UNIT-II

Inheritance

Inheritance is one of the properties of object oriented programming. Inheritance means acquiring the properties of parent class to child class. A class that is inherited is called super class. The class that does the inheriting is called sub class. Inheritance is achieved by the keywords extends or implements.

Eg

```
class A {
int i, j;
void showij() {
 S.o.pln("i=" + i + " " + "j=" + j);
 }
}
class B extends A {
 int k;
 void showk() {
 S.o.pln("k=" + k);
 }
 void sum() {
 S.o.pln("i+j+k=" + (i+j+k));
 }
}
```

```
class Sample{
 psvm(){
 A a=new A();
 B b=new B();
 b.i=10;
 b.j=20;
 b.showij();
 b.showk();
 b.sum();
 a.showij();
 }
}
```

Member Access and Inheritance

Although a subclass includes all the members of its superclass, it cannot access those members of the superclass that have been declared as private.

Eg

```
class A{
 int i;
 private int j;
 void setij(int x, int y){
  i=x;
  j=y;
 }
}
class B extends A {
 int total;
 void sum(){
 total=i+j; //Error, j is not accessible here
 }
}
```

```
class Demo{
 psvm(){
 B b = new B();
 b.setij(10,20);
 b.sum();
 S.o.println("Total is "+ b.total);
 }
}
```

Eg2

```
class Box {
 double width;
 double height;
 double depth;
 Box(double w, double h, double d) {
  width = w;
  height = h;
  depth = d;
 }
 Box() {
  width = 1;
  height = 1;
  depth = 1;
 }
 Box(double len) {
  width = height = depth = len;
 }
 double Volume()
 {
  return width * height * depth;
 }
```

```
class BoxWeight extends Box{
 double weight;
BoxWeight(double w, double h, double d, double m){
 w;dth=w;
 height=h;
 depth=d;
 weight=m;
}
class BoxDemo{
 psvm(){
 BoxWeight mybox1=new BoxWeight(10,20,25,33.3);
 BoxWeight mybox2=new BoxWeight(1,2,3,4.4);
 double vol=mybox1.volume();
 S.o.pln("Volume is "+vol);
 S.o.pln("Volume is "+mybox2.volume());
  }
}
```

The super keyword

Whenever a subclass needs to refer to its immediate superclass, super keyword is used. super has generally used to call,

1. Super class constructor (Hidden by subclass constructor)
2. Super class method (Hidden by subclass method)
3. Super class reference variable (Hidden by subclass reference variable)

```
class A {
  A() {
  S.o.pln("A's Constructor");
  }
  A(int x, int y) {
  int z = x+y;
  S.o.pln("z = "+z); }
  }
class B extends A {
   B() {
   S.o.pln("B's constructor");
   }
  B(int c, int d) {
  super(c,d);
  int e = c*d;
  S.o.pln("e = "+e);
  }
  }
class Demo {
  psvm() {
  B b = new B();
  B b1 = new B(10,20);
```

o/p:

```
A's Constructor
B's Constructor
z = 30
e = 200
```

[AC

B·

In the above, both the sub class constructor hide the super class constructors. If you want to access the super class constructors using child class objects, super keyword is used.

By default, whether we use super keyword or not, java implicitly calls the super class constructor first and child class constructor next, if both are non-parametrized constructors.

But when both the super class as well as child class constructors are parametrized constructor, then super keyword should be used in child class constructor as first line to call the hidden super class constructor.

Eg:

```
class A{
A(){
S.o.pln("constructor A");}
}
class B extends A{
B(){
S.o.pln("Constructor B");
 }
}
class C extends B{
C(){
S.o.pln("Constructor C");
 }
}
```

```
class Demo{
  psvm(){
  C c= new C();
  }
}
```

o/p
```
Constructor A
Constructor B
Constructor C
```

Method Overriding

If two or more methods, with same name, same return type and same parameters exist, but available in parent and child class, then the child class method is said to be overriding (hiding) the super class method. This concept is called method overriding. through method overriding, Run-time polymorphism is achieved.

Eg
```
class A{
  void show(){
  S.o.pln("Inside A");
  }
  }
  class B extends A{
  void show(){
  S.o.pln("Inside B");
  }
}
```

```
class Demo2
 psvm(){
 B b=new B();
 b.show();
 }
}
```

o/p

Inside B.

In the above program, the child class show() hides (overrides) super class show().

If you wish to access the superclass version of an overridden method using child class object, you can do so by using super keyword.

2nd use of super keyword is used to call superclass overridden method. In the above program, change as below

```
class B extends A {
  void show(){
   super.show();
   S.o.println("Inside B");
  }
}
class Demo2{
psvm(){
B b=new B();
b.show();
}
}
```

o/p

Inside A
Inside B

Super keyword to access superclass

reference variable

The 2nd use of super keyword to call the superclass instance variable, if the child class instance variable is also similar.

Eg

```
class A {
  int i;
}
class B extends A {
  int i;
  B(int a, int b) {
    super.i = a;
    i = b;
  }
  void show() {
    S.o.pln("i in superclass "+super.i);
    S.o.pln("i in sub class" + i);
  }
}
class Demo {
  psvm() {
    B b = new B(1, 2);
    b.show();
  }
}
```

o/p

```
i in superclass 1
i in sub class 2
```

Dynamic Method Dispatch

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at runtime, rather than compile time. It is very important because this is how java implements run-time polymorphism. It works under the rule:

A super class reference variable can refer to a sub class object.

Eg

```
class A{
 Void callme(){
 S.o.pln("A's callme method");
 }
}
class B extends A{
 Void callme(){
 S.o.pln("B's callme method");
 }
}
class c extends B{
 Void callme(){
 S.o.pln("c's callme method");
 }
}
class Demo{
 psvm(){
 A a= new A(); // object a
 B b= new B(); // object b
 C c= new c(); // object c
```

```
r.callme();
r=b;
r.callme();   [or]
r=c;
r.callme();
}
}
```

```
A r;  // Reference Variable of Super class A
r=new A();
r.callme();
r=new B();
r.callme();
r=new C();
r.callme();
}
}
```

Abstract class

There are situations in which you will want to define a super class that declares a method without providing complete implementation for it. A subclass can inherit those classes and fill the details for the partially implemented method.

1. An abstract class has a keyword abstract.
2. It may have constructor, instance variables, concrete methods. But it should definitely contain one abstract method.
3. An abstract method is a method, which provides only declaration and not definition.
   Eg: `void show();`
   `void add();`
4. An abstract class cannot be instantiated.
   [Cannot create object for an abstract class]
5. The sub class which extends the abstract class should definitely provide definition for the super class abstract methods.

6. Any subclass which extends an abstract class, but does not provide implementation for the super class abstract methods, should declare itself as abstract.

```
abstract class A {
abstract void callme();
void callmetoo() {
S.o.println("Concrete method");
}
}
class B extends A {
void callme() {
S.o.println("Callme method implemented");
}
}
class Demo {
psvm() {
B b = new B();
b.callme();
b.callmetoo();
}
}
```

Indirect implementation

```
abstract class A {
abstract void show();
}
abstract class B extends A {          // Doesn't provide implementation for abstract(), hence class as abstract
void display() {
S.o.println("Concrete method"); }
}
```

```
class c extends B{
void show(){
System.out.println("Abstract() implementation");
}
}
class Sample{
psvm(){
C c = new c();
c.show();
c.display();
}
}
```

The final keyword

The keyword final has three uses.

1. A variable which is declared with keyword final acts as constant.
2. final keyword is used to prevent overriding
3. final keyword is used to prevent inheritance

Eg 1:

```
final int FILE_NEW = 2;
final int FILE_OPEN = 1;
final double PI = 3.14;
```

The variables which are declared as final cannot be modified. Variables declared as final do not occupy separate memory space for each object created. All the objects share the common memory space for constants.

Eg2
```
class A {
final void meth() {
S.o.pln("final method");
}
}
class B extends A {
  void meth() { // Error, Can't override
  S.o.pln("final method in B");
  }
```

Eg3
```
final class A {
  //...
  }
class B extends A { // Error, Can't inherit A
  //...
  }
```

The object class

1. It is a special class defined by Java.
2. All the other classes are subclasses of Object
3. (i.e) Object class is the super class of all other classes.
4. It means, a reference variable of type Object can refer to an object of any other cl
5. Since arrays are implemented as classes, a variable of type Object can refer to any array

Object class methods are :

1. Object clone() → creates a new object that is the same as the object being clo
2. boolean Equals (Object object) → determines whether one object is Equal to another
3. Void finalize() → called for collecting an unused object [Garbage collectio
4. class getClass() → Obtains the class of an object at runtime.
5. Void notify() → resumes execution of a thread waiting.
6. Void notifyAll() → resumes execution of all threads waiting.
7. String toString() → returns a string that describes the object.
8. Void wait()
   Void wait(long milliseconds)
   Void wait(long milliseconds, int nanoseconds)

   → waits on another thread of execution.

The above getClass(), notify(), notifyAll(), wait() are declared as final.

Eg/

```
class A {
Void show() {
S.o.pln("show");
}
}
```

```
class Demo {
psvm() {
Object o = new A();
o.show();
}
}
```

Object cloning

The object cloning is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.

The java.lang.Cloneable interface must be implemented by the class, whose object clone we want to create. The clone() method is defined in the Object class.

Syntax

```
Object clone() throws CloneNotSupportedException
```

Eg

```
class Student implements Cloneable{
 int rollno;
 String name;
 Student(int rno, String sname){
  rollno = rno;
  name = sname;
 }
public Object clone() throws CloneNotSupportedExcep
{
 return super.clone();
}
public static void main(String args[]){
try{
Student s1 = new Student(101, "Raj");
Student s2 = (Student)s1.clone();
S.o.pln(s1.rollno+" "+s1.name);
S.o.pln(s2.rollno+" "+s2.name);
}
```

```
catch (Exception e)
  {
   S.o.pln(e);
  }
 }
}
```

o/p 101 Raj
101 Raj

The above program should throw CloneNotSupportedException, if we don't implement the Cloneable interface.

The clone object has its own space in the memory, where it copies the content of the original object. Hence, if we change the content of the original object after cloning, the changes does not reflect in the clone object. In the abov program, if we make the change,

```
psvm(){
try {
   Student s1=new Student(101,"Raj");
   Student s2 = (Student) s1.clone();
    s2.rollno = 102;
    s2.name = "Ravi";
   S.o.pln(s1.rollno+" "+s1.name);
   S.o.pln(s2.rollno+" "+s2.name);
  }
 catch()
  {
  }
}
```

o/p 101 Raj → S1 (original object)
102 Ravi → S2 (clone object)

Interface

Unlike abstract class (0-100% abstraction) Interface achieves 100% abstraction. An interface is syntactically similar to classes. But they lack instance variables and their methods are declared without any body. Any number of classes can inherit an interface using the keyword implements. Also one class can implement any number of interfaces.

To implement an interface, a class must create the complete set of methods defined by the interface.

Eg

```
interface Callback {
  void callback(int param);
}
class Client implements Callback {
public void callback(int p) {
s.o.pln(" p is "+p);
}
  void meth() {
  s.o.pln("Concrete method");
  }
}
class Test{
  psvm(){
Client c=new Client();
c.callback(10);
c.meth();
}
}
```

[or]

```
psvm(){
Callback c= new Client();
c.callback(10);
}
```

Partial implementation

If a class inherits an interface, but does not fully implement the methods defined by that interface, then that class should be declared abstract.

Eg

```
abstract class Incomplete implements Callback{
 int a,b;
 void show(){
 System.out.println(a+" "+b);
 }
```

Interfaces can be extended

```
interface A{
 void meth1();
}
interface B extends A{
 void meth2();
}
class C implements B{
 public void meth1(){
  System.out.println("Interface A's method");
 }
 public void meth2(){
  System.out.println("Interface B's method");
 }
}
```

```
class Sample {
 psvm() {
  C c = new CC();
  c.meth1();
  c.meth2();
 }
}
```

Inner classes

Inner class means one class which is a member of another class. Inner classes are of three types depending on how and where you define them. They are:

1) Inner class
2) Method-local inner class
3) Anonymous inner class.

Inner class

Inner class means one class which is available directly inside an other class.

Eg:

```
class Outer {
  int num;
  class Inner {
    public void print() {
    S.o.pln("Inner class");
    }
  }
  void display() {
    Inner in = new Inner();
    in.print();
  }
```

```
public class Demo{
  parm(){
  Outer o = new Outer();
   o.display(); // Calling the inner class method
                   indirectly.
  o.new Inner.print(); // Calling the inner class
                         method directly.
  }
}
```

Method-local Inner class

In java, a class can be written within a method of other class. Like local variables, the scope of the inner class is restricted with the method.

A method-local inner class can be instantiated only within the method, where the inner class is defined.

Eg 2

```
class Outer{
Void outerMethod(){
 S.o.pln("Inside outer method");
 class Inner{
  Void innerMethod(){
  S.o.pln("Inside inner method");
  }
 }
 Inner y = New Inner();
  y.innerMethod();
}
}
```

```
class Demo{
  psvm(){
  Outer x = new Outer();
  x.outerMethod();
  }
}
```

Anonymous Inner class

A class that has no name is called anonymous inner class. It should be used, if you have to override the method of class or interface.
Anonymous inner class can be created in two ways

1. class (may be abstract or concrete)
2. Interface

Eg:

```
abstract class Person{
  abstract void eat();
}
class Demo{
  psvm(){
  Person p = new Person(){
  void eat(){
    S.o.pln("Method local inner class");
  }
  };
  p.eat();
  }
}
```

Eg:

```
interface Eatable{
 void eat();
}
class Demo{
psvm(){
Eatable e = new Eatable{
Public Void eat(){
 S.o.pln("Nice fruit");
 }
};
e.eat();
 }
}
```

ArrayList class

Java ArrayList class uses a dynamic array for storing the elements.

1. ArrayList stores Objects of any types as elements.
2. It can contain duplicate elements.
3. It maintains insertion order.
4. It allows random access, because array works at the index basis.

Constructor

ArrayList() – It is used to build an empty array list.

ArrayList(Collection c) – It is used to build an array list that is initialized with the elements of the collection

ArrayList(int capacity) – It is used to build an array list that has the specified initial capacity

Methods of ArrayList

Void add(int index, Object element) – used to insert the specified element at the specified position (index) in a list.

boolean addAll(Collection c) – used to append all the elements in the specified collection to the end of this list.

Void clear() – used to remove all of the elements from this list.

boolean add(Object o) – used to append the specified element to the end of the list.

boolean addAll(int index, Collection c) – used to insert all of the elements of the specified collection into this list, starting at the specified position.

Creating an ArrayList

ArrayList<class type> al = new ArrayList<class type>();

Two ways to iterate the elements of ArrayList

1. By Iterator interface
2. By for-each loop

Eg:

```
import java.util.*;
class Demo{
 psvm(){
ArrayList<String> l=new ArrayList<String>();
 l.add("Ravi");
 l.add("vijay");
 l.add("Ravi");
 l.add("Ajay");
Iterator itr = l.iterator();
while(itr.hasNext()){
 So.pln(itr.next());
  }
 }
}
```

o/p
```
Ravi
vijay
Ravi
Ajay
```

The above Iterator code can be replaced by enhanced for Loop as follows:

```
l.add("Ravi");
l.add("vijay");
l.add("Ravi");
l.add("Ajay");
for(String obj : l)
  S.o.pln(obj);
```

o/p
```
Ravi
vijay
Ravi
Ajay
```

User-defined class Objects in ArrayList

Ex:

```
class Student{
  int rollno;
  String sname;
  Student(int rno, String name){
    rollno = rno;
    sname = name;
  }
}

import java.util.*;
public class Demo{
  public(){
  Student s1 = new Student(101, "Raj");
  Student s2 = new Student(102, "Rani");
ArrayList<Student> al = new ArrayList<Student>();
al.add(s1);
al.add(s2);
Iterator itr = al.iterator();
while(itr.hasNext()){
Student st = (Student)itr.next();
S.o.println(st.rollno+" "+st.sname);
  }
}
}
```

o/p

```
101 Raj
102 Rani
```

Eg3

```
ArrayList<String> al = new ArrayList<String>();
al.add("Ravi");
al.add("Vijay");
al.add("Ajay");
ArrayList<String> al1 = new ArrayList<String>();
al1.add("Kumar");
al1.add("Raj");
al.add(al1);
Iterator itr = al.iterator();
while(itr.hasNext()){
S.o.println(itr.next());
}
}
}
```

o/p
```
Ravi
Vijay
Ajay
Kumar
Raj
```

Eg4

```
ArrayList<Integer> al = new ArrayList<Integer>();
al.add(1);
al.add(2);
al.add(1,3); // adding element 3 to index 1.
S.o.println("The size is:"+al.size());
S.o.println("The contents are:"+al);
al.removeAll();
S.o.println("The size is:"+al.size());
S.o.println("The contents are:"+al);
```

o/p
```
1
3
2
The size is : 3
The contents are : [1 3 2]
The size is : 0
The contents are :
```

Eg:

```
ArrayList<String> al = new ArrayList<String>();
al.add("Ravi");
al.add("vijay");
ArrayList<String> al1 = new ArrayList<String>();
al1.add("Ravi");
al1.add("Raj");
al.removeAll(al1);
Iterator itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
```

O/P Ravi

Java - Strings Class

Strings, which are widely used in Java programming, are a sequence of characters. In Java programming language, strings are treated as objects. The Java platform provides the String class to create and manipulate strings. When you create a String object, you are creating a string that cannot be changed. That is, once a String object has been created, you cannot change the characters that comprise that string.

The String Constructors

The String class supports several constructors. To create an empty String. you call the default constructor. For example,
String s = new String();
will create an instance of String with no characters in it.

To create a String initialized by an array
of characters, use the constructor shown here:
String(char chars[ ])
Here is an example:
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
This constructor initializes s with the string "abc".

You can specify a subrange of a character array as an initializer using the following constructor:
String(char chars[ ], int startIndex, int numChars)
Here, startIndex specifies the index at which the subrange begins, and numChars specifies the number of characters to use. Here is an example:
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
This initializes s with the characters cde.

You can construct a String object that contains the same character sequence as another String object using this constructor:
String(String strObj)
Here, strObj is a String object. Consider this example:

```
char c[] = {'J', 'a', 'v', 'a'};
String s1 = new String(c);
String s2 = new String(s1);
System.out.println(s1);
System.out.println(s2);
```

The output from this program is as follows:
Java
Java

the String class provides
constructors that initialize a string when given a byte array. Their forms are shown here:
String(byte asciiChars[ ])

String(byte asciiChars[ ], int startIndex, int numChars)
Here, asciiChars specifies the array of bytes. The second form allows you to specify a subrange. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform. The following program illustrates these constructors:

```
// Construct string from subset of char array.
class SubStringCons {
public static void main(String args[]) {
byte ascii[] = {65, 66, 67, 68, 69, 70 };
String s1 = new String(ascii);
System.out.println(s1);
String s2 = new String(ascii, 2, 3);
System.out.println(s2);
}
}
```

This program generates the following output:

```
ABCDEF
CDE
```

The contents of the array are copied whenever you create a String object from an array. If you modify the contents of the array after you have created the string, the String will be unchanged.

String Length

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

```
int length( )
```

The following fragment prints "3", since there are three characters in the string s:

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

String Literals

The earlier examples showed how to explicitly create a String instance from an array of characters by using the new operator. However, there is an easier way to do this using a string literal.

```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);
String s2 = "abc"; // use string literal
```

String Concatenation

The + operator, which concatenates two strings, producing a String object as the result. This allows you to chain together a series of + operations. For example, the following fragment concatenates three strings:

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

This displays the string "He is 9 years old."

```
int age = 9;
```

```
String s = "He is " + age + " years old.";
System.out.println(s);

String s = "four: " + 2 + 2;
System.out.println(s);
```

This fragment displays
four: 22

Character Extraction

charAt( )

To extract a single character from a String, you can refer directly to an individual character via the charAt( ) method. It has this general form:

```
char charAt(int where)
```

Here, where is the index of the character that you want to obtain. The value of where must be nonnegative and specify a location within the string. charAt( ) returns the character at the specified location. For example,

```
char ch;
ch = "abc".charAt(1);
```

assigns the value "b" to ch.

getChars( )

If you need to extract more than one character at a time, you can use the getChars( ) method. It has this general form:

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

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

```
String s = "This is a demo of the getChars method.";
int start = 10;
int end = 14;
char buf[] = new char[end - start];
s.getChars(start, end, buf, 0);
System.out.println(buf);
```

Demo

toCharArray( )

If you want to convert all the characters in a String object into a character array, the easiest way is to call toCharArray( ). It returns an array of characters for the entire string. It has this general form:

```
char[ ] toCharArray( )
```

String Comparison

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

equals( ) and equalsIgnoreCase( )

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

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

Here, str is the String object being compared with the invoking String object. It returns

true if the strings contain the same characters in the same order, and false otherwise. The comparison is case-sensitive.
To perform a comparison that ignores case differences, call equalsIgnoreCase( ). When it compares two strings, it considers A-Z to be the same as a-z. It has this general form:

boolean equalsIgnoreCase(String str)
Here, str is the String object being compared with the invoking String object. It, too, returns true if the strings contain the same characters in the same order, and false otherwise.
Here is an example that demonstrates equals( ) and equalsIgnoreCase( ):

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = "Hello";
String s3 = "Good-bye";
String s4 = "HELLO";
System.out.println(s1 + " equals " + s2 + " -> " +
s1.equals(s2));
System.out.println(s1 + " equals " + s3 + " -> " +
s1.equals(s3));
System.out.println(s1 + " equals " + s4 + " -> " +
s1.equals(s4));
System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
s1.equalsIgnoreCase(s4));
}
}
```

The output from the program is shown here:

```
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

startsWith( ) and endsWith( )

The startsWith( ) method determines whether a given String begins with a specified string. Conversely, endsWith( ) determines whether the String in question ends with a specified string. They have the following general forms:

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

Here, str is the String being tested. If the string matches, true is returned. Otherwise, false is returned. For example,

```
"Foobar".endsWith("bar")
```

and

```
"Foobar".startsWith("Foo")
```

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

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

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

```
"Foobar".startsWith("bar", 3)
```

returns true.

equals( ) Versus ==

It is important to understand that the equals( ) method and the == operator perform two different operations. As just explained, the equals( ) method compares the characters inside a String object. The == operator compares two object references to see whether they refer to the same instance.

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

The variable s1 refers to the String instance created by "Hello". The object referred to by s2 is created with s1 as an initializer. Thus, the contents of the two String objects are identical, but they are distinct objects. This means that s1 and s2 do not refer to the same objects and are, therefore, not ==, as is shown here by the output of the preceding example:

```
Hello equals Hello -> true
Hello == Hello -> false
```

Searching Strings

The String class provides two methods that allow you to search a string for a specified character or substring:

- indexOf( ) Searches for the first occurrence of a character or substring.
- lastIndexOf( ) Searches for the last occurrence of a character or substring.

To search for the first occurrence of a character, use

```
int indexOf(int ch)
```

To search for the last occurrence of a character, use

```
int lastIndexOf(int ch)
```

Here, ch is the character being sought.

To search for the first or last occurrence of a substring, use

```
int indexOf(String str)
int lastIndexOf(String str)
```

Here, str specifies the substring.

You can specify a starting point for the search using these forms:

```
int indexOf(int ch, int startIndex)
int lastIndexOf(int ch, int startIndex)
int indexOf(String str, int startIndex)
Int lastIndexOf(String str, int startIndex)
```

Here, startIndex specifies the index at which point the search begins. For indexOf( ), the search runs from startIndex to the end of the string. For lastIndexOf( ), the search runs from startIndex to zero.

The following example shows how to use the various index methods to search inside of Strings:

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
public static void main(String args[]) {
String s = "Now is the time for all good men " +
```

```
"to come to the aid of their country.";
System.out.println(s);
System.out.println("indexOf(t) = " +
s.indexOf('t'));
System.out.println("lastIndexOf(t) = " +
s.lastIndexOf('t'));
System.out.println("indexOf(the) = " +
s.indexOf("the"));
System.out.println("lastIndexOf(the) = " +
s.lastIndexOf("the"));
System.out.println("indexOf(t, 10) = " +
s.indexOf('t', 10));
System.out.println("lastIndexOf(t, 60) = " +
s.lastIndexOf('t', 60));
System.out.println("indexOf(the, 10) = " +
s.indexOf("the", 10));
System.out.println("lastIndexOf(the, 60) = " +
s.lastIndexOf("the", 60));
}
}
```

```
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

Modifying a String

substring( )

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

String substring(int startIndex)

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

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

String substring(int startIndex, int endIndex)

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

concat( )

You can concatenate two strings using concat( ), shown here:

String concat(String str)

This method creates a new object that contains the invoking string with the contents of str appended to the end. concat( ) performs the same function as +. For example,

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

puts the string "onetwo" into s2. It generates the same result as the following sequence:

```
String s1 = "one";
String s2 = s1 + "two";
```

replace( )

The replace( ) method has two forms. The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char original, char replacement)

Here, original specifies the character to be replaced by the character specified by replacement. The resulting string is returned. For example,

String s = "Hello".replace('l', 'w');

puts the string "Hewwo" into s.

trim( )

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

String trim( )

Here is an example:

String s = " Hello World ".trim();

This puts the string "Hello World" into s.

Changing the Case of Characters Within a String

The method toLowerCase( ) converts all the characters in a string from uppercase to lowercase. The toUpperCase( ) method converts all the characters in a string from lowercase to uppercase. Nonalphabetical characters, such as digits, are unaffected. Here are the general forms of these methods:

String toLowerCase( )

String toUpperCase()

```
String s = "This is a test.";
System.out.println("Original: " + s);
String upper = s.toUpperCase();
String lower = s.toLowerCase();
System.out.println("Uppercase: " + upper);
System.out.println("Lowercase: " + lower);
}
}
```

The output produced by the program is shown here:

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