentes, resent, tenser, ternes,
   treens]
8: [arles, earls, lares, laser, lears, rales, reals, serial]
8: [earings, erasing, gainers, reagins, regains, reginas,
   searing, seringa]
8: [peris, piers, pries, prise, ripes, speir, spier, spire]
12: [apers, apres, asper, pares, parse, pears, prase, presa,
    rapes, reaps, spare, spear]
11: [alerts, alters, artels, estral, laster, ratels, salter,
    slater, staler, stelar, talers]
9: [capers, crapes, escarp, pacers, parsec, recaps, scrape,
   secpar, spacer]
9: [palest, palets, pastel, petals, plates, pleats, septal,
   staple, tepals]

```

```

9: [anestri, antsier, nastier, ratines, retains, retinas,
   retsina, stainer, stearin]
8: [ates, east, eats, etas, sate, seat, seta, teas]
8: [carets, cartes, caster, caters, crates, reacts, recast,
   traces]
```

## The SortedSet Interface

A [SortedSet](#) is a [Set](#) that maintains its elements in ascending order, sorted according to the elements' natural ordering or according to a [Comparator](#) provided at [SortedSet](#) creation time. In addition to the normal [Set](#) operations, the [SortedSet](#) interface provides operations for the following:

- Range view — allows arbitrary range operations on the sorted set
- Endpoints — returns the first or last element in the sorted set
- Comparator access — returns the [Comparator](#), if any, used to sort the set

The code for the [SortedSet](#) interface follows.

```

public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

## Set Operations

The operations that [SortedSet](#) inherits from [Set](#) behave identically on sorted sets and normal sets with two exceptions:

- The [Iterator](#) returned by the [iterator](#) operation traverses the sorted set in order.
- The array returned by [toArray](#) contains the sorted set's elements in order.

Although the interface doesn't guarantee it, the [toString](#) method of the Java platform's [SortedSet](#) implementations returns a string containing all the elements of the sorted set, in order.

## Standard Constructors

---

By convention, all general-purpose `Collection` implementations provide a standard conversion constructor that takes a `Collection`; `SortedSet` implementations are no exception. In `TreeSet`, this constructor creates an instance that sorts its elements according to their natural ordering. This was probably a mistake. It would have been better to check dynamically to see whether the specified collection was a `SortedSet` instance and, if so, to sort the new `TreeSet` according to the same criterion (comparator or natural ordering). Because `TreeSet` took the approach that it did, it also provides a constructor that takes a `SortedSet` and returns a new `TreeSet` containing the same elements sorted according to the same criterion. Note that it is the compile-time type of the argument, not its runtime type, that determines which of these two constructors is invoked (and whether the sorting criterion is preserved).

`SortedSet` implementations also provide, by convention, a constructor that takes a `Comparator` and returns an empty set sorted according to the specified `Comparator`. If `null` is passed to this constructor, it returns a set that sorts its elements according to their natural ordering.

## Range-view Operations

The `range-view` operations are somewhat analogous to those provided by the `List` interface, but there is one big difference. Range views of a sorted set remain valid even if the backing sorted set is modified directly. This is feasible because the endpoints of a range view of a sorted set are absolute points in the element space rather than specific elements in the backing collection, as is the case for lists. A `range-view` of a sorted set is really just a window onto whatever portion of the set lies in the designated part of the element space. Changes to the `range-view` write back to the backing sorted set and vice versa. Thus, it's okay to use `range-views` on sorted sets for long periods of time, unlike `range-views` on lists.

Sorted sets provide three `range-view` operations. The first, `subSet`, takes two endpoints, like `subList`. Rather than indices, the endpoints are objects and must be comparable to the elements in the sorted set, using the set's `Comparator` or the natural ordering of its elements, whichever the set uses to order itself. Like `subList`, the range is half open, including its low endpoint but excluding the high one.

Thus, the following line of code tells you how many words between "doorbell" and "pickle", including "doorbell" but excluding "pickle", are contained in a `SortedSet` of strings called `dictionary`:

```
int count = dictionary.subSet("doorbell", "pickle").size();
```

In like manner, the following one-liner removes all the elements beginning with the letter `f`.

```
dictionary.subSet("f", "g").clear();
```

A similar trick can be used to print a table telling you how many words begin with each letter.

```
for (char ch = 'a'; ch <= 'z'; ) {  
    String from = String.valueOf(ch++);
```

```

String to = String.valueOf(ch);
System.out.println(from + ":" + dictionary.subSet(from, to).size());
}

```

Suppose you want to view a *closed interval*, which contains both of its endpoints, instead of an open interval. If the element type allows for the calculation of the successor of a given value in the element space, merely request the subSet from lowEndpoint to successor(highEndpoint). Although it isn't entirely obvious, the successor of a string s in String's natural ordering is s + "\0" — that is, s with a null character appended.

Thus, the following one-liner tells you how many words between "doorbell" and "pickle", including doorbell *and* pickle, are contained in the dictionary.

```
count = dictionary.subSet("doorbell", "pickle\0").size();
```

A similar technique can be used to view an *open interval*, which contains neither endpoint. The open-interval view from lowEndpoint to highEndpoint is the half-open interval from successor(lowEndpoint) to highEndpoint. Use the following to calculate the number of words between "doorbell" and "pickle", excluding both.

```
count = dictionary.subSet("doorbell\0", "pickle").size();
```

The SortedSet interface contains two more range-view operations — headSet and tailSet, both of which take a single Object argument. The former returns a view of the initial portion of the backing SortedSet, up to but not including the specified object. The latter returns a view of the final portion of the backing SortedSet, beginning with the specified object and continuing to the end of the backing SortedSet. Thus, the following code allows you to view the dictionary as two disjoint volumes (a-m and n-z).

```
SortedSet<String> volume1 = dictionary.headSet("n");
SortedSet<String> volume2 = dictionary.tailSet("n");
```

## Endpoint Operations

The SortedSet interface contains operations to return the first and last elements in the sorted set, not surprisingly called `first` and `last`. In addition to their obvious uses, `last` allows a workaround for a deficiency in the SortedSet interface. One thing you'd like to do with a SortedSet is to go into the interior of the set and iterate forward or backward. It's easy enough to go forward from the interior: Just get a `tailSet` and iterate over it. Unfortunately, there's no easy way to go backward.

The following idiom obtains the first element that is less than a specified object o in the element space.

```
Object predecessor = ss.headSet(o).last();
```

This is a fine way to go one element backward from a point in the interior of a sorted set. It could be applied repeatedly to iterate backward, but this is very inefficient, requiring a lookup for each element returned.

## Comparator Accessor

The `SortedSet` interface contains an accessor method called `comparator` that returns the `Comparator` used to sort the set, or `null` if the set is sorted according to the *natural ordering* of its elements. This method is provided so that sorted sets can be copied into new sorted sets with the same ordering.

# The SortedMap Interface

A `SortedMap` is a `Map` that maintains its entries in ascending order, sorted according to the keys' natural ordering, or according to a `Comparator` provided at the time of the `SortedMap` creation. Natural ordering and `Comparators` are discussed in the [Object Ordering](#) section. The `SortedMap` interface provides operations for normal `Map` operations and for the following:

- Range view — performs arbitrary range operations on the sorted map
- Endpoints — returns the first or the last key in the sorted map
- Comparator access — returns the `Comparator`, if any, used to sort the map

The following interface is the `Map` analog of [`SortedSet`](#).

```
public interface SortedMap<K, V> extends Map<K, V>{  
    Comparator<? super K> comparator();  
    SortedMap<K, V> subMap(K fromKey, K toKey);  
    SortedMap<K, V> headMap(K toKey);  
    SortedMap<K, V> tailMap(K fromKey);  
    K firstKey();  
    K lastKey();  
}
```

## Map Operations

The operations `SortedMap` inherits from `Map` behave identically on sorted maps and normal maps with two exceptions:

- The `Iterator` returned by the `iterator` operation on any of the sorted map's `Collection` views traverse the collections in order.
- The arrays returned by the `Collection` views' `toArray` operations contain the keys, values, or entries in order.

Although it isn't guaranteed by the interface, the `toString` method of the `Collection` views in all the Java platform's `SortedMap` implementations returns a string containing all the elements of the view, in order.

## Standard Constructors

By convention, all general-purpose `Map` implementations provide a standard conversion constructor that takes a `Map`; `SortedMap` implementations are no exception. In `TreeMap`, this constructor creates an instance that orders its entries according to their keys' natural ordering. This was probably a mistake. It would have been better to check dynamically to see whether the specified `Map` instance was a `SortedMap` and, if so, to sort the new map according to the same criterion (comparator or natural ordering). Because `TreeMap` took the approach it did, it also provides a constructor that takes a `SortedMap` and returns a new `TreeMap` containing the same mappings as the given `SortedMap`, sorted according to the same criterion. Note that it is the compile-time type of the argument, not its runtime type, that determines whether the `SortedMap` constructor is invoked in preference to the ordinary `map` constructor.

`SortedMap` implementations also provide, by convention, a constructor that takes a `Comparator` and returns an empty map sorted according to the specified `Comparator`. If `null` is passed to this constructor, it returns a `Map` that sorts its mappings according to their keys' natural ordering.

## Comparison to SortedSet

Because this interface is a precise `Map` analog of `SortedSet`, all the idioms and code examples in [The SortedSet Interface](#) section apply to `SortedMap` with only trivial modifications.

## MODULE – 3

### STRING METHODS

```
public final class String  
extends Object  
implements Serializable, Comparable<String>, CharSequence
```

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};  
String str = new String(data);
```

Here are some more examples of how strings can be used:

```
System.out.println("abc");  
String cde = "cde";  
System.out.println("abc" + cde);  
String c = "abc".substring(2,3);  
String d = cde.substring(1, 2);
```

The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase. Case mapping is based on the Unicode Standard version specified by the [Character](#) class.

The Java language provides special support for the string concatenation operator ( + ), and for conversion of other objects to strings. String concatenation is implemented through the [StringBuilder](#)(or [StringBuffer](#)) class and its append method. String conversions are implemented through the method [toString](#), defined by Object and inherited by all classes in Java. For additional information on string concatenation and conversion, see Gosling, Joy, and Steele, *The Java Language Specification*.

Unless otherwise noted, passing a null argument to a constructor or method in this class will cause a [NullPointerException](#) to be thrown.

A String represents a string in the UTF-16 format in which *supplementary characters* are represented by *surrogate pairs* (see the section [Unicode Character Representations](#) in the Character class for more information). Index values refer to char code units, so a supplementary character uses two positions in a String.

The String class provides methods for dealing with Unicode code points (i.e., characters), in addition to those for dealing with Unicode code units (i.e., char values).

Constructors

### Constructor and Description

[String\(\)](#)

Initializes a newly created String object so that it represents an empty character sequence.

[String\(byte\[\] bytes\)](#)

Constructs a new String by decoding the specified array of bytes using the platform's default charset.

[String\(byte\[\] bytes, Charset charset\)](#)

Constructs a new String by decoding the specified array of bytes using the specified [charset](#).

[String\(byte\[\] ascii, int hibyte\)](#)

Deprecated.

This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a [Charset](#), charset name, or that use the platform's default charset.

[String\(byte\[\] bytes, int offset, int length\)](#)

Constructs a new String by decoding the specified subarray of bytes using the platform's default charset.

[String\(byte\[\] bytes, int offset, int length, Charset charset\)](#)

Constructs a new String by decoding the specified subarray of bytes using the specified [charset](#).

[String\(byte\[\] ascii, int hibyte, int offset, int count\)](#)

Deprecated.

This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a [Charset](#), charset name, or that use the platform's default charset.

[String\(byte\[\] bytes, int offset, int length, String charsetName\)](#)

Constructs a new String by decoding the specified subarray of bytes using the specified charset.

[String\(byte\[\] bytes, String charsetName\)](#)

Constructs a new String by decoding the specified array of bytes using the specified [charset](#).

[String\(char\[\] value\)](#)

Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

[String\(char\[\] value, int offset, int count\)](#)

Allocates a new String that contains characters from a subarray of the character array argument.

[String\(int\[\] codePoints, int offset, int count\)](#)

Allocates a new String that contains characters from a subarray of the [Unicode code point](#) array argument.

**[String\(String original\)](#)**

Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

**[String\(StringBuffer buffer\)](#)**

Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.

**[String\(StringBuilder builder\)](#)**

Allocates a new string that contains the sequence of characters currently contained in the string builder argument.

`public static final Comparator<String> CASE_INSENSITIVE_ORDER`

A Comparator that orders String objects as by compareToIgnoreCase. This comparator is serializable.

Note that this Comparator does *not* take locale into account, and will result in an unsatisfactory ordering for certain locales. The java.text package provides *Collators* to allow locale-sensitive ordering.

- - **String**

`public String()`

Initializes a newly created String object so that it represents an empty character sequence. Note that use of this constructor is unnecessary since Strings are immutable.

- **String**

`public String(String original)`

Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string. Unless an explicit copy of original is needed, use of this constructor is unnecessary since Strings are immutable.

Parameters:

original - A String

- **String**

`public String(char[] value)`

Allocates a new String so that it represents the sequence of characters currently contained in the character array argument. The contents of the character array are copied; subsequent modification of the character array does not affect the newly created string.

Parameters:

value - The initial value of the string

- **String**
- public String(char[] value,
- int offset,
- int count)

Allocates a new String that contains characters from a subarray of the character array argument. The offset argument is the index of the first character of the subarray and the count argument specifies the length of the subarray. The contents of the subarray are copied; subsequent modification of the character array does not affect the newly created string.

Parameters:

value - Array that is the source of characters

offset - The initial offset

count - The length

Throws:

[IndexOutOfBoundsException](#) - If the offset and count arguments index characters outside the bounds of the value array

- **String**
- public String(int[] codePoints,
- int offset,
- int count)

Allocates a new String that contains characters from a subarray of the [Unicode code point](#) array argument. The offset argument is the index of the first code point of the subarray and the count argument specifies the length of the subarray. The contents of the subarray are converted to chars; subsequent modification of the int array does not affect the newly created string.

Parameters:

codePoints - Array that is the source of Unicode code points

offset - The initial offset

count - The length

Throws:

[IllegalArgumentException](#) - If any invalid Unicode code point is found in codePoints

[IndexOutOfBoundsException](#) - If the offset and count arguments index characters outside the bounds of the codePoints array

Since: 1.5

- **String**
- [@Deprecated](#)
- public String(byte[] ascii,  
                  int hibyte,  
                  int offset,  
                  int count)

Deprecated. This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a [Charset](#), charset name, or that use the platform's default charset.

Allocates a new String constructed from a subarray of an array of 8-bit integer values.

The offset argument is the index of the first byte of the subarray, and the count argument specifies the length of the subarray.

Each byte in the subarray is converted to a char as specified in the method above.

Parameters:

ascii - The bytes to be converted to characters  
 hibyte - The top 8 bits of each 16-bit Unicode code unit  
 offset - The initial offset  
 count - The length

Throws:

[IndexOutOfBoundsException](#) - If the offset or count argument is invalid

See Also:

[String\(byte\[\], int\)](#), [String\(byte\[\], int, int, java.lang.String\)](#), [String\(byte\[\], int, int, java.nio.charset.Charset\)](#), [String\(byte\[\], int, int\)](#), [String\(byte\[\], java.lang.String\)](#), [String\(byte\[\], java.nio.charset.Charset\)](#), [String\(byte\[\]\)](#)

- **String**
- [@Deprecated](#)
- public String(byte[] ascii,  
                  int hibyte)

Deprecated. This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a [Charset](#), charset name, or that use the platform's default charset.

Allocates a new String containing characters constructed from an array of 8-bit integer values. Each character *c* in the resulting string is constructed from the corresponding component *b* in the byte array such that:

$$c == (\text{char})(((\text{hibyte} \& 0xff) << 8) \\ | (\text{b} \& 0xff))$$

Parameters:

ascii - The bytes to be converted to characters

hibyte - The top 8 bits of each 16-bit Unicode code unit

See Also:

[String\(byte\[\], int, int, java.lang.String\)](#), [String\(byte\[\], int, int, java.nio.charset.Charset\)](#),  
[String\(byte\[\], int, int\)](#), [String\(byte\[\], java.lang.String\)](#), [String\(byte\[\], java.nio.charset.Charset\)](#),  
[String\(byte\[\]\)](#)

- **String**
- public String(byte[] bytes,
- int offset,
- int length,
- **String** charsetName)  
    throws [UnsupportedEncodingException](#)

Constructs a new String by decoding the specified subarray of bytes using the specified charset.

The length of the new String is a function of the charset, and hence may not be equal to the length of the subarray.

The behavior of this constructor when the given bytes are not valid in the given charset is unspecified. The [CharsetDecoder](#) class should be used when more control over the decoding process is required.

Parameters:

bytes - The bytes to be decoded into characters

offset - The index of the first byte to decode

length - The number of bytes to decode

charsetName - The name of a supported [charset](#)

Throws:

[UnsupportedEncodingException](#) - If the named charset is not supported

[IndexOutOfBoundsException](#) - If the offset and length arguments index characters outside the bounds of the bytes array

Since:

JDK1.1

- **String**
- public String(byte[] bytes,
- int offset,
- int length,
- **Charset** charset)

Constructs a new String by decoding the specified subarray of bytes using the specified [charset](#).

The length of the new String is a function of the charset, and hence may not be equal to the length of the subarray.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement string. The [CharsetDecoder](#) class should be used when more control over the decoding process is required.

Parameters:

bytes - The bytes to be decoded into characters  
offset - The index of the first byte to decode  
length - The number of bytes to decode  
charset - The [charset](#) to be used to decode the bytes

Throws:

[IndexOutOfBoundsException](#) - If the offset and length arguments index characters outside the bounds of the bytes array

Since:

1.6

- o **String**
- o public String(byte[] bytes,
- o       [String](#) charsetName)  
throws [UnsupportedEncodingException](#)

Constructs a new String by decoding the specified array of bytes using the specified [charset](#). The length of the new String is a function of the charset, and hence may not be equal to the length of the byte array.

The behavior of this constructor when the given bytes are not valid in the given charset is unspecified. The [CharsetDecoder](#) class should be used when more control over the decoding process is required.

Parameters:

bytes - The bytes to be decoded into characters  
charsetName - The name of a supported [charset](#)

Throws:

[UnsupportedEncodingException](#) - If the named charset is not supported

Since:

JDK1.1

- o **String**
- o public String(byte[] bytes,  
[Charset](#) charset)

Constructs a new String by decoding the specified array of bytes using the specified [charset](#). The length of the new String is a function of the charset, and hence may not be equal to the length of the byte array.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement string. The [CharsetDecoder](#) class should be used when more control over the decoding process is required.

Parameters:

bytes - The bytes to be decoded into characters  
charset - The [charset](#) to be used to decode the bytes

Since:

1.6

- **String**
- public String(byte[] bytes,
- int offset,
- int length)

Constructs a new String by decoding the specified subarray of bytes using the platform's default charset. The length of the new String is a function of the charset, and hence may not be equal to the length of the subarray.

The behavior of this constructor when the given bytes are not valid in the default charset is unspecified. The [CharsetDecoder](#) class should be used when more control over the decoding process is required.

Parameters:

bytes - The bytes to be decoded into characters  
offset - The index of the first byte to decode  
length - The number of bytes to decode

Throws:

[IndexOutOfBoundsException](#) - If the offset and the length arguments index characters outside the bounds of the bytes array

Since:

JDK1.1

- **String**

public String(byte[] bytes)

Constructs a new String by decoding the specified array of bytes using the platform's default charset. The length of the new String is a function of the charset, and hence may not be equal to the length of the byte array.

The behavior of this constructor when the given bytes are not valid in the default charset is unspecified. The [CharsetDecoder](#) class should be used when more control over the decoding process is required.

Parameters:

bytes - The bytes to be decoded into characters

Since:

JDK1.1

- o **String**

`public String(StringBuffer buffer)`

Allocates a new string that contains the sequence of characters currently contained in the string buffer argument. The contents of the string buffer are copied; subsequent modification of the string buffer does not affect the newly created string.

Parameters:

buffer - A StringBuffer

- o **String**

`public String(StringBuilder builder)`

Allocates a new string that contains the sequence of characters currently contained in the string builder argument. The contents of the string builder are copied; subsequent modification of the string builder does not affect the newly created string.

This constructor is provided to ease migration to StringBuilder. Obtaining a string from a string builder via the `toString` method is likely to run faster and is generally preferred.

Parameters:

builder - A StringBuilder

Since:

1.5

- **Method Detail**

- o **length**

`public int length()`

Returns the length of this string. The length is equal to the number of [Unicode code units](#) in the string.

Specified by:

[length](#) in interface [CharSequence](#)

Returns:

the length of the sequence of characters represented by this object.

- o **isEmpty**

public boolean isEmpty()

Returns true if, and only if, [length\(\)](#) is 0.

Returns:

true if [length\(\)](#) is 0, otherwise false

Since:

1.6

- o **charAt**

public char charAt(int index)

Returns the char value at the specified index. An index ranges from 0 to [length\(\)](#) - 1. The first char value of the sequence is at index 0, the next at index 1, and so on, as for array indexing.

If the char value specified by the index is a [surrogate](#), the surrogate value is returned.

Specified by:

[charAt](#) in interface [CharSequence](#)

Parameters:

index - the index of the char value.

Returns:

the char value at the specified index of this string. The first char value is at index 0.

Throws:

[IndexOutOfBoundsException](#) - if the index argument is negative or not less than the length of this string.

- o **codePointAt**

public int codePointAt(int index)

Returns the character (Unicode code point) at the specified index. The index refers to char values (Unicode code units) and ranges from 0 to [length\(\)](#)- 1.

If the char value specified at the given index is in the high-surrogate range, the following index is less than the length of this String, and the char value at the following index is in the low-surrogate range, then the supplementary code point corresponding to this surrogate pair is returned. Otherwise, the char value at the given index is returned.

Parameters:

index - the index to the char values

Returns:

the code point value of the character at the index

Throws:

[IndexOutOfBoundsException](#) - if the index argument is negative or not less than the length of this string.

Since:

1.5

- o **codePointBefore**

```
public int codePointBefore(int index)
```

Returns the character (Unicode code point) before the specified index. The index refers to char values (Unicode code units) and ranges from 1 to [length](#).

If the char value at (index - 1) is in the low-surrogate range, (index - 2) is not negative, and the char value at (index - 2) is in the high-surrogate range, then the supplementary code point value of the surrogate pair is returned. If the char value at index - 1 is an unpaired low-surrogate or a high-surrogate, the surrogate value is returned.

Parameters:

index - the index following the code point that should be returned

Returns:

the Unicode code point value before the given index.

Throws:

[IndexOutOfBoundsException](#) - if the index argument is less than 1 or greater than the length of this string.

Since:

1.5

- o **codePointCount**
- o 

```
public int codePointCount(int beginIndex,
                           int endIndex)
```

Returns the number of Unicode code points in the specified text range of this String. The text range begins at the specified beginIndex and extends to the char at index endIndex - 1. Thus the length (in chars) of the text range is endIndex-beginIndex. Unpaired surrogates within the text range count as one code point each.

Parameters:

beginIndex - the index to the first char of the text range.

endIndex - the index after the last char of the text range.

Returns:

the number of Unicode code points in the specified text range

Throws:

[IndexOutOfBoundsException](#) - if the beginIndex is negative, or endIndex is larger than the length of this String, or beginIndex is larger than endIndex.

Since:

1.5

- **offsetByCodePoints**
- public int offsetByCodePoints(int index,  
                          int codePointOffset)

Returns the index within this String that is offset from the given index by codePointOffset code points. Unpaired surrogates within the text range given by index and codePointOffset count as one code point each.

Parameters:

index - the index to be offset

codePointOffset - the offset in code points

Returns:

the index within this String

Throws:

[IndexOutOfBoundsException](#) - if index is negative or larger than the length of this String, or if codePointOffset is positive and the substring starting with index has fewer than codePointOffset code points, or if codePointOffset is negative and the substring before index has fewer than the absolute value of codePointOffset code points.

Since:

1.5

- **getChars**
- public void getChars(int srcBegin,  
                          int srcEnd,  
                          char[] dst,  
                          int dstBegin)

Copies characters from this string into the destination character array.

The first character to be copied is at index srcBegin; the last character to be copied is at index srcEnd-1 (thus the total number of characters to be copied is srcEnd-srcBegin). The characters are copied into the subarray of dst starting at index dstBegin and ending at index:

$$\text{dstBegin} + (\text{srcEnd}-\text{srcBegin}) - 1$$

Parameters:

srcBegin - index of the first character in the string to copy.

srcEnd - index after the last character in the string to copy.

dst - the destination array.

dstBegin - the start offset in the destination array.

Throws:

[IndexOutOfBoundsException](#) - If any of the following is true:

- srcBegin is negative.
- srcBegin is greater than srcEnd
- srcEnd is greater than the length of this string

- dstBegin is negative
- dstBegin+(srcEnd-srcBegin) is larger than dst.length
- **getBytes**
- [@Deprecated](#)
- public void getBytes(int srcBegin,  
                          int srcEnd,  
                          byte[] dst,  
                          int dstBegin)

Deprecated. This method does not properly convert characters into bytes. As of JDK 1.1, the preferred way to do this is via the [getBytes\(\)](#) method, which uses the platform's default charset.

Copies characters from this string into the destination byte array. Each byte receives the 8 low-order bits of the corresponding character. The eight high-order bits of each character are not copied and do not participate in the transfer in any way.

The first character to be copied is at index srcBegin; the last character to be copied is at index srcEnd-1. The total number of characters to be copied is srcEnd-srcBegin. The characters, converted to bytes, are copied into the subarray of dst starting at index dstBegin and ending at index:

$$\text{dstBegin} + (\text{srcEnd}-\text{srcBegin}) - 1$$

Parameters:

srcBegin - Index of the first character in the string to copy

srcEnd - Index after the last character in the string to copy

dst - The destination array

dstBegin - The start offset in the destination array

Throws:

[IndexOutOfBoundsException](#) - If any of the following is true:

- srcBegin is negative
- srcBegin is greater than srcEnd
- srcEnd is greater than the length of this String
- dstBegin is negative
- dstBegin+(srcEnd-srcBegin) is larger than dst.length
- **getBytes**
- public byte[] getBytes([String](#) charsetName)  
    throws [UnsupportedEncodingException](#)

Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.

The behavior of this method when this string cannot be encoded in the given charset is unspecified. The [CharsetEncoder](#) class should be used when more control over the encoding process is required.

Parameters:

charsetName - The name of a supported [charset](#)

Returns:

The resultant byte array

Throws:

[UnsupportedEncodingException](#) - If the named charset is not supported

Since:

JDK1.1

- **getBytes**

public byte[] getBytes([Charset](#) charset)

Encodes this String into a sequence of bytes using the given [charset](#), storing the result into a new byte array.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement byte array. The [CharsetEncoder](#) class should be used when more control over the encoding process is required.

Parameters:

charset - The [Charset](#) to be used to encode the String

Returns:

The resultant byte array

Since:

1.6

- **getBytes**

public byte[] getBytes()

Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

The behavior of this method when this string cannot be encoded in the default charset is unspecified. The [CharsetEncoder](#) class should be used when more control over the encoding process is required.

Returns:

The resultant byte array

Since:

JDK1.1

- o **equals**

public boolean equals([Object](#) anObject)

C.compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

Overrides:

[equals](#) in class [Object](#)

Parameters:

anObject - The object to compare this String against

Returns:

true if the given object represents a String equivalent to this string, false otherwise

See Also:

[compareTo\(String\)](#), [equalsIgnoreCase\(String\)](#)

- o **contentEquals**

public boolean contentEquals([StringBuffer](#) sb)

C.compares this string to the specified StringBuffer. The result is true if and only if this String represents the same sequence of characters as the specified StringBuffer. This method synchronizes on the StringBuffer.

Parameters:

sb - The StringBuffer to compare this String against

Returns:

true if this String represents the same sequence of characters as the specified StringBuffer, false otherwise

Since:

1.4

- o **contentEquals**

public boolean contentEquals([CharSequence](#) cs)

C.compares this string to the specified CharSequence. The result is true if and only if this String represents the same sequence of char values as the specified sequence. Note that if the CharSequence is a StringBuffer then the method synchronizes on it.

Parameters:

cs - The sequence to compare this String against

Returns:

true if this String represents the same sequence of char values as the specified sequence, false otherwise

Since:

1.5

- o **equalsIgnoreCase**

```
public boolean equalsIgnoreCase(String anotherString)
```

Compares this String to another String, ignoring case considerations. Two strings are considered equal ignoring case if they are of the same length and corresponding characters in the two strings are equal ignoring case.

Two characters c1 and c2 are considered the same ignoring case if at least one of the following is true:

- The two characters are the same (as compared by the == operator)
- Applying the method [Character.toUpperCase\(char\)](#) to each character produces the same result
- Applying the method [Character.toLowerCase\(char\)](#) to each character produces the same result

Parameters:

anotherString - The String to compare this String against

Returns:

true if the argument is not null and it represents an equivalent String ignoring case; false otherwise

See Also:

[equals\(Object\)](#)

- o **compareTo**

```
public int compareTo(String anotherString)
```

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this String object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this String object lexicographically precedes the argument string. The result is a positive integer if this String object lexicographically follows the argument string. The result is zero if the strings are equal; compareTo returns 0 exactly when the [equals\(Object\)](#) method would return true.

This is the definition of lexicographic ordering. If two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let k be the smallest such index; then the string whose character at position k has the smaller value, as determined by using the < operator, lexicographically precedes the other string. In this case, compareTo returns the difference of the two character values at position k in the two string -- that is, the value:

this.charAt(k)-anotherString.charAt(k)

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, `compareTo` returns the difference of the lengths of the strings -- that is, the value:

`this.length()-anotherString.length()`

Specified by:

[compareTo](#) in interface [Comparable<String>](#)

Parameters:

`anotherString` - the String to be compared.

Returns:

the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.

- o **compareToIgnoreCase**

`public int compareToIgnoreCase(String str)`

Compares two strings lexicographically, ignoring case differences. This method returns an integer whose sign is that of calling `compareTo` with normalized versions of the strings where case differences have been eliminated by calling `Character.toLowerCase(Character.toUpperCase(character))` on each character.

Note that this method does *not* take locale into account, and will result in an unsatisfactory ordering for certain locales. The `java.text` package provides *collators* to allow locale-sensitive ordering.

Parameters:

`str` - the String to be compared.

Returns:

a negative integer, zero, or a positive integer as the specified String is greater than, equal to, or less than this String, ignoring case considerations.

Since:

1.2

See Also:

[Collator.compare\(String, String\)](#)

- o **regionMatches**
- o `public boolean regionMatches(int toffset,`  
o       [String](#) `other,`  
o       int `ooffset,`  
o       int `len)`

Tests if two string regions are equal.

A substring of this String object is compared to a substring of the argument other. The result is true if these substrings represent identical character sequences. The substring of this String object to be compared begins at index toffset and has length len. The substring of other to be compared begins at index ooffset and has length len. The result is false if and only if at least one of the following is true:

- toffset is negative.
- ooffset is negative.
- toffset+len is greater than the length of this String object.
- ooffset+len is greater than the length of the other argument.
- There is some nonnegative integer  $k$  less than len such that: this.charAt(toffset + k) != other.charAt(ooffset + k)

Parameters:

toffset - the starting offset of the subregion in this string.

other - the string argument.

ooffset - the starting offset of the subregion in the string argument.

len - the number of characters to compare.

Returns:

true if the specified subregion of this string exactly matches the specified subregion of the string argument; false otherwise.

- **regionMatches**
- public boolean regionMatches(boolean ignoreCase,  
○                   int toffset,  
○                   String other,  
○                   int ooffset,  
○                   int len)

Tests if two string regions are equal.

A substring of this String object is compared to a substring of the argument other. The result is true if these substrings represent character sequences that are the same, ignoring case if and only if ignoreCase is true. The substring of this String object to be compared begins at index toffset and has length len. The substring of other to be compared begins at index ooffset and has length len. The result is false if and only if at least one of the following is true:

- toffset is negative.
- ooffset is negative.
- toffset+len is greater than the length of this String object.
- ooffset+len is greater than the length of the other argument.
- ignoreCase is false and there is some nonnegative integer  $k$  less than len such that:  
▪ this.charAt(toffset+k) != other.charAt(ooffset+k)

- ignoreCase is true and there is some nonnegative integer  $k$  less than len such that:
- Character.toLowerCase(this.charAt(toffset+k)) !=
- Character.toLowerCase(other.charAt(ooffset+k))

and:

```
Character.toUpperCase(this.charAt(toffset+k)) !=  
    Character.toUpperCase(other.charAt(ooffset+k))
```

Parameters:

ignoreCase - if true, ignore case when comparing characters.

toffset - the starting offset of the subregion in this string.

other - the string argument.

ooffset - the starting offset of the subregion in the string argument.

len - the number of characters to compare.

Returns:

true if the specified subregion of this string matches the specified subregion of the string argument; false otherwise. Whether the matching is exact or case insensitive depends on the ignoreCase argument.

- **startsWith**

```
public boolean startsWith(String prefix,  
                        int toffset)
```

Tests if the substring of this string beginning at the specified index starts with the specified prefix.

Parameters:

prefix - the prefix.

toffset - where to begin looking in this string.

Returns:

true if the character sequence represented by the argument is a prefix of the substring of this object starting at index toffset; false otherwise. The result is false if toffset is negative or greater than the length of this String object; otherwise the result is the same as the result of the expression

```
    this.substring(toffset).startsWith(prefix)
```

- **startsWith**

```
public boolean startsWith(String prefix)
```

Tests if this string starts with the specified prefix.

Parameters:

prefix - the prefix.

Returns:

true if the character sequence represented by the argument is a prefix of the character sequence represented by this string; false otherwise. Note also that true will be returned if the argument is an empty string or is equal to this String object as determined by the [equals\(Object\)](#) method.

Since:

1.0

- **endsWith**

`public boolean endsWith(String suffix)`

Tests if this string ends with the specified suffix.

Parameters:

suffix - the suffix.

Returns:

true if the character sequence represented by the argument is a suffix of the character sequence represented by this object; false otherwise. Note that the result will be true if the argument is the empty string or is equal to this String object as determined by the [equals\(Object\)](#) method.

- **hashCode**

`public int hashCode()`

Returns a hash code for this string. The hash code for a String object is computed as

$$s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \dots + s[n-1]$$

using int arithmetic, where  $s[i]$  is the  $i$ th character of the string,  $n$  is the length of the string, and  $^$  indicates exponentiation. (The hash value of the empty string is zero.)

Overrides:

[hashCode](#) in class [Object](#)

Returns:

a hash code value for this object.

See Also:

[Object.equals\(java.lang.Object\)](#), [System.identityHashCode\(java.lang.Object\)](#)

- **indexOf**

`public int indexOf(int ch)`

Returns the index within this string of the first occurrence of the specified character. If a character with value ch occurs in the character sequence represented by this String object, then the index (in Unicode code units) of the first such occurrence is returned. For values of ch in the range from 0 to 0xFFFF (inclusive), this is the smallest value k such that:

`this.charAt(k) == ch`

is true. For other values of ch, it is the smallest value k such that:

`this.codePointAt(k) == ch`

is true. In either case, if no such character occurs in this string, then -1 is returned.

Parameters:

ch - a character (Unicode code point).

Returns:

the index of the first occurrence of the character in the character sequence represented by this object, or -1 if the character does not occur.

- **indexOf**
- `public int indexOf(int ch,  
int fromIndex)`

Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

If a character with value ch occurs in the character sequence represented by this String object at an index no smaller than fromIndex, then the index of the first such occurrence is returned. For values of ch in the range from 0 to 0xFFFF (inclusive), this is the smallest value k such that:

`(this.charAt(k) == ch) && (k >= fromIndex)`

is true. For other values of ch, it is the smallest value k such that:

`(this.codePointAt(k) == ch) && (k >= fromIndex)`

is true. In either case, if no such character occurs in this string at or after position fromIndex, then -1 is returned.

There is no restriction on the value of fromIndex. If it is negative, it has the same effect as if it were zero: this entire string may be searched. If it is greater than the length of this string, it has the same effect as if it were equal to the length of this string: -1 is returned.

All indices are specified in char values (Unicode code units).

Parameters:

ch - a character (Unicode code point).

fromIndex - the index to start the search from.

Returns:

the index of the first occurrence of the character in the character sequence represented by this object that is greater than or equal to fromIndex, or -1 if the character does not occur.

- **lastIndexOf**

public int lastIndexOf(int ch)

Returns the index within this string of the last occurrence of the specified character. For values of ch in the range from 0 to 0xFFFF (inclusive), the index (in Unicode code units) returned is the largest value  $k$  such that:

this.charAt( $k$ ) == ch

is true. For other values of ch, it is the largest value  $k$  such that:

this.codePointAt( $k$ ) == ch

is true. In either case, if no such character occurs in this string, then -1 is returned. The String is searched backwards starting at the last character.

Parameters:

ch - a character (Unicode code point).

Returns:

the index of the last occurrence of the character in the character sequence represented by this object, or -1 if the character does not occur.

- **lastIndexOf**

○ public int lastIndexOf(int ch,  
int fromIndex)

Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index. For values of ch in the range from 0 to 0xFFFF (inclusive), the index returned is the largest value  $k$  such that:

(this.charAt(*k*) == ch) && (*k* <= fromIndex)

is true. For other values of ch, it is the largest value *k* such that:

(this.codePointAt(*k*) == ch) && (*k* <= fromIndex)

is true. In either case, if no such character occurs in this string at or before position fromIndex, then -1 is returned.

All indices are specified in char values (Unicode code units).

Parameters:

ch - a character (Unicode code point).

fromIndex - the index to start the search from. There is no restriction on the value of fromIndex.

If it is greater than or equal to the length of this string, it has the same effect as if it were equal to one less than the length of this string: this entire string may be searched. If it is negative, it has the same effect as if it were -1: -1 is returned.

Returns:

the index of the last occurrence of the character in the character sequence represented by this object that is less than or equal to fromIndex, or -1 if the character does not occur before that point.

- o **indexOf**

public int indexOf([String](#) str)

Returns the index within this string of the first occurrence of the specified substring.

The returned index is the smallest value *k* for which:

this.startsWith(str, *k*)

If no such value of *k* exists, then -1 is returned.

Parameters:

str - the substring to search for.

Returns:

the index of the first occurrence of the specified substring, or -1 if there is no such occurrence.

- o **indexOf**
- o public int indexOf([String](#) str,  
                  int fromIndex)

Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

The returned index is the smallest value  $k$  for which:

$k \geq \text{fromIndex} \ \&\& \ \text{this.startsWith(str, } k\text{)}$

If no such value of  $k$  exists, then -1 is returned.

Parameters:

str - the substring to search for.

fromIndex - the index from which to start the search.

Returns:

the index of the first occurrence of the specified substring, starting at the specified index, or -1 if there is no such occurrence.

- o **lastIndexOf**

public int lastIndexOf([String](#) str)

Returns the index within this string of the last occurrence of the specified substring. The last occurrence of the empty string "" is considered to occur at the index value this.length().

The returned index is the largest value  $k$  for which:

$\text{this.startsWith(str, } k\text{)}$

If no such value of  $k$  exists, then -1 is returned.

Parameters:

str - the substring to search for.

Returns:

the index of the last occurrence of the specified substring, or -1 if there is no such occurrence.

- o **lastIndexOf**

o public int lastIndexOf([String](#) str,  
int fromIndex)

Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

The returned index is the largest value  $k$  for which:

$k \leq \text{fromIndex} \ \&\& \ \text{this.startsWith(str, } k\text{)}$

If no such value of  $k$  exists, then -1 is returned.

Parameters:

str - the substring to search for.

fromIndex - the index to start the search from.

Returns:

the index of the last occurrence of the specified substring, searching backward from the specified index, or -1 if there is no such occurrence.

- o **substring**

public [String](#) substring(int beginIndex)

Returns a string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

Examples:

"unhappy".substring(2) returns "happy"

"Harbison".substring(3) returns "bison"

"emptiness".substring(9) returns "" (an empty string)

Parameters:

beginIndex - the beginning index, inclusive.

Returns:

the specified substring.

Throws:

[IndexOutOfBoundsException](#) - if beginIndex is negative or larger than the length of this String object.

- o **substring**

o public [String](#) substring(int beginIndex,  
int endIndex)

Returns a string that is a substring of this string. The substring begins at the specified beginIndex and extends to the character at index endIndex - 1. Thus the length of the substring is endIndex-beginIndex.

Examples:

"hamburger".substring(4, 8) returns "urge"

"smiles".substring(1, 5) returns "mile"

Parameters:

---

beginIndex - the beginning index, inclusive.

endIndex - the ending index, exclusive.

Returns:

the specified substring.

Throws:

[IndexOutOfBoundsException](#) - if the beginIndex is negative, or endIndex is larger than the length of this String object, or beginIndex is larger than endIndex.

- o **subSequence**
- o public [CharSequence](#) subSequence(int beginIndex,  
int endIndex)

Returns a character sequence that is a subsequence of this sequence.

An invocation of this method of the form

str.subSequence(begin, end)

behaves in exactly the same way as the invocation

str.substring(begin, end)

Specified by:

[subSequence](#) in interface [CharSequence](#)

API Note:

This method is defined so that the String class can implement the [CharSequence](#) interface.

Parameters:

beginIndex - the begin index, inclusive.

endIndex - the end index, exclusive.

Returns:

the specified subsequence.

Throws:

[IndexOutOfBoundsException](#) - if beginIndex or endIndex is negative, if endIndex is greater than length(), or if beginIndex is greater than endIndex

Since:

1.4

- o **concat**

public [String](#) concat([String](#) str)

Concatenates the specified string to the end of this string.

If the length of the argument string is 0, then this String object is returned. Otherwise, a String object is returned that represents a character sequence that is the concatenation of the character sequence represented by this String object and the character sequence represented by the argument string.

Examples:

"cares".concat("s") returns "caress"  
"to".concat("get").concat("her") returns "together"

Parameters:

str - the String that is concatenated to the end of this String.

Returns:

a string that represents the concatenation of this object's characters followed by the string argument's characters.

- o **replace**
- o public [String](#) replace(char oldChar,  
                  char newChar)

Returns a string resulting from replacing all occurrences of oldChar in this string with newChar.

If the character oldChar does not occur in the character sequence represented by this String object, then a reference to this String object is returned. Otherwise, a String object is returned that represents a character sequence identical to the character sequence represented by this String object, except that every occurrence of oldChar is replaced by an occurrence of newChar.

Examples:

"mesquite in your cellar".replace('e', 'o')  
                  returns "mosquito in your collar"  
"the war of baronets".replace('r', 'y')  
                  returns "the way of bayonets"  
"sparring with a purple porpoise".replace('p', 't')  
                  returns "starring with a turtle tortoise"  
"JonL".replace('q', 'x') returns "JonL" (no change)

Parameters:

oldChar - the old character.

newChar - the new character.

Returns:

a string derived from this string by replacing every occurrence of oldChar with newChar.

- o **matches**

public boolean matches([String](#) regex)

Tells whether or not this string matches the given [regular expression](#).

An invocation of this method of the form `str.matches(regex)` yields exactly the same result as the expression

[Pattern.matches\(regex, str\)](#)

Parameters:

regex - the regular expression to which this string is to be matched

Returns:

true if, and only if, this string matches the given regular expression

Throws:

[PatternSyntaxException](#) - if the regular expression's syntax is invalid

Since:

1.4

See Also:

[Pattern](#)

- o **contains**

`public boolean contains(CharSequence s)`

Returns true if and only if this string contains the specified sequence of char values.

Parameters:

s - the sequence to search for

Returns:

true if this string contains s, false otherwise

Since:

1.5

- o **replaceFirst**
- o `public String replaceFirst(String regex, String replacement)`

Replaces the first substring of this string that matches the given [regular expression](#) with the given replacement.

An invocation of this method of the form `str.replaceFirst(regex, repl)` yields exactly the same result as the expression

[Pattern.compile\(regex\).matcher\(str\).replaceFirst\(repl\)](#)

Note that backslashes (\) and dollar signs (\$) in the replacement string may cause the results to be different than if it were being treated as a literal replacement string; see [Matcher.replaceFirst\(java.lang.String\)](#). Use [Matcher.quoteReplacement\(java.lang.String\)](#) to suppress the special meaning of these characters, if desired.

---

Parameters:

regex - the regular expression to which this string is to be matched

replacement - the string to be substituted for the first match

Returns:

The resulting String

Throws:

[PatternSyntaxException](#) - if the regular expression's syntax is invalid

Since:

1.4

See Also:

[Pattern](#)

- **replaceAll**

- public [String](#) replaceAll([String](#) regex,  
[String](#) replacement)

Replaces each substring of this string that matches the given [regular expression](#) with the given replacement.

An invocation of this method of the form *str.replaceAll(regex, repl)* yields exactly the same result as the expression

[Pattern.compile\(regex\).matcher\(str\).replaceAll\(repl\)](#)

Note that backslashes (\) and dollar signs (\$) in the replacement string may cause the results to be different than if it were being treated as a literal replacement string; see [Matcher.replaceAll](#). Use [Matcher.quoteReplacement\(java.lang.String\)](#) to suppress the special meaning of these characters, if desired.

Parameters:

regex - the regular expression to which this string is to be matched

replacement - the string to be substituted for each match

Returns:

The resulting String

Throws:

[PatternSyntaxException](#) - if the regular expression's syntax is invalid

Since:

1.4

See Also:

[Pattern](#)

- **replace**

- public [String](#) replace([CharSequence](#) target,  
[CharSequence](#) replacement)

Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence. The replacement proceeds from the beginning of the string to the end, for example, replacing "aa" with "b" in the string "aaa" will result in "ba" rather than "ab".

Parameters:

target - The sequence of char values to be replaced  
replacement - The replacement sequence of char values

Returns:

The resulting string

Since:

1.5

- o **split**
- o public [String\[\] split\(String regex, int limit\)](#)

Splits this string around matches of the given [regular expression](#).

The array returned by this method contains each substring of this string that is terminated by another substring that matches the given expression or is terminated by the end of the string. The substrings in the array are in the order in which they occur in this string. If the expression does not match any part of the input then the resulting array has just one element, namely this string.

When there is a positive-width match at the beginning of this string then an empty leading substring is included at the beginning of the resulting array. A zero-width match at the beginning however never produces such empty leading substring.

The limit parameter controls the number of times the pattern is applied and therefore affects the length of the resulting array. If the limit  $n$  is greater than zero then the pattern will be applied at most  $n - 1$  times, the array's length will be no greater than  $n$ , and the array's last entry will contain all input beyond the last matched delimiter. If  $n$  is non-positive then the pattern will be applied as many times as possible and the array can have any length. If  $n$  is zero then the pattern will be applied as many times as possible, the array can have any length, and trailing empty strings will be discarded.

The string "boo:and:foo", for example, yields the following results with these parameters:

#### **Regex Limit Result**

|   |    |                               |
|---|----|-------------------------------|
| : | 2  | { "boo", "and:foo" }          |
| : | 5  | { "boo", "and", "foo" }       |
| : | -2 | { "boo", "and", "foo" }       |
| o | 5  | { "b", "", ":and:f", "", "" } |
| o | -2 | { "b", "", ":and:f", "", "" } |
| o | 0  | { "b", "", ":and:f" }         |

An invocation of this method of the form `str.split(regex, n)` yields the same result as the expression

[Pattern.compile\(regex\).split\(str, n\)](#)

Parameters:

regex - the delimiting regular expression

limit - the result threshold, as described above

Returns:

the array of strings computed by splitting this string around matches of the given regular expression

Throws:

[PatternSyntaxException](#) - if the regular expression's syntax is invalid

Since:

1.4

See Also:

[Pattern](#)

- o **split**

public [String\[\] split\(String regex\)](#)

Splits this string around matches of the given [regular expression](#).

This method works as if by invoking the two-argument [split](#) method with the given expression and a limit argument of zero. Trailing empty strings are therefore not included in the resulting array.

The string "boo:and:foo", for example, yields the following results with these expressions:

**Regex Result**

: { "boo", "and", "foo" }  
o { "b", "", ":and:f" }

Parameters:

regex - the delimiting regular expression

Returns:

the array of strings computed by splitting this string around matches of the given regular expression

Throws:

[PatternSyntaxException](#) - if the regular expression's syntax is invalid

Since:

1.4

See Also:

[Pattern](#)

- o **join**

- public static [String](#) join([CharSequence](#) delimiter,  
[CharSequence](#)... elements)

Returns a new String composed of copies of the CharSequence elements joined together with a copy of the specified delimiter.

For example,

```
String message = String.join("-", "Java", "is", "cool");
// message returned is: "Java-is-cool"
```

Note that if an element is null, then "null" is added.

Parameters:

delimiter - the delimiter that separates each element

elements - the elements to join together.

Returns:

a new String that is composed of the elements separated by the delimiter

Throws:

[NullPointerException](#) - If delimiter or elements is null

Since:

1.8

See Also:

[StringJoiner](#)

- **join**
- public static [String](#) join([CharSequence](#) delimiter,  
[Iterable](#)<? extends [CharSequence](#)> elements)

Returns a new String composed of copies of the CharSequence elements joined together with a copy of the specified delimiter.

For example,

```
List<String> strings = new LinkedList<>();
strings.add("Java");strings.add("is");
strings.add("cool");
String message = String.join(" ", strings);
//message returned is: "Java is cool"
```

```
Set<String> strings = new LinkedHashSet<>();
strings.add("Java"); strings.add("is");
strings.add("very"); strings.add("cool");
```

```
String message = String.join("-", strings);
//message returned is: "Java-is-very-cool"
```

Note that if an individual element is null, then "null" is added.

Parameters:

delimiter - a sequence of characters that is used to separate each of the elements in the resulting String

elements - an Iterable that will have its elements joined together.

Returns:

a new String that is composed from the elements argument

Throws:

[NullPointerException](#) - If delimiter or elements is null

Since:

1.8

See Also:

[join\(CharSequence,CharSequence...\)](#), [StringJoiner](#)

- o **toLowerCase**

public [String](#) toLowerCase([Locale](#) locale)

Converts all of the characters in this String to lower case using the rules of the given Locale.

Case mapping is based on the Unicode Standard version specified by the [Character](#) class.

Since case mappings are not always 1:1 char mappings, the resulting String may be a different length than the original String.

Examples of lowercase mappings are in the following table:

| Language Code of Locale | Upper Case   | Lower Case   | Description                                       |
|-------------------------|--------------|--------------|---------------------------------------------------|
| tr (Turkish)            | \u0130       | \u0069       | capital letter I with dot above -> small letter i |
| tr (Turkish)            | \u0049       | \u0131       | capital letter I -> small letter dotless i        |
| (all)                   | French Fries | french fries | lowercased all chars in String                    |
| (all)                   | IXØYΣ        | 1XØYΣ        | lowercased all chars in String                    |

Parameters:

locale - use the case transformation rules for this locale

Returns:

the String, converted to lowercase.

Since:

1.1

See Also:

[toLowerCase\(\)](#), [toUpperCase\(\)](#), [toUpperCase\(Locale\)](#)

- o **toLowerCase**

public [String](#) toLowerCase()

Converts all of the characters in this String to lower case using the rules of the default locale.  
This is equivalent to calling toLowerCase(Locale.getDefault()).

**Note:** This method is locale sensitive, and may produce unexpected results if used for strings that are intended to be interpreted locale independently. Examples are programming language identifiers, protocol keys, and HTML tags. For instance, "TITLE".toLowerCase() in a Turkish locale returns "t\u0131tle", where "\u0131" is the LATIN SMALL LETTER DOTLESS I character. To obtain correct results for locale insensitive strings, use toLowerCase(Locale.ROOT).

Returns:

the String, converted to lowercase.

See Also:

[toLowerCase\(Locale\)](#)

- o **toUpperCase**

public [String](#) toUpperCase([Locale](#) locale)

Converts all of the characters in this String to upper case using the rules of the given Locale.  
Case mapping is based on the Unicode Standard version specified by the [Character](#) class.  
Since case mappings are not always 1:1 char mappings, the resulting String may be a different length than the original String.

Examples of locale-sensitive and 1:M case mappings are in the following table.

| Language Code of Locale | Lower Case    | Upper Case    | Description                                       |
|-------------------------|---------------|---------------|---------------------------------------------------|
| tr (Turkish)            | \u0069        | \u0130        | small letter i -> capital letter I with dot above |
| tr (Turkish)            | \u0131        | \u0049        | small letter dotless i -> capital letter I        |
| (all)                   | \u00df        | \u0053 \u0053 | small letter sharp s -> two letters: SS           |
| (all)                   | Fahrvergnügen | FAHRVERGNÜGEN |                                                   |

Parameters:

locale - use the case transformation rules for this locale

Returns:

the String, converted to uppercase.

Since:

1.1

See Also:

[toUpperCase\(\)](#), [toLowerCase\(\)](#), [toLowerCase\(Locale\)](#)

- **toUpperCase**

public [String](#) toUpperCase()

Converts all of the characters in this String to upper case using the rules of the default locale.

This method is equivalent to toUpperCase(Locale.getDefault()).

**Note:** This method is locale sensitive, and may produce unexpected results if used for strings that are intended to be interpreted locale independently. Examples are programming language identifiers, protocol keys, and HTML tags. For instance, "title".toUpperCase() in a Turkish locale returns "T\u0130TITLE", where '\u0130' is the LATIN CAPITAL LETTER I WITH DOT ABOVE character. To obtain correct results for locale insensitive strings, use toUpperCase(Locale.ROOT).

Returns:

the String, converted to uppercase.

See Also:

[toUpperCase\(Locale\)](#)

- **trim**

public [String](#) trim()

Returns a string whose value is this string, with any leading and trailing whitespace removed.

If this String object represents an empty character sequence, or the first and last characters of character sequence represented by this String object both have codes greater than '\u0020' (the space character), then a reference to this String object is returned.

Otherwise, if there is no character with a code greater than '\u0020' in the string, then a String object representing an empty string is returned.

Otherwise, let  $k$  be the index of the first character in the string whose code is greater than '\u0020', and let  $m$  be the index of the last character in the string whose code is greater than '\u0020'. A String object is returned, representing the substring of this string that begins with the character at index  $k$  and ends with the character at index  $m$ -that is, the result of this.substring( $k$ ,  $m + 1$ ).

This method may be used to trim whitespace (as defined above) from the beginning and end of a string.

Returns:

A string whose value is this string, with any leading and trailing white space removed, or this string if it has no leading or trailing white space.

- **toString**

public [String](#) **toString()**

This object (which is already a string!) is itself returned.

Specified by:

[toString](#) in interface [CharSequence](#)

Overrides:

[toString](#) in class [Object](#)

Returns:

the string itself.

- **toCharArray**

public char[] **toCharArray()**

Converts this string to a new character array.

Returns:

a newly allocated character array whose length is the length of this string and whose contents are initialized to contain the character sequence represented by this string.

- **format**
- public static [String](#) **format([String](#) format,  
[Object](#)... args)**

Returns a formatted string using the specified format string and arguments.

The locale always used is the one returned by [Locale.getDefault\(\)](#).

Parameters:

format - A [format string](#)

args - Arguments referenced by the format specifiers in the format string. If there are more arguments than format specifiers, the extra arguments are ignored. The number of arguments is variable and may be zero. The maximum number of arguments is limited by the maximum dimension of a Java array as defined by *The Java™ Virtual Machine Specification*. The behaviour on a null argument depends on the [conversion](#).

Returns:

A formatted string

Throws:

[IllegalFormatException](#) - If a format string contains an illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions. For specification of all possible formatting errors, see the [Details](#) section of the formatter class specification.

Since:

1.5

See Also:

[Formatter](#)

- o **format**
- o public static [String](#) format([Locale](#) l,  
                  [String](#) format,  
                  [Object](#)... args)

Returns a formatted string using the specified locale, format string, and arguments.

Parameters:

l - The [locale](#) to apply during formatting. If l is null then no localization is applied.

format - A [format string](#)

args - Arguments referenced by the format specifiers in the format string. If there are more arguments than format specifiers, the extra arguments are ignored. The number of arguments is variable and may be zero. The maximum number of arguments is limited by the maximum dimension of a Java array as defined by *The Java™ Virtual Machine Specification*. The behaviour on a null argument depends on the [conversion](#).

Returns:

A formatted string

Throws:

[IllegalFormatException](#) - If a format string contains an illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions. For specification of all possible formatting errors, see the [Details](#) section of the formatter class specification

Since:

1.5

See Also:

[Formatter](#)

- o **valueOf**

public static [String](#) valueOf([Object](#) obj)

Returns the string representation of the Object argument.

Parameters:

obj - an Object.

Returns:

if the argument is null, then a string equal to "null"; otherwise, the value of obj.toString() is returned.

See Also:

[Object.toString\(\)](#)

- o **valueOf**

public static [String](#) valueOf(char[] data)

Returns the string representation of the char array argument. The contents of the character array are copied; subsequent modification of the character array does not affect the returned string.

Parameters:

data - the character array.

Returns:

a String that contains the characters of the character array.

- o **valueOf**
- o public static [String](#) valueOf(char[] data,
  - o int offset,
  - int count)

Returns the string representation of a specific subarray of the char array argument.

The offset argument is the index of the first character of the subarray. The count argument specifies the length of the subarray. The contents of the subarray are copied; subsequent modification of the character array does not affect the returned string.

Parameters:

data - the character array.

offset - initial offset of the subarray.

count - length of the subarray.

Returns:

a String that contains the characters of the specified subarray of the character array.

Throws:

[IndexOutOfBoundsException](#) - if offset is negative, or count is negative, or offset+count is larger than data.length.

- o **copyValueOf**
- o public static [String](#) copyValueOf(char[] data,
  - o int offset,
  - int count)

Equivalent to [valueOf\(char\[\], int, int\)](#).

Parameters:

data - the character array.

offset - initial offset of the subarray.

count - length of the subarray.

Returns:

a String that contains the characters of the specified subarray of the character array.

Throws:

[IndexOutOfBoundsException](#) - if offset is negative, or count is negative, or offset+count is larger than data.length.

- **copyValueOf**

public static [String](#) copyValueOf(char[] data)

Equivalent to [valueOf\(char\[\]\)](#).

Parameters:

data - the character array.

Returns:

a String that contains the characters of the character array.

- **valueOf**

public static [String](#) valueOf(boolean b)

Returns the string representation of the boolean argument.

Parameters:

b - a boolean.

Returns:

if the argument is true, a string equal to "true" is returned; otherwise, a string equal to "false" is returned.

- **valueOf**

public static [String](#) valueOf(char c)

Returns the string representation of the char argument.

Parameters:

c - a char.

Returns:

a string of length 1 containing as its single character the argument c.

- **valueOf**

public static [String](#) valueOf(int i)

Returns the string representation of the int argument.

The representation is exactly the one returned by the Integer.toString method of one argument.

Parameters:

i - an int.

Returns:

a string representation of the int argument.

See Also:

[Integer.toString\(int, int\)](#)

- **valueOf**

public static [String](#) valueOf(long l)

Returns the string representation of the long argument.

The representation is exactly the one returned by the Long.toString method of one argument.

Parameters:

l - a long.

Returns:

a string representation of the long argument.

See Also:

[Long.toString\(long\)](#)

- **valueOf**

public static [String](#) valueOf(float f)

Returns the string representation of the float argument.

The representation is exactly the one returned by the Float.toString method of one argument.

Parameters:

f - a float.

Returns:

a string representation of the float argument.

See Also:

[Float.toString\(float\)](#)

- **valueOf**

public static [String](#) valueOf(double d)

Returns the string representation of the double argument.

The representation is exactly the one returned by the Double.toString method of one argument.

Parameters:

d - a double.

Returns:

a string representation of the double argument.

See Also:

[Double.toString\(double\)](#)

- o **intern**

public [String](#) intern()

Returns a canonical representation for the string object.

A pool of strings, initially empty, is maintained privately by the class String.

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the [equals\(Object\)](#) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

It follows that for any two strings s and t, s.intern() == t.intern() is true if and only if s.equals(t) is true.

All literal strings and string-valued constant expressions are interned. String literals are defined in section 3.10.5 of the *The Java™ Language Specification*.

Returns:

a string that has the same contents as this string, but is guaranteed to be from a pool of unique strings.

## Class StringBuffer

- [java.lang.Object](#)
  - 
  - o [java.lang.StringBuffer](#)
  
  - All Implemented Interfaces:  
[Serializable](#), [Appendable](#), [CharSequence](#)
-

```
public final class StringBuffer  
extends Object  
implements Serializable, CharSequence
```

A thread-safe, mutable sequence of characters. A string buffer is like a [String](#), but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

String buffers are safe for use by multiple threads. The methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

The principal operations on a `StringBuffer` are the `append` and `insert` methods, which are overloaded so as to accept data of any type. Each effectively converts a given datum to a string and then appends or inserts the characters of that string to the string buffer. The `append` method always adds these characters at the end of the buffer; the `insert` method adds the characters at a specified point.

For example, if `z` refers to a string buffer object whose current contents are "start", then the method call `z.append("le")` would cause the string buffer to contain "startle", whereas `z.insert(4, "le")` would alter the string buffer to contain "starlet".

In general, if `sb` refers to an instance of a `StringBuffer`, then `sb.append(x)` has the same effect as `sb.insert(sb.length(), x)`.

Whenever an operation occurs involving a source sequence (such as appending or inserting from a source sequence), this class synchronizes only on the string buffer performing the operation, not on the source. Note that while `StringBuffer` is designed to be safe to use concurrently from multiple threads, if the constructor or the `append` or `insert` operation is passed a source sequence that is shared across threads, the calling code must ensure that the operation has a consistent and unchanging view of the source sequence for the duration of the operation. This could be satisfied by the caller holding a lock during the operation's call, by using an immutable source sequence, or by not sharing the source sequence across threads.

Every string buffer has a capacity. As long as the length of the character sequence contained in the string buffer does not exceed the capacity, it is not necessary to allocate a new internal buffer array. If the internal buffer overflows, it is automatically made larger.

Unless otherwise noted, passing a null argument to a constructor or method in this class will cause a [NullPointerException](#) to be thrown.

As of release JDK 5, this class has been supplemented with an equivalent class designed for use by a single thread, [StringBuilder](#). The `StringBuilder` class should generally be used in

preference to this one, as it supports all of the same operations but it is faster, as it performs no synchronization.

### Constructor and Description

#### [StringBuffer\(\)](#)

Constructs a string buffer with no characters in it and an initial capacity of 16 characters.

#### [StringBuffer\(CharSequence seq\)](#)

Constructs a string buffer that contains the same characters as the specified CharSequence.

#### [StringBuffer\(int capacity\)](#)

Constructs a string buffer with no characters in it and the specified initial capacity.

#### [StringBuffer\(String str\)](#)

Constructs a string buffer initialized to the contents of the specified string.

•

- **StringBuffer**

#### [public StringBuffer\(\)](#)

Constructs a string buffer with no characters in it and an initial capacity of 16 characters.

- **StringBuffer**

#### [public StringBuffer\(int capacity\)](#)

Constructs a string buffer with no characters in it and the specified initial capacity.

Parameters:

capacity - the initial capacity.

Throws:

[NegativeArraySizeException](#) - if the capacity argument is less than 0.

- **StringBuffer**

#### [public StringBuffer\(String str\)](#)

Constructs a string buffer initialized to the contents of the specified string. The initial capacity of the string buffer is 16 plus the length of the string argument.

Parameters:

str - the initial contents of the buffer.

- o **StringBuffer**

```
public StringBuffer(CharSequence seq)
```

Constructs a string buffer that contains the same characters as the specified CharSequence. The initial capacity of the string buffer is 16 plus the length of the CharSequence argument.

If the length of the specified CharSequence is less than or equal to zero, then an empty buffer of capacity 16 is returned.

Parameters:

seq - the sequence to copy.

Since:

1.5

- **Method Detail**

- o **length**

```
public int length()
```

Returns the length (character count).

Specified by:

[length](#) in interface [CharSequence](#)

Returns:

the length of the sequence of characters currently represented by this object

- o **capacity**

```
public int capacity()
```

Returns the current capacity. The capacity is the amount of storage available for newly inserted characters, beyond which an allocation will occur.

Returns:

the current capacity

- o **ensureCapacity**

```
public void ensureCapacity(int minimumCapacity)
```

Ensures that the capacity is at least equal to the specified minimum. If the current capacity is less than the argument, then a new internal array is allocated with greater capacity. The new capacity is the larger of:

- The minimumCapacity argument.
- Twice the old capacity, plus 2.

If the minimumCapacity argument is nonpositive, this method takes no action and simply returns. Note that subsequent operations on this object can reduce the actual capacity below that requested here.

Parameters:

minimumCapacity - the minimum desired capacity.

- **trimToSize**

`public void trimToSize()`

Attempts to reduce storage used for the character sequence. If the buffer is larger than necessary to hold its current sequence of characters, then it may be resized to become more space efficient. Calling this method may, but is not required to, affect the value returned by a subsequent call to the [capacity\(\)](#) method.

Since:

1.5

- **setLength**

`public void setLength(int newLength)`

Sets the length of the character sequence. The sequence is changed to a new character sequence whose length is specified by the argument. For every nonnegative index  $k$  less than newLength, the character at index  $k$  in the new character sequence is the same as the character at index  $k$  in the old sequence if  $k$  is less than the length of the old character sequence; otherwise, it is the null character '\u0000'. In other words, if the newLength argument is less than the current length, the length is changed to the specified length.

If the newLength argument is greater than or equal to the current length, sufficient null characters ('\u0000') are appended so that length becomes the newLength argument.

The newLength argument must be greater than or equal to 0.

Parameters:

newLength - the new length

Throws:

[IndexOutOfBoundsException](#) - if the newLength argument is negative.

See Also:

[length\(\)](#)

- o **charAt**

`public char charAt(int index)`

Returns the char value in this sequence at the specified index. The first char value is at index 0, the next at index 1, and so on, as in array indexing.

The index argument must be greater than or equal to 0, and less than the length of this sequence.

If the char value specified by the index is a [surrogate](#), the surrogate value is returned.

Specified by:

[charAt](#) in interface [CharSequence](#)

Parameters:

index - the index of the desired char value.

Returns:

the char value at the specified index.

Throws:

[IndexOutOfBoundsException](#) - if index is negative or greater than or equal to `length()`.

See Also:

[length\(\)](#)

- o **codePointAt**

`public int codePointAt(int index)`

Returns the character (Unicode code point) at the specified index. The index refers to char values (Unicode code units) and ranges from 0 to [length\(\)](#)- 1.

If the char value specified at the given index is in the high-surrogate range, the following index is less than the length of this sequence, and the char value at the following index is in the low-surrogate range, then the supplementary code point corresponding to this surrogate pair is returned. Otherwise, the char value at the given index is returned.

Parameters:

index - the index to the char values

Returns:

the code point value of the character at the index

Since:

1.5

- o **codePointBefore**

public int codePointBefore(int index)

Returns the character (Unicode code point) before the specified index. The index refers to char values (Unicode code units) and ranges from 1 to [length\(\)](#).

If the char value at (index - 1) is in the low-surrogate range, (index - 2) is not negative, and the char value at (index - 2) is in the high-surrogate range, then the supplementary code point value of the surrogate pair is returned. If the char value at index - 1 is an unpaired low-surrogate or a high-surrogate, the surrogate value is returned.

Parameters:

index - the index following the code point that should be returned

Returns:

the Unicode code point value before the given index.

Since:

1.5

- o **codePointCount**
- o public int codePointCount(int beginIndex,  
int endIndex)

Returns the number of Unicode code points in the specified text range of this sequence. The text range begins at the specified beginIndex and extends to the char at index endIndex - 1. Thus the length (in chars) of the text range is endIndex-beginIndex. Unpaired surrogates within this sequence count as one code point each.

Parameters:

beginIndex - the index to the first char of the text range.

endIndex - the index after the last char of the text range.

Returns:

the number of Unicode code points in the specified text range

Since:

1.5

- o **offsetByCodePoints**
- o public int offsetByCodePoints(int index,  
                          int codePointOffset)

Returns the index within this sequence that is offset from the given index by codePointOffset code points. Unpaired surrogates within the text range given by index and codePointOffset count as one code point each.

Parameters:

index - the index to be offset

codePointOffset - the offset in code points

Returns:

the index within this sequence

Since:

1.5

- o **getChars**
- o public void getChars(int srcBegin,  
                          int srcEnd,

- `char[] dst,`  
`int dstBegin)`

Characters are copied from this sequence into the destination character array dst. The first character to be copied is at index srcBegin; the last character to be copied is at index srcEnd-1. The total number of characters to be copied is srcEnd-srcBegin. The characters are copied into the subarray of dst starting at index dstBegin and ending at index:

`dstbegin + (srcEnd-srcBegin) - 1`

Parameters:

`srcBegin` - start copying at this offset.

`srcEnd` - stop copying at this offset.

`dst` - the array to copy the data into.

`dstBegin` - offset into dst.

Throws:

[IndexOutOfBoundsException](#) - if any of the following is true:

- `srcBegin` is negative
  - `dstBegin` is negative
  - the `srcBegin` argument is greater than the `srcEnd` argument.
  - `srcEnd` is greater than `this.length()`.
  - `dstBegin+srcEnd-srcBegin` is greater than `dst.length`
- **setCharAt**
  - `public void setCharAt(int index,`  
`char ch)`

The character at the specified index is set to ch. This sequence is altered to represent a new character sequence that is identical to the old character sequence, except that it contains the character ch at position index.

The index argument must be greater than or equal to 0, and less than the length of this sequence.

Parameters:

index - the index of the character to modify.

ch - the new character.

Throws:

[IndexOutOfBoundsException](#) - if index is negative or greater than or equal to length().

See Also:

[length\(\)](#)

o **append**

public [StringBuffer](#) append([Object](#) obj)

Appends the string representation of the Object argument.

The overall effect is exactly as if the argument were converted to a string by the method [String.valueOf\(Object\)](#), and the characters of that string were then [appended](#) to this character sequence.

Parameters:

obj - an Object.

Returns:

a reference to this object.

o **append**

public [StringBuffer](#) append([String](#) str)

Appends the specified string to this character sequence.

The characters of the String argument are appended, in order, increasing the length of this sequence by the length of the argument. If str is null, then the four characters "null" are appended.

Let  $n$  be the length of this character sequence just prior to execution of the append method. Then the character at index  $k$  in the new character sequence is equal to the character at index  $k$  in the old character sequence, if  $k$  is less than  $n$ ; otherwise, it is equal to the character at index  $k-n$  in the argument str.

Parameters:

str - a string.

Returns:

a reference to this object.

- o **append**

public [StringBuffer](#) append([StringBuffer](#) sb)

Appends the specified StringBuffer to this sequence.

The characters of the StringBuffer argument are appended, in order, to the contents of this StringBuffer, increasing the length of this StringBuffer by the length of the argument. If sb is null, then the four characters "null" are appended to this StringBuffer.

Let  $n$  be the length of the old character sequence, the one contained in the StringBuffer just prior to execution of the append method. Then the character at index  $k$  in the new character sequence is equal to the character at index  $k$  in the old character sequence, if  $k$  is less than  $n$ ; otherwise, it is equal to the character at index  $k-n$  in the argument sb.

This method synchronizes on this, the destination object, but does not synchronize on the source (sb).

Parameters:

sb - the StringBuffer to append.

Returns:

a reference to this object.

Since:

1.4

- o **append**

public [StringBuffer](#) append([CharSequence](#) s)

Appends the specified CharSequence to this sequence.

The characters of the CharSequence argument are appended, in order, increasing the length of this sequence by the length of the argument.

The result of this method is exactly the same as if it were an invocation of this.append(s, 0, s.length());

This method synchronizes on this, the destination object, but does not synchronize on the source (s).

If s is null, then the four characters "null" are appended.

Specified by:

[append](#) in interface [Appendable](#)

Parameters:

s - the CharSequence to append.

Returns:

a reference to this object.

Since:

1.5

- o **append**
- o public [StringBuffer](#) append([CharSequence](#) s,
  - o int start,
  - int end)

Appends a subsequence of the specified CharSequence to this sequence.

Characters of the argument s, starting at index start, are appended, in order, to the contents of this sequence up to the (exclusive) index end. The length of this sequence is increased by the value of end - start.

Let  $n$  be the length of this character sequence just prior to execution of the append method. Then the character at index  $k$  in this character sequence becomes equal to the character at index  $k$  in this sequence, if  $k$  is less than  $n$ ; otherwise, it is equal to the character at index  $k+start-n$  in the argument s.

If s is null, then this method appends characters as if the s parameter was a sequence containing the four characters "null".

Specified by:

### [append](#) in interface [Appendable](#)

Parameters:

s - the sequence to append.

start - the starting index of the subsequence to be appended.

end - the end index of the subsequence to be appended.

Returns:

a reference to this object.

Throws:

[IndexOutOfBoundsException](#) - if start is negative, or start is greater than end or end is greater than s.length()

Since:

1.5

- o **append**

`public StringBuffer append(char[] str)`

Appends the string representation of the char array argument to this sequence.

The characters of the array argument are appended, in order, to the contents of this sequence.

The length of this sequence increases by the length of the argument.

The overall effect is exactly as if the argument were converted to a string by the method [String.valueOf\(char\[\]\)](#), and the characters of that string were then [appended](#) to this character sequence.

Parameters:

str - the characters to be appended.

Returns:

a reference to this object.

- o **append**

`o public StringBuffer append(char[] str,`

`o int offset,`

int len)

Appends the string representation of a subarray of the char array argument to this sequence.

Characters of the char array str, starting at index offset, are appended, in order, to the contents of this sequence. The length of this sequence increases by the value of len.

The overall effect is exactly as if the arguments were converted to a string by the method [String.valueOf\(char\[\],int,int\)](#), and the characters of that string were then [appended](#) to this character sequence.

Parameters:

str - the characters to be appended.

offset - the index of the first char to append.

len - the number of chars to append.

Returns:

a reference to this object.

Throws:

[IndexOutOfBoundsException](#) - if offset < 0 or len < 0 or offset+len > str.length

o **append**

public [StringBuffer](#) append(boolean b)

Appends the string representation of the boolean argument to the sequence.

The overall effect is exactly as if the argument were converted to a string by the method [String.valueOf\(boolean\)](#), and the characters of that string were then [appended](#) to this character sequence.

Parameters:

b - a boolean.

Returns:

a reference to this object.

o **append**

public [StringBuffer](#) append(char c)

Appends the string representation of the char argument to this sequence.

The argument is appended to the contents of this sequence. The length of this sequence increases by 1.

The overall effect is exactly as if the argument were converted to a string by the method [String.valueOf\(char\)](#), and the character in that string were then [appended](#) to this character sequence.

Specified by:

[append](#) in interface [Appendable](#)

Parameters:

c - a char.

Returns:

a reference to this object.

- **append**

public [StringBuffer](#) append(int i)

Appends the string representation of the int argument to this sequence.

The overall effect is exactly as if the argument were converted to a string by the method [String.valueOf\(int\)](#), and the characters of that string were then [appended](#) to this character sequence.

Parameters:

i - an int.

Returns:

a reference to this object.

- **appendCodePoint**

public [StringBuffer](#) appendCodePoint(int codePoint)

Appends the string representation of the codePoint argument to this sequence.

The argument is appended to the contents of this sequence. The length of this sequence increases by [Character.charCount\(codePoint\)](#).

The overall effect is exactly as if the argument were converted to a char array by the method [Character.toChars\(int\)](#) and the character in that array were then [appended](#) to this character sequence.

Parameters:

codePoint - a Unicode code point

Returns:

a reference to this object.

Since:

1.5

o **append**

`public StringBuffer append(long lng)`

Appends the string representation of the long argument to this sequence.

The overall effect is exactly as if the argument were converted to a string by the method [String.valueOf\(long\)](#), and the characters of that string were then [appended](#) to this character sequence.

Parameters:

lng - a long.

Returns:

a reference to this object.

o **append**

`public StringBuffer append(float f)`

Appends the string representation of the float argument to this sequence.

The overall effect is exactly as if the argument were converted to a string by the method [String.valueOf\(float\)](#), and the characters of that string were then [appended](#) to this character sequence.

Parameters:

f - a float.

Returns:

a reference to this object.

- o **append**

public [StringBuffer](#) append(double d)

Appends the string representation of the double argument to this sequence.

The overall effect is exactly as if the argument were converted to a string by the method [String.valueOf\(double\)](#), and the characters of that string were then [appended](#) to this character sequence.

Parameters:

d - a double.

Returns:

a reference to this object.

- o **delete**
- o public [StringBuffer](#) delete(int start,  
int end)

Removes the characters in a substring of this sequence. The substring begins at the specified start and extends to the character at index end - 1 or to the end of the sequence if no such character exists. If start is equal to end, no changes are made.

Parameters:

start - The beginning index, inclusive.

end - The ending index, exclusive.

Returns:

This object.

Throws:

[StringIndexOutOfBoundsException](#) - if start is negative, greater than length(), or greater than end.

Since:

1.2

- o **deleteCharAt**

```
public StringBuffer deleteCharAt(int index)
```

Removes the char at the specified position in this sequence. This sequence is shortened by one char.

Note: If the character at the given index is a supplementary character, this method does not remove the entire character. If correct handling of supplementary characters is required, determine the number of chars to remove by calling Character.charCount(thisSequence.codePointAt(index)), where thisSequence is this sequence.

Parameters:

index - Index of char to remove

Returns:

This object.

Throws:

[StringIndexOutOfBoundsException](#) - if the index is negative or greater than or equal to length().

Since:

1.2

- o **replace**

```
o public StringBuffer replace(int start,  
           int end,  
           String str)
```

Replaces the characters in a substring of this sequence with characters in the specified String. The substring begins at the specified start and extends to the character at index end - 1 or to the end of the sequence if no such character exists. First the characters in the substring are removed and then the specified String is inserted at start. (This sequence will be lengthened to accommodate the specified String if necessary.)

Parameters:

start - The beginning index, inclusive.

end - The ending index, exclusive.

str - String that will replace previous contents.

Returns:

This object.

Throws:

[StringIndexOutOfBoundsException](#) - if start is negative, greater than length(), or greater than end.

Since:

1.2

- o **substring**

`public String substring(int start)`

Returns a new String that contains a subsequence of characters currently contained in this character sequence. The substring begins at the specified index and extends to the end of this sequence.

Parameters:

start - The beginning index, inclusive.

Returns:

The new string.

Throws:

[StringIndexOutOfBoundsException](#) - if start is less than zero, or greater than the length of this object.

Since:

1.2

- o **subSequence**

`o public CharSequence subSequence(int start,  
int end)`

Returns a new character sequence that is a subsequence of this sequence.

An invocation of this method of the form

sb.subSequence(begin,&nbsp;end)

behaves in exactly the same way as the invocation

sb.substring(begin,&nbsp;end)

This method is provided so that this class can implement the [CharSequence](#) interface.

Specified by:

[subSequence](#) in interface [CharSequence](#)

Parameters:

start - the start index, inclusive.

end - the end index, exclusive.

Returns:

the specified subsequence.

Throws:

[IndexOutOfBoundsException](#) - if start or end are negative, if end is greater than length(), or if start is greater than end

Since:

1.4

- o **substring**
- o public [String](#) substring(int start,  
int end)

Returns a new String that contains a subsequence of characters currently contained in this sequence. The substring begins at the specified start and extends to the character at index end - 1.

Parameters:

start - The beginning index, inclusive.

end - The ending index, exclusive.

Returns:

The new string.

Throws:

[StringIndexOutOfBoundsException](#) - if start or end are negative or greater than length(), or start is greater than end.

Since:

1.2

- o **insert**
- o public [StringBuffer](#) insert(int index,  
                  char[] str,  
                  int offset,  
                  int len)

Inserts the string representation of a subarray of the str array argument into this sequence. The subarray begins at the specified offset and extends len chars. The characters of the subarray are inserted into this sequence at the position indicated by index. The length of this sequence increases by len chars.

Parameters:

index - position at which to insert subarray.

str - A char array.

offset - the index of the first char in subarray to be inserted.

len - the number of chars in the subarray to be inserted.

Returns:

This object

Throws:

[StringIndexOutOfBoundsException](#) - if index is negative or greater than length(), or offset or len are negative, or (offset+len) is greater than str.length.

Since:

1.2

- o **insert**
- o public [StringBuffer](#) insert(int offset,  
[Object](#) obj)

Inserts the string representation of the Object argument into this character sequence.

The overall effect is exactly as if the second argument were converted to a string by the method [String.valueOf\(Object\)](#), and the characters of that string were then inserted into this character sequence at the indicated offset.

The offset argument must be greater than or equal to 0, and less than or equal to the [length](#) of this sequence.

Parameters:

offset - the offset.

obj - an Object.

Returns:

a reference to this object.

Throws:

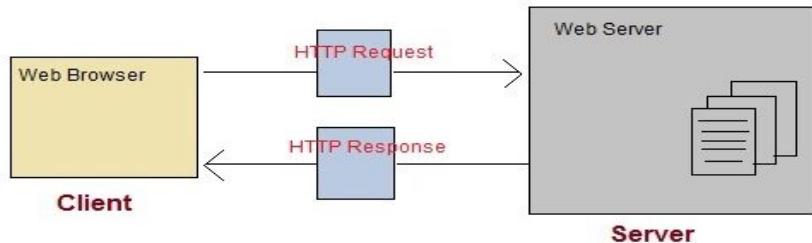
[StringIndexOutOfBoundsException](#) - if the offset is invalid.

## MODULE - 4

# SERVLETS

### Introduction to Web

The web clients make requests to web server. When a server answers a request, it usually sends some type of content(**MIME**- Multi purpose Internet Mail Exchange) to the client. The client uses web browser to send request to the server. The server often sends response to the browser with a set of instructions written in **HTML**(HyperText Markup Language)



Before Servlets, **CGI(Common Gateway Interface)** programming was used to create web applications. Here's how a CGI program works :

#### *Drawbacks of CGI programs*

- High response time because CGI programs execute in their own OS shell.
- CGI is not scalable.
- CGI programs are not always secure or object-oriented.
- It is Platform dependent.

Because of these disadvantages, developers started looking for better CGI solutions. And then Sun Microsystems developed **Servlet** as a solution.

### Servlet

**Servlet** technology is used to create web application (resides at server side and generates dynamic web page).

Servlet can be described in many ways, depending on the context.

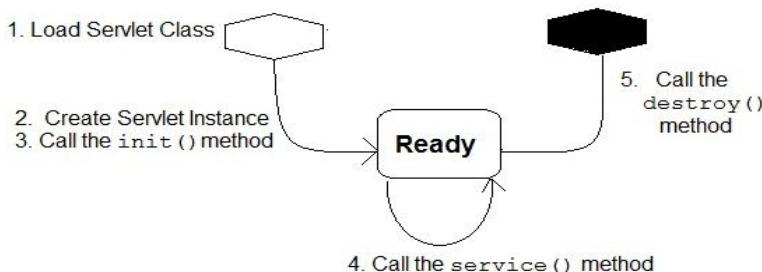
- Servlet is server side program.
- Servlet is an API that provides many interfaces and classes.
- Servlet is a web component that is deployed on the server to create dynamic web page.

### *Advantages of using Servlets*

1. **better performance:** because it creates a thread for each request not process.
2. **Portability:** Servlets are platform independent because it uses java language.
3. **Robust:** Servlets are managed by JVM so we don't need to worry about memory leak, garbage collection etc.
4. **Secure:** servlets are object oriented and runs inside JVM.
5. Servlets are scalable.

### Life Cycle of a Servlet

The web container maintains the life cycle of a servlet instance



1. **Loading Servlet Class :** A Servlet class is loaded when first request for the servlet is received by the Web Container.
2. **Servlet instance creation :** After the Servlet class is loaded, Web Container creates the instance of it. Servlet instance is created only once in the life cycle.
3. **Call to the init() method :** init() method is called by the Web Container on servlet instance to initialize the servlet.

public void **init**(ServletConfig config) throws ServletException

4. **Call to the service() method :** The containers call the service() method each time the request for servlet is received. The service() method will then call the doGet() or doPost() methos based ont eh type of the HTTP request, as explained in previous lessons.

```
public void service(ServletRequest request, ServletResponse response) throws ServletException,  
IOException
```

5. **Call to destroy() method:** The Web Container call the destroy() method before removing servlet instance, giving it a chance for cleanup activity.

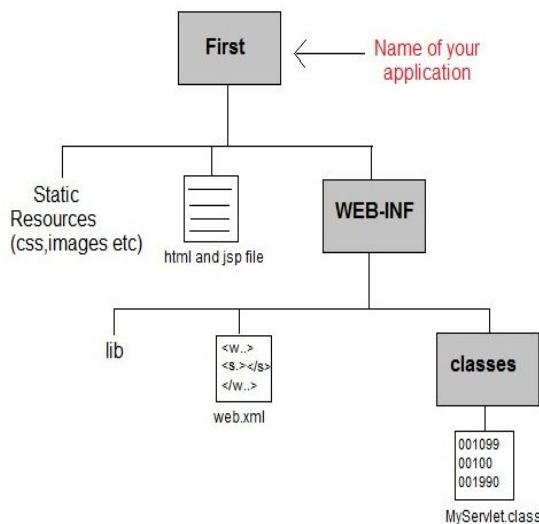
### **Steps to Create Servlet Application using tomcat server**

To create a Servlet application you need to follow the below mentioned steps. These steps are common for all the Web server. In our example we are using Apache Tomcat server. Apache Tomcat is an open source web server for testing servlets and JSP technology. Create directory structure for your application.

1. Create directory structure for your application.
2. Create a Servlet
3. Compile the Servlet
4. Create Deployment Descriptor for your application
5. Start the server and deploy the application
- 6.

#### ***1. Creating the Directory Structure***

Sun Microsystem defines a unique directory structure that must be followed to create a servlet application.



Create the above directory structure inside **Apache-Tomcat\webapps** directory.

All HTML, static files(images, css etc) are kept directly under **Web application** folder.

While all the Servlet classes are kept inside **classes** folder.

The **web.xml** (deployment descriptor) file is kept under **WEB-INF** folder.

## 2. Creating a Servlet

There are three different ways to create a servlet.

- By extending **HttpServlet** class
- By extending **GenericServlet** class
- By implementing **Servlet** interface

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

// extending HttpServlet class
  
```

```

public MyServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException
    {
        response.setContentType("text/html"); // set content type
        //get the stream to write the data
  
```

```
    PrintWriter out = response.getWriter(); // create printwriter object
    out.println("<h1>Hello Readers</h1>"); // print ur content on client web browser
}
}
```

Write above code and save it as **MyServlet.java** anywhere on your PC. Compile it from there and paste the class file into WEB-INF/classes/ directory that you have to create inside **Tomcat/webapps** directory.

---

---

```
import javax.servlet.*;
import java.io.*;

// extending GenericServlet class

public MyServlet extends GenericServlet
{
    public void service(ServletRequest request,ServletResponse response)
        throws IOException,ServletException{
    {
        response.setContentType("text/html"); // set content type
        //get the stream to write the data
        PrintWriter out = response.getWriter(); // create printwriter object
        out.println("<h1>Hello Readers</h1>"); // print ur content on client web browser
    }
}
```

### 3. Compiling a Servlet

To compile a Servlet a JAR file is required. Different servers require different JAR files. In Apache Tomcat server servlet-api.jar file is required to compile a servlet class.

- Download  **servlet-api.jar** file.
  - Paste the servlet-api.jar file inside Java\jdk\jre\lib\ext directory.
-

**NOTE:** After compiling your Servlet class you will have to paste the class file into WEB-INF/classes/directory.

#### *4. Create Deployment Descriptor*

**Deployment Descriptor(DD)** is an XML document that is used by Web Container to run Servlets and JSP pages.

##### **web.xml file**

```
<web-app>
    <servlet>
        <servlet-name> MyServlet </servlet-name>
        <servlet-class> MyServlet </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name> MyServlet </servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
</web-app>
```

#### *5. Starting Tomcat Server for the first time*

set JAVA\_HOME or JRE\_HOME in environment variable (It is required to start server).

Go to My Computer properties -> Click on advanced tab then environment variables -> Click on the new tab of user variable -> Write JAVA\_HOME in variable name and paste the path of jdk folder in variable value -> ok

#### *Run Servlet Application*

Open Browser and type **http:localhost:8080/First/hello**

#### **Servlet API**

Servlet API consists of two important packages that encapsulates all the important classes and interface, namely :

1. **`javax.servlet.*;`**

INTERFACES	CLASSES
Servlet	ServletInputStream
ServletContext	ServletOutputStream
ServletConfig	ServletException
ServletRequest	UnavailableException
ServletResponse	GenericServlet

### Interface Summary

<a href="#"><u>Servlet</u></a>	Defines methods that all servlets must implement.
<a href="#"><u>ServletRequest</u></a>	Defines an object to provide client request information to a servlet.
<a href="#"><u>ServletResponse</u></a>	Defines an object to assist a servlet in sending a response to the client.
<a href="#"><u>ServletConfig</u></a>	A servlet configuration object used by a servlet container to pass information to a servlet during initialization.
<a href="#"><u>ServletContext</u></a>	Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.

## Interface Servlet

Method	Description
void destroy( )	Called when the servlet is unloaded.
ServletConfig getServletConfig( )	Returns a <b>ServletConfig</b> object that contains any initialization parameters.
String getServletInfo( )	Returns a string describing the servlet.
void init(ServletConfig sc) throws ServletException	Called when the servlet is initialized. Initialization parameters for the servlet can be obtained from <i>sc</i> . An <b>UnavailableException</b> should be thrown if the servlet cannot be initialized.
void service(ServletRequest req, ServletResponse res) throws ServletException, IOException	Called to process a request from a client. The request from the client can be read from <i>req</i> . The response to the client can be written to <i>res</i> . An exception is generated if a servlet or IO problem occurs.

## Interface ServletRequest

Method	Description
Object getAttribute(String attr)	Returns the value of the attribute named <i>attr</i> .
String getCharacterEncoding( )	Returns the character encoding of the request.
int getContentLength( )	Returns the size of the request. The value -1 is returned if the size is unavailable.
String getContentType( )	Returns the type of the request. A <b>null</b> value is returned if the type cannot be determined.
ServletInputStream getInputStream( ) throws IOException	Returns a <b>ServletInputStream</b> that can be used to read binary data from the request. An <b>IllegalStateException</b> is thrown if <b>getReader( )</b> has already been invoked for this request.
String getParameter(String pname)	Returns the value of the parameter named <i>pname</i> .
Enumeration getParameterNames( )	Returns an enumeration of the parameter names for this request.
String[ ] getParameterValues(String name)	Returns an array containing values associated with the parameter specified by <i>name</i> .
String getProtocol( )	Returns a description of the protocol.
BufferedReader getReader( ) throws IOException	Returns a buffered reader that can be used to read text from the request. An <b>IllegalStateException</b> is thrown if <b>getInputStream( )</b> has already been invoked for this request.
String getRemoteAddr( )	Returns the string equivalent of the client IP address.
String getRemoteHost( )	Returns the string equivalent of the client host name.
String getScheme( )	Returns the transmission scheme of the URL used for the request (for example, "http", "ftp").
String getServerName( )	Returns the name of the server.
int getServerPort( )	Returns the port number.

**Interface ServletResponse**

<b>Method</b>	<b>Description</b>
String getCharacterEncoding( )	Returns the character encoding for the response.
ServletOutputStream getOutputStream( ) throws IOException	Returns a <b>ServletOutputStream</b> that can be used to write binary data to the response. An <b>IllegalStateException</b> is thrown if <b>getWriter( )</b> has already been invoked for this request.
PrintWriter getWriter( ) throws IOException	Returns a <b>PrintWriter</b> that can be used to write character data to the response. An <b>IllegalStateException</b> is thrown if <b>getOutputStream( )</b> has already been invoked for this request.
void setContentLength(int size)	Sets the content length for the response to <i>size</i> .
void setContentType(String type)	Sets the content type for the response to <i>type</i> .

**Interface ServletConfig**

<b>Method</b>	<b>Description</b>
ServletContext getServletContext( )	Returns the context for this servlet.
String getInitParameter(String param)	Returns the value of the initialization parameter named <i>param</i> .
Enumeration getInitParameterNames( )	Returns an enumeration of all initialization parameter names.
String getServletName( )	Returns the name of the invoking servlet.

**Interface ServletContext**

<b>Method</b>	<b>Description</b>
Object getAttribute(String attr)	Returns the value of the server attribute named <i>attr</i> .
String getMimeType(String file)	Returns the MIME type of <i>file</i> .
String getRealPath(String vpath)	Returns the real path that corresponds to the virtual path <i>vpath</i> .
String getServerInfo( )	Returns information about the server.
void log(String s)	Writes <i>s</i> to the servlet log.
void log(String s, Throwable e)	Writes <i>s</i> and the stack trace for <i>e</i> to the servlet log.
void setAttribute(String attr, Object val)	Sets the attribute specified by <i>attr</i> to the value passed in <i>val</i> .

### Class Summary

<a href="#"><u>GenericServlet</u></a>	Defines a generic, protocol-independent servlet.
<a href="#"><u>ServletInputStream</u></a>	Provides an input stream for reading binary data from a client request, including an efficient readLine method for reading data one line at a time.
<a href="#"><u>ServletOutputStream</u></a>	Provides an output stream for sending binary data to the client.
<a href="#"><u>ServletException</u></a>	Defines a general exception a servlet can throw when it encounters difficulty.
<a href="#"><u>UnavailableException</u></a>	Defines an exception that a servlet or filter throws to indicate that it is permanently or temporarily unavailable.

### Class GenericServlet

java.lang.Object

└ javax.servlet.GenericServlet

All Implemented Interfaces: [Servlet](#), [ServletConfig](#)

### Class ServletInputStream

java.lang.Object

└ java.io.InputStream

└ javax.servlet.ServletInputStream

### Method Summary

int	<a href="#"><u>readLine</u></a> (byte[] b,	int off,	int len)
Reads the input stream, one line at a time.			

### Class ServletOutputStream

### Method Summary

void	<a href="#"><u>println</u></a> ()
void	<a href="#"><u>print</u></a> (java.lang.String s)  Writes a String to the client, without a carriage return-line feed (CRLF) character at the end.

void	<a href="#"><b>println()</b></a> Writes a carriage return-line feed (CRLF) to the client.
void	<a href="#"><b>println(java.lang.String s)</b></a> Writes a String to the client, followed by a carriage return-line feed (CRLF).

2. **javax.servlet.http.\*;**

<b>CLASSES</b>	
<b>INTERFACES</b>	
Cookie	HttpServletRequest
HttpServlet	HttpServletResponse
HttpSessionBindingEvent	HttpSession

### Interface Summary

<a href="#"><b>HttpServletRequest</b></a>	Extends the <a href="#">ServletRequest</a> interface to provide request information for HTTP servlets.
<a href="#"><b>HttpServletResponse</b></a>	Extends the <a href="#">ServletResponse</a> interface to provide HTTP-specific functionality in sending a response.
<a href="#"><b>HttpSession</b></a>	Provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

## Interface HttpServletResponse

Method	Description
void addCookie(Cookie cookie)	Adds <i>cookie</i> to the HTTP response.
boolean containsHeader(String field)	Returns <b>true</b> if the HTTP response header contains a field named <i>field</i> .
String encodeURL(String url)	Determines if the session ID must be encoded in the URL identified as <i>url</i> . If so, returns the modified version of <i>url</i> . Otherwise, returns <i>url</i> . All URLs generated by a servlet should be processed by this method.
String encodeRedirectURL(String url)	Determines if the session ID must be encoded in the URL identified as <i>url</i> . If so, returns the modified version of <i>url</i> . Otherwise, returns <i>url</i> . All URLs passed to <b>sendRedirect( )</b> should be processed by this method.
void sendError(int c) throws IOException	Sends the error code <i>c</i> to the client.
void sendError(int c, String s) throws IOException	Sends the error code <i>c</i> and message <i>s</i> to the client.
void sendRedirect(String url) throws IOException	Redirects the client to <i>url</i> .
void setDateHeader(String field, long msec)	Adds <i>field</i> to the header with date value equal to <i>msec</i> (milliseconds since midnight, January 1, 1970, GMT).
void setHeader(String field, String value)	Adds <i>field</i> to the header with value equal to <i>value</i> .
void setIntHeader(String field, int value)	Adds <i>field</i> to the header with value equal to <i>value</i> .
void setStatus(int code)	Sets the status code for this response to <i>code</i> .

## Interface HttpServletRequest

Method	Description
<code>String getAuthType( )</code>	Returns authentication scheme.
<code>Cookie[ ] getCookies( )</code>	Returns an array of the cookies in this request.
<code>long getDateHeader(String field)</code>	Returns the value of the date header field named <i>field</i> .
<code>String getHeader(String field)</code>	Returns the value of the header field named <i>field</i> .
<code>Enumeration getHeaderNames( )</code>	Returns an enumeration of the header names.
<code>int getIntHeader(String field)</code>	Returns the <code>Int</code> equivalent of the header field named <i>field</i> .
<code>String getMethod( )</code>	Returns the HTTP method for this request.
<code>String getPathInfo( )</code>	Returns any path information that is located after the servlet path and before a query string of the URL.
<code>String getPathTranslated( )</code>	Returns any path information that is located after the servlet path and before a query string of the URL after translating it to a real path.
<code>String getQueryString( )</code>	Returns any query string in the URL.
<code>String getRemoteUser( )</code>	Returns the name of the user who issued this request.
<code>String getRequestedSessionId( )</code>	Returns the ID of the session.
<code>String getRequestURI( )</code>	Returns the URI.
<code>StringBuffer getRequestURL( )</code>	Returns the URL.
<code>String getServletPath( )</code>	Returns that part of the URL that identifies the servlet.
<code>HttpSession getSession( )</code>	Returns the session for this request. If a session does not exist, one is created and then returned.
<code>HttpSession getSession(boolean new)</code>	If <i>new</i> is <code>true</code> and no session exists, creates and returns a session for this request. Otherwise, returns the existing session for this request.
<code>boolean isRequestedSessionIdFromCookie( )</code>	Returns <code>true</code> if a cookie contains the session ID. Otherwise, returns <code>false</code> .
<code>boolean isRequestedSessionIdFromURL( )</code>	Returns <code>true</code> if the URL contains the session ID. Otherwise, returns <code>false</code> .
<code>boolean isRequestedSessionIdValid( )</code>	Returns <code>true</code> if the requested session ID is valid in the current session context.

**Interface****HttpSession**

<b>Method</b>	<b>Description</b>
<code>Object getAttribute(String attr)</code>	Returns the value associated with the name passed in <code>attr</code> . Returns <code>null</code> if <code>attr</code> is not found.
<code>Enumeration getAttributeNames( )</code>	Returns an enumeration of the attribute names associated with the session.
<code>long getCreationTime( )</code>	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when this session was created.
<code>String getId( )</code>	Returns the session ID.
<code>long getLastAccessedTime( )</code>	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the client last made a request for this session.
<code>void invalidate( )</code>	Invalidates this session and removes it from the context.
<code>boolean isNew( )</code>	Returns <code>true</code> if the server created the session and it has not yet been accessed by the client.
<code>void removeAttribute(String attr)</code>	Removes the attribute specified by <code>attr</code> from the session.
<code>void setAttribute(String attr, Object val)</code>	Associates the value passed in <code>val</code> with the attribute name passed in <code>attr</code> .

**Class Summary**

<b><u>Cookie</u></b>	Creates a cookie, a small amount of information sent by a servlet to a Web browser, saved by the browser, and later sent back to the server.
<b><u>HttpServlet</u></b>	Provides an abstract class to be subclassed to create an HTTP servlet suitable for a Web site.
<b><u>HttpSessionBindingEvent</u></b>	Events of this type are either sent to an object that implements <a href="#">HttpSessionBindingListener</a> when it is bound or unbound from a session, or to a <a href="#">HttpSessionAttributeListener</a> that has been configured in the deployment descriptor when any attribute is bound, unbound or replaced in a session.
<b><u>HttpSessionEvent</u></b>	This is the class representing event notifications for changes to sessions within a web application.

## Class Cookie

Method	Description
Object clone( )	Returns a copy of this object.
String getComment( )	Returns the comment.
String getDomain( )	Returns the domain.
int getMaxAge( )	Returns the maximum age (in seconds).
String getName( )	Returns the name.
String getPath( )	Returns the path.
boolean getSecure( )	Returns <b>true</b> if the cookie is secure. Otherwise, returns <b>false</b> .
String getValue( )	Returns the value.
int getVersion( )	Returns the version.
void setComment(String <i>c</i> )	Sets the comment to <i>c</i> .
void setDomain(String <i>d</i> )	Sets the domain to <i>d</i> .
void setMaxAge(int <i>secs</i> )	Sets the maximum age of the cookie to <i>secs</i> . This is the number of seconds after which the cookie is deleted.
void setPath(String <i>p</i> )	Sets the path to <i>p</i> .
void setSecure(boolean <i>secure</i> )	Sets the security flag to <i>secure</i> .
void setValue(String <i>v</i> )	Sets the value to <i>v</i> .
void setVersion(int <i>v</i> )	Sets the version to <i>v</i> .

**Class****HttpServlet**

<b>Method</b>	<b>Description</b>
<code>void doDelete(HttpServletRequest req,                   HttpServletResponse res) throws IOException, ServletException</code>	Handles an HTTP DELETE request.
<code>void doGet(HttpServletRequest req,               HttpServletResponse res) throws IOException, ServletException</code>	Handles an HTTP GET request.
<code>void doHead(HttpServletRequest req,               HttpServletResponse res) throws IOException,               ServletException</code>	Handles an HTTP HEAD request.
<code>void doOptions(HttpServletRequest req,               HttpServletResponse res) throws IOException, ServletException</code>	Handles an HTTP OPTIONS request.
<code>void doPost(HttpServletRequest req,               HttpServletResponse res) throws IOException, ServletException</code>	Handles an HTTP POST request.
<code>void doPut(HttpServletRequest req,               HttpServletResponse res) throws IOException, ServletException</code>	Handles an HTTP PUT request.
<code>void doTrace(HttpServletRequest req,               HttpServletResponse res) throws IOException, ServletException</code>	Handles an HTTP TRACE request.
<code>long getLastModified(HttpServletRequest req)</code>	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the requested resource was last modified.
<code>void service(HttpServletRequest req,               HttpServletResponse res) throws IOException, ServletException</code>	Called by the server when an HTTP request arrives for this servlet. The arguments provide access to the HTTP request and response, respectively.

**Class HttpSessionBindingEvent**

<b>Method Summary</b>	
<code>java.lang.String <a href="#">getName()</a></code>	Returns the name with which the attribute is bound to or unbound from the session.
<code><a href="#">HttpSession</a></code>	<code><a href="#">getSession()</a></code> Return the session that changed.
<code>java.lang.Object <a href="#">getValue()</a></code>	

Returns the value of the attribute that has been added, removed or replaced.

### Class HttpSessionEvent

#### Method Summary

[HttpSession](#) [\*\*getSession\(\)\*\*](#)

Return the session that changed.

#### Reading Servlet parameters

In this example, we will show how a parameter is passed to a  
**index.html**

```
<form method="post" action="check">  
    Name <input type="text" name="user" >  
    <input type="submit" value="submit">  
</form>
```

#### **MyServlet.java**

```
public class MyServlet extends HttpServlet {  
  
    protected void doPost(request, response){  
        ... // set content type . . .  
        String user=request.getParameter("user");  
  
        out.println("<h2> Welcome "+user+"</h2>");  
    }  
}
```

**NOTE:** getParameter( ) returns string to get int value use

**Integer.parseInt("string");**

## Handling HTTP request and Response

HttpServlet class provides various methods that handle various types of HTTP request.

A servlet typically must override at least one method, usually one of these:

- doGet, if the servlet supports HTTP GET requests
- doPost, for HTTP POST requests
- doPut, for HTTP PUT requests
- doDelete, for HTTP DELETE requests

GET and POST methods are commonly used when handling form input.

**NOTE: By default a request is Get request.**

### *Difference between GET and POST requests*

GET Request	POST Request
Data is sent in header to the server	Data is sent in the request body
Get request can send only limited amount of data	Large amount of data can be sent.
Get request is not secured because data is exposed in URL	Post request is secured because data is not exposed in URL.

## Session

**Session** simply means a particular interval of time. **Session Tracking** is a way to maintain state (data) of an user. It is also known as **session management** in servlet.

Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.

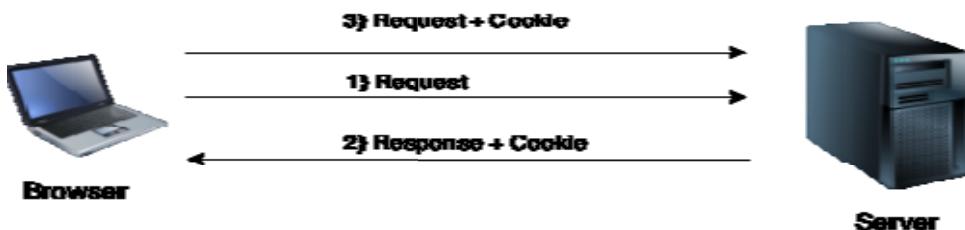
There are 2 techniques used in Session tracking:

1. **Cookies**
2. **HttpSession**

## Cookies in Servlet

A **cookie** is a small piece of information that is persisted between the multiple client requests.

By default, each request is considered as a new request. In cookies technique, we add cookie with response from the servlet. So cookie is stored in the cache of the browser. After that if request is sent by the user, cookie is added with request by default. Thus, we recognize the user as the old user.



### Advantage of Cookies

1. Simplest technique of maintaining the state.
2. Cookies are maintained at client side.

### Disadvantage of Cookies

1. It will not work if cookie is disabled from the browser.
2. Only textual information can be set in Cookie object.

#### Creating a new Cookie

```
Cookie ck = new Cookie("username", name);
```

creating a new cookie object

#### Setting up lifespan for a cookie

```
ck.setMaxAge(30*60);
```

setting maximum age of cookie

#### Sending the cookie to the client

```
response.addCookie(ck);
```

adding cookie to response object

#### Getting cookies from client request

```
Cookie[] cks = request.getCookies();
```

getting the cookie for request object

### *Example demonstrating usage of Cookies*

#### **index.html**

```
<form method="post" action=" MyServlet ">
    Name:<input type="text" name="user" /><br/>
    Password:<input type="text" name="pass" ><br/>
    <input type="submit" value="submit">
</form>
```

**MyServlet.java**

```
public class MyServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    {
        // .....
        String name = request.getParameter("user");
        String pass = request.getParameter("pass");

        if(pass.equals("1234"))
        {
            Cookie ck = new Cookie("username",name);
            response.addCookie(ck);

            //response.sendRedirect("First");//call ur servlet

            //creating submit button
            out.print("<form action= First >");
            out.print("<input type='submit' value='go'>");
            out.print("</form>");
        }
    }
}
```

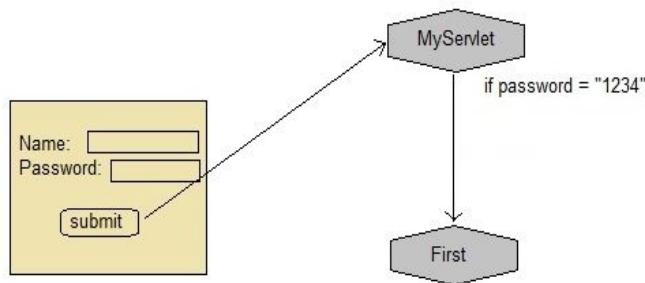
**First.java**

```
public class First extends HttpServlet {
```

```

protected void doGet(HttpServletRequest request, HttpServletResponse response)
{
    // ...
    Cookie[] cks = request.getCookies();
    out.println("Welcome "+cks[0].getValue());
}
}

```



### Useful Methods of Cookie class

Method	Description
public void setMaxAge(int expiry)	Sets the maximum age of the cookie in seconds.
public String getName()	Returns the name of the cookie. The name cannot be changed after creation.
public String getValue()	Returns the value of the cookie.
public void setName(String name)	Changes the name of the cookie.
public void setValue(String value)	Changes the value of the cookie.

### Other methods required for using Cookies

For adding cookie or getting the value from the cookie, we need some methods provided by other interfaces. They are:

1. **public void addCookie(Cookie ck):** method of HttpServletResponse interface is used to add cookie in response object.
2. **public Cookie[ ] getCookies():** method of HttpServletRequest interface is used to return all the cookies from the browser.

## HttpSession

**HttpSession** object is used to store entire session with a specific client. We can store, retrieve and remove attribute from **HttpSession** object. Any servlet can have access to **HttpSession** object throughout the getSession() method.

### Creating a new session

```
HttpSession session = request.getSession();
```

getSession() method returns a session.  
If the session already exist, it return the existing session else create a new session

```
HttpSession session = request.getSession(true);
```

getSession(true) always return a new session

### Getting a pre-existing session

```
HttpSession session = request.getSession(false);
```

return a pre-existing session

### Destroying a session

```
session.invalidate();
```

destroy a session

## *Some Important Methods of HttpSession*

Methods	Description
long getCreationTime()	returns the time when the session was created, measured in milliseconds since midnight January 1, 1970 GMT.

String getId()	returns a string containing the unique identifier assigned to the session.
int getMaxInactiveInterval()	returns the maximum time interval, in seconds.
void invalidate()	destroy the session
boolean isNew()	returns true if the session is new else false

### *Complete Example demonstrating usage of HttpSession*

#### **index.html**

```
<form method="post" action="Validate">
    User: <input type="text" name="uname" /><br/>
    <input type="submit" value="submit">
</form>
```

#### **Validate.java**

```
public class Validate extends HttpServlet {
    protected void doPost(request, response) {
        // ...
        String name = request.getParameter("user");
        //creating a session
        HttpSession session = request.getSession();
        session.setAttribute("user", uname);
        response.sendRedirect("Welcome");
    }
}
```

**Welcome.java**

```
public class Welcome extends HttpServlet {  
  
    protected void doGet(request, response){  
        // ....  
        HttpSession session = request.getSession();  
        String user = (String)session.getAttribute("user");  
        out.println("Hello "+user);  
    }  
}
```

## JSP

- **Java Server Page** technology is used to create dynamic web applications.
- JSP pages are easier to maintain than a **Servlet**.
- JSP pages are opposite of Servlets as a servlet adds HTML code inside Java code, while JSP adds Java code inside HTML using JSP tags.
- Everything a Servlet can do, a JSP page can also do it.
- JSP enables us to write HTML pages containing tags, inside which we can include powerful Java programs

### Why JSP is preferred over servlets?

- JSP provides an **easier** way to code **dynamic web pages**.
- JSP **does not require** additional files like, java class files, **web.xml** etc
- Any **change** in the JSP code is handled by **Web Container**(Application server like tomcat), and doesn't require re-compilation.
- JSP pages can be directly accessed, and web.xml mapping is not required like in servlets.

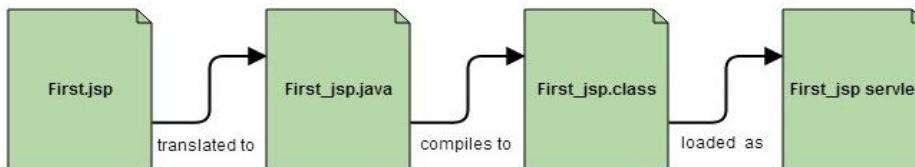
### Advantage of JSP

- Easy to maintain and code.
- High Performance and Scalability.
- JSP is built on Java technology, so it is **platform independent**.

### Life cycle of a JSP Page

The JSP pages follows these phases:

- Initialization ( `jspInit()` method is invoked by the container).
- Request processing ( `_jspService()` method is invoked by the container).
- Destroy ( `jspDestroy()` method is invoked by the container).



### In the end a JSP becomes a Servlet

- As depicted in the above diagram, JSP page is translated into servlet by the help of JSP translator. The JSP translator is a part of webserver that is responsible to translate the JSP page into servlet. After that Servlet page is compiled by the compiler and gets converted into the class file.
- **Web Container** translates JSP code into a **servlet class source(.java)** file, then compiles that into a java servlet class. In the third step, the servlet class bytecode is loaded using classloader. The Container then creates an instance of that servlet class.
- The initialized servlet can now service request. For each request the **Web Container** call the `_jspService()` method. When the Container removes the servlet instance from service, it calls the `jspDestroy()` method to perform any required clean up.

### Do we need to follow directory structure to run a simple JSP ?

No, put jsp files in a folder directly and deploy that folder. It will be running fine. But if you are using bean class, Servlet or tld file then directory structure is required.

### JSP Scripting Element

JSP Scripting element are written inside `<% %>` tags. These code inside `<% %>` tags are processed by the JSP engine during translation of the JSP page. Any other text in the JSP page is considered as HTML code or plain text.

### There are five different types of scripting elements

1. **JSP scriptlet tag** A scriptlet tag is used to execute java source code in JSP.  
`<% java source code %>`

In this example, we are displaying a welcome message.

```
<html>
<body>
<% out.print("welcome to jsp"); %>
</body>
</html>
```

### 2. **JSP Declaration Tag**

The **JSP declaration tag** is used to *declare variables, objects and methods*.

The code written inside the jsp declaration tag is placed outside the `service()` method of auto generated servlet.

So it doesn't get memory at each request.

```
<%! field or method declaration %>
```

### Difference between JSP Scriptlet tag and Declaration tag

Jsp Scriptlet Tag	Jsp Declaration Tag
The jsp scriptlet tag can only declare variables not methods.	The jsp declaration tag can declare variables as well as methods.
The declaration of scriptlet tag is placed inside the <code>_jspService()</code> method.	The declaration of jsp declaration tag is placed outside the <code>_jspService()</code> method.

#### declaration tag with variable

In  
**index.jsp**

```
<html>
<body>
<%! int data=50; %>
<%= "Value of the variable is:"+data %>
</body>
</html>
```

#### declaration tag that declares method

**index.jsp**

```
<html>
<body>
<%!
    int cube(int n){
        return n*n*n;
    }
%>
<%= "Cube of 3 is:"+cube(3) %>
```

### 3. **JSP Expression Tag**

Expression Tag is used to print out java language expression that is put between the tags. An expression tag can hold any java language expression that can be used as an argument to the `out.print()` method.

Syntax of Expression Tag

`<%= JavaExpression %>`

`<%=(2*5)%>` //note no ; at end of statement.

#### 4. JSP directives

The **jsp directives** are messages that tells the web container how to translate a JSP page into the corresponding servlet.

Syntax `<%@ directive attribute="value" %>`

There are three types of directives:

- import directive
- include directive
- taglib directive

##### a. import

The import attribute is used to import class,interface or all the members of a package. It is similar to import keyword in java class or interface.

`<%@ page import="java.util.Date" %>`

Today is: `<%= new Date() %>`

##### b. Include Directive

The *include* directive tells the Web Container to copy everything in the included file and paste it into current JSP file. The include directive is used to include the contents of any resource it may be jsp file, html file or text file. Syntax of **include** directive is:

`<%@ include file="filename.jsp" %>`

##### c. JSP Taglib directive

The JavaServer Pages API allow you to define custom JSP tags that look like HTML or XML tags and a tag library is a set of user-defined tags that implement custom behavior. The **taglib** directive declares that your JSP page uses a set of custom tags, identifies the location of the library, and provides means for identifying the custom tags in your JSP page.

Syntax `<%@ taglib uri = "uri" prefix = "prefixOfTag" %>`

For example, suppose the **custlib** tag library contains a tag called **hello**. If you wanted to use the hello tag with a prefix of **mytag**, your tag would be `<mytag:hello>` and it will be used in your JSP file as follows

`<%@ taglib uri = "http://www.example.com/custlib" prefix = "mytag" %>`

```
<html>
  <mytag:hello/>
</body>
</html>
```

#### 5. JSP Comments

JSP comment marks text or statements that the JSP container should ignore.

Syntax of the JSP comments `<%-- This is JSP comment --%>`

#### Request String

- The query string contains the attribute & the value of the html form or JSP form, which sends with the POST /GET method to Servlet or JSP page for any request.

- A query string is the part of a URL which is attached to the end, after the file name. It begins with a question mark and usually includes information in pairs. The format is parameter=value, as in the following example:  
www.mediacollege.com/cgi-bin/myscript.cgi?name="Umesh"
- Query strings can contain multiple sets of parameters, separated by an ampersand (&) like so:  
www.mediacollege.com/cgi-bin/myscript.cgi?fname="Umesh"&lname="M"
- to parse this info we use method of JSP **request** object
- we can easily get using **request.getParameter()** with name of parameter as argument, to get its value.

index.html

```
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

welcome.jsp

```
<%
String name=request.getParameter("uname");
out.print("welcome "+name);
%>
```

### welcome.jsp

```
<html>
<body>
<%
    String
    name=request.getParameter("uname");
    out.print("Welcome "+name);
    session.setAttribute("user",name);

<a href="second.jsp">second jsp page</a>
%>
</body>
</html>
```

### Following are the JSP implicit object

Implicit Object	Description
<b>request</b>	The <b>HttpServletRequest</b> object associated with the request.
<b>response</b>	The <b>HttpServletRequest</b> object associated with the response that is sent back to the browser.
<b>out</b>	The <b>JspWriter</b> object associated with the output stream of the response.
<b>session</b>	The <b>HttpSession</b> object associated with the session for the given user of request.

### Session implicit object (concept same as servlet)

In JSP, session is an implicit object of type HttpSession. The Java developer can use this object to set, get or remove attribute or to get session information.

Example of session implicit object

index.html

```
<html>
<body>
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

```
</body>
</html>
```

### **second.jsp**

```
<html>
<body>
<%
    String name=(String)session.getAttribute("user");
    out.print("Hello "+name);
%
</body>
</html>
```

*//other methods of Session what u used in servlets can be used here.*

#### **Cookie (concept same as servlet)**

##### **Step 1: Creating a Cookie object**

```
Cookie cookie = new Cookie("key","value");
```

##### **Step 2: Setting the maximum age**

You use **setMaxAge** to specify how long (in seconds) the cookie should be valid. The following code will set up a cookie for 24 hours.

```
cookie.setMaxAge(60*60*24);
```

##### **Step 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);
```

```
<%
// Create cookies for first and last names.
Cookie firstName = new Cookie("first_name", request.getParameter("first_name"));
Cookie lastName = new Cookie("last_name", request.getParameter("last_name"));

// Set expiry date after 24 Hrs for both the cookies.
firstName.setMaxAge(60*60*24);
lastName.setMaxAge(60*60*24);

// Add both the cookies in the response header.
response.addCookie( firstName );
response.addCookie( lastName );
%>
```

```
<html>
<body>


```

```
<%= request.getParameter("last_name")%>
</p>
</body>
</html>
```

### **Reading Cookies with JSP**

```
<html>
<body>
<%
Cookie cookie = null;
Cookie[] cookies = null;

// Get an array of Cookies associated with the this domain
cookies = request.getCookies();

if( cookies != null ) {
    out.println("<h2> Found Cookies Name and Value</h2>");

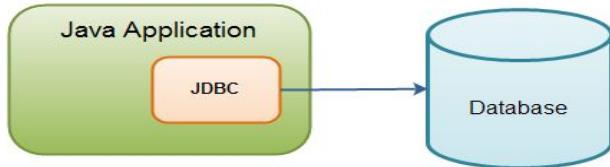
    for (int i = 0; i < cookies.length; i++) {
        cookie = cookies[i];
        out.print("Name : " + cookie.getName() + ", ");
        out.print("Value: " + cookie.getValue() + "<br/>");
    }
} else {
    out.println("<h2>No cookies founds</h2>");
}
%>
</body>

</html>
//other methods of Cookie what u used in servlets can be used here.
```

## MODULE -5

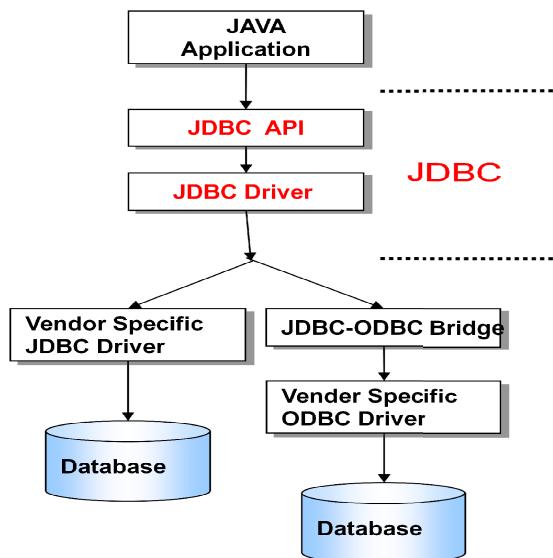
# JDBC

- **Java Database Connectivity(JDBC)** is an **Application Programming Interface(API)** used to connect Java application with Database.



- JDBC is used to interact with various type of Database such as Oracle, MS Access, MySQL and SQL Server.
- JDBC can also be defined as the platform-independent interface between a relational database and Java programming.
- It allows java program to execute SQL statement and retrieve result from database.

### JDBC Model



- JDBC consists of two parts:
  - JDBC API, a purely Java-based API
  - JDBC driver
    - Communicates with vendor-specific drivers
  - JDBC driver classified into 4 categories

## JDBC Driver

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

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

### **1) JDBC-ODBC bridge driver**

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



#### **Advantages:**

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

#### **Disadvantages:**

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

---

### **2) Native-API driver**

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

**Advantage:**

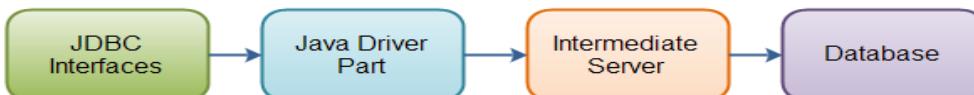
- performance upgraded than JDBC-ODBC bridge driver.

**Disadvantage:**

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

**3) Network Protocol driver**

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

**Advantage:**

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

**Disadvantages:**

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

**4) Thin driver**

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

**Advantage:**

- Better performance than all other drivers.

- No software is required at client side or server side.

### **Disadvantage:**

- Drivers depends on the Database.

### **Various Database drivers**

<b>Database name</b>	<b>Driver name</b>
MS Access	<b>sun.jdbc.odbc.JdbcOdbcDriver</b>
Oracle	<b>oracle.jdbc.driver.OracleDriver</b>
Microsoft SQL Server 2000	<b>com.microsoft.sqlserver.jdbc.SQLServerDriver</b>
MySQL	<b>org.gjt.mm.mysql.Driver</b>

### **JDBC Packages**

JDBC API is contained in 2 packages.

- **import java.sql.\*;**
  - contains core java data objects of JDBC API. It's a part of J2SE.
- **import javax.sql.\* ;**
  - It extends java.sql and is in J2EE

## **5 Steps to connect to the database in java**

There are 5 steps to connect any java application with the database in java using JDBC. They are as follows:

- Register the driver class
- Creating connection
- Creating statement
- Executing queries
- Closing connection

---

### **1) Register the driver class**

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

```
Class.forName("driverClassName");
```

## 2) Create the connection object

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

```
Connection con = DriverManager.getConnection(url, user, password);
```

## 3) Create & Execute query

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

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

```
Statement st = con.createStatement();
ResultSet rs = st.executeQuery(sql);
```

## 4) Process data returned from DBMS

```
while(rs.next()){
    System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

## 5) Close the connection object

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

```
rs.close();
st.close();
con.close();
```

## complete example

```
import java.sql.*;
import java.sql.*;
Class Dbconnection
```

```
{    public static void main(String args[])
    {    //Dynamically loads a driver class, for Oracle database
Class.forName("oracle.jdbc.driver.OracleDriver");

        //Establishes connection to database by obtaining a Connection object
Connection con = DriverManager.getConnection( "jdbc:oracle:thin:@localhost:1521:XE",
"scott", "tiger");

/*      jdbc- is API
        oracle- is DB name
        thin- is the driver
        localhost-is sever name on which oracle is running (can give IP address)
        1521- is port number
        XE- is oracle service name      */

Statement statement = con.createStatement();
String sql = "select * from users";
ResultSet result = statement.executeQuery(sql);
while(result.next()) {
    String name = result.getString("name");
    long age = result.getInt("age");
    System.out.println(name);
    System.out.println(age);
}
result.close();
statement.close();
con.close();
}
```

## Process results

When you execute an SQL query you get back a ResultSet. The ResultSet contains the result of your SQL query. The result is returned in rows with columns of data. You iterate the rows of the ResultSet like this:

```
while(result.next()) {
```

```

String name = result.getString("name");
long age = result.getLong ("age");

}

```

The `ResultSet.next()` method moves to the next row in the `ResultSet`, if there are anymore rows. If there are anymore rows, it returns true. If there were no more rows, it will return false.

You can also pass an index of the column instead, like this:

```

while(result.next()) {
    result.getString(1);
    result.getInt (2);
}

```

## Statement Object

There are 3 types of statement objects to execute the sql query

Interfaces	Recommended Use
Statement	Use the for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use the when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use the when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

### 1. Statement object

- The **Statement interface** provides methods to execute queries with the database.
- The statement interface is a factory of `ResultSet` i.e. it provides factory method to get the object of `ResultSet`.

The important methods of Statement interface are as follows:

**1) ResultSet executeQuery(String sql):** is used to execute SELECT query. It returns the object of ResultSet.

Example show above

**2) int executeUpdate(String sql):** is used to execute specified query, it may be create, drop, insert, update, delete etc.

#### Snippet

```
Statement stmt=con.createStatement();
// for insert
int result=stmt.executeUpdate("insert into emp values(33,'Irfan',50000)");
// for update
int result=stmt.executeUpdate("update emp set name='Vimal',salary=10000 where id=33");
// for delete
int result=stmt.executeUpdate("delete from emp where id=33");
System.out.println(result+" records affected");
con.close();
```

**3) boolean execute(String sql):** is used to execute queries that may return multiple results.

```
boolean status = stmt.execute(anyquery);
if(status){
    //query is a select query.
    ResultSet rs = stmt.getResultSet()
```

## 2. PreparedStatement object

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.

**Improves performance:** The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.

```
// PreparedStatement to insert record
PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
stmt.setInt(1,101);//1 specifies the first parameter in the query
stmt.setString(2,"Ratan");
int i=stmt.executeUpdate();
```

---

```
System.out.println(i+" records inserted");
```

The **setXXX()** methods are used to supply values to the parameters

All of the **Statement object's** methods for interacting with the database execute(), executeQuery(), and executeUpdate() also work with the PreparedStatement object.

```
// PreparedStatement to update record
```

```
PreparedStatement stmt=con.prepareStatement("update emp set name=? where id=?");
stmt.setString(1," Ratan ");//1 specifies the first parameter in the query i.e. name
stmt.setInt(2,101);
int i=stmt.executeUpdate();
System.out.println(i+" records updated");
```

```
// PreparedStatement to delete record
```

```
PreparedStatement stmt=con.prepareStatement("delete from emp where id=?");
stmt.setInt(1,101);
int i=stmt.executeUpdate();
System.out.println(i+" records deleted");
```

### 3. CallableStatement Objects

CallableStatement interface is used to call the **stored procedures and functions**.

Suppose you need the get the age of the employee based on the date of birth, you may create a function that receives date as the input and returns age of the employee as the output.

#### Snippet

```
String SQL = "{call getEmpName (?, ?)}"; // stored procedure called
cs = conn.prepareCall (SQL);
cs.setInt(100);
// registerOutParameter() used to register OUT type used by stored procedure
cs.registerOutParameter(2, VARCHAR);
cs.execute();
String Name = cs.getString(1);
Cs.close();
```

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;

```

**Three types of parameters exist: IN, OUT, and INOUT.**

Parameter	Description
IN	Data that needs to be passed to stored procedure. values to IN parameters with the setXXX() methods.
OUT	contains value supplied by the SQL statement it returns. You retrieve values from the OUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values.

ResultSet interface

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

Method Name	Description
first()	Moves the cursor to the first row
last()	Moves the cursor to the last row.
previous()	Moves the cursor to the previous row. This method returns false if the previous row is off the result set
next()	Moves the cursor to the next row. This method returns false if there are no more rows in the result set

absolute(int row)	Moves the cursor to the specified row
relative(int row)	Moves the cursor the given number of rows from where it currently is pointing.
getRow()	Returns the row number that the cursor is pointing to.
beforeFirst()	Moves the cursor to just before the first row
afterLast()	Moves the cursor to just after the last row

When you create a ResultSet there are three attributes you can set. These are:

### 1. Type

1. **ResultSet.TYPE\_FORWARD\_ONLY** (default type)- TYPE\_FORWARD\_ONLY means that the ResultSet can only be navigated forward
2. **ResultSet.TYPE\_SCROLL\_INSENSITIVE**- TYPE\_SCROLL\_INSENSITIVE means that the ResultSet can be navigated (scrolled) both forward and backwards. The ResultSet is insensitive to changes while the ResultSet is open. That is, if a record in the ResultSet is changed in the database by another thread or process, it will not be reflected in already opened ResultSet's of this type.
3. **ResultSet.TYPE\_SCROLL\_SENSITIVE**- means that the ResultSet can be navigated (scrolled) both forward and backwards. The ResultSet is sensitive to changes in the underlying data source while the ResultSet is open.

### 2. Concurrency determines whether the ResultSet can be updated, or only read.

1. **ResultSet.CONCUR\_READ\_ONLY**- means that the ResultSet can only be read.
2. **ResultSet.CONCUR\_UPDATABLE**- means that the ResultSet can be both read and updated.

```
//Inserting Rows into a ResultSet u have updateXXX( )
rs.updateString (1, "raj");
rs.updateInt (2, 55);
rs.insertRow(); // call this method
```

```
//updating Rows into a ResultSet u have updateXXX()
```

Updates the current row by updating the corresponding row in the database.

```
rs.updateString ("name" , "ram");
```

```
rs.updateInt ("age" , 55);
```

```
rs.updateRow();// call this method
```

```
//Delete Rows from a ResultSet
```

Deletes the current row from the database

```
rs.deleteRow(3); //delete current row
```

3. **Holdability**- determines if a ResultSet is closed when the commit() method of the underlying connection is called.

1. **ResultSet.CLOSE\_CURSORS\_OVER\_COMMIT**- means that all ResultSet instances are closed when connection.commit() method is called on the connection that created the ResultSet.
2. **ResultSet.HOLD\_CURSORS\_OVER\_COMMIT**- means that the ResultSet is kept open when the connection.commit() method is called on the connection that created the ResultSet.

The HOLD\_CURSORS\_OVER\_COMMIT holdability might be useful if you use the ResultSet to update values in the database. Thus, you can open a ResultSet, update rows in it, call connection.commit() and still keep the same ResultSet open for future transactions on the same rows.

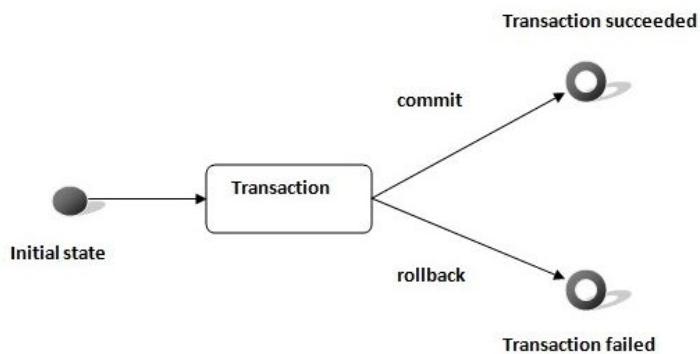
- U can set resultset attribute for Statement or PreparedStatement, like this:

```
Statement statement = connection.createStatement(
    ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY,
    ResultSet.CLOSE_CURSORS_OVER_COMMIT
);
```

```
PreparedStatement statement = connection.prepareStatement(sql,
    ResultSet.TYPE_FORWARD_ONLY,ResultSet.CONCUR_READ_ONLY,
    ResultSet.CLOSE_CURSORS_OVER_COMMIT
);
```

## Transaction

- A transaction is a set of actions to be carried out as a single, atomic action. Either all of the actions are carried out, or none of them are.
- *Advantage is fast performance It makes the performance fast because database is hit at the time of commit.*



In JDBC, **Connection interface** provides methods to manage transaction.

Method	Description
void setAutoCommit(boolean status)	It is true by default means each transaction is committed by default.
void commit()	commits the transaction.
void rollback()	cancels the transaction.

```

try{
    //Assume a valid connection object conn
    conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();

    String SQL = "INSERT INTO Employees " +
        "VALUES (106, 20, 'Rita', 'Tez')";
    stmt.executeUpdate(SQL);
    //Submit a malformed SQL statement that breaks
    String SQL = "INSERTED IN Employees " +
        "VALUES (107, 22, 'Sita', 'Singh')";
  
```

```
stmt.executeUpdate(SQL);
// If there is no error.
conn.commit();
}catch(SQLException se){
// If there is any error.
conn.rollback();
}
```

## Savepoints

- When you set a savepoint you define a logical rollback point within a transaction. If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.

The Connection object has two new methods that help you manage savepoints –

- **setSavepoint(String savepointName):** Defines a new savepoint. It also returns a Savepoint object.
- **releaseSavepoint(Savepoint savepointName):** Deletes a savepoint. Notice that it requires a Savepoint object as a parameter. This object is usually a savepoint generated by the setSavepoint() method.

```
try{
//Assume a valid connection object conn
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();

//set a Savepoint
Savepoint savepoint1 = conn.setSavepoint("Savepoint1");
String SQL = "INSERT INTO Employees " + "VALUES (106, 20, 'Rita', 'Tez')";
stmt.executeUpdate(SQL);

//Submit a malformed SQL statement that breaks
String SQL = "INSERTED IN Employees " + "VALUES (107, 22, 'Sita', 'Tez')";
stmt.executeUpdate(SQL);
```

```

// If there is no error, commit the changes.

conn.commit();

}catch(SQLException se){
    // If there is any error.

conn.rollback(savepoint1);
}

```

## Batch Processing in JDBC

Instead of executing a single query, we can execute a batch (group) of queries. It makes the performance fast.

### *Methods of Statement interface*

The required methods for batch processing are given below:

Method	Description
<code>void addBatch(String query)</code>	It adds query into batch.
<code>int[] executeBatch()</code>	It executes the batch of queries.
<code>clearBatch()</code>	Clears the batch

```

Statement stmt=con.createStatement();
stmt.addBatch("insert into user values(190,'abhi',40000)");
stmt.addBatch("insert into user values(191,'umesh',50000)");
int[] count = stmt.executeBatch();
// u can get number of sql stmt that was executed by count[] array.

```

## MetaData

DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, name of total number of tables, name of total number of views etc.

## methods of DatabaseMetaData interface

- **String getDriverName():** it returns the name of the JDBC driver.
- **String getDriverVersion():** it returns the version number of the JDBC driver.
- **String getUserName():** it returns the username of the database.
- **String getDatabaseProductName():** it returns the product name of the database.
- **String getDatabaseProductVersion():** it returns the product version of the database.
- **String getSchemas()**
- **String getPrimaryKeys()**
- **String getProcedures()**
- **String getTables()**

```
DatabaseMetaData dbmd=con.getMetaData();
System.out.println("Driver Name: "+dbmd.getDriverName());
System.out.println("Driver Version: "+dbmd.getDriverVersion());
System.out.println("UserName: "+dbmd.getUserName());
System.out.println("Database Product Name: "+dbmd.getDatabaseProductName());
System.out.println("Database Product
```

## ResultSetMetaData Interface

The metadata means data about data i.e. we can get further information from the data.

If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

## methods of ResultSetMetaData interface

Method	Description
public int getColumnCount()	it returns the total number of columns in the ResultSet object.
public String getColumnName(int index)	it returns the column name of the specified column index.
public String getColumnTypeName(int	it returns the column type name for the specified index.

index)	
public String getTableName(int index)	it returns the table name for the specified column index.

## Exceptions

JDBC methods throws 3 types of exceptions

1. **SQLException**
  2. **SQLWarnings**
  3. **DataTruncation.**
1. **SQLException Methods** - An SQLException can occur both in the driver and the database.

<b>Method</b>	<b>Description</b>
getErrorCode( )	Gets the error number associated with the exception.
getMessage( )	Gets the JDBC driver's error message for an error, handled by the driver or gets the Oracle error number and message for a database error.
getNextException( )	Gets the next Exception object in the exception chain.

2. **SQLWarnings** - An exception that provides information on database access warnings.

getWarnings()-Retrieves the first warning reported by calls on this Connection object.

getNextWarning()-Retrieves subsequent warnings.

3. **DataTruncation** – this exception is thrown when a data values is unexpectedly truncated