Chapter 3

Java Object-Oriented Programming for Enterprise Applications

- An Overview of Enterprise Applications
- Object-Oriented Features: class members and methods, access control, overloading, constructors, destructors, finalizers, inheritance, abstract class, and interface
- Using existing classes
- Building User Defined Classes: members, methods, fields
- Object reference
- Static Members
- Properties of Interfaces and Inner classes
- Inner classes
- Using Packages
- Java Language Package
- Java Utilities Package
- Inheritance: extending classes, overriding, interfaces, and finalizer methods, overloading methods, subclasses
- Examples and Case Studies

An Overview of Enterprise Applications

Information gathering and processing is the essence of the modern enterprises, which deal with their global customers, suppliers, shippers, and their own internal corporate divisions.

Earlier Three-Tier Architecture:

- Presentation Tier (clients): dumb terminals -> sophisticated GUI
- Business logic (middle tier): Transaction-Processing Monitor; COBOL or PL/I
- Resource Tier (Databases and others): some sort of databases such as DB2, ORACLE, Microsoft SQL, etc

Modern Three-Tier Architecture (Distributed-component systems):

- Presentation Tier (clients): sophisticated GUI with remote proxies that communicate requests to the distributed components (Java Remote Method Invocation, CORBA - Component Object Request Broker Architecture, and DCOM) over the network
- Business logic (middle tier): Distributed objects on the middle tier
- Resource Tier (Databases and others): some sort of databases such as IBM DB2, ORACLE, Microsoft SQL relational databases

N-Tier Computing Model that currently used in enterprise and Web applications allow a mixture of Computer Hardware and/or Software layers to provide a modular collection of Information Services. This model is often referred to as:

- Thin-client
- Browser-based
- Network-centric
- Multi-tiered computing

Advantages of this model may include

- Component-based Clients, Interfaces, Agents, Transactions, Middleware, and Servers flexibly arranged into a variety of configurations.
- Programs partitioned into Tiers allow each layer or component part to be developed, managed, deployed and enhanced independently.

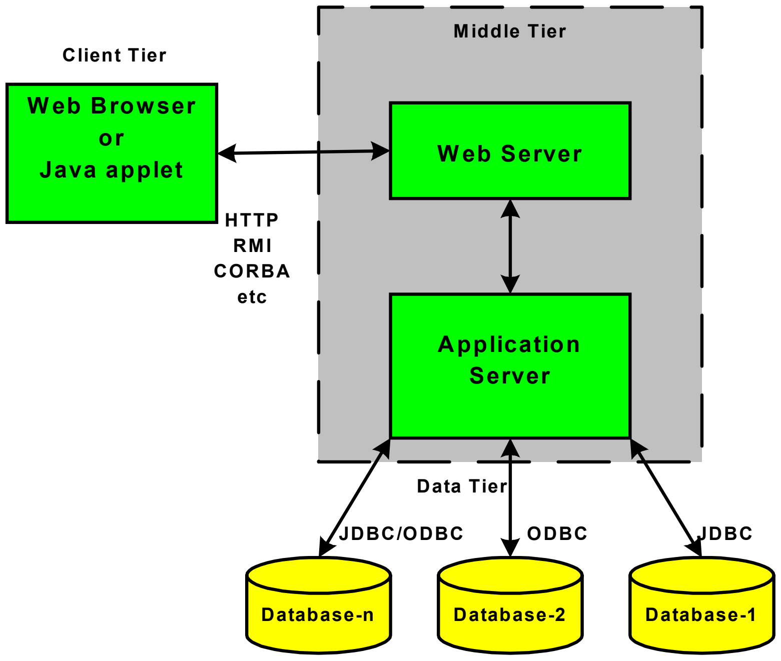


Figure 3.1 N-Tier Distributed Computing Model

Java Object-Oriented Features

In modern business systems, object-oriented languages are used to

- Improve development of GUIs
- Simplify access to information
- Encapsulate the business logic

Object-oriented principles of encapsulation, inheritance, and polymorphism will allow the software that models the business to be encapsulated into business objects, and become flexible, extensible, and reusable.

- In object-oriented programming, classes are templates for defining properties and behavior functions within a program.
- Each instance of a class represents an object
- Inheritance is a form of software reuse by deriving a more specialized new class from previously defined superclass.
- Classes (subclasses and superclasses) are the fundamental building blocks of any Java program.
- Each Java class must be derived from a superclass. This establishes an "is a" relationship.
- Java supports only single inheritance. A new Java class (subclass or child class) is derived from only one superclass or parent class. Additional instance variables and instance methods of its own are also added.
- Java classes are split into three basic components:
    - a class declaration
    - data
    - methods
- The **extends** keyword is used to derived a new class.

The Object class

- In Java, the **Object** class is the root superclass of all Java classes. All other classes inherit methods from this Object class.
- The Object class does not contain any data

Classes

- An **abstract** may contain non-abstract methods. Abstract classes are classes that must never be instantiated in a hierarchy. An abstract class is normally created with features and behavior common to be inherited by their subclasses.

- An **interface** is similar to a class. It is created to offer **abstract** method definitions and/or **static final** data (if nay). All methods in an interface must be abstract. To create a new class that uses interface, the **implements** keyword is used.
- A **class** may only inherit from one abstract super class, but it can implement any number of interfaces.
- The public class is saved as a disk file with the same name as that of the public class that it contains
- A **public** keyword precedes the class keyword will make that class accessible to all other classes; omit the public keyword to limit the access to within the package which contains the class.

Class Members

- A Java **class** consists of public, protected and/or private methods and data variables.
- Methods are normally declared public and instance variables are usually declared private.
- All **public** methods and data variables can be considered as a public interface of a class. It means that public class members have global visibility and can be accessed by any objects and offers services that the class provides to all other classes
- Java class **private** methods and private variables are only accessible to instances within their own class. A private method cannot be called from outside its class. The only way to access a private variable is by calling one of the public class methods.
- The **protected** class members (methods and variables) are accessible only to methods of classes in the same package and to all subclasses (derived classes) of that class (parent class).

- A **static** variable will be created only one copy to be shared by all objects of a class. A beneficial side effect is that the static class members exist and accessible even when no objects of that class exist. A user can access this static class member (variable) by prefixing its class name and the dot operator. For example:
    - public static namedVariable // accessible within the class
    - private static namedVariable // accessible to outside
    - System.out,println() // Can be used without instantiating a System object
    - System.out.print() // Can be used without instantiating a System object
- The **final** modifier can be used to specify that a variable has a constant value; or that a method cannot be over ridden in a subclass. For example:
    - final public int hourlyRate = 6.5;
- The **synchronized** modifier is used to specify that a method for a safer thread in a multithreading environment
- The **native** modifier is used to identify methods that have native implementations in C/C++
- Constructors
    - A constructor is a special method with the same name as that of the class name.
    - A class may have more than one constructor.
    - A call to the superclass constructor is via the **super** reference.
- The **finalizer** method or destructor
    - Each class can have a finalizer method that is declared with the name **finalize**.
    - The finalizer method should always be defined as **protected** so that it can only be accessed by it subclasses only.
    - The superclass finalizer should be used (if any) so that it can ensure all parts of an object are properly garbage collected through

the garbage collection thread (a low-priority thread) and returning resources to the system.

- An example:

```
protected void finalize()
 {
 // statements
super.finalize();
 }
```

Java Class Declaration

- To declare a Java class, we will need the following components:
    - Optional access modifiers: **public, protected** or **private**
    - The **class** keyword
    - A valid class name or identifier
    - Optional keyword: **extends** and **implements**
    - Class members: methods, and data fields (variables)

Example 3-1: Subclass "is a" super class relationship

Super class	Subclass
Vehicle	Car, trucks
Student	UndergraduateStudent, GraduateStudent
Shape	Circle, Triangle, Rectangle
Loan	CarLoan, HomeImrovementLoan, MorgageLoan
Account	CheckingAccount, SavingAccount
Employee	Staff, Engineer, StaffEngineer, Consultant, Contractor

Some Notes on Methods

Protected Method

When a method should be visible to subclass but not to the rest of the system

Abstract Method

A method `abstract` (in an abstract class) when that method must be implemented by all subclasses

Final Method

Declare a method `final` when that method should be inherited but not overridden by any subclass

Super Method

When overriding a super-class method, use that super-class method (`super`)

main Method

main()

JVM sends the message main() to a program object

Execution of the method main()

Declaring Java Class Methods

- May have the following access modifiers: public, protected, private, or package
- A **static** method can be declared to be used for the class-wide information which can be used by the class rather than an instance of a class
- The **public static void main(String [ ] args)** method is the entry point of a standalone Java program. The main must be declared static so that it can be invoked even when on object of the class has been created.
- Argument list - a comma-separated list of parameter declarations; for main() it must be an array of **String** array with **args** as the named object
- A return type:

- void - for the method that does not return any value, such as the main which does not return a value to the Java Virtual Machine (JVM)
- A primitive type - boolean, byte, char, short, int, long, float, and double
- Object type - return a reference to an object of any class or interface
- Array type - return a reference to a Java array

Example 3-2: An example of declaring a new class

```
public class Employee
 {
 // Data Members
public String firstName;
public String mInitialName;
public String lastName;
public int employeeID;
protected float salary;

// Constructors
public Employee (String firstName, String mInitialName,
                 String lastName, int empID, float salary)
  {
   this.firstName = firstName;
   this.mInitialName = mInitialName;
   this.lastName  = lastName;
   this.employeeID = empID;
   this.salary = salary;
  }

// Methods
// public float  increSalary(float percent);
// public float  getSlary();
// public String getLastName();
// public String getFirstName();
// public String getmInitialName();
// public boolean validateEmpID();
}
```

Object Instance

- Once a class is declared, a programmer can create or instantiate objects of that class:

```
        className ObjectReference = new className;
```

- In Java, the **new** is the only operator used to invoke a constructor to instantiate a single instance of a named class and returns a reference to that object.
- An object can be referenced by multiple object variables. If the class **shippedItems** is already declared, the shippedItem object can be created and shipToMexico and airToMexico are called object references to the shippedItems object:

```
shippedItems shipToMexico  = new shippedItems();
shippedItems airToMexico = shipToMexico;
```

- The keywords for object reference are:
  - **this** - an object reference to the current object itself for each object
  - **super** - a reference to the object of its parent class
- To see if one particular object is an instance of a specific class or an implementation of a specified **interface**, the **instanceof** operator can be used:

```
if(shipToMexico instanceof shippedItems)
//some statements
```

- The dot (**.**) operator is required to access instance variables within an object
- The dot (.) operator is also used to invoke methods within an object.

Using Java Class Methods

- Passing arguments
  - A primitive type(s): value(s) will be copied
  - Object: object reference will be copied

- Java methods are invoked using the following syntax:

  - The keyword **this** and **super** may be used instead of an object's name

Class Inheritance

- In java, class inheritance is a kind of code reuse which is achieved through **extends** some class and **implements** some interface
- Classes inherits the instance variables and methods of the classes above them in the hierarchy
- A class can extend its inherited characteristics by adding instance variables and methods
- A class can extend its inherited characteristics by overriding inherited methods

Example 3-3: An example of class inheritance.

```
public class Employee extends Object{ };  // Super class
public class FullTimeEmployee extends Employee{ }; // Subclass
public class PartTimeEmployee extends Employee{ }; // Subclass
public class Contractor extends Employee{ }; // Subclass
public class StudentEmployee extends PartTimeEmployee; // Sub -sub class
```

Inner Classes

- Most Java classes are defined at the package level, meaning that each class is a member of a particular package
- Inner classes are classes defined inside other classes for use mainly as event handling purposes.
- Inner classes can be private, protected or public
- Naming convention for inner classes
  - Inner classes with names: OutClassName$InnerClassName.class
  - Anonymous inner classes: OuterClassName$#.class, where # starts from 1
- To access the outer class's this reference, use OuterClassName.this.

Implementing Java Interface

- Interfaces are abstract classes that are left completely unimplemented.
- Java interfaces may include a set of public abstract methods or a set of constants for class implementation.
- A Java interface includes the following definition:
    - public interface InterfaceName
    - a set of **public abstract** methods

Example 3-4: An example of creating Java interfaces.

```
public interface TruckShipping
      {
            public abstract double volume();
            public abstract String getCustomer();
            public abstract String getDistance();
            public abstract String getdestination():
       }

public interface MathConstants
      {
            public static final int PI = 3.14159;
            public static final int E  = 2.73.14159;
            public static final int PI = 3.14159;
      }

public interface EmailMessage
      {
            public abstract String setSender();
            public abstract String addRecipient();
            public abstract String addCC();
            public abstract String addContent();
            public abstract String setSubject();
            public abstract String send();
      }
```

The Language Package

The Java language package is known as java.lang, which make up the core of the Java language. The most important classes contained in this package are:

- public class **Object**
- Data type wrapper classes:
  - public final class **Boolean** extends Object implements Serializable
  - public final class **Character** extends Objects implements Serializable, Comparable
  - public final class **Double** extends Number implements Comparable
  - public final class **Float** extends Number implements Comparable
  - public final class **Integer** extends Number implements Comparable
  - public final class **Long** extends Number implements Comparable
- public final class **Math** extends Object
- public final class **Sting** extends Object implements Serializable, Comparable
- public final class **System** extends Object
- public interface **Runtime** extends RunTimeOperations, Objects, IDLEtity
- Thread classes
  - public class **Thread** extends Object implements Runnable - used to create a thread of execution in program
  - public class **ThreadDeath** extend Errors - used to clean up thread
  - public class **ThreadGroup** extends Object -
  - public class **ThreadLocal** extends Object - ThreadLocal objects are typically private static variables in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).
  - public interface **Runnable**{ void run(); }
- Classes

  - public final class **Class** extends Object implements Serializable - provides such run time information as name, type, and parent for a class
  - public abstract class **ClassLoader** extends Object - for loading a class file
- Exception-handling classes:
  - public class **Throwable** extends Object implements Serializable
  - public class **Exception** extends Throwable
  - public class **Error** extends Throwable
- public abstract class **Process** extends Object

The Utilities Package

The java.utili is the name for referring to this package. It contains the following classes:

- public class **Date** extends Object implements Serializable, Cloneable, Comparable - calendar date and time
- Data structure classes
  - public class **BitSet** extends Object implements Cloneable, Serializable - this class implements a growable vector of bits
  - public abstract class **Dictionary** extends Object - for hashtable (key, value)
  - public class **Hashtable** extends Dictionary implements Map, Cloneable, Serializable
  - public class **Properties** extends Hashtable
  - public class **Vector** extends AbstractList implements Lists, Cloneable, Serializable - for implementing a growable array of objects
  - public class **Stack** extends Vector - Last-In-First-Out
  - public interface Enumeration - for generating a series of elements
- public class **Random** extends Object implements Serializable - for generating a stream of pseudorandom numbers

- public class **StringTokenizer** extends Object implements Enumeration - for breaking a string into tokens
- public interface **Observer** - for observing changes of objects

Example 3-5: Instantiating employee objects

```
// declare paulLin as an employee
Employee paulLin;
// Then instantiate the object paulLin
paulLin = new Employee("Paul", "I", "Lin", 1234, 100.00E3);

// or
// declare and instantiate the object paulLin in one step
Employee paulLin = new Employee("Paul", "I", "Lin", 1234, 100.00E3);
```

Example 3-6: The Object class specification as shown below is defined in the J2SE API.

```
public class Object
 {
 // Constructor
public Object()
// Methods
      // Returns the runtime class of an object. That Class object
      // is the object that is locked by static synchronized methods
      // of the represented class.
public final class getClass()

      // Returns a hash code value for the object. This method is
      // supported for the benefit of hashtables such as those
      // provided by java.util.Hashtable.
public int hashCode()


      // Indicates whether some other object is "equal to" this one.
public boolean equals(Object obj)


      // Creates and returns a copy of this object. The precise
      // meaning of "copy" may depend on the class of the object.
protected Object clone()throws CloneSupportedException


      // Returns a string representation of the object.
public String toString()


      // Wakes up a single thread that is waiting on this object's
      // monitor. If any threads are waiting on this object, one of
      // them is chosen to be awakened. The choice is arbitrary and
      // occurs at the discretion of the implementation. A thread
      // waits on an object's monitor by calling one of the wait
      // methods.
public final void notify()
```

```
      // Wakes up all threads that are waiting on this object's monitor.
      // A thread waits on an object's monitor by calling one of the
      // wait methods.
public final void notifyAll()


      // Causes current thread to wait until either another thread
      // invokes the notify() method or the notifyAll() method for
      // this object, or a specified amount of time has elapsed.
public final void wait(long timeout)throws InterruptException


      // Causes current thread to wait until another thread invokes the
      // notify() method or the notifyAll() method for this object,
      // or some other thread interrupts the current thread, or a
      // certain amount of real time has elapsed.
public final void wait(long timeout,
                       int nanos)
                throws InterruptedException

      // Causes current thread to wait until another thread invokes the
      // notify() method or the notifyAll() method for this object.
      // In other word's this method behaves exactly as if it simply
      // performs the call wait(0).
public final void wait()
                throws InterruptedException

      // Called by the garbage collector on an object when garbage
      // collection determines that there are no more references to
      // the object. A subclass overrides the finalize method to
      // dispose of system resources or to perform other cleanup.
protected void finalize()
                 throws Throwable
```

Example 3-7: The Point class as shown below is defined in the J2SE API specification.

```
public class Point extends Point2D implements Serializable
 {
  // Field Variables
    public int x; // The x coordinate, set to 0 by default.
    public int y; // The y coordinate, set to 0 by default.
  // Constructors
      // Constructs and initializes a point at the origin
      // (0, 0) of the coordinate space.
    public Point()
      { x = 0; y =0; }
      // Constructs and initializes a point at the specified
      // (x, y) location in the coordinate space.
    public Point(int thisX, int thisY)
```

```
      { x = thisX;      y = thisY;  }
      // Constructs and initializes a point with the same
      // location as the specified Point object.
    Point(Point p)
      { x = p.x; y = p.y; }
  // Methods
      // Determines whether two points are equal.
    boolean equals(Object obj)
    Points getLocation()
    double getX()
    double getY()

   // Moves this point to the specificed point in the (x, y) coordinate
   // plane. This method is identical with setLocation(int, int).
    void move(int newX, int newY)
       ( x = newX; y = newY;}
    void setLocation(double newX, double newY)
       { x = (int) newX;   y = (int) newY; }
    void setLocation(int newX, int newY)
       { x = newX;   y =  newY; }
    void setLocation(point P)

   // toString() Returns a string representation of this point and its
   // location in the (x, y) coordinate space. This method is intended
   // to be used only for debugging purposes
    String toString()

   // Translates this point, at location (x, y), by dx along the x axis
   // and dy along the y axis so that it now represents the point
   // (x + dx, y + dy).
    void translate(int dx, int dy)
     { x += dx;  y += dy; }
}
```

Example 3-8: The Random class for generating random numbers and sequences as listed below is extracted from the J2SE API specification:

```
public class Ramdom extends Object implments Serializable
```

```
{
 // Constructors:
 public Random(){ this(System.currentTimeMillis(); }
    // Generates a random number
 public Random(long seed){ setSeed(seed); }
    // Generates a random number using a long seed number

 // Class Methods
 // Generates next random number
 synchronized protected int next(int bits)
   {
     seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
      return (int)(seed >>> (48 - bits));
    }

 // Returns the next pseudorandom, uniformly distributed
 // boolean value from this random number generator's sequence.
 boolean nextBoolean()

 void nextBytes(byte[] bytes)
    Generates random bytes and places them
      into a user-supplied byte array.
 float nextFloat()
    Returns the next pseudorandom, Gaussian ("normally")
    distributed double value with mean 0.0 and standard
    deviation 1.0 from this random number generator's
      sequence.
 double nextGaussian()
    Returns the next pseudorandom, Gaussian ("normally")
      distributed double value with mean 0.0 and standard
      deviation 1.0 from this random number generator's sequence.
    // Returns the next pseudorandom, uniformly distributed
    // int value from this random number generator's sequence.
 public int nextInt() {  return next(32); }
    // Returns a pseudorandom, uniformly distributed int value
    // between 0 (inclusive) and the specified value (exclusive),
    // drawn from this random number generator's sequence.
 public int nextInt(int n)
   {
      if (n<=0)
      throw new IllegalArgumentException("n must be positive");

      if ((n & -n) == n)  // i.e., n is a power of 2
      return (int)((n * (long)next(31)) >> 31);
```

```
        int bits, val;
        do {
            bits = next(31);
            val = bits % n;
           } while(bits - val + (n-1) < 0);
           return val;
       }

   long nextLong()
      Returns the next pseudorandom, uniformly distributed
      long value from this random number generator's sequence.
      // Sets the seed of this random number generator using a
      // single long seed.
   synchronized public void setSeed(long seed)
      {
       this.seed = (seed ^ 0x5DEECE66DL) & ((1L << 48) - 1);
            haveNextNextGaussian = false;
      }
```

Creating and Using Packages

- A **package** is a group of related classes and interfaces
- It is a mechanism for managing a large group of classes and interfaces while avoiding naming conflicts
- To create a package
  - Include package statement at the beginning of a class file, for example:

    ```
    package company.employee;
    ```
  - The package naming mirror the folder or directory structure of the class files:

    ```
    company\employee
    ```
- You can **import** packages to your program. It does not read in, or load the referenced packages.

  ```
  import company.employee;
  import java.awt.*;
  import java.lang.*;
  ```

Guidelines for Creating Packages

- Classes in a package must be declared public

- Use directory structure, for examples
  - java.lang //
  - java.swing //
- Choose a unique Internet domain name as class path name
- Choose a package name and add a package keyword
- Write a package and package name at the first line of the source code file

Examples of Using Java Defined Classes

Example 3-9: Using methods in the System class

The class is defined as follows:

```
      public final class System extends Objects
```

in the package called java.lang.System. It is a final class that cannot be instantiated. The methods and data fields contained in this final class are standard input stream, standard output stream, error output stream; a means of loading files and libraries; and a utility method for quickly copying a portion of an array.

All methods defined in this class can be used directly in your program. For examples:

// To terminate or exit current JVM

```
System.exit(0);
```

// To obtain the system's time

```
long systemTime = System.currentTimeMills();
```

Some class methods are listed below:

```
      // Terminates the current JVM

static void exit(int status)

      // Runs the system garbage collector
static void gc()
```

```
      // Return the current system time in milliseconds
static long currentTimeMills()

      // Copies an array from the specified source array, beginning at
      // the specified position, to the specified position of the
      // destination array.

static void arrayCopy(Object src, int src position, Object dst, int
dst_position, int length)

static String getProperty()
```

Returns system's properties with the following: KEY and VALUE.

Key	**Description of Associated Value**
java.version	Java Runtime Environment version
java.vendor	Java Runtime Environment vendor
java.vendor.url	Java vendor URL
java.home	Java installation directory
java.vm.specification.version	Java Virtual Machine specification version
java.vm.specification.vendor	Java Virtual Machine specification vendor
java.vm.specification.name	Java Virtual Machine specification name
java.vm.version	Java Virtual Machine implementation version
java.vm.vendor	Java Virtual Machine implementation vendor
java.vm.name	Java Virtual Machine implementation name
java.specification.version	Java Runtime Environment specification version
java.specification.vendor	Java Runtime Environment specification vendor
java.specification.name	Java Runtime Environment specification name
java.class.version	Java class format version number
java.class.path	Java class path
java.ext.dirs	Path of extension directory or directories
os.name	Operating system name
os.arch	Operating system architecture
os.version	Operating system version
file.separator	File separator ("/" on UNIX)
path.separator	Path separator (":" on UNIX)
line.separator	Line separator ("\n" on UNIX)
user.name	User's account name
user.home	User's home directory
user.dir	User's current working directory

```
static String getProperty(String key)
static String getProperty(String key)
```

Example 3-10: Write a Java program that reads an ASCII digit from keyboard and display a number sequence: 1, 2.. up to the value of the entered number.

Solution:

Java classes and methods that under our consideration:

Read keyboard input:

System.out.println()

System.out.print()

JoptionPane.showInputDialog()

Display message and numbers

JoptionPane.showMessageDialog()

System.in.read()

Study the J2SE API we found the following information:

InputStream class

`public abstract class InputStream extends Object`

This abstract class is the superclass of all classes representing an input stream of bytes. Applications that need to define a subclass of InputStream must always provide a method that returns the next byte of input.

// Return the number of bytes that can be read from the input stream
`int available()`

// Close the input stream and return all the resources associated with this stream
`void close()`

// Read the next byte of data from the input stream
`read()`

// Read a number of bytes from input stream and store them in the array
`int read(byte[ ] b)`

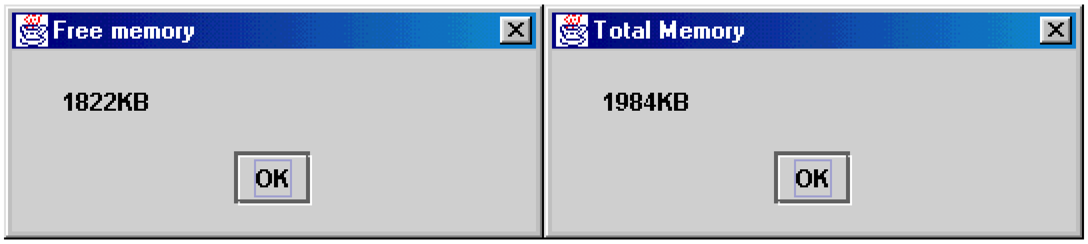
// Read up to a number of bytes as specified from input stream and store them in the array
`int read(byte[ ] b, int offset, int length)`

```
// ReadAChar.java
//
// Running Results:
```

```
// E:\LinJavaExs\3_ObjClass>java ReadAChar
// Enter a number in the range:0 - 9:
// 9
// 1       2       3       4       5       6       7       8       9
public class ReadAChar
  {
  public static void main (String args[])
   {
    char aChar = (char)-1;
    int  loopCountInt;
    String loopCountStr;

    // Solution 1: Using dialog box to read keyboard input
    // loopCountStr = JOptionPane.showInputDialog( "Enter a number in
                      the range:0 - 9" );
    // loopCountInt = Interger.parseInt(loopCountStr);

    // Solution 2: Using Console I/O to read keyboard input
    System.out.println("Enter a number in the range:0 - 9: ");
    try
     {
      aChar = (char)System.in.read();
     }
    catch (Exception e)
     {
      System.out.println("Error: " + e.toString());
     }
    loopCountInt = Character.digit(aChar, 10);
    if ((loopCountInt > 0) && (loopCountInt < 10))
    {
      for (int n = 1; n <= loopCountInt; n++)
      {
        System.out.print(n);
        System.out.print("\t");
      }
    }
    else
      System.out.println("The number was out of the range: 0 - 9!");
  }
}
```

Example 3-11: This Java application examines the system memory information. The instance of the class Runtime allows a Java application to interface with the run time environment. Some useful methods of an Runtime object are
long freeMemory() // Return the amount of free memory in the system
long totalMemory() // Returns the total amount of memory in the Java Virtual Machine.

```
/* AvaiMemory.java
 *
 *public class Runtime extends Object

 */
import javax.swing.JOptionPane;
import java.util.*;
import java.lang.*;
public class AvaiMemory
  {
  public static void main (String args[])
  {
    Runtime runtime = Runtime.getRuntime();
    long freeMem = runtime.freeMemory() / 1024;
    long totalMem = runtime.totalMemory() / 1024;
    JOptionPane.showMessageDialog(null, freeMem + "KB", "Free memory",
    JOptionPane.PLAIN_MESSAGE);
    JOptionPane.showMessageDialog(null, totalMem + "KB", "Total Memory",
    JOptionPane.PLAIN_MESSAGE);
  }
}
```



```
1822KB
```

```
1984KB
```

Some light-weight GUI components defined in javax.swing.* and java.awt.* packages

- public class **JLabel** extends Jcomponents implements SwingConstants, Accessible

- public class **JTextField** extends JtextComponents implements SwingConstants

  addActionListener()
  setEditable()

- public class **JButton** extends AbstractButton implements Accessible

  addActionListener()

- public class **Container** extends Component

  ```
  // A generic Abstract Window Toolkit (AWT) container object that can
  // contain other AWT components
  // Some methods:
  Container c = getContentPane();
  c.setLayout() = (new FlowLayout()); // Set the Layout Manager for this
                                      //  container
  c.add(minuteLabel)
  c.add(minuteField)
  ```

- public class **FlowLayout** extends Objects implements LayoutManager, Serializable
  // Constructor: FlowLayout()

A flow layout arranges components in a left-to-right flow, much like lines of text in a paragraph. Flow layouts are typically used to arrange buttons in a panel. It will arrange buttons left to right until no more buttons fit on the same line. Each line is centered.