Chapter 5 – Object-Based Programming

- ✓ **Introduction**
- ✓ **Implementing a Time Abstract Data Type with a Class**
- ✓ **Class Scope**
- ✓ **Controlling Access to Members**
- ✓ **Referring to the Current Object's Members with `this`**
- ✓ **Initializing Class Objects: Constructors**
- ✓ **Using Overloaded Constructors**
- ✓ **Using Set and Get Methods**
- ✓ **Composition**
- ✓ **Garbage Collection**
- ✓ **Static Class Members**
- ✓ **Final Instance Variables**
- ✓ **Creating Packages**
- ✓ **Package Access**

Introduction

- Object Oriented Programming (OOP)
  - *Encapsulates* data (attributes) and methods (behaviors)
    - Objects
  - Allows objects to communicate
    - Well-defined *interfaces*

Introduction (cont.)

- Procedural programming language
  - C is an example
  - Action-oriented
  - Functions are units of programming
- Object-oriented programming language
  - Java is an example
  - Object-oriented
  - Classes are units of programming
    - Functions, or *methods*, are encapsulated in classes

Implementing a Time Abstract Data Type with a Class

- We introduce classes `Time1` and `TimeTest`
  - `Time1.java` declares class `Time1`
  - `TimeTest.java` declares class `TimeTest`
  - `public` classes must be declared in separate files
  - Class `Time1` will not execute by itself
    - Does not have method `main`
    - `TimeTest`, which has method `main`, creates (*instantiates*) and uses `Time1` object

```
1     // Fig. 8.1: Time1.java
2     // Time1 class declaration maintains the time in 24-hour format.
3     import java.text.DecimalFormat;
4
5     public class Time1 extends Object {
6        private int hour;     // 0 - 23
7        private int minute;   // 0 - 59
8        private int second;   // 0 - 59
9
10       // Time1 constructor initializes each instance variable to zero;
11       // ensures that each Time1 object starts in a consistent state
12       public Time1()
13       {
14          setTime( 0, 0, 0 );
15       }
16
17       // set a new time value using universal tim
18       // validity checks on the data; set invalid
19       public void setTime( int h, int m, int s )
20       {
21          hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
22          minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
23          second = ( ( s >= 0 && s < 60 ) ? s : 0 );
24       }
25
```

Time1 (subclass) extends superclass java.lang.Object (Chapter 9 discusses inheritance)

private variables (and methods) are accessible only to methods in this class

Time1 constructor creates Time1 object then invokes method setTime

public methods (and variables) are accessible wherever program has Time1 reference

```
26       // convert to String in universal-time format
27       public String toUniversalString()
28       {
29          DecimalFormat twoDigits = new DecimalFormat( "00" );
30
31          return twoDigits.format( hour ) + ":" +
32             twoDigits.format( minute ) + ":" + twoDigits.format( second );
33       }
34
35       // convert to String in standard-time format
36       public String toStandardString()
37       {
38          DecimalFormat twoDigits = new DecimalFormat( "00" );
39
40          return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) + ":" +
41             twoDigits.format( minute ) + ":" + twoDigits.format( second ) +
42             ( hour < 12 ? " AM" : " PM" );
43       }
44
45    } // end class Time1
```

Implementing a Time Abstract Data Type with a Class (cont.)

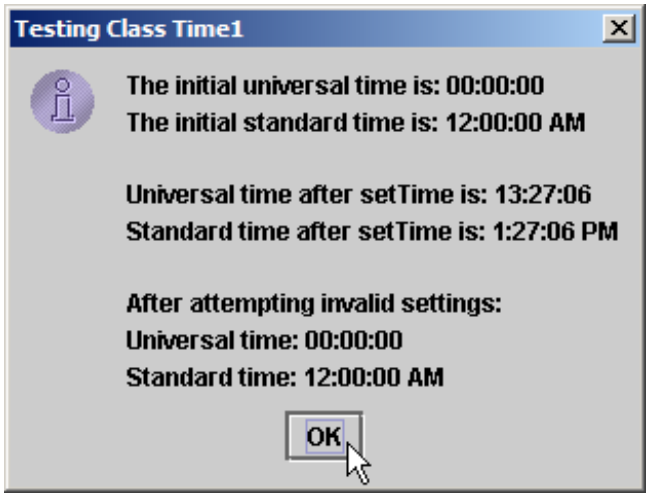
- Every Java class must extend another class
  - `Time1` extends `java.lang.Object`
  - If class does not explicitly `extend` another class
    - class implicitly extends `Object`
- Class *constructor*
  - Same name as class
  - Initializes instance variables of a class object
  - Called when program instantiates an object of that class
  - Can take arguments, but *cannot return values*
  - Class can have several constructors, through *overloading*
  - Class `Time1` constructor(lines 12-15)

```
1   // Fig. 8.2: TimeTest1.java
2   // Class TimeTest1 to exercise class
3   import javax.swing.JOptionPane;
4
5   public class TimeTest1 {
6
7      public static void main( String args[] )
8      {
9         Time1 time = new Time1();  // calls Time1
10
11       // append String version of time to String output
12       String output = "The initial universal time is: " +
13          time.toUniversalString() + "\nThe initial standard time is: " +
14          time.toStandardString();
15
16       // change time and append updated time to output
17       time.setTime( 13, 27, 6 );
18       output += "\n\nUniversal time after setTime is: " +
19          time.toUniversalString() +
20          "\nStandard time after setTime is: " + time.toStandardString();
21
22       // set time with invalid values; append updated time to output
23       time.setTime( 99, 99, 99 );
24       output += "\n\nAfter attempting invalid settings: " +
25          "\nUniversal time: " + time.toUniversalString() +
26          "\nStandard time: " + time.toStandardString();
27
```

Declare and create instance of class `Time1` by calling `Time1` constructor

`TimeTest1` interacts with `Time1` by calling `Time1` `public` methods

```
28          JOptionPane.showMessageDialog( null, output,
29             "Testing Class Time1", JOptionPane.INFORMATION_MESSAGE );
30
31          System.exit( 0 );
32
33       } // end main
34
35    } // end class TimeTest1
```



Class Scope

- Class scope
  - Class variables and methods
  - Members are accessible to all class methods
  - Members can be referenced by name
    - objectReferenceName.objectMemberName
  - Shadowed (hidden) class variables
    - `this`.variableName

Controlling Access to Members

- Member access modifiers
  - Control access to class's variables and methods
  - `public`
    - Variables and methods accessible to clients of the class
  - `private`
    - Variables and methods not accessible to clients of the class

```
1      // Fig. 8.3: TimeTest2.java
2      // Errors resulting from attempts to access private members of Time1.
3      public class TimeTest2 {
4
5         public static void main( String args[] )
6         {
7            Time1 time = new Time1();
8
9            time.hour = 7;    // error: hour is a private instance variable
10           time.minute = 15; // error: minute is a private instance variable
11           time.second = 30; // error: second is a private instance variable
12        }
13
14    } // end class TimeTest2
```

Compiler error – `TimeTest2` cannot directly access `Time1`'s `private` data

```
TimeTest2.java:9: hour has private access in Time1
      time.hour = 7;    // error: hour is a private instance variable
          ^
TimeTest2.java:10: minute has private access in Time1
      time.minute = 15; // error: minute is a private instance variable
          ^
TimeTest2.java:11: second has private access in Time1
      time.second = 30; // error: second is a private instance variable
          ^
3 errors
```

Referring to the Current Object's Members with `this`

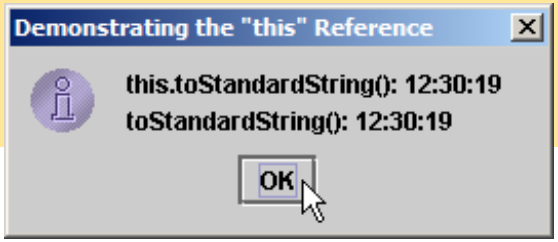
- Keyword `this` (*`this` reference*)
  - Allows an object to refers to itself

```java
// Fig. 8.4: ThisTest.java
// Using the this reference to refer to instance variables and methods.
import javax.swing.*;
import java.text.DecimalFormat;

public class ThisTest {

   public static void main( String args[] )
   {
      SimpleTime time = new SimpleTime( 12, 30, 19 );

      JOptionPane.showMessageDialog( null, time.buildString(),
         "Demonstrating the \"this\" Reference",
         JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );
   }

} // end class ThisTest

// class SimpleTime demonstrates the "this" reference
class SimpleTime {
   private int hour;
   private int minute;
   private int second;

```

```
27       // constructor uses parameter names identical to instance variable
28       // names; "this" reference required to distinguish between names
29       public SimpleTime( int hour, int minute, int second )
30       {
31          this.hour = hour;       // set "this" object's
32          this.minute = minute;   // set "this" object's
33          this.second = second;   // set "this" object's
34       }
35
36       // use explicit and implicit "this" to call toStandardString
37       public String buildString()
38       {
39          return "this.toStandardString(): " + this.toStandardString() +
40             "\ntoStandardString(): " + toStandardString();
41       }
42
43       // return String representation of SimpleTime
44       public String toStandardString()
45       {
46          DecimalFormat twoDigits = new DecimalFormat( "00" );
47
48          // "this" is not required here, because method does not
49          // have local variables with same names as instance variables
50          return twoDigits.format( this.hour ) + ":" +
51             twoDigits.format( this.minute ) + ":" +
52             twoDigits.format( this.second );
53       }
54
55    } // end class SimpleTime
```

`this` used to distinguish between arguments and `ThisTest` variables

Use explicit and implicit this to call toStandardString



Initializing Class Objects: Constructors

- Class constructor
  - Same name as class
  - Initializes instance variables of a class object
  - Call class constructor to instantiate object of that class

```
new ClassName( argument1, argument2, …, arugmentN );
```

  - `new` indicates that new object is created
  - *ClassName* indicates type of object created
  - *arguments* specifies constructor argument values

Using Overloaded Constructors

- Overloaded constructors
  - Methods (in same class) may have same name
  - Must have different parameter lists

```
1   // Fig. 8.5: Time2.java
2   // Time2 class declaration with overloaded constructors.
3   import java.text.DecimalFormat;
4
5   public class Time2 {
6      private int hour;    // 0 - 23
7      private int minute;  // 0 - 59
8      private int second;  // 0 - 59
9
10     // Time2 constructor initializes each
11     // ensures that Time object starts in a consistent state
12     public Time2()
13     {
14        this( 0, 0, 0 ); // invoke Time2 constr          s
15     }
16
17     // Time2 constructor: hour supplied, minute and second defaulted to 0
18     public Time2( int h )
19     {
20        this( h, 0, 0 ); // invoke Time2 con
21     }
22
23     // Time2 constructor: hour and minute supplied, second defaulted to 0
24     public Time2( int h, int m )
25     {
26        this( h, m, 0 ); // invoke Time2 constructor with three arguments
27     }
28
```

No-argument (default) constructor

Use this to invoke the Time2 constructor declared at lines 30-33

Overloaded constructor has one `int` argument

Second overloaded constructor has two `int` arguments

```
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       }
34
35       // Time2 constructor: another Time2 object supplied
36       public Time2( Time2 time )
37       {
38          // invoke Time2 constructor with three arguments
39          this( time.hour, time.minute, time.second );
40       }
41
42       // set a new time value using universal time; perform
43       // validity checks on data; set invalid values to zero
44       public void setTime( int h, int m, int s )
45       {
46          hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
47          minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
48          second = ( ( s >= 0 && s < 60 ) ? s : 0 );
49       }
50
51       // convert to String in universal-time format
52       public String toUniversalString()
53       {
54          DecimalFormat twoDigits = new DecimalFormat( "00" );
55
56          return twoDigits.format( hour ) + ":" +
57             twoDigits.format( minute ) + ":" + twoDigits.format( second );
58       }
```

Third overloaded constructor has three `int` arguments

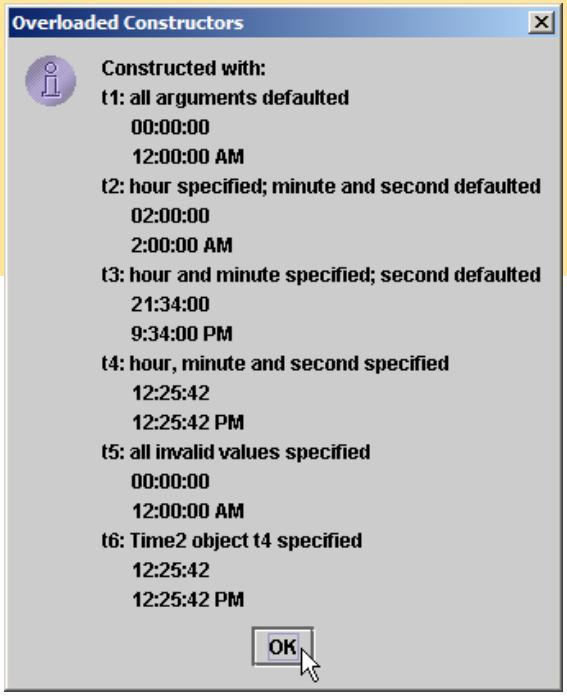
Fourth overloaded constructor has `Time2` argument

```
59
60       // convert to String in standard-time format
61       public String toStandardString()
62       {
63          DecimalFormat twoDigits = new DecimalFormat( "00" );
64
65          return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) + ":" +
66             twoDigits.format( minute ) + ":" + twoDigits.format( second ) +
67             ( hour < 12 ? " AM" : " PM" );
68       }
69
70    } // end class Time2
```

```
1  // Fig. 8.6: TimeTest3.java
2  // Overloaded constructors used to initialize Time2 objects.
3  import javax.swing.*;
4
5  public class TimeTest3 {
6
7     public static void main( String args[] )
8     {
9        Time2 t1 = new Time2();           // 00:00:00
10       Time2 t2 = new Time2( 2 );        // 02:00:00
11       Time2 t3 = new Time2( 21, 34 );   // 21:34:00
12       Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
13       Time2 t5 = new Time2( 27, 74, 99 ); // 00:00:00
14       Time2 t6 = new Time2( t4 );       // 12:25:42
15
16       String output = "Constructed with: " +
17          "\nt1: all arguments defaulted" +
18          "\n      " + t1.toUniversalString() +
19          "\n      " + t1.toStandardString();
20
21       output += "\nt2: hour specified; minute and second defaulted" +
22          "\n      " + t2.toUniversalString() +
23          "\n      " + t2.toStandardString();
24
25       output += "\nt3: hour and minute specified; second defaulted" +
26          "\n      " + t3.toUniversalString() +
27          "\n      " + t3.toStandardString();
```

Instantiate each `Time2` reference using a different constructor

```
28
29          output += "\nt4: hour, minute and second specified" +
30             "\n      " + t4.toUniversalString() +
31             "\n      " + t4.toStandardString();
32
33          output += "\nt5: all invalid values specified" +
34             "\n      " + t5.toUniversalString() +
35             "\n      " + t5.toStandardString();
36
37          output += "\nt6: Time2 object t4 specified" +
38             "\n      " + t6.toUniversalString() +
39             "\n      " + t6.toStandardString();
40
41          JOptionPane.showMessageDialog( null, output,
42             "Overloaded Constructors", JOptionPane.INFORMATION_MESSAGE );
43
44          System.exit( 0 );
45
46       } // end main
47
48    } // end class TimeTest3
```



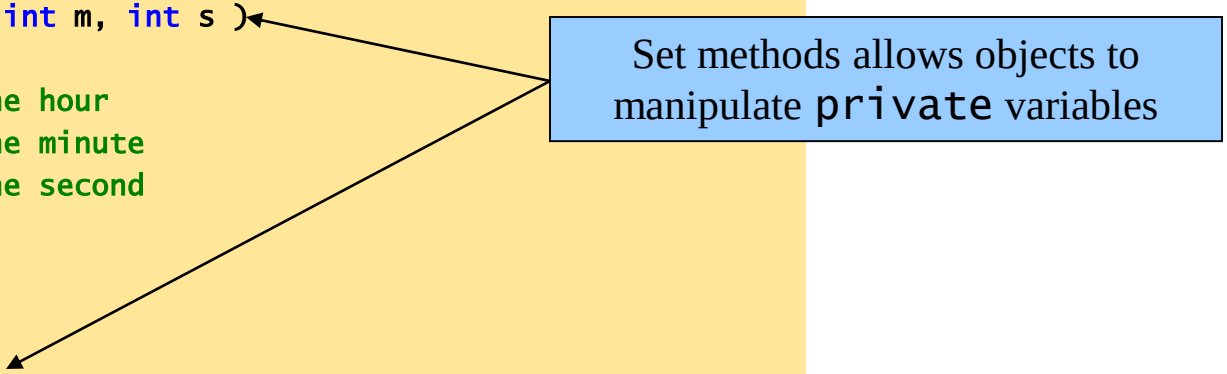
Using Set and Get Methods

- Accessor method ("get" method)
  - `public` method
  - Allow clients to read `private` data
- Mutator method ("set" method)
  - `public` method
  - Allow clients to modify `private` data

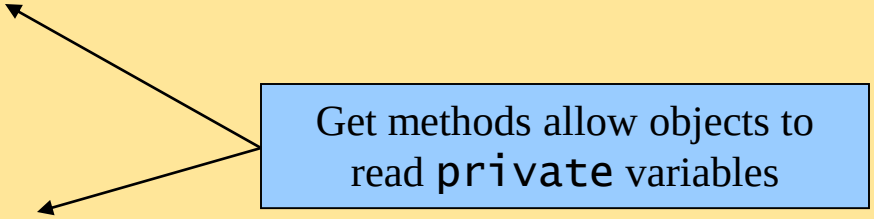
```
1     // Fig. 8.7: Time3.java
2     // Time3 class declaration with set and get methods.
3     import java.text.DecimalFormat;
4
5     public class Time3 {
6        private int hour;     // 0 - 23
7        private int minute;   // 0 - 59
8        private int second;   // 0 - 59
9
10       // Time3 constructor initializes each
11       // ensures that Time object starts in a consistent state
12       public Time3()
13       {
14          this( 0, 0, 0 ); // invoke Time3 constructor with three arguments
15       }
16
17       // Time3 constructor: hour supplied, minute and second defaulted to 0
18       public Time3( int h )
19       {
20          this( h, 0, 0 ); // invoke Time3 constructor with three arguments
21       }
22
23       // Time3 constructor: hour and minute supplied, second defaulted to 0
24       public Time3( int h, int m )
25       {
26          this( h, m, 0 ); // invoke Time3 constructor with three arguments
27       }
28
```

`private` variables cannot be accessed directly by objects in different classes

```
29       // Time3 constructor: hour, minute and second supplied
30       public Time3( int h, int m, int s )
31       {
32          setTime( h, m, s );
33       }
34
35       // Time3 constructor: another Time3 object supplied
36       public Time3( Time3 time )
37       {
38          // invoke Time3 constructor with three arguments
39          this( time.getHour(), time.getMinute(), time.getSecond() );
40       }
41
42       // Set Methods
43       // set a new time value using universal time; perform
44       // validity checks on data; set invalid values to zero
45       public void setTime( int h, int m, int s )
46       {
47          setHour( h );   // set the hour
48          setMinute( m ); // set the minute
49          setSecond( s ); // set the second
50       }
51
52       // validate and set hour
53       public void setHour( int h )
54       {
55          hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
56       }
57
```



```
58       // validate and set minute
59       public void setMinute( int m )
60       {
61          minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
62       }
63
64       // validate and set second
65       public void setSecond( int s )
66       {
67          second = ( ( s >= 0 && s < 60 ) ? s : 0 );
68       }
69
70       // Get Methods
71       // get hour value
72       public int getHour()
73       {
74          return hour;
75       }
76
77       // get minute value
78       public int getMinute()
79       {
80          return minute;
81       }
82
```



```
83       // get second value
84       public int getSecond()
85       {
86          return second;
87       }
88
89       // convert to String in universal-time format
90       public String toUniversalString()
91       {
92          DecimalFormat twoDigits = new DecimalFormat( "00" );
93
94          return twoDigits.format( getHour() ) + ":" +
95             twoDigits.format( getMinute() ) + ":" +
96             twoDigits.format( getSecond() );
97       }
98
99       // convert to String in standard-time format
100      public String toStandardString()
101      {
102         DecimalFormat twoDigits = new DecimalFormat( "00" );
103
104         return ( ( getHour() == 12 || getHour() == 0 ) ?
105            12 : getHour() % 12 ) + ":" + twoDigits.format( getMinute() ) +
106            ":" + twoDigits.format( getSecond() ) +
107            ( getHour() < 12 ? " AM" : " PM" );
108      }
109
110   } // end class Time3
```

Composition

- Composition
  - Class contains references to objects of other classes
    - These references are members

```
1  // Fig. 8.9: Date.java
2  // Date class declaration.
3
4  public class Date {
5     private int month;  // 1-12
6     private int day;    // 1-31 based on month
7     private int year;   // any year
8
9     // constructor: call checkMonth to confirm proper value for month;
10    // call checkDay to confirm proper value for day
11    public Date( int theMonth, int theDay, int theYear )
12    {
13       month = checkMonth( theMonth ); // validate month
14       year = theYear;                 // could validate year
15       day = checkDay( theDay );       // validate d
16
17       System.out.println( "Date object constructor
18          toDateString() );
19
20    } // end Date constructor
21
22    // utility method to confirm proper month value
23    private int checkMonth( int testMonth )
24    {
25       if ( testMonth > 0 && testMonth <= 12 )  // validate month
26          return testMonth;
```

Class `Date` encapsulates data that describes date

`Date` constructor instantiates `Date` object based on specified arguments

```
27
28          else { // month is invalid
29             System.out.println( "Invalid month (" + testMonth +
30                ") set to 1." );
31             return 1;  // maintain object in consistent state
32          }
33
34       } // end method checkMonth
35
36       // utility method to confirm proper day value based on month and year
37       private int checkDay( int testDay )
38       {
39          int daysPerMonth[] =
40             { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
41
42          // check if day in range for month
43          if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
44             return testDay;
45
46          // check for leap year
47          if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
48               ( year % 4 == 0 && year % 100 != 0 ) ) )
49             return testDay;
50
51          System.out.println( "Invalid day (" + testDay + ") set to 1." );
52
53          return 1;  // maintain object in consistent state
54
55       } // end method checkDay
```

```
56
57       // return a String of the form month/day/year
58       public String toDateString()
59       {
60          return month + "/" + day + "/" + year;
61       }
62
63    } // end class Date
```

```java
// Fig. 8.10: Employee.java
// Employee class declaration.

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;
   }

   // convert Employee to String format
   public String toEmployeeString()
   {
      return lastName + ", " + firstName +
         "  Hired: " + hireDate.toDateString() +
         "  Birthday: " + birthDate.toDateString();
   }

} // end class Employee
```

Employee is composed of two references to `Date` objects

```
1     // Fig. 8.11: EmployeeTest.java
2     // Demonstrating an object with a member object.
3     import javax.swing.JOptionPane;
4
5     public class EmployeeTest {
6
7        public static void main( String args[] )
8        {
9           Date birth = new Date( 7, 24, 1949 );
10          Date hire = new Date( 3, 12, 1988 );
11          Employee employee = new Employee( "Bob", "Jones", birth, hire );
12
13          JOptionPane.showMessageDialog( null, employee.toEmployeeString(),
14             "Testing Class Employee", JOptionPane.INFORMATION_MESSAGE );
15
16          System.exit( 0 );
17       }
18
19    } // end class EmployeeTest
```


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

Garbage Collection

- Garbage collection
  - Returns memory to system
  - Java performs this automatically
    - object marked for garbage collection if no references to object
- Finalizer method
  - Returns resources to system
  - Java provides method `finalize`
    - Defined in `java.lang.Object`
    - Receives no parameters
    - Returns `void`

Static Class Members

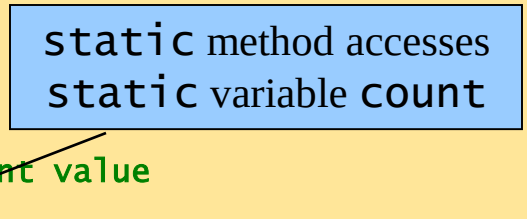
- `static` keyword
  - `static` class variable
    - Class-wide information
      - All class objects share same data
- Access to a class's `public static` members
  - Qualify the member name with the class name and a dot (.)
    - e.g., Math.random()

```
1  // Fig. 8.12: Employee.java
2  // Employee class declaration.
3  public class Employee {
4     private String firstName;
5     private String lastName;
6     private static int count = 0; // number of objects in memory
7
8     // initialize employee, add 1 to static count and
9     // output String indicating that constructor was called
10    public Employee( String first, String last )
11    {
12       firstName = first;
13       lastName = last;
14
15       ++count;  // increment static count of employees
16       System.out.println( "Employee constructor: " +
17          firstName + " " + lastName );
18    }
19
20    // subtract 1 from static count when garba
21    // calls finalize to clean up object and output String
22    // indicating that finalize was called
23    protected void finalize()
24    {
25       --count;  // decrement static count of employees
26       System.out.println( "Employee finalizer: " +
27          firstName + " " + lastName + "; count = " + count );
28    }
29
```

`Employee` objects share one instance of `count`

Called when `Employee` is marked for garbage collection

```
30     // get first name
31     public String getFirstName()
32     {
33        return firstName;
34     }
35
36     // get last name
37     public String getLastName()
38     {
39        return lastName;
40     }
41
42     // static method to get static count value
43     public static int getCount()
44     {
45        return count;
46     }
47
48  } // end class Employee
```



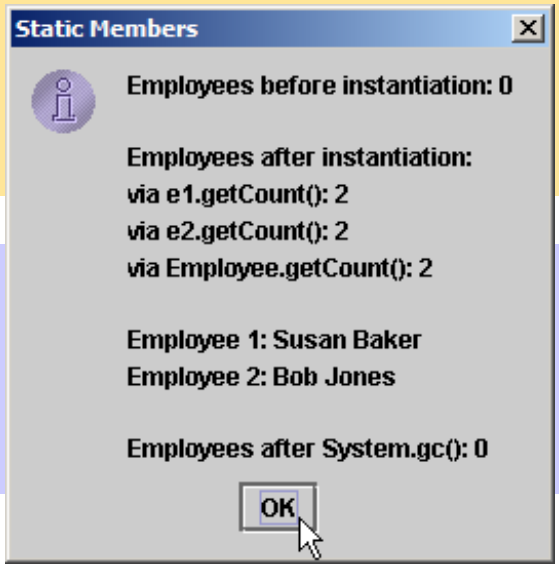
```
1   // Fig. 8.13: EmployeeTest.java
2   // Test Employee class with static class variable,
3   // static class method, and dynamic memory.
4   import javax.swing.*;
5
6   public class EmployeeTest {
7
8      public static void main( String args[] )
9      {
10       // prove that count is 0 before creating Employees
11       String output = "Employees before instantiation: " +
12          Employee.getCount();
13
14       // create two Employees; count should be 2
15       Employee e1 = new Employee( "Susan", "Baker" );
16       Employee e2 = new Employee( "Bob", "Jones" );
17
18       // prove that count is 2 after creating two Employees
19       output += "\n\nEmployees after instantiation: " +
20          "\nvia e1.getCount(): " + e1.getCount() +
21          "\nvia e2.getCount(): " + e2.getCount() +
22          "\nvia Employee.getCount(): " + Employee.getCount();
23
24       // get names of Employees
25       output += "\n\nEmployee 1: " + e1.getFirstName() +
26          " " + e1.getLastName() + "\nEmployee 2: " +
27          e2.getFirstName() + " " + e2.getLastName();
28
```

`EmployeeTest` can invoke `Employee` `static` method, even though `Employee` has not been instantiated

```
29          // decrement reference count for each Employee object; in this
30          // example, there is only one reference to each Employee, so these
31          // statements mark each                                    tion
32          e1 = null;
33          e2 = null;
34
35          System.gc(); // suggest call to garbage collector
36
37          // show Employee count after calling garbage collector; count
38          // displayed may be 0, 1 or 2 based on whether garbage collector
39          // executes immediately and number of Employee objects collected
40          output += "\n\nEmployees after System.gc(): " +
41             Employee.getCount();
42
43          JOptionPane.showMessageDialog( null, output,
44             "Static Members", JOptionPane.INFORMATION_MESSAGE );
45
46          System.exit( 0 );
47       }
48
49    } // end class EmployeeTest
```

Calls Java's automatic garbage-collection mechanism

```
Employee constructor: Susan Baker
Employee constructor: Bob Jones
Employee finalizer: Susan Baker; count = 1
Employee finalizer: Bob Jones; count = 0
```



Final Instance Variables

- `final` keyword
  - Indicates that variable is not modifiable
    - Any attempt to modify `final` variable results in error

```
private final int INCREMENT = 5;
```

    - Declares variable `INCREMENT` as a *constant*

```
32    // class containing constant variable
33    class Increment {
34       private int count = 0;       // number of increments
35       private int total = 0;       // total of all inc
36       private final int INCREMENT; // constant variabl
37
38       // initialize constant INCREMENT
39       public Increment( int incrementValue )
40       {
41          INCREMENT = incrementValue; // intialize constant variable (once)
42       }
43
44       // add INCREMENT to total and add 1 to count
45       public void increment()
46       {
47          total += INCREMENT;
48          ++count;
49       }
50
51       // return String representation of an Increment object's data
52       public String toIncrementString()
53       {
54          return "After increment " + count + ": total = " + total;
55       }
56
57    } // end class Increment
```

`final` keyword declares INCREMENT as constant

`final` variable INCREMENT must be initialized before using it

```
IncrementTest.java:40: variable INCREMENT might not have been initialized
   {
   ^
1 error
```

Creating Packages

- We can `import` *packages* into programs
  - Group of related classes and interfaces
  - Help manage complexity of application components
  - Facilitate software reuse
  - Provide convention for unique class names
    - Popular package-naming convention
      - Reverse Internet domain name
        - e.g., `com.deitel`

```
1  // Fig. 8.16: Time1.java
2  // Time1 class declaration maintains the time
3  package com.deitel.ch08;
4
5  import java.text.DecimalFormat;
6
7  public class Time1 extends Object {
8     private int hour;     // 0 - 23
9     private int minute;   // 0 - 59
10    private int second;   // 0 - 59
11
12    // Time1 constructor initializes each instance variable to zero;
13    // ensures that each Tim                                    te
14    public Time1()
15    {
16       setTime( 0, 0, 0 );
17    }
18
19    // set a new time value using universal time; perform
20    // validity checks on the data; set invalid values to zero
21    public void setTime( int h, int m, int s )
22    {
23       hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
24       minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
25       second = ( ( s >= 0 && s < 60 ) ? s : 0 );
26    }
27
```

Class `Time1` is placed in this `package`

Class `Time1` is in directory `com/deitel/ch08`

`import` class `DecimalFormat` from `package java.text`

```
28       // convert to String in universal-time format
29       public String toUniversalString()
30       {
31          DecimalFormat twoDigits = new DecimalFormat( "00" );
32
33          return twoDigits.format( hour ) + ":" +
34             twoDigits.format( minute ) + ":" + twoDigits.format( second );
35       }
36
37       // convert to String in standard-time format
38       public String toStandardString()
39       {
40          DecimalFormat twoDigits = new DecimalFormat( "00" );
41
42          return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) + ":" +
43             twoDigits.format( minute ) + ":" + twoDigits.format( second ) +
44             ( hour < 12 ? " AM" : " PM" );
45       }
46
47    } // end class Time1
```

DecimalFormat from package java.text

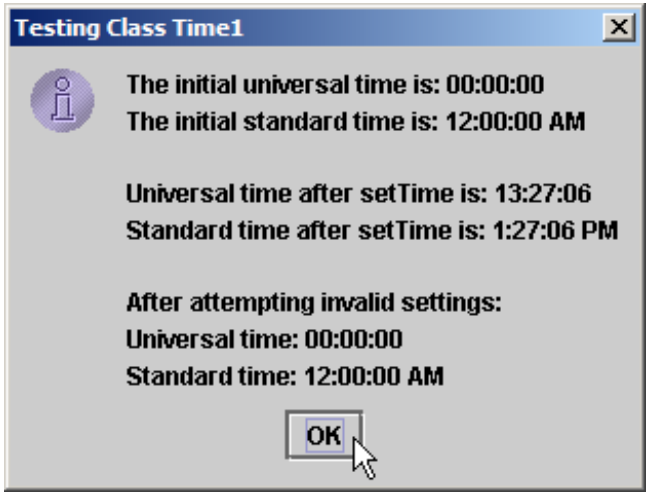
```
1  // Fig. 8.17: TimeTest1.java
2  // Class TimeTest1 to exercise class Time1.
3
4  // Java packages
5  import javax.swing.JOptionPane;
6
7  // Deitel packages
8  import com.deitel.ch08.Time1;  // import Time1 class
9
10 public class TimeTest1 {
11
12    public static void main( String arg
13    {
14       Time1 time = new Time1();  // calls Time1 constructor
15
16       // append String version of
17       String output = "The initial universal time is: " +
18          time.toUniversalString() + "\nThe initial standard time is: " +
19          time.toStandardString();
20
21       // change time and append updated time to output
22       time.setTime( 13, 27, 6 );
23       output += "\n\nUniversal time after setTime is: " +
24          time.toUniversalString() +
25          "\nStandard time after setTime is: " + time.toStandardString();
26
```

import class JOptionPane from package javax.swing

import class Time1 from package com.deitel.ch08

TimeTest1 can declare Time1 object

```
27          // set time with invalid values; append updated time to output
28          time.setTime( 99, 99, 99 );
29          output += "\n\nAfter attempting invalid settings: " +
30             "\nUniversal time: " + time.toUniversalString() +
31             "\nStandard time: " + time.toStandardString();
32
33          JOptionPane.showMessageDialog( null, output,
34             "Testing Class Time1", JOptionPane.INFORMATION_MESSAGE );
35
36          System.exit( 0 );
37
38       } // end main
39
40    } // end class TimeTest1
```



Package access

Variable or method does not have member access modifier

```
1  // Fig. 8.18: PackageDataTest.java
2  // Classes in the same package (i.e., the same directory) can
3  // use package access data of other classes in the sa
4  import javax.swing.JOptionPane;
5
6  public class PackageDataTest {
7
8     public static void main( String args[] )
9     {
10       PackageData packageData = new PackageData();
11
12       // append String representation of packageData to output
13       String output = "After instantiation:\n" +
14          packageData.toPackageDataString();
15
16       // change package access data in packageData ob
17       packageData.number = 77;
18       packageData.string = "Goodbye";
19
20       // append String representation of packageData to output
21       output += "\nAfter changing values:\n" +
22          packageData.toPackageDataString();
23
24       JOptionPane.showMessageDialog( null, output, "Package Access",
25          JOptionPane.INFORMATION_MESSAGE );
26
```

Instantiate reference to `PackageData` object

`PackageDataTest` can access `PackageData` data, because each class shares same `package`

```
27          System.exit( 0 );
28       }
29
30    } // end class PackageDataTest
31
32    // class with package access instance variables
33    class PackageData {
34       int number;     // package-access instance variable
35       String string;  // package-access instance variable
36
37       // constructor
38       public PackageData()
39       {
40          number = 0;
41          string = "Hello";
42       }
43
44       // return PackageData object String representation
45       public String toPackageDataString()
46       {
47          return "number: " + number + "    string: " + string;
48       }
49
50    } // end class PackageData
```

No access modifier, so class has package-access variables