Chapter 0

Object-Oriented Design

Chapter Objectives

- Review the core concepts underlying object-oriented programming
- Review how these concepts are accomplished in a Java program
- Discuss the use of generic types to define collection classes

Object Orientation

- An *object* is a fundamental entity in a Java program
- Java programs also manage *primitive data* such as numbers and characters
- An object usually represents a more complex concept such as a bank account

Object Orientation

- Objects often contain primitive data
- An Object that represents a bank account might contain a numeric account balance
- An object is defined by a *class*
- A class contains the definition of the data values and operations for an object

Object Orientation

- Once a class has been created, multiple objects can be instantiated from that class
- For example, multiple bank accounts can be created from a single class
- Each bank account object keeps track of its own balance and other account information
- This ability to protect and manage its own information is called *Encapsulation*

Object Orientation

- Classes interact with each other by invoking the methods of another class
- Classes can also be created from other classes using *inheritance*
- Inheritance means that the definition of one class can be based on another class that already exists
- Inheritance leads to reuse of existing solutions

Object Orientation

- A class that inherits from or is derived from another class is said to be a child of that class
- Classes may also be derived from that child creating a hierarchy
- Common characteristics are defined in high-level classes
- Specific differences are defined in derived classes

FIGURE 2.1 Various aspects of object-oriented software

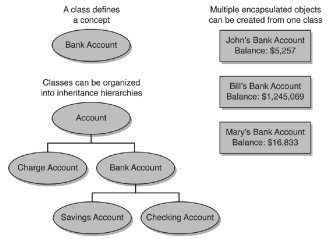

FIGURE 2.1 Various aspects of object-oriented software

Using Objects

- Objects provide services via their public methods
- For example

```
System.out.println ("Hello");
```

illustrates a call to the println method of the out object which is stored in the System class

Abstraction

- An object is an *abstraction* meaning that the precise details of how it works are irrelevant to the user
- A watch is an example of an abstraction
- We need not know how it works, only that it provides the correct time

Abstraction

- Abstraction does not eliminate the details
- It simply confines them to the minimum scope necessary
- It is not necessary that we know how an automatic transmission works
- However it is necessary that the designers, builders, and maintainers of that transmission know how it works

Creating Objects

- Java variables can hold either a primitive value or a reference to an object
- For example:

```
int x;   /*creates a variable x to hold a
  primitive integer*/

Integer y; /*creates a variable y to hold an
  object of the Integer class*/
```

Creating Objects

- The declaration (`int x;`) actually creates a primitive integer (and initializes it to 0)
- The declaration (`Integer y;`) simply creates an object reference variable that can point to an Integer object (and initializes it to null)
  - No Integer object exists until we instantiate one

```
y = new Integer(74); /*creates a new Integer
object with value 74 and sets the object
reference variable y to point to the new
object*/
```


Creating Objects

- An object reference variable, such as y, actually stores the address where the object is stored in memory
- Some objects are so frequently used that Java allows their use without explicit instantiation
- For example:

```
String name = "James Gosling";
```


Class Libraries and Packages

- A *class library* is a set of classes that supports the development of programs
- Class libraries are often supplied as part of a language or by third-party vendors
- The String class, for example, is not an inherent part of the Java language
- Instead, it is part of the Java *standard class library* that can be found in any Java environment

Class Libraries and Packages

- The standard class library is made up of several clusters of related classes, sometimes called Java APIs
- For example, the Java Database API includes the set of classes that support database applications
- The classes of the standard class library are grouped into *packages* grouping related classes by one name
- The String class and the System class are both part of the java.lang package
- The package organization of classes is more fundamental than the API names

The import Declaration

- The classes of the package java.lang are automatically available for use when writing a program
- To use classes from any other package, we must either fully qualify the class (e.g. java.util.Random) or use an import declaration
- For example:

```
import java.util.Random;
```

- Another form of the declaration is

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

- This second form of the declaration allows us to reference any class in the given package

State and Behavior

- All objects have a state and a set of behaviors
- State refers to the values of an objects variables
- Note that each object, even objects of the same class, have their own state
- Behavior refers to the actions that an object can take
- Behaviors are accomplished through method calls
- Unlike state, behaviors are consistent across all objects of a given class

Classes

- An object is defined by a class
- A class is the model, pattern, or blueprint from which an object is created
- A class is not an object anymore than a blueprint is a house
- A class contains the declarations of the data that will be stored in each instantiated object and the declarations of the methods that can be invoked using an object of this class.
- Collectively, these are called the *members* of the class

FIGURE 2.2 The members of a class: data and method declarations

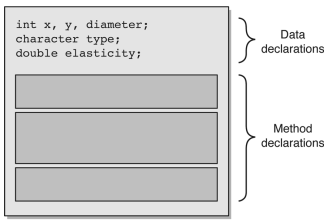

FIGURE 2.2 The members of a class: data and method declarations

The Coin Class

- The Coin class is an example that represents a coin that can be flipped and that at any point in time shows a face of either heads or tails

Listing 2.1

Listing 2.1

```
//********************************************************************
//  Coin.java        Authors: Lewis/Loftus
//
//  Represents a coin with two sides that can be flipped.
//********************************************************************

import java.util.Random;

public class Coin
{
   private final int HEADS = 0;
   private final int TAILS = 1;

   private int face;

   //-----------------------------------------------------------------
   //  Sets up the coin by flipping it initially.
   //-----------------------------------------------------------------
   public Coin ()
   {
        flip();
   }
```


Listing 2.1 (cont.)

```
   //-----------------------------------------------------------------
   //  Flips the coin by randomly choosing a face value.
   //-----------------------------------------------------------------
   public void flip ()
   {
     face = (int) (Math.random() * 2);
   }

   //-----------------------------------------------------------------
   //  Returns true if the current face of the coin is heads.
   //-----------------------------------------------------------------
   public boolean isHeads ()
       {
      return (face == HEADS);
   }

   //-----------------------------------------------------------------
   //  Returns the current face of the coin as a string.
   //-----------------------------------------------------------------
```


Listing 2.1 (cont.)

Listing 2.1 continued

```
   public String toString()
   {
      String faceName;

      if (face == HEADS)
         faceName = "Heads";
      else
         faceName = "Tails";

      return faceName;
   }
}
```

Instance Data

- Note that in the `Coin` class, the constants `HEADS` and `TAILS` and the variable `face` are declared inside the class but not inside of any method
- The location at which a variable is declared defines its *scope*
- By being declared at the class level, these variables and constants, called *instance data*, can be referenced in any method of the class

Encapsulation

- We can think about an object in two ways:
  - As designer and/or developer of a particular object, we need to think about the details of how that object works
  - As designer and/or developer of other objects that might interact with that object in a larger context, we need only concern ourselves with the services that the object provides
- Objects should be *self-governing* meaning that the variables contained within an object should only be modified by the methods of that object

Encapsulation

- We should make it difficult for any other object to "reach in" and change the value of variable inside the class
- This characteristic is called Encapsulation
- The methods of a class define the interface between objects of that class and the program(s) that use them
- The code that uses an object, sometimes called the *client*, should not be allowed to access variables directly

Visibility Modifiers

- A modifier is a Java reserved word used to specify particular characteristics of a programming language construct
- For example, the `final` modifier is used to declare a constant
- *Visibility modifiers* control access to the members (variables and methods) of a class
- The reserve words `public` and `private` are visibility modifiers that can be applied to variables and methods of a class

Visibility Modifiers

- If a member of a class has public visibility, then it can be accessed from outside of the class
- If a member of a class has private visibility, then it can be used anywhere inside the class but cannot be referenced externally
- A third visibility modifier, `protected`, is relevant only in the context of inheritance and will be discussed later in this chapter

FIGURE 2.3 The effects of public and private visibility

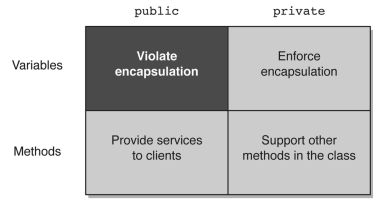

FIGURE 2.3 The effects of public and private visibility

Local Data

- Unlike instance data, which is declared at the class level, local data is declared inside of a method
- Local data has a scope limited to only the method in which it is declared
- Formal parameter names in a method header are also consider local data
- They do not exist until the method is called and cease to exist when the method is exited

Constructors

- A constructor is similar to a method that is invoked when an object is instantiated
- We generally create constructors to help us establish the initial state of an object
- A constructor differs from a regular method in two ways:
  - The name of the constructor will be the same as the name of the class
  - A constructor cannot return a value and does not have a return type specified in the header
- Each class has a *default constructor* if we do not provide one

Method Overloading

- In Java, as in other object-oriented languages, you can use the same method name with different parameter lists for multiple methods
- This technique is called *method overloading*
- A method's name along with the number, type, and order of its parameters is called the method's *signature*
- As long as each method's signature is unique, then the compiler can tell which method is being referenced
- Note that the return type is not part of the signature

Method Overloading

- The System.out.println method is a good example of an overloaded method as illustrated by this partial list of signatures:

```
println(String s)
println(int i)
println(double d)
println(char c)
println(boolean b)
```


References Revisited

- An object reference variable that does not currently point to an object is a *null* reference
- An object may refer to itself using the `this` reference
- It is possible for two or more variables to point to the same object
- This situation is called *aliasing*
- For example if x and y are both object reference variables pointing to the same bank account and we change the balance on bank account x then we have also altered the balance of y.

Garbage Collection

- Once all references to an object have been discarded, we can no longer access that object and it is then called *garbage*
- Java performs automatic garbage collection meaning that occasionally the Java run time environment collects all of the objects that have been marked as garbage and returns their memory to the system for future use
- In many other languages, the programmer would have to explicitly return memory that has become garbage or risk running out of memory

Passing Objects as Parameters

- Formal parameters are those listed in the header of a method
- Actual parameters are those listed in the call to a method
- In Java, all parameters are passed by value meaning that at the time of the method call, the value of the actual parameter is copied into the formal parameter
- However, when object reference variables are passed as parameters, the net effect is that the address is passed and thus the formal parameter becomes an alias to the actual parameter

Passing Objects as Parameters

- Thus if we change the state of the object that is referenced by the formal parameter, we are also changing the state of the object referenced by the actual parameter (since it is the same object)
- However, if we change the formal parameter itself, by assigning it to a different object, we will have no impact on the actual parameter

The static Modifier

- Another kind of variable, called a *static variable*, is shared among all instances of a class
- There is only one copy of a static variable for all objects of a class
- The reserved word static is used as a modifier to declare a static variable:

```
private static int count = 0;
```


Static Methods

- A static method can be invoked through the class name without having to instantiate an object of the class
- All of the methods of the `Math` class are static
- For example:

```
System.out.println("Square root of 27:
" + Math.sqrt(27));
```

- A method is made static by using the static modifier in the method declaration

Wrapper Classes

- At times, we will need to convert primitive types to classes so that we may use them in an object context (e.g. collections)
- Java provides wrapper classes for all primitive types to allow for this conversion
- For example
  - int - Integer
  - char - Character
- The wrapper classes also provide useful methods and constants such as the Integer.parseInt method that takes a string and returns its numeric value or the MIN_VALUE and MAX_VALUE constants of the Integer class that represent the smallest and largest int values respectively

Interfaces

- A Java *interface* is a collection of constants and abstract methods
- An *abstract method* is a method that does not have an implementation
- The following interface, called Complexity, contains two abstract methods, `setComplexity` and `getComplexity`:

```
interface Complexity
{
  void setComplexity (int complexity);
   int getComplexity ();
}
```

Interfaces

- A class implements an interface by providing method implementations for each of the abstract methods
- A class that implements an interface uses the reserve word `implements` followed by the interface name

```
class Questions implements Complexity
{
    int difficulty;

    void setComplexity (int complexity);
    {
      difficulty = complexity;
    }

    int getComplexity()
    {
      return difficulty;
    }
}
```

- Multiple classes may implement the same interface and a single class may implement multiple interfaces

Interfaces - Comparable

- The `Comparable` interface is defined in the java.lang package
- It contains one method, `compareTo`, which takes an object as parameter and returns an integer
  - e.g. int result = obj1.compareTo(obj2);
- The intention of this interface is to provide a common mechanism to compare one object to another
- The integer that is returned should be negative if obj1 is less than obj2, 0 if they are equal, and positive if obj1 is greater than obj2

Interfaces - Iterator

- The `Iterator` interface is also defined in the Java standard class library
- It is used by classes that represent a collection of objects, providing a means to move through the collection one object at time
- The two primary methods of the `Iterator` interface are `hasNext`, which returns a `boolean`, and `next`, which returns an object

Interfaces - Iterator

- The Iterator interface also has a method called `remove` which takes no parameters and has a void return type
- The purpose of this method is to remove the object most recently returned by the next method

Interfaces - Iterable

- The `Iterable` interface provides one method called `iterator` which takes no parameters and returns an `Iterator` over the given collection

Inheritance

- In object-oriented programming, *inheritance* means to derive a class from the definition of another class
- In general, creating a new class via inheritance is faster, easier, and cheaper
- Inheritance is at the heart of software reuse

Inheritance

- The word class comes from the idea of classifying groups of objects with similar characteristics
- For example, all mammals share certain characteristics:
  - Warm blooded
  - Have hair
  - Bear live offspring

Inheritance

- In software, a `mammal` class would have variables and methods that describe the state and behavior of mammals
- From that class, we could derive a `horse` class
- The `horse` class would inherit all of the variables and methods of the `mammal` class
- The horse class could also define additional variables and methods of its own

Inheritance

- The original class that is used to derive a new one is called the *parent class*, *superclass*, or *base class*
- The derivation process establishes an *is-a relationship* between the two classes
- For example, a `horse` is-a `mammal`

Inheritance Example

```
class Book
{
   protected int numPages;

   protected void pages(){
   {
    System.out.println("Number of Pages:  "+ numPages);
   }
}

class Dictionary extends Book
{
    private int numDefs;

    public void info(){
     {
       System.out.println("Number of definitions:"+ numDefs);
       System.out.println("Definitions per page:"+ numDefs/numPages);
     }
}

Usage could be
        Dictionary webster = new Dictionary();
        System.out.println(webster.info());
```


Inheritance - the protected Modifier

- Earlier, we discussed the `public` and `private` visibility modifiers
- An additional visibility modifier, `protected`, only applies to inheritance
- A child class inherits all of the variables and methods that are declared `public` in the parent
- This violates our earlier discussion of encapsulation

Inheritance - the protected Modifier

- The `protected` modifier allows the child class to inherit the variable or method but maintains some of the properties of encapsulation
- Specifically, a variable or method declared with `protected` visibility may be accessed by any class in the same package

Inheritance - the super Reference

- The reserve word `super` can be used in a child class to refer to its parent class
- Like the `this` reference, what the word `super` refers to depends on the class in which it is used
- Unlike the `this` reference which refers to a specific instance, `super` is a general reference to the members of the parent class
- The super reference is often used to call the constructor of the parent class:

```
super();
```


Inheritance - Overriding Methods

- A child class inherits all of the variables and methods of the parent
- A child class may also declare its own variables and methods
- Among these declarations may be methods with the same name and signature as those of the parent
- This is called *overriding*
- If a method is defined with the `final` modifier then a child class cannot override it

Inheritance - Class Hierarchies

- A child class derived from one parent can be the parent of its own child class
- Multiple child classes may also be derived from a single parent
- Thus inheritance relationships often develop into *class hierarchies*

FIGURE 2.4 A UML class diagram showing a class hierarchy

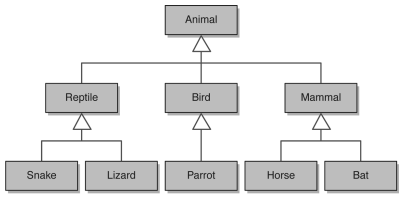

FIGURE 2.4 A UML class diagram showing a class hierarchy

Inheritance - the `Object` Class

- In Java, all classes are derived ultimately from the `Object` class
- The `Object` class provides a `toString` method which is often overridden
- The `Object` class provides an `equals` method that tests to see if two objects are aliases and is also often overridden

Inheritance - Abstract Classes

- An *abstract class* may contain fully implemented methods and abstract methods
- Like an interface, an abstract class cannot be instantiated
- A class is declared as abstract by including the `abstract` modifier in the class header

Inheritance - Abstract Classes

- Some classes may be too conceptually abstract to be instantiated
- Consider a `vehicle` class
- Without knowing what kind of vehicle it is, what would it mean to instantiate one

FIGURE 2.5
A vehicle class hierarchy

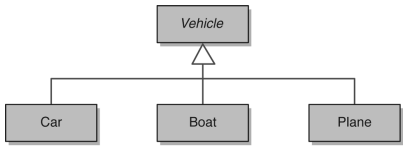

FIGURE 2.5 A vehicle class hierarchy

Interface Hierarchies

- Like hierarchies of classes, inheritance may also apply to interfaces creating hierarchies of interfaces
- A child interface inherits the abstract methods and constants of the parent interface

Polymorphism

- The term *polymorphism* can be defined as “having many forms”
- A *polymorphic reference* is a reference variable that can refer to different types of objects at different points in time
- The specific method invoked by a polymorphic reference can change from one invocation to the next

Polymorphism

- Polymorphism results in the need for *dynamic binding*
- Usually binding of a call to the code occurs at compile time
- Dynamic binding means that this binding cannot occur until run time

Polymorphism

- In Java, a reference that is declared to refer to an object of a particular class can also be used to refer to an object of any class related to it by inheritance
- For example:

```
Mammal pet;
Horse secretariat = new Horse();
pet = secretariat;  // a valid assignment
```

- This is polymorphism as a result of inheritance

FIGURE 2.6
A class hierarchy of employees

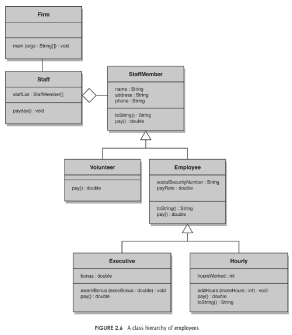

FIGURE 2.6 A class hierarchy of employees

Polymorphism

- Polymorphism can also result from interfaces
- If we declare a reference variable to be of an interface type, it can then refer to any object that implements that interface

Generic Types

- Java enables us to define a class based upon a *generic type*
- This means that we can define a class so that it stores, operates on, and manages objects whose type is not specified until the class is instantiated

Generic Types

Assume that we need to define a class named `Array` that stores and manages array of any type.

```
class Array<T>{
      T[] arr;
      int size=0;
      public Array(int capacity){
          arr = new T[capacity];
          size = capacity;
      }
      public T get(int index){
            return(arr[index]);
     }
     public void set(int index, T elem){
             arr[index] = elem;
     }
     //etc
```


Generic Types

- Then if we wanted to instantiate an to hold Strings

```
final int SIZE = 50;
Array<String> arrS = new Array(SIZE);
arrS.set(0, "Apples");
String first = arrS.get(0);
Array<Coin> arrC = new Array(SIZE/2);
arrC.set(0, new Coin());
```


Generic Types

- The type supplied at the time of instantiation replaces the type T wherever it is used in the declaration of the class
- A generic type such as T cannot be instantiated
- We will use generics throughout the book to develop collection classes

Exceptions

- Problems that arise in a Java program may generate exceptions or errors
- An exception is an object that defines an unusual or erroneous situation
- An error is similar to an exception, except that an error generally represents an unrecoverable situation

Exceptions

- A program can be designed to process an exception in one of three ways:
  - Not handle the exception at all
  - Handle the exception where it occurs
  - Handle the exception at another point in the program

Exceptions

- If an exception is not handled at all by the program, the program will produce an exception message and terminate
- For example:

Exception in three "main" java.lang.ArithmeticException: / by zero
at Zero.main (Zero.java:17)

- This message provides the name of the exception, description of the exception, the class and method, as well as the filename and line number where the exception occurred

Exceptions

- To handle an exception when it is thrown, we use a *try statement*
- A try statement consists of a try block followed by one or more catch clauses

```
try
{
 // statements in the try block
}
catch (IOException exception)
{
 // statements that handle the I/O problem
}
```


Exceptions

- When a try statement is executed, the statements in the try block are executed
- If no exception is thrown, processing continues with the statement following the try statement
- If an exception is thrown, control is immediately passed to the first catch clause whose specified exception corresponds to the class of the exception that was thrown
- If an exception is not caught and handled where it occurs, control is immediately returned to the method that invoked the method that produced the exception

Exceptions

- If that method does not handle the exception (via a try statement with an appropriate catch clause) then control returns to the method that called it
- This process is called propagating the exception
- Exception propagation continues until the exception is caught and handled or until it is propagated out of the main method resulting in the termination of the program

FIGURE 2.7
Part of the Error and Exception class hierarchy

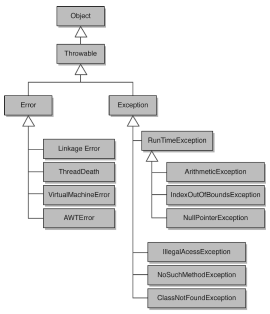

FIGURE 2.7 Part of the Error and Exception class hierarchy

Zero.java

```
//  Demonstrates an uncaught exception.
public class Zero{
   public static void main (String[] args){
   int numerator = 10;
   int denominator = 0;
   System.out.println (numerator / denominator);
   System.out.println ("This text will not be printed.");
 }
}
Output:
Exception in thread "main" java.lang.ArithmeticException:
/ by zero at Zero.main(Zero.java:11)
```


ProductCodes.java

```
// Demonstrates an exception that is caught.

public class ProductCodes{
   Scanner sc = new Scanner(System.in);
   public static void main (String[] args){
    String code;
    char zone;
    Scanner sc = new Scanner(System.in);
    int district, valid = 0, banned = 0;
    System.out.print ("Enter product code (XXX to quit): ");
    code =sc.nextLine();
```


ProductCode.java

```
while (!code.equals ("XXX")){
         try{
            zone = code.charAt(9);
            district = Integer.parseInt(code.substring(3, 7));
            valid++;
            if (zone == 'R' && district > 2000)
               banned++;
         }
         catch (StringIndexOutOfBoundsException exception){
            System.out.println ("Improper code length: " + code);
         }
         catch (NumberFormatException exception){
            System.out.println ("District is not numeric:" +code);
         }
         System.out.print ("Enter product code (XXX to quit): ");
         code =sc.nextLine();
   }
```


ProductCodes.java

```
      System.out.println ("# of valid codes entered: " + valid);
      System.out.println ("# of banned codes entered: " + banned);
         }
      }
```


The *throw* Statement

- A programmer can define an exception by extending the appropriate class in the Exception class hierarchy.
- Exceptions are thrown using the *throw* statement
- Usually a throw statement is nested inside an if statement that evaluates the condition to see if the exception should be thrown

The *throw* Statement

```
class Array<T>{
      T[] arr;
      int size = 0;
      public Array(int capacity){
          arr = new T[capacity];
          size = capacity;
      }
     public T get(int index) throws IndexOutOfBoundsException{
           if (index <0 || index > size-1)
               throw(new IndexOutOfBoundsException);
           else
               return(arr[index]);
     }
     public void set(int index, T elem){
            arr[index] = elem;
     }
}
```


Checked Exceptions

- An exception is either *checked* or *unchecked*
- A checked exception must either be caught by a method or it must be in the throws clause of any method that may throw it or propagate it.
- The compiler will complain if a checked exception is not handled appropriately
- All exceptions in java are checked except RuntimeException and its descendants.
- If you are calling a method from JDK you may verify if it throws an exception from the description of the method and catch it if needed.

Standard I/O

- There are three standard I/O streams:
  - *standard output* – defined by `System.out`
  - *standard input* – defined by `System.in`
  - *standard error* – defined by `System.err`
- We use `System.out` when we execute `println` statements
- `System.out` and `System.err` typically represent a particular window on the monitor screen
- `System.in` typically represents keyboard input, which we've used many times with `Scanner` objects

The IOException Class

- Operations performed by some I/O classes may throw an `IOException`
  - A file might not exist
  - Even if the file exists, a program may not be able to find it
  - The file might not contain the kind of data we expect
- An `IOException` is a checked exception therefore it must either be caught from a method that performs I/O or a throw must be propagated from that method all the way to main.

Reading From and Writing To Text Files

- We can use the Scanner class to read data from a text file

```
try{
   Scanner scan = new Scanner(new File("myfile.dat"));
   while(scan.hasNext()
      String line = scan.nextLine();
     // process the line
  }
}
catch (FileNotFoundException e){
      System.out.println("File specified is not found");
}
scan.close();
```


Writing Text Files

```
Let's now examine other classes that let us write data to a text file
The FileWriter class represents a text output file, but with minimal
support for manipulating data. Therefore, we also rely on PrintStream
objects, which have print and println methods defined for them

// TestData.java       Author: Lewis/Loftus
//
//  Demonstrates I/O exceptions and the use of a character file
//  output stream.
import java.io.*;

public class TestData
{
   public static void main (String[] args) {
      final int MAX = 10;
      int value =234;;
      String file = "test.dat";
      try{
          FileWriter fw = new FileWriter (file);
          BufferedWriter bw = new BufferedWriter (fw);
          PrintWriter outFile = new PrintWriter (bw);
```

Writing Text Files

```
     for (int line=1; line <= MAX; line++) {
          outFile.print (value + "   ");
           value++;
     }
     outFile.println ();
     outFile.close();
     System.out.println ("Output file has been created: " + file);
    }
    catch (IOException e){
       System.out.println(" Error with file");
    }
} //end main
} //end class
```