Classes and Objects: A Deeper Look
Instead of this absurd division into sexes, they ought to class people as static and dynamic.
   — Evelyn Waugh

Is it a world to hide virtues in?
   — William Shakespeare

But what, to serve our private ends, Forbids the cheating of our friends?
   — Charles Churchill

This above all: to thine own self be true.
   — William Shakespeare

Don’t be “consistent,” but be simply true.
   — Oliver Wendell Holmes, Jr.
OBJECTIVES

In this chapter you will learn:

- Encapsulation and data hiding.
- The notions of data abstraction and abstract data types (ADTs).
- To use keyword `this`.
- To use `static` variables and methods.
- To import `static` members of a class.
- To use the `enum` type to create sets of constants with unique identifiers.
- How to declare `enum` constants with parameters.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>8.2</td>
<td>Time Class Case Study</td>
</tr>
<tr>
<td>8.3</td>
<td>Controlling Access to Members</td>
</tr>
<tr>
<td>8.4</td>
<td>Referring to the Current Object’s Members with the <code>this</code> Reference</td>
</tr>
<tr>
<td>8.5</td>
<td>Time Class Case Study: Overloaded Constructors</td>
</tr>
<tr>
<td>8.6</td>
<td>Default and No-Argument Constructors</td>
</tr>
<tr>
<td>8.7</td>
<td>Notes on <code>Set</code> and <code>Get</code> Methods</td>
</tr>
<tr>
<td>8.8</td>
<td>Composition</td>
</tr>
<tr>
<td>8.9</td>
<td>Enumerations</td>
</tr>
<tr>
<td>8.10</td>
<td>Garbage Collection and Method <code>finalize</code></td>
</tr>
</tbody>
</table>
8.11 static Class Members
8.12 static Import
8.13 final Instance Variables
8.14 Software Reusability
8.15 Data Abstraction and Encapsulation
8.16 Time Class Case Study: Creating Packages
8.17 Package Access
8.18 (Optional) GUI and Graphics Case Study: Using Objects with Graphics
8.19 (Optional) Software Engineering Case Study: Starting to Program the Classes of the ATM System
8.20 Wrap-Up
8.2 Time Class Case Study

- **public services (or public interface)**
  - public methods available for a client to use

- If a class does not define a constructor the compiler will provide a default constructor

- **Instance variables**
  - Can be initialized when they are declared or in a constructor
  - Should maintain consistent (valid) values
Software Engineering Observation 8.1

Methods that modify the values of private variables should verify that the intended new values are proper. If they are not, the set methods should place the private variables into an appropriate consistent state.
// Fig. 8.1: Time1.java
// Time1 class declaration maintains the time in 24-hour format.

public class Time1 {
    private int hour;   // 0 – 23
    private int minute; // 0 - 59
    private int second; // 0 - 59

    // set a new time value using universal time; ensure that
    // the data remains consistent by setting invalid values to zero
    public void setTime( int h, int m, int s ) {
        hour = ( ( h >= 0 && h < 24 ) ? h : 0 );   // validate hour
        minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
        second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
    } // end method setTime
} // end class Time1

private instance variables

Declare public method setTime

Validate parameter values before setting instance variables
19. // convert to String in universal-time format (HH:MM:SS)
20. public String toUniversalString()
21. {
22.     return String.format("%02d:%02d:%02d", hour, minute, second);
23. } // end method toUniversalString
24.
25. // convert to String in standard-time format (H:MM:SS AM or PM)
26. public String toString()
27. {
28.     return String.format("%d:%02d:%02d %s",
29.         ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
30.         minute, second, ( hour < 12 ? "AM" : "PM" ) );
31. } // end method toString
32. } // end class Time1
• **String method format**
  
  – Similar to `printf` except it returns a formatted string instead of displaying it in a command window

• **new implicitly invokes Time1’s default constructor since Time1 does not declare any constructors**
Software Engineering Observation 8.2

Classes simplify programming, because the client can use only the public methods exposed by the class. Such methods are usually client oriented rather than implementation oriented. Clients are neither aware of, nor involved in, a class’s implementation. Clients generally care about what the class does but not how the class does it.
Software Engineering Observation 8.3

Interfaces change less frequently than implementations. When an implementation changes, implementation-dependent code must change accordingly. Hiding the implementation reduces the possibility that other program parts will become dependent on class-implementation details.
// Fig. 8.2: Time1Test.java
// Time1 object used in an application.

class Time1Test
{
    public static void main( String args[] )
    {
        // create and initialize a Time1 object
        Time1 time = new Time1(); // invokes Time1 constructor

        // output string representations of the time
        System.out.print( "The initial universal time is: " );
        System.out.println( time.toUniversalString() );
        System.out.print( "The initial standard time is: " );
        System.out.println( time.toString() );
        System.out.println(); // output a blank line
    }
}

Create a Timel object

Call toUniversalString method

Call toString method
// change time and output updated time

time.setTime(13, 27, 6);
System.out.print( "Universal time after setTime is: " );
System.out.println( time.toUniversalString() );
System.out.print( "Standard time after setTime is: " );
System.out.println( time.toString() );
System.out.println(); // output a blank line

// set time with invalid values; output updated time

time.setTime(99, 99, 99);
System.out.println( "After attempting invalid settings:" );
System.out.print( "Universal time: " );
System.out.println( time.toUniversalString() );
System.out.print( "Standard time: " );
System.out.println( time.toString() );

} // end main

} // end class Time1Test

The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
8.3 Controlling Access to Members

• A class’s public interface
  – public methods a view of the services the class provides to the class’s clients

• A class’s implementation details
  – private variables and private methods are not accessible to the class’s clients
Common Programming Error 8.1

An attempt by a method that is not a member of a class to access a private member of that class is a compilation error.
// Fig. 8.3: MemberAccessTest.java  
// Private members of class Time1 are not accessible.
public class MemberAccessTest
{
    public static void main( String args[] )
    {
        Time1 time = new Time1(); // create and initialize Time1 object
        time.hour = 7;    // error: hour has private access in Time1
        time.minute = 15; // error: minute has private access in Time1
        time.second = 30; // error: second has private access in Time1
    } // end main
} // end class MemberAccessTest

Attempting to access private instance variables

MemberAccessTest.java:9: hour has private access in Time1
  time.hour = 7;  // error: hour has private access in Time1
  ^
MemberAccessTest.java:10: minute has private access in Time1
  time.minute = 15; // error: minute has private access in Time1
  ^
MemberAccessTest.java:11: second has private access in Time1
  time.second = 30; // error: second has private access in Time1
  ^
3 errors
8.4 Referring to the Current Object’s Members with the this Reference

• The this reference
  – Any object can access a reference to itself with keyword this
  – Non-static methods implicitly use this when referring to the object’s instance variables and other methods
  – Can be used to access instance variables when they are shadowed by local variables or method parameters

• A .java file can contain more than one class
  – But only one class in each .java file can be public
```java
// Fig. 8.4: ThisTest.java
// this used implicitly and explicitly to refer to members of an object.

public class ThisTest
{
    public static void main( String args[] )
    {
        SimpleTime time = new SimpleTime( 15, 30, 19 );
        System.out.println( time.buildString() );
    } // end main
} // end class ThisTest

// class SimpleTime demonstrates the "this" reference
class SimpleTime
{
    private int hour;   // 0-23
    private int minute; // 0-59
    private int second; // 0-59

    // if the constructor uses parameter names identical to
    // instance variable names the "this" reference is
    // required to distinguish between names
    public SimpleTime( int hour, int minute, int second )
    {
        this.hour = hour;      // set "this" object's hour
        this.minute = minute;  // set "this" object's minute
        this.second = second;  // set "this" object's second
    } // end SimpleTime constructor

Create new SimpleTime object

Declare instance variables

Method parameters shadow instance variables

Using this to access the object’s instance variables
```

© 1992-2007 Pearson Education, Inc. All rights reserved.
// use explicit and implicit "this" to call toUniversalString
public String buildString()
{
    return String.format( "%24s: %s
%24s: %s",
    "this.toUniversalString()",{this.toUniversalString()},
    "toUniversalString()",{toUniversalString()});
} // end method buildString

// convert to String in universal-time format (HH:MM:SS)
public String toUniversalString()
{
    // "this" is not required here to access instance variables,
    // because method does not have local variables with same
    // names as instance variables
    return String.format( "%02d:%02d:%02d",
        this.hour, this.minute, this.second );
} // end method toUniversalString

) // end class SimpleTime

this.toUniversalString(): 15:30:19
this.toUniversalString(): 15:30:19

toUniversalString(): 15:30:19

toUniversalString(): 15:30:19

Using this explicitly and implicitly to call toUniversalString

Use of this not necessary here
Common Programming Error 8.2

It is often a logic error when a method contains a parameter or local variable that has the same name as a field of the class. In this case, use reference `this` if you wish to access the field of the class—otherwise, the method parameter or local variable will be referenced.
Error-Prevention Tip 8.1

Avoid method parameter names or local variable names that conflict with field names. This helps prevent subtle, hard-to-locate bugs.
Performance Tip 8.1

Java conserves storage by maintaining only one copy of each method per class—this method is invoked by every object of the class. Each object, on the other hand, has its own copy of the class’s instance variables (i.e., non-\texttt{static} fields). Each method of the class implicitly uses \texttt{this} to determine the specific object of the class to manipulate.
8.5 Time Class Case Study: Overloaded Constructors

• Overloaded constructors
  – Provide multiple constructor definitions with different signatures

• No-argument constructor
  – A constructor invoked without arguments

• The this reference can be used to invoke another constructor
  – Allowed only as the first statement in a constructor’s body
// Fig. 8.5: Time2.java
// Time2 class declaration with overloaded constructors.

class Time2
{
    int hour; // 0 - 23
    int minute; // 0 - 59
    int second; // 0 - 59

    // Time2 no-argument constructor: initializes each instance variable
    // to zero; ensures that Time2 objects start in a consistent state
    public Time2()
    {
        this(0, 0, 0); // invoke Time2 constructor with three arguments
    } // end Time2 no-argument constructor

    // Time2 constructor: hour supplied, minute and second defaulted to 0
    public Time2(int h)
    {
        this(h, 0, 0); // invoke Time2 constructor with three arguments
    } // end Time2 one-argument constructor

    // Time2 constructor: hour and minute supplied, second defaulted to 0
    public Time2(int h, int m)
    {
        this(h, m, 0); // invoke Time2 constructor with three arguments
    } // end Time2 two-argument constructor
} // end Time2
```java
29 // Time2 constructor: hour, minute and second supplied
30 public Time2( int h, int m, int s )
31 {
32     setTime( h, m, s ); // Invoke setTime to validate time
33 } // end Time2 three-argument constructor

// Time2 constructor: another Time2 object supplied
35 public Time2( Time2 time )
36 {
37     // Invoke Time2 three-argument constructor
38     this( time.getHour(), time.getMinute(), time.getSecond() );
39 } // end Time2 constructor with a Time2 object argument

// Set Methods
// set a new time value using universal time; ensure that
// the data remains consistent by setting invalid values to zero
42 public void setTime( int h, int m, int s )
43 {
44     setHour( h );   // set the hour
45     setMinute( m ); // set the minute
46     setSecond( s ); // set the second
47 } // end method setTime
```

The constructor takes a reference to another `Time2` object as a parameter. Could have directly accessed instance variables of object `time` here.
// validate and set hour
public void setHour(int h)
{
    hour = ((h >= 0 && h < 24) ? h : 0);
} // end method setHour

// validate and set minute
public void setMinute(int m)
{
    minute = ((m >= 0 && m < 60) ? m : 0);
} // end method setMinute

// validate and set second
public void setSecond(int s)
{
    second = ((s >= 0 && s < 60) ? s : 0);
} // end method setSecond

// Get Methods
// get hour value
public int getHour()
{
    return hour;
} // end method getHour
// get minute value
public int getMinute()
{
    return minute;
} // end method getMinute

// get second value
public int getSecond()
{
    return second;
} // end method getSecond

// convert to String in universal-time format (HH:MM:SS)
public String toUniversalString()
{
    return String.format("%02d:%02d:%02d", getHour(), getMinute(), getSecond());
} // end method toUniversalString

// convert to String in standard-time format (H:MM:SS AM or PM)
public String toString()
{
    return String.format("%d:%02d:%02d %s",
        ( (getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12 ),
        getMinute(), getSecond(), ( getHour() < 12 ? "AM" : "PM" ));
} // end method toString

} // end class Time2
Common Programming Error 8.3

It is a syntax error when \texttt{this} is used in a constructor’s body to call another constructor of the same class if that call is not the first statement in the constructor. It is also a syntax error when a method attempts to invoke a constructor directly via \texttt{this}. 
Common Programming Error 8.4

A constructor can call methods of the class. Be aware that the instance variables might not yet be in a consistent state, because the constructor is in the process of initializing the object. Using instance variables before they have been initialized properly is a logic error.
Software Engineering Observation 8.4

When one object of a class has a reference to another object of the same class, the first object can access all the second object’s data and methods (including those that are private).
• Using *set* methods
  – Having constructors use *set* methods to modify instance variables instead of modifying them directly simplifies implementation changing
When implementing a method of a class, use the class’s *set* and *get* methods to access the class’s private data. This simplifies code maintenance and reduces the likelihood of errors.
// Fig. 8.6: Time2Test.java
// Overloaded constructors used to initialize Time2 objects.

public class Time2Test {
   public static void main( String args[] ) {
      Time2 t1 = new Time2();             // 00:00:00
      Time2 t2 = new Time2( 2 );          // 02:00:00
      Time2 t3 = new Time2( 21, 34 );     // 21:34:00
      Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
      Time2 t5 = new Time2( 27, 74, 99 ); // 00:00:00
      Time2 t6 = new Time2( t4 );         // 12:25:42

      System.out.println( "Constructed with:"
      System.out.println( "t1: all arguments defaulted" );
      System.out.printf( " %s\n", t1.toUniversalString() );
      System.out.printf( " %s\n", t1.toString() );
   }
}

Call overloaded constructors
System.out.println("t2: hour specified; minute and second defaulted");
System.out.printf(" %s\n", t2.toUniversalString());
System.out.printf(" %s\n", t2.toString());

System.out.println("t3: hour and minute specified; second defaulted");
System.out.printf(" %s\n", t3.toUniversalString());
System.out.printf(" %s\n", t3.toString());

System.out.println("t4: hour, minute and second specified");
System.out.printf(" %s\n", t4.toUniversalString());
System.out.printf(" %s\n", t4.toString());

System.out.println("t5: all invalid values specified");
System.out.printf(" %s\n", t5.toUniversalString());
System.out.printf(" %s\n", t5.toString());
System.out.println( "t6: Time2 object t4 specified" );
System.out.printf( " %s
", t6.toUniversalString() );
System.out.printf( " %s
", t6.toString() );
}
} // end class Time2Test

<table>
<thead>
<tr>
<th>Time2Test.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>38 System.out.println( &quot;t6: Time2 object t4 specified&quot; );</td>
</tr>
</tbody>
</table>
| 39 System.out.printf( " %s
", t6.toUniversalString() ); |
| 40 System.out.printf( " %s
", t6.toString() ); |
| 41 } // end main |
| 42 } // end class Time2Test |

- **t1**: all arguments defaulted
  - 00:00:00
  - 12:00:00 AM
- **t2**: hour specified; minute and second defaulted
  - 02:00:00
  - 2:00:00 AM
- **t3**: hour and minute specified; second defaulted
  - 21:34:00
  - 9:34:00 PM
- **t4**: hour, minute and second specified
  - 12:25:42
  - 12:25:42 PM
- **t5**: all invalid values specified
  - 00:00:00
  - 12:00:00 AM
- **t6**: Time2 object t4 specified
  - 12:25:42
  - 12:25:42 PM

© 1992-2007 Pearson Education, Inc. All rights reserved.
8.6 Default and No-Argument Constructors

• Every class must have at least one constructor
  – If no constructors are declared, the compiler will create a default constructor
    • Takes no arguments and initializes instance variables to their initial values specified in their declaration or to their default values
      – Default values are zero for primitive numeric types, false for boolean values and null for references
  – If constructors are declared, the default initialization for objects of the class will be performed by a no-argument constructor (if one is declared)
Common Programming Error 8.5

If a class has constructors, but none of the public constructors are no-argument constructors, and a program attempts to call a no-argument constructor to initialize an object of the class, a compilation error occurs. A constructor can be called with no arguments only if the class does not have any constructors (in which case the default constructor is called) or if the class has a public no-argument constructor.
Software Engineering Observation 8.6

Java allows other methods of the class besides its constructors to have the same name as the class and to specify return types. Such methods are not constructors and will not be called when an object of the class is instantiated. Java determines which methods are constructors by locating the methods that have the same name as the class and do not specify a return type.
8.7 Notes on Set and Get Methods

- **Set methods**
  - Also known as mutator methods
  - Assign values to instance variables
  - Should validate new values for instance variables
    - Can return a value to indicate invalid data

- **Get methods**
  - Also known as accessor methods or query methods
  - Obtain the values of instance variables
  - Can control the format of the data it returns
Software Engineering Observation 8.7

When necessary, provide public methods to change and retrieve the values of private instance variables. This architecture helps hide the implementation of a class from its clients, which improves program modifiability.
Software Engineering Observation 8.8

Class designers need not provide *set* or *get* methods for each *private* field. These capabilities should be provided only when it makes sense.
8.7 Notes on Set and Get Methods (Cont.)

• Predicate methods
  – Test whether a certain condition on the object is true or false and returns the result
  – Example: an `isEmpty` method for a container class (a class capable of holding many objects)

• Encapsulating specific tasks into their own methods simplifies debugging efforts
8.8 Composition

• Composition
  – A class can have references to objects of other classes as members
  – Sometimes referred to as a *has-a* relationship
One form of software reuse is composition, in which a class has as members references to objects of other classes.
// Fig. 8.7: Date.java
// Date class declaration.

public class Date {
    private int month; // 1-12
    private int day;   // 1-31 based on month
    private int year;  // any year

    // constructor: call checkMonth to confirm proper value for month;
    // call checkDay to confirm proper value for day
    public Date( int theMonth, int theDay, int theYear )
    {
        month = checkMonth( theMonth ); // validate month
        year = theYear; // could validate year
        day = checkDay( theDay ); // validate day

        System.out.printf(
            "Date object constructor for date %s\n", this);
    } // end Date constructor

    // methods
// utility method to confirm proper month value
private int checkMonth(int testMonth)
{
    if (testMonth > 0 && testMonth <= 12) // validate month
        return testMonth;
    else // month is invalid
    {
        System.out.printf("Invalid month (%d) set to 1.", testMonth);
        return 1; // maintain object in consistent state
    } // end else
} // end method checkMonth

// utility method to confirm proper day value based on month and year
private int checkDay(int testDay)
{
    int daysPerMonth[] =
    { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
// check if day in range for month
if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
  return testDay;

// check for leap year
if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
          ( year % 4 == 0 && year % 100 != 0 ) ) )
  return testDay;

  System.out.printf( "Invalid day (%d) set to 1.", testDay );
  return 1;  // maintain object in consistent state
}  // end method checkDay

// return a String of the form month/day/year
public String toString()
{
  return String.format( "%d/%d/%d", month, day, year );
}  // end method toString

}  // end class Date

Check if the day is February 29 on a leap year
public class Employee {

private String firstName;
private String lastName;
private Date birthDate;
private Date hireDate;

// constructor to initialize name, birth date and hire date
public Employee( String first, String last, Date dateOfBirth, Date dateOfHire ) {
    firstName = first;
    lastName = last;
    birthDate = dateOfBirth;
    hireDate = dateOfHire;
} // end Employee constructor

// convert Employee to String format
public String toString() {
    return String.format( "%s, %s  Hired: %s  Birthday: %s", lastName, firstName, hireDate, birthDate );
} // end method toString

} // end class Employee

Employee contains references to two Date objects

Implicit calls to hireDate and birthDate’s toString methods
// Fig. 8.9: EmployeeTest.java
// Composition demonstration.

public class EmployeeTest
{
    public static void main( String args[] )
    {
        Date birth = new Date( 7, 24, 1949 );
        Date hire = new Date( 3, 12, 1988 );
        Employee employee = new Employee( "Bob", "Blue", birth, hire );

        System.out.println( employee );
    } // end main
} // end class EmployeeTest

Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988

Create an Employee object

Display the Employee object
8.9 Enumerations

- **enum types**
  - Declared with an `enum` declaration
    - A comma-separated list of `enum` constants
    - Declares an `enum` class with the following restrictions:
      - `enum` types are implicitly `final`
      - `enum` constants are implicitly `static`
      - Attempting to create an object of an `enum` type with `new` is a compilation error
    - `enum` constants can be used anywhere constants can
  - `enum` constructor
    - Like class constructors, can specify parameters and be overloaded
public enum Book
{

    // declare constants of enum type
    JHTP6( "Java How to Program 6e", "2005" ),
    CHTP4( "C How to Program 4e", "2004" ),
    IW3HTP3( "Internet & World Wide Web How to Program 3e", "2004" ),
    CPPHTP4( "C++ How to Program 4e", "2003" ),
    VBHTP2( "Visual Basic .NET How to Program 2e", "2002" ),
    CSHARPHTP( "C# How to Program", "2002" );


    // instance fields
    private final String title; // book title
    private final String copyrightYear; // copyright year

    // enum constructor
    Book( String bookTitle, String year )
    {
        title = bookTitle;
        copyrightYear = year;
    } // end enum Book constructor

Declare six `enum` constants
Arguments to pass to the `enum` constructor
Declare `enum` constructor `Book`
Declare instance variables
// accessor for field title
public String getTitle()
{
    return title;
} // end method getTitle

// accessor for field copyrightYear
public String getCopyrightYear()
{
    return copyrightYear;
} // end method getCopyrightYear
8.9 Enumerations (Cont.)

• **static method values**
  – Generated by the compiler for every `enum`
  – Returns an array of the `enum`’s constants in the order in which they were declared

• **static method range of class EnumSet**
  – Takes two parameters, the first and last `enum` constants in the desired range
  – Returns an `EnumSet` containing the constants in that range, inclusive
  – An enhanced for statement can iterate over an `EnumSet` as it can over an array
// Fig. 8.11: EnumTest.java
// Testing enum type Book.
import java.util.EnumSet;

public class EnumTest
{
    public static void main( String args[] )
    {
        System.out.println( "All books:\n" );

        // print all books in enum Book
        for ( Book book : Book.values() )
            System.out.printf( "%-10s%-45s%s\n", book,
                                book.getTitle(), book.getCopyrightYear() );

        System.out.println( "Display a range of enum constants:\n" );

        // print first four books
        for ( Book book : EnumSet.range( Book.JHTP6, Book.CPPHTP4 ) )
            System.out.printf( "%-10s%-45s%s\n", book,
                                book.getTitle(), book.getCopyrightYear() );
    } // end main
} // end class EnumTest

Enhanced for loop iterates for each enum constant in the array returned by method value

Enhanced for loop iterates for each enum constant in the EnumSet returned by method range
<table>
<thead>
<tr>
<th>All books:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>JHTP6 Java How to Program 6e</td>
<td>2005</td>
</tr>
<tr>
<td>CHTP4 C How to Program 4e</td>
<td>2004</td>
</tr>
<tr>
<td>IWBHTP3 Internet &amp; World Wide Web How to Program 3e</td>
<td>2004</td>
</tr>
<tr>
<td>CPPHTP4 C++ How to Program 4e</td>
<td>2003</td>
</tr>
<tr>
<td>VBHTP2 Visual Basic .NET How to Program 2e</td>
<td>2002</td>
</tr>
<tr>
<td>CSHARPHTP C# How to Program</td>
<td>2002</td>
</tr>
</tbody>
</table>

Display a range of enum constants:

<table>
<thead>
<tr>
<th>All books:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>JHTP6 Java How to Program 6e</td>
<td>2005</td>
</tr>
<tr>
<td>CHTP4 C How to Program 4e</td>
<td>2004</td>
</tr>
<tr>
<td>IWBHTP3 Internet &amp; World Wide Web How to Program 3e</td>
<td>2004</td>
</tr>
<tr>
<td>CPPHTP4 C++ How to Program 4e</td>
<td>2003</td>
</tr>
</tbody>
</table>
In an `enum` declaration, it is a syntax error to declare `enum` constants after the `enum` type’s constructors, fields and methods in the `enum` declaration.
8.10 Garbage Collection and Method finalize

• Garbage collection
  – JVM marks an object for garbage collection when there are no more references to that object
  – JVM’s garbage collector will retrieve those objects memory so it can be used for other objects

• finalize method
  – All classes in Java have the finalize method
    • Inherited from the Object class
  – finalize is called by the garbage collector when it performs termination housekeeping
  – finalize takes no parameters and has return type void
A class that uses system resources, such as files on disk, should provide a method to eventually release the resources. Many Java API classes provide `close` or `dispose` methods for this purpose. For example, class `Scanner` (java.sun.com/javase/6/docs/api/java/util/Scanner.html) has a `close` method.
8.11 static Class Members

• static fields
  – Also known as class variables
  – Represents class-wide information
  – Used when:
    • all objects of the class should share the same copy of this
      instance variable or
    • this instance variable should be accessible even when no
      objects of the class exist
  – Can be accessed with the class name or an object name and
    a dot (.)
  – Must be initialized in their declarations, or else the
    compiler will initialize it with a default value (0 for ints)
Software Engineering Observation 8.11

Use a static variable when all objects of a class must use the same copy of the variable.
Software Engineering Observation 8.12

Static class variables and methods exist, and can be used, even if no objects of that class have been instantiated.
// Fig. 8.12: Employee.java
// Static variable used to maintain a count of the number of
// Employee objects in memory.

public class Employee
{
    private String firstName;
    private String lastName;
    private static int count = 0; // number of objects in memory

    // initialize employee, add 1 to static count and
    // output String indicating that constructor was called
    public Employee( String first, String last )
    {
        firstName = first;
        lastName = last;

        count++;  // increment static count of employees
        System.out.printf( "Employee constructor: %s %s; count = %d\n",
            firstName, lastName, count );
    } // end Employee constructor
23  // subtract 1 from static count when garbage
24  // collector calls finalize to clean up object;
25  // confirm that finalize was called
26  protected void finalize() {
27       count--; // decrement static count of employees
28       System.out.printf("Employee finalizer: %s %s; count = %d\n", 
29                firstName, lastName, count );
30  } // end method finalize
31
32  // get first name
33  public String getFirstName() {
34       return firstName;
35  } // end method getFirstName
36
37  // get last name
38  public String getLastName() {
39       return lastName;
40  } // end method getLastName
41
42  // static method to get static count value
43  public static int getCount() {
44       return count;
45  } // end method getCount
46  } // end class Employee

Declare method **finalize**

Declare **static** method **getCount** to get **static** field **count**
public class EmployeeTest {

    public static void main( String args[] ) {

        // show that count is 0 before creating Employees
        System.out.printf( "Employees before instantiation: %d\n", Employee.getCount() );

        // create two Employees; count should be 2
        Employee e1 = new Employee( "Susan", "Baker" );
        Employee e2 = new Employee( "Bob", "Blue" );
    }
}
// show that count is 2 after creating two Employees
System.out.println("\nEmployees after instantiation: ");
System.out.printf("via e1.getCount(): %d\n", e1.getCount());
System.out.printf("via e2.getCount(): %d\n", e2.getCount());
System.out.printf("via Employee.getCount(): %d\n", Employee.getCount());

// get names of Employees
System.out.printf("\nEmployee 1: %s %s\nEmployee 2: %s %s\n", e1.getFirstName(), e1.getLastName(), e2.getFirstName(), e2.getLastName());

// in this example, there is only one reference to each Employee,
// so the following two statements cause the JVM to mark each
// Employee object for garbage collection
e1 = null;
e2 = null;
System.gc(); // ask for garbage collection to occur now
// show Employee count after calling garbage collector; count
// displayed may be 0, 1 or 2 based on whether garbage collector
// executes immediately and number of Employee objects collected
System.out.printf("\nEmployees after System.gc(): %d\n", Employee.getCount());
}
}
}

Call static method getCount

Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2

Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2

Employee 1: Susan Baker
Employee 2: Bob Blue

Employee finalizer: Bob Blue; count = 1
Employee finalizer: Susan Baker; count = 0

Employees after System.gc(): 0
Good Programming Practice 8.1

Invoke every static method by using the class name and a dot (.) to emphasize that the method being called is a static method.
8.11 static Class Members (Cont.)

• **String objects are immutable**
  – String concatenation operations actually result in the creation of a new String object

• **static method gc of class System**
  – Indicates that the garbage collector should make a best-effort attempt to reclaim objects eligible for garbage collection
  – It is possible that no objects or only a subset of eligible objects will be collected

• **static methods cannot access non-static class members**
  – Also cannot use the this reference
Common Programming Error 8.7

A compilation error occurs if a static method calls an instance (non-static) method in the same class by using only the method name. Similarly, a compilation error occurs if a static method attempts to access an instance variable in the same class by using only the variable name.
Common Programming Error 8.8

Referring to this in a static method is a syntax error.
8.12 *static Import*

- **static import declarations**
  - Enables programmers to refer to imported *static* members as if they were declared in the class that uses them
  - **Single static import**
    - `import static packageName.ClassName.staticMemberName;`
  - **static import on demand**
    - `import static packageName.ClassName.*;`
      - Imports all *static* members of the specified class
// Fig. 8.14: StaticImportTest.java

// Using static import to import static methods of class Math.

import static java.lang.Math.*;

public class StaticImportTest {
    public static void main( String args[] ) {
        System.out.printf( "sqrt( 900.0 ) = %.1f\n", sqrt( 900.0 ) );
        System.out.printf( "ceil( -9.8 ) = %.1f\n", ceil( -9.8 ) );
        System.out.printf( "log( E ) = %.1f\n", log( E ) );
        System.out.printf( "cos( 0.0 ) = %.1f\n", cos( 0.0 ) );
    } // end main
} // end class StaticImportTest

sqrt( 900.0 ) = 30.0
ceil( -9.8 ) = -9.0
log( E ) = 1.0
cos( 0.0 ) = 1.0

Use Math's static methods and instance variable without preceding them with Math.
A compilation error occurs if a program attempts to import `static` methods that have the same signature or `static` fields that have the same name from two or more classes.
8.13 final Instance Variables

• Principle of least privilege
  – Code should have only the privilege and access it needs to
    accomplish its task, but no more

• final instance variables
  – Keyword final
    • Specifies that a variable is not modifiable (is a constant)
  – final instance variables can be initialized at their
    declaration
    • If they are not initialized in their declarations, they must be
      initialized in all constructors
Declaring an instance variable as `final` helps enforce the principle of least privilege. If an instance variable should not be modified, declare it to be `final` to prevent modification.
// Fig. 8.15: Increment.java
// final instance variable in a class.

public class Increment
{
    private int total = 0; // total of all increments
    private final int INCREMENT; // constant variable (uninitialized)

    // constructor initializes final instance variable INCREMENT
    public Increment( int incrementValue )
    {
        INCREMENT = incrementValue; // initialize constant variable (once)
    } // end Increment constructor

    // add INCREMENT to total
    public void addIncrementToTotal()
    {
        total += INCREMENT;
    } // end method addIncrementToTotal

    // return String representation of an Increment object's data
    public String toString()
    {
        return String.format( "total = %d", total );
    } // end method toString

} // end class Increment
public class IncrementTest
{
  public static void main( String args[] )
  {
    Increment value = new Increment( 5 );
    System.out.printf( "Before incrementing: %s\n\n", value );
    for ( int i = 1; i <= 3; i++ )
    {
      value.addIncrementToTotal();
      System.out.printf( "After increment %d: %s\n", i, value );
    } // end for
  } // end main
} // end class IncrementTest

Before incrementing: total = 0
After increment 1: total = 5
After increment 2: total = 10
After increment 3: total = 15
Common Programming Error 8.10

Attempting to modify a final instance variable after it is initialized is a compilation error.
Error-Prevention Tip 8.2

Attempts to modify a final instance variable are caught at compilation time rather than causing execution-time errors. It is always preferable to get bugs out at compilation time, if possible, rather than allow them to slip through to execution time (where studies have found that the cost of repair is often many times more expensive).
Software Engineering Observation 8.14

A final field should also be declared static if it is initialized in its declaration. Once a final field is initialized in its declaration, its value can never change. Therefore, it is not necessary to have a separate copy of the field for every object of the class. Making the field static enables all objects of the class to share the final field.
Common Programming Error 8.11

Not initializing a final instance variable in its declaration or in every constructor of the class yields a compilation error indicating that the variable might not have been initialized. The same error occurs if the class initializes the variable in some, but not all, of the class’s constructors.
Increment.java:13: variable INCREMENT might not have been initialized
} // end Increment constructor
^ 1 error
8.14 **Software Reusability**

- **Rapid application development**
  - Software reusability speeds the development of powerful, high-quality software

- **Java’s API**
  - provides an entire framework in which Java developers can work to achieve true reusability and rapid application development
  - Documentation:
    - [java.sun.com/javase/6/docs/api/](http://java.sun.com/javase/6/docs/api/)
    - Or [http://java.sun.com/javase/downloads/index.jsp](http://java.sun.com/javase/downloads/index.jsp) to download
8.15 Data Abstraction and Encapsulation

• Data abstraction
  – Information hiding
    • Classes normally hide the details of their implementation from their clients
  – Abstract data types (ADTs)
    • Data representation
      – example: primitive type int is an abstract representation of an integer
        • ints are only approximations of integers, can produce arithmetic overflow
    • Operations that can be performed on data
Avoid reinventing the wheel. Study the capabilities of the Java API. If the API contains a class that meets your program’s requirements, use that class rather than create your own.
8.15 Data Abstraction and Encapsulation (Cont.)

- **Queues**
  - Similar to a “waiting line”
    - Clients place items in the queue (enqueue an item)
    - Clients get items back from the queue (dequeue an item)
    - First-in, first out (FIFO) order
  - Internal data representation is hidden
    - Clients only see the ability to enqueue and dequeue items
Programmers create types through the class mechanism. New types can be designed to be as convenient to use as the built-in types. This marks Java as an extensible language. Although the language is easy to extend via new types, the programmer cannot alter the base language itself.
8.16 Time Class Case Study: Creating Packages

• To declare a reusable class
  – Declare a public class
  – Add a package declaration to the source-code file
    • must be the first executable statement in the file
    • package name should consist of your Internet domain name in reverse order followed by other names for the package
      – example: com.deitel.jhtp7.ch08
    – package name is part of the fully qualified class name
      • Distinguishes between multiple classes with the same name belonging to different packages
      • Prevents name conflict (also called name collision)
    – Class name without package name is the simple name
Time1.java

1 // Fig. 8.18: Time1.java
2 // Time1 class declaration maintains the time in 24-hour format.
3 package com.deitel.jhtp7.ch08;
4
5 public class Time1
6 {
7     private int hour;   // 0 - 23
8     private int minute; // 0 - 59
9     private int second; // 0 - 59
10
11     // set a new time value using universal time; perform
12     // validity checks on the data; set invalid values to zero
13     public void setTime( int h, int m, int s )
14     {
15         hour = ( ( h >= 0 && h < 24 ) ? h : 0 );   // validate hour
16         minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
17         second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
18     } // end method setTime
19
20 }

Outline
// convert to String in universal-time format (HH:MM:SS)
public String toUniversalString()
{
    return String.format( "%02d:%02d:%02d", hour, minute, second );
} // end method toUniversalString

// convert to String in standard-time format (H:MM:SS AM or PM)
public String toString()
{
    return String.format( "%d:%02d:%02d %s",
        ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
        minute, second, ( hour < 12 ? "AM" : "PM" ) );
} // end method toString

} // end class Time1
8.16 Time Class Case Study: Creating Packages (Cont.)

- Compile the class so that it is placed in the appropriate package directory structure
  - Example: our package should be in the directory
    
    \[
    \text{com} \\
    \downarrow \text{deitel} \\
    \downarrow \text{jhpt7} \\
    \downarrow \text{ch08}
    \]
  
  - javac command-line option `-d`
    - javac creates appropriate directories based on the class’s package declaration
    - A period (.) after `-d` represents the current directory
8.16 Time Class Case Study: Creating Packages (Cont.)

- Import the reusable class into a program
  - Single-type-import declaration
    - Imports a single class
    - Example: import java.util.Random;
  - Type-import-on-demand declaration
    - Imports all classes in a package
    - Example: import java.util.*;
Common Programming Error 8.12

Using the `import declaration import java.*;` causes a compilation error. You must specify the exact name of the package from which you want to import classes.
// Fig. 8.19: Time1PackageTest.java
// Time1 object used in an application.
import com.deitel.jhtp6.ch08.Time1; // import class Time1

public class Time1PackageTest
{
    public static void main( String args[] )
    {
        // create and initialize a Time1 object
        Time1 time = new Time1(); // calls Time1 constructor

        // output string representations of the time
        System.out.println( "The initial universal time is: ");
        System.out.println( time.toUniversalString() );
        System.out.println( "The initial standard time is: ");
        System.out.println( time.toString() );
        System.out.println(); // output a blank line
    }
}

Outline

Single-type import declaration

Refer to the Time1 class by its simple name
// change time and output updated time
time.setTime(13, 27, 6);
System.out.print("Universal time after setTime is: ");
System.out.println(time.toUniversalString());
System.out.print("Standard time after setTime is: ");
System.out.println(time.toString());
System.out.println(); // output a blank line

// set time with invalid values; output updated time
time.setTime(99, 99, 99);
System.out.println("After attempting invalid settings:");
System.out.print("Universal time: ");
System.out.println(time.toUniversalString());
System.out.print("Standard time: ");
System.out.println(time.toString());

} // end main

} // end class Time1PackageTest

The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
8.16 Time Class Case Study: Creating Packages (Cont.)

• Class loader
  – Locates classes that the compiler needs
    • First searches standard Java classes bundled with the JDK
    • Then searches for optional packages
      – These are enabled by Java’s extension mechanism
    • Finally searches the classpath
      – List of directories or archive files separated by directory separators
        • These files normally end with .jar or .zip
        • Standard classes are in the archive file rt.jar
8.16 **Time Class Case Study: Creating Packages (Cont.)**

- To use a classpath other than the current directory
  - `-classpath` option for the `javac` compiler
  - Set the `CLASSPATH` environment variable
- The JVM must locate classes just as the compiler does
  - The `java` command can use other classpaths by using the same techniques that the `javac` command uses
Common Programming Error 8.13

Specifying an explicit classpath eliminates the current directory from the classpath. This prevents classes in the current directory (including packages in the current directory) from loading properly. If classes must be loaded from the current directory, include a dot (.) in the classpath to specify the current directory.
Software Engineering Observation 8.16

In general, it is a better practice to use the `-classpath` option of the compiler, rather than the `CLASSPATH` environment variable, to specify the classpath for a program. This enables each application to have its own classpath.
Error-Prevention Tip 8.3

Specifying the classpath with the `CLASSPATH` environment variable can cause subtle and difficult-to-locate errors in programs that use different versions of the same package.
8.17 Package Access

• Package access
  – Methods and variables declared without any access modifier are given package access
  – This has no effect if the program consists of one class
  – This does have an effect if the program contains multiple classes from the same package
    • Package-access members can be directly accessed through the appropriate references to objects in other classes belonging to the same package
// Fig. 8.20: PackageDataTest.java  
// Package-access members of a class are accessible by other classes  
// in the same package.

public class PackageDataTest {
    public static void main( String args[] ) {
        PackageData packageData = new PackageData();

        // output String representation of packageData
        System.out.printf( "After instantiation:
%s
", packageData );

        // change package access data in packageData object
        packageData.number = 77;
        packageData.string = "Goodbye";

        // output String representation of packageData
        System.out.printf( "
After changing values:
%s
", packageData );
    } // end main
} // end class PackageDataTest

Can directly access package-access members
// class with package access instance variables

class PackageData {

    int number; // package-access instance variable
    String string; // package-access instance variable

    // constructor
    public PackageData() {
        number = 0;
        string = "Hello";
    } // end PackageData constructor

    // return PackageData object String representation
    public String toString() {
        return String.format( "number: %d; string: %s", number, string );
    } // end method toString

} // end class PackageData

After instantiation:
number: 0; string: Hello

After changing values:
number: 77; string: Goodbye
8.18  (Optional) GUI and Graphics Case Study: Using Objects with Graphics

• To create a consistent drawing that remains the same each time it is drawn
  – Store information about the displayed shapes so that they can be reproduced exactly the same way each time `paintComponent` is called
public class MyLine
{
    private int x1; // x coordinate of first endpoint
    private int y1; // y coordinate of first endpoint
    private int x2; // x coordinate of second endpoint
    private int y2; // y coordinate of second endpoint
    private Color myColor; // color of this shape

    public MyLine( int x1, int y1, int x2, int y2, Color color )
    {
        this.x1 = x1; // set x coordinate of first endpoint
        this.y1 = y1; // set y coordinate of first endpoint
        this.x2 = x2; // set x coordinate of second endpoint
        this.y2 = y2; // set y coordinate of second endpoint
        myColor = color; // set the color
    } // end MyLine constructor

    public void draw( Graphics g )
    {
        g.setColor( myColor );
        g.drawLine( x1, y1, x2, y2 );
    } // end method draw
} // end class MyLine
1 // Fig. 8.22: DrawPanel.java
2 // Program that uses class MyLine
3 // to draw random lines.
4 import java.awt.Color;
5 import java.awt.Graphics;
6 import java.util.Random
7 import javax.swing.JPanel;
8
9 public class DrawPanel extends JPanel
10 {
11    private Random randomNumbers = new Random();
12    private MyLine lines[]; // array of lines
13
14    // constructor, creates a panel with random shapes
15    public DrawPanel()
16    {
17        setBackground( Color.WHITE );
18
19        lines = new MyLine[ 5 + randomNumbers.nextInt( 5 ) ];
20    }
```java
// create lines
for ( int count = 0; count < lines.length; count++ )
{
    // generate random coordinates
    int x1 = randomNumbers.nextInt( 300 );
    int y1 = randomNumbers.nextInt( 300 );
    int x2 = randomNumbers.nextInt( 300 );
    int y2 = randomNumbers.nextInt( 300 );

    // generate a random color
    Color color = new Color( randomNumbers.nextInt( 256 ),
                            randomNumbers.nextInt( 256 ),
                            randomNumbers.nextInt( 256 ) );

    // add the line to the list of lines to be displayed
    lines[ count ] = new MyLine( x1, y1, x2, y2, color );
} // end for
} // end DrawPanel constructor

// for each shape array, draw the individual shapes
public void paintComponent( Graphics g )
{
    super.paintComponent( g );

    // draw the lines
    for ( MyLine line : lines )
        line.draw( g );
} // end method paintComponent

// end class DrawPanel
```

Generate coordinates for this line

Generate a color for this line

Create the new `MyLine` object with the generated attributes

Draw each `MyLine`
```java
// Fig. 8.23: TestDraw.java
// Test application to display a DrawPanel.
import javax.swing.JFrame;

public class TestDraw
{
    public static void main( String args[] )
    {
        DrawPanel panel = new DrawPanel();
        JFrame application = new JFrame();

        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        application.add( panel );
        application.setSize( 300, 300 );
        application.setVisible( true );
    } // end main
} // end class TestDraw
```
8.19 Starting to Program the Classes of the ATM System

• Visibility
  – Attributes normally should be private, methods invoked by clients should be public
  – Visibility markers in UML
    • A plus sign (+) indicates public visibility
    • A minus sign (-) indicates private visibility

• Navigability
  – Navigability arrows indicate in which direction an association can be traversed
  – Bidirectional navigability
    • Associations with navigability arrows at both ends or no navigability arrows at all can be traversed in either direction
8.19 Starting to Program the Classes of the ATM System (Cont.)

• Implementing the ATM system from its UML design (for each class)
  – Declare a `public` class with the name in the first compartment and an empty no-argument constructor
  – Declare instance variables based on attributes in the second compartment
  – Declare references to other objects based on associations described in the class diagram
  – Declare the shells of the methods based on the operations in the third compartment
    • Use the return type `void` if no return type has been specified
Fig. 8.24 | Class diagram with visibility markers.
Fig. 8.25 | Class diagram with navigability arrows.
// Class Withdrawal represents an ATM withdrawal transaction

public class Withdrawal {
    // no-argument constructor
    public Withdrawal () {
        // end no-argument Withdrawal constructor
    } // end class Withdrawal
// Class Withdrawal represents an ATM withdrawal transaction

public class Withdrawal
{
    // attributes
    private int accountNumber; // account to withdraw funds from
    private double amount; // amount to withdraw

    // no-argument constructor
    public Withdrawal()
    {
        // end no-argument Withdrawal constructor
    } // end class Withdrawal

Declare instance variables
// Class Withdrawal represents an ATM withdrawal transaction

public class Withdrawal
{

    // attributes
    private int accountNumber; // account to withdraw funds from
    private double amount; // amount to withdraw

    // references to associated objects
    private Screen screen; // ATM's screen
    private Keypad keypad; // ATM's keypad
    private CashDispenser cashDispenser; // ATM's cash dispenser
    private BankDatabase bankDatabase; // account info database

    // no-argument constructor
    public Withdrawal()
    {
    } // end no-argument Withdrawal constructor

} // end class Withdrawal

Declare references to other objects
public class Withdrawal {

    // attributes
    private int accountNumber; // account to withdraw funds from
    private double amount; // amount to withdraw

    // references to associated objects
    private Screen screen; // ATM's screen
    private Keypad keypad; // ATM's keypad
    private CashDispenser cashDispenser; // ATM's cash dispenser
    private BankDatabase bankDatabase; // account info database

    // no-argument constructor
    public Withdrawal() {
    }

    // operations
    public void execute() {
    }
public class Keypad {

    // no attributes have been specified yet

    // no-argument constructor
    public Keypad() {
    }

    // operations
    public int getInput() {
    }

    } // end class Keypad