Лекция 7

Моделиране на класове с композиция и делегиране

OBJECTIVES

In this lecture you will learn:

- Encapsulation and data hiding.
- To apply the Single Responsibility Principle with properties of mutable reference data types
- 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.
- Build classes using composition and delegation

7.1 Introduction

`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

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.

7.2 Time Class Case Study

- `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

Outline

Time1.java

(1 of 2)

```
1  // Fig. 8.1: Time1.java
2  // Time1 class declaration maintains the time in 24-hour format.
3
4  public class Time1
5  {
6     private int hour;   // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // set a new time value using universal time; ensure that
11    // the data remains consistent by setting invalid values to zero
12    public void setTime( int h, int m, int s )
13
14       hour = ( ( h >= 0 && h < 24 ) ? h : 0 );   // validate hour
15       minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
16       second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
17    } // end method setTime
18
```

private instance variables

Declare **public** method **setTime**

Validate parameter values before setting instance variables

Outline

Time1.java

(2 of 2)

```
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
```

format strings

Software Engineering Observation

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

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.

Outline

Time1Test.java

(1 of 2)

```
1  // Fig. 8.2: Time1Test.java
2  // Time1 object used in an application.
3
4  public class Time1Test
5  {
6     public static void main( String args[] )
7     {
8        // create and initialize a Time1 object
9        Time1 time = new Time1(); // invokes Time1 constructor
10
11       // output string representations of the time
12       System.out.print( "The initial universal time is: " );
13       System.out.println( time.toUniversalString() );
14       System.out.print( "The initial standard time is: " );
15       System.out.println( time.toString() );
16       System.out.println(); // output a blank line
17
```

Create a **Time1** object

Call **toUniversalString** method

Call **toString** method

Outline

Time1Test.java

```
18       // change time and output updated time
19       time.setTime( 13, 27, 6 );
20       System.out.print( "Universal time after setTime is: " );
21       System.out.println( time.toUniversalString() );
22       System.out.print( "Standard time after setTime is: " );
23       System.out.println( time.toString() );
24       System.out.println(); // output a blank line
25
26       // set time with invalid values; output updated time
27       time.setTime( 99, 99, 99 );
28       System.out.println( "After attempting invalid settings:" );
29       System.out.print( "Universal time: " );
30       System.out.println( time.toUniversalString() );
31       System.out.print( "Standard time: " );
32       System.out.println( time.toString() );
33    } // end main
34 } // end class Time1Test
```

Call **setTime** method

Call **setTime** method with invalid values

```
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
```

7.3 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

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.

Outline

MemberAccessTest.java

```
1  // Fig. 8.3: MemberAccessTest.java
2  // Private members of class Time1 are not accessible.
3  public class MemberAccessTest
4  {
5     public static void main( String args[] )
6     {
7        Time1 time = new Time1(); // create and initialize Time1 object
8
9        time.hour = 7;    // error: hour has private access in Time1
10       time.minute = 15; // error: minute has private access in Time1
11       time.second = 30; // error: second has private access in Time1
12    } // end main
13 } // 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
```

7.4 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`

Outline

ThisTest.java

(1 of 2)

```
1  // Fig. 8.4: ThisTest.java
2  // this used implicitly and explicitly to refer to members of an object.
3
4  public class ThisTest
5  {
6     public static void main( String args[] )
7     {
8        SimpleTime time = new SimpleTime( 15, 30, 19 );
9        System.out.println( time.buildString() );
10    } // end main
11 } // end class ThisTest
12
13 // class SimpleTime demonstrates the "this" reference
14 class SimpleTime
15 {
16    private int hour;   // 0-23
17    private int minute; // 0-59
18    private int second; // 0-59
19
20    // if the constructor uses parameter names identical to
21    // instance variable names the "this" reference is
22    // required to distinguish between names
23    public SimpleTime( int hour, int minute, int second )
24    {
25       this.hour = hour;       // set "this" object's hour
26       this.minute = minute;   // set "this" object's minute
27       this.second = second;   // set "this" object's second
28    } // end SimpleTime constructor
29
```

Create new **SimpleTime** object

Declare instance variables

Method parameters shadow instance variables

Using this to access the object's instance variables

Outline

```
30    // use explicit and implicit "this" to call toUniversalString
31    public String buildString()
32    {
33       return String.format( "%24s: %s\n%24s: %s",
34          "this.toUniversalString()", this.toUniversalString(),
35          "toUniversalString()", toUniversalString() );
36    } // end method buildString
37
38    // convert to String in universal-time format (HH:MM:SS)
39    public String toUniversalString()
40    {
41       // "this" is not required here to access instance variables,
42       // because method does not have local variables with same
43       // names as instance variables
44       return String.format( "%02d:%02d:%02d",
45          this.hour, this.minute, this.second );
46    } // end method toUniversalString
47 } // end class SimpleTime
```

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

ThisTest.java

Using **this** explicitly and implicitly to call **toUniversalString**

(2 of 2)

Use of **this** not necessary here

Common Programming Error

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

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

Performance Tip

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-`static` fields). Each method of the class implicitly uses `this` to determine the specific object of the class to manipulate.

7.5 Time Class Case Study: Overloaded Constructors

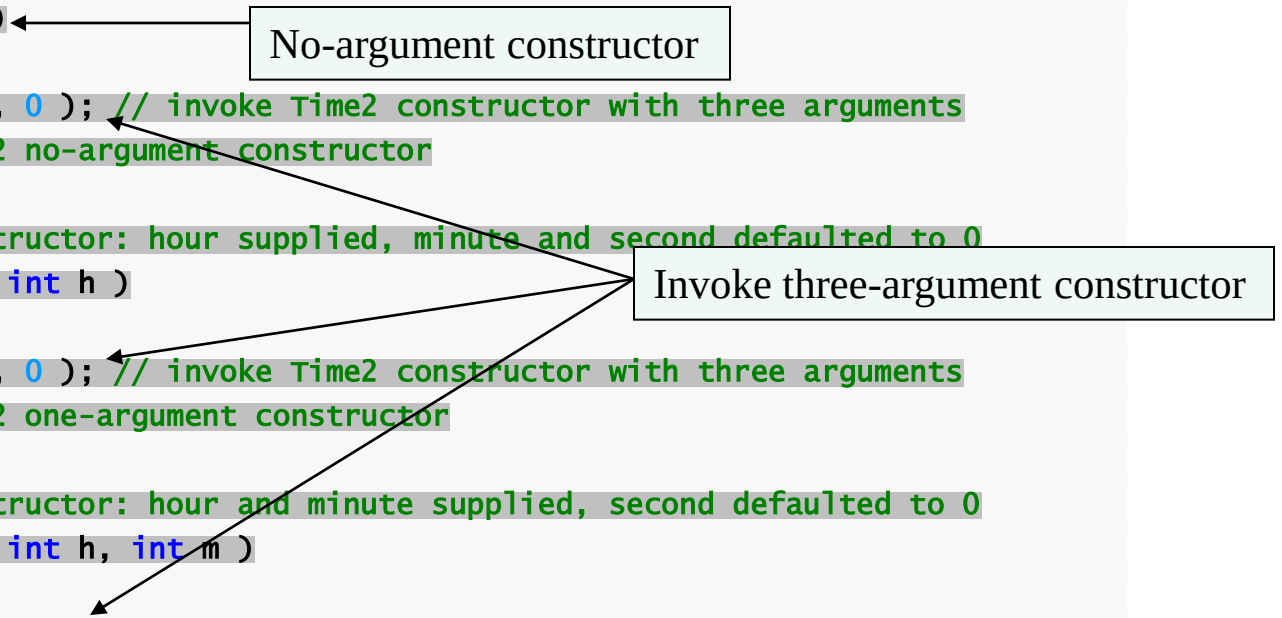
- **Overloaded constructors**
  - **Provide multiple constructor definitions with different signatures**
- **No-argument constructor**
  - **A constructor invoked without arguments**
- **The `this` reference can be used to invoke another constructor**
  - **Allowed only as the first statement in a constructor's body**

Outline

Time2.java

(1 of 4)

```
1  // Fig. 8.5: Time2.java
2  // Time2 class declaration with overloaded constructors.
3
4  public class Time2
5  {
6     private int hour;   // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // Time2 no-argument constructor: initializes each instance variable
11    // to zero; ensures that Time2 objects start in a consistent state
12    public Time2()
13    {
14       this( 0, 0, 0 ); // invoke Time2 constructor with three arguments
15    } // end Time2 no-argument constructor
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 constructor with three arguments
21    } // end Time2 one-argument constructor
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    } // end Time2 two-argument constructor
28
```



Outline

Time2.java

(2 of 4)

```
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
34
35    // Time2 constructor: another Time2 object supplied
36    public Time2( Time2 time )
37    {
38       // invoke Time2 three-argument c
39       this( time.hour, time.minute, time.second );
40    } // end Time2 constructor with a Time2 object argument
41
42    // Set Methods
43    // set a new time value using universal time; ensure that
44    // the data remains consistent by setting 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    } // end method setTime
51
```

Call **`setTime`** method

Constructor takes a reference to another **`Time2`** object as a parameter

Could have directly accessed instance variables of object **`time`** here

Outline

Time2.java

(3 of 4)

```
52    // validate and set hour
53    public void setHour( int h )
54    {
55       hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
56    } // end method setHour
57
58    // validate and set minute
59    public void setMinute( int m )
60    {
61       minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
62    } // end method setMinute
63
64    // validate and set second
65    public void setSecond( int s )
66    {
67       second = ( ( s >= 0 && s < 60 ) ? s : 0 );
68    } // end method setSecond
69
70    // Get Methods
71    // get hour value
72    public int getHour()
73    {
74       return hour;
75    } // end method getHour
76
```

```
77    // get minute value
78    public int getMinute()
79    {
80       return minute;
81    } // end method getMinute
82
83    // get second value
84    public int getSecond()
85    {
86       return second;
87    } // end method getSecond
88
89    // convert to String in universal-time format (HH:MM:SS)
90    public String toUniversalString()
91    {
92       return String.format(
93          "%02d:%02d:%02d", hour, minute, second );
94    } // end method toUniversalString
95
96    // convert to String in standard-time format (H:MM:SS AM or PM)
97    public String toString()
98    {
99       return String.format( "%d:%02d:%02d %s",
100         ( (hour== 0 || hour == 12) ? 12 : hour % 12 ),
101         minute, second, ( hour < 12 ? "AM" : "PM" ) );
102   } // end method toString
103} // end class Time2
```

Outline

Time2.java

(4 of 4)

Common Programming Error

It is a syntax error when `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 `this`.

Common Programming Error

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

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`).

7.5 Time Class Case Study: Overloaded Constructors (Cont.)

Using *set* methods

- Having constructors use *set* methods to modify instance variables instead of modifying them directly simplifies implementation changing

Software Engineering Observation

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.

Outline

Call overloaded constructors

Time2Test.java

(1 of 3)

```
1  // Fig. 8.6: Time2Test.java
2  // Overloaded constructors used to initialize Time2 objects.
3
4  public class Time2Test
5  {
6     public static void main( String args[] )
7     {
8        Time2 t1 = new Time2();           // 00:00:00
9        Time2 t2 = new Time2( 2 );        // 02:00:00
10       Time2 t3 = new Time2( 21, 34 );   // 21:34:00
11       Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
12       Time2 t5 = new Time2( 27, 74, 99 ); // 00:00:00
13       Time2 t6 = new Time2( t4 );       // 12:25:42
14
15       System.out.println( "Constructed with:" );
16       System.out.println( "t1: all arguments defaulted" );
17       System.out.printf( "   %s\n", t1.toUniversalString() );
18       System.out.printf( "   %s\n", t1.toString() );
19
```

Outline

Time2Test.java

(2 of 3)

```
20      System.out.println(
21         "t2: hour specified; minute and second defaulted" );
22      System.out.printf( "   %s\n", t2.toUniversalString() );
23      System.out.printf( "   %s\n", t2.toString() );
24
25      System.out.println(
26         "t3: hour and minute specified; second defaulted" );
27      System.out.printf( "   %s\n", t3.toUniversalString() );
28      System.out.printf( "   %s\n", t3.toString() );
29
30      System.out.println( "t4: hour, minute and second specified" );
31      System.out.printf( "   %s\n", t4.toUniversalString() );
32      System.out.printf( "   %s\n", t4.toString() );
33
34      System.out.println( "t5: all invalid values specified" );
35      System.out.printf( "   %s\n", t5.toUniversalString() );
36      System.out.printf( "   %s\n", t5.toString() );
37
```

Outline

```
38       System.out.println( "t6: Time2 object t4 specified" );
39       System.out.printf( "   %s\n", t6.toUniversalString() );
40       System.out.printf( "   %s\n", 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
```

Time2Test.java

7.6 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

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

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.

7.7 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

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

Class designers need not provide *set* or *get* methods for each `private` field. These capabilities should be provided only when it makes sense.

7.7 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**

7.8 Composition

The **Single responsibility principle** states that **every context** (class, method, variable) should have a **single responsibility**, and that responsibility should be entirely encapsulated by the context. All its services should be narrowly aligned with that responsibility.

Definition: **Responsibility** is a *reason to change*,

Hence, a class, method, variable should have one, and only one, reason to change.

For example, a datamember should change only when the its set method is called.

7.8 Composition

Example for a change at class level

Consider a class that compiles and prints a report. It may change for two reasons. First, the **content of the report can change**. Second, **the format of the report can change**. The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and **should therefore be in separate classes**.

It would be a **bad design** to **couple two things that change for different reasons** at different times.

Software Engineering Observation

The reason it is important to keep a class focused on a **single concern** is that it **makes the class more robust**. Continuing with the foregoing example, if there is a change to the report compilation process, there is greater danger that the printing code will break, if it is part of the same class.

7.8 Composition

Composition

- A class can have references to objects of other classes as members
- Sometimes referred to as a *has-a* relationship

Software Engineering Observation

One form of software reuse is composition, in which a class has as members references to objects of other classes.

Outline

Date.java

(1 of 3)

```
1  // Fig. 8.7: Date.java
2  // Date class declaration.
3
4  public class Date
5  {
6     private int month; // 1-12
7     private int day;   // 1-31 based on month
8     private int year;  // any year
9
10    // constructor: call checkMonth to confirm proper value for month;
11    // call checkDay to confirm proper value for day
12    public Date( int theMonth, int theDay, int theYear )
13    {
14       month = checkMonth( theMonth ); // validate month
15       year = theYear; // could validate year
16       day = checkDay( theDay ); // validate day
17
18       System.out.printf(
19          "Date object constructor for date %s\n", this );
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;
27       else // month is invalid
28       {
29          System.out.printf(
30             "Invalid month (%d) set to 1.", testMonth );
31          return 1; // maintain object in consistent state
32       } // end else
33    } // end method checkMonth
34
35    // utility method to confirm proper day value based on month and year
36    private int checkDay( int testDay )
37    {
38       int daysPerMonth[] =
39          { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
40
```

Validates month value

Validates day value

Outline

Date.java

(2 of 3)

Outline

Date.java

```
41       // check if day in range for month
42       if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
43          return testDay;
44
45       // check for leap year
46       if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
47            ( year % 4 == 0 && year % 100 != 0 ) ) )
48          return testDay;
49
50       System.out.printf( "Invalid day (%d) set to 1.", testDay );
51       return 1;  // maintain object in consistent state
52    } // end method checkDay
53
54    // return a String of the form month/day/year
55    public String toString()
56    {
57       return String.format( "%d/%d/%d", month, day, year );
58    } // end method toString
59 } // end class Date
```

Check if the day is February 29 on a leap year

Outline

Employee.java

```
1  // Fig. 8.8: Employee.java
2  // Employee class with references to other objects.
3
4  public class Employee
5  {
6     private String firstName;
7     private String lastName;
8     private Date birthDate;
9     private Date hireDate;
10
11    // constructor to initialize name, birth date and hire date
12    public Employee( String first, String last, Date dateOfBirth,
13       Date dateOfHire )
14    {
15       firstName = first;
16       lastName = last;
17       birthDate = dateOfBirth;
18       hireDate = dateOfHire;
19    } // end Employee constructor
20
21    // convert Employee to String format
22    public String toString()
23    {
24       return String.format( "%s, %s  Hired: %s  Birthday: %s",
25          lastName, firstName, hireDate, birthDate );
26    } // end method toString
27 } // end class Employee
```

Employee contains references to two **Date** objects

Implicit calls to **hireDate** and **birthDate**'s **toString** methods

Outline

EmployeeTest.java

```
1  // Fig. 8.9: EmployeeTest.java
2  // Composition demonstration.
3
4  public class EmployeeTest
5  {
6     public static void main( String args[] )
7     {
8        Date birth = new Date( 7, 24, 1949 );
9        Date hire = new Date( 3, 12, 1988 );
10       Employee employee = new Employee( "Bob", "Blue", birth, hire );
11
12       System.out.println( employee );
13    } // end main
14 } // end class EmployeeTest
```

Create an **Employee** object

Display the **Employee** object

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
```

7.8 Composition

❑ Immutable and mutable reference data types

In object-oriented and functional programming, **an immutable object** (unchangeable object) is an **object whose state cannot be modified after it is created**. For example **class Date and class Employee are immutable**. This is in contrast to **a mutable object** (changeable object), which **can be modified after it is created**. In some cases, an object is considered immutable even if some internally used attributes change but the object's state appears to be unchanging from an external point of view.

7.8 Composition

- **Immutable and mutable reference data types**

  `String`s, `LocalDate` and wrapper classes for primitive datatypes are typically employed as immutable objects to improve readability and run time efficiency in object-oriented programming. Immutable objects are also useful because they are inherently thread-safe. Other benefits are that they are simpler to understand and reason about and offer higher security than mutable objects.

7.8 Composition

Consider the case when class `Date` and class `Employee` are mutable i.e. there are set- methods in these class. We note that the set and get methods of class `Employee` must take in consideration the Single Responsibility Principle.

Otherwise we observe the following anomaly- methods of an instance, different of class `Employee` may change the values of mutable data members of an instance of class `Employee`. In other words, such data members may have more than one reason to change in addition the set methods of class `Employee`.

```
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
```

```
// set month method
public void setMonth(int testMonth) {
    month = checkMonth(testMonth);
} // end set month method

// 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
```

```
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;
      // memory sharing is not desirable here!!!
      birthDate = dateOfBirth;  // a copy constructor is required
      hireDate  = dateOfHire;   // a copy constructor is required
   } // end Employee constructor
```

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

```
public static void main( String args[] )
{
   Date  hire =  new Date( 7, 24, 2018);
   Date  birth = new Date( 3, 12, 1989 );
   Employee employee = new Employee( "Bob", "Blue", birth, hire );

   System.out.println(employee);
   System.out.println( "Note: Hiredate and Birthdate have more than one " +
                       " reason to change!");
   System.out.println( "Wrong management of mutable reference types...");
   System.out.println("\n\nChange the Hiredate month to 1960 without the” +
                      " knowledge of employee\n");
   hire.setYear(1960);
   System.out.println("\n\nGet Employee Birthdate and change month to 5 "
                         “without the knowledge of employee\n");
   Date spy = employee.getBirthDate();
   spy.setMonth(5); //
   System.out.println("Note: The employee gets hired before he has been born.");
   System.out.println("Note: The employee birthday changed without his “ +
                      " knowledge.");
   System.out.println(employee);
   System.out.println("This effect is referred to as Memory sharing!");
} // end main
```

Date object constructor for date 7/24/2018
Date object constructor for date 3/12/1989
Blue, Bob Hired: **7/24/2018** Birthday: **3/12/1989**
Note: Hiredate and Birthdate have more than one reason to change!
Wrong management of mutable reference types...

Change the Hiredate month to 1960 without the knowledge of employee

Get Employee Birthdate and change month to 5 without the knowledge of employee

Note: The employee gets hired before he has been born.
Note: The employee birthday has changed without his knowledge.
Blue, Bob Hired: **7/24/1960** Birthday: **5/12/1989**
This effect is referred to as Memory sharing!

```
run:
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949


Change the Hire date month to 10 without the knowldge of employee

Blue, Bob  Hired: 10/12/1988  Birthday: 7/24/1949


Get Employee Birthdate and change month to 5 without the knowldge of employee

Blue, Bob  Hired: 10/12/1988  Birthday: 5/24/1949
This effect is referred to as Memory sharing!
```

7.8 Composition

To define a simple immutable class follow the below mentioned rules

1. Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
2. Make all fields `final` and `private`.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as `final`. A more sophisticated approach is to make the constructor `private` and construct instances in factory methods.

7.8 Composition

4. ***If the instance fields include references to mutable objects, don't allow those objects to be changed*****:**

5. Don't provide methods that modify the mutable objects.

6. Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

```
public final class MyEmployee {// Wrong:
  private final String name;       // immutable reference type
  private final double salary;     // primitive datatype
  private final Date dateOfBirth; //mutable reference type
  // Wrong:The constructor stores references to external, mutable objects
  public MyEmployee(String name, double salary, Date dateOfBirth) {
        this.name=name!= null? name: "default name" ;
        this.salary=salary> 0? salary: 1;
        this.dateOfBirth= dateOfBirth !=null? dateOfBirth:
                                                  new Date (1, 1, 1900);
  }
  public String getName() {
        return name; // OK, reference to a mutable reference type
  }
  public double getSalary() {
        return salary; // OK, reference to a mutable reference type
  }
  public Date getDateOfBirth() {
        return  dateOfBirth ; // Wrong: returns a reference to a mutable
  }
}
```

```
public final class MyEmployee { // Correct class design
private String firstName; //String is a  immutable reference type
private String lastName;  // String is a immutable reference type
private Date birthDate;//  Date is a  mutable reference type
private Date hireDate; // Date is a mutable reference type
private Date[] visits; // array is a mutable reference type

public MyEmployee(String first, String last,
                  Date dateOfBirth, Date dateOfHire, Date[] visits) {
    setFirstName(first);
    setLastName(last);
    setHireDate(dateOfHire);
    setBirthDate(dateOfBirth);
    setVisits(visits);
}
// mutable reference type getter
public Date getBirthDate() {
    return new Date(birthDate);
}

// mutable reference type setter
private void setBirthDate(Date dateOfBirth) {
    this.birthDate = dateOfBirth != null ?
                     new Date(dateOfBirth) : new Date(1, 1, 1900);
}
```

```
public Date[] getVisits() {
    // deep copy of visits
    Date[] temp = new Date[visits.length];

    for (int i = 0; i < visits.length; i++) {
        temp[i]= visits[i]!= null? new Date(visits[i]): null;
    }
    return temp; // return  deep copy of visits
}

public void setVisits(Date[] visits) {
    // set a deep copy of visits
    this.visits = visits!=null? new Date[visits.length]: new Date[0];

    for (int i = 0; i <visits.length ; i++) {
        this.visits[i] = visits[i]!= null? new Date(visits[i]): null;
    }
}
```

Note: Hiredate and Birthdate have more than one reason to change!
Correct management of mutable reference types...

Change the Hiredate month to 1960 without the knowledge of employee

Get Employee Birthdate and change month to 5 without the knowldge of employee

Date object constructor for date 3/12/1989
Note: The employee is hired as originally defined.
Note: The employee birthday remains as originally defined.
Blue, Bob Hired: **7/24/2018** Birthday: **3/12/1989** Visits: [4/10/2020]

Process finished with exit code 0

7.9 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`

7.10 `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 `int`s)

Software Engineering Observation

Use a `static` variable when all objects of a class must use the same copy of the variable.

Software Engineering Observation

Static class variables and methods exist, and can be used, even if no objects of that class have been instantiated.

Quiz- what is the output?

```
class Test {

      public static String foo(){
            System.out.println("Test foo called");
            return "";
      }

      public static void main(String args[]){
            Test obj = null;
            System.out.println(obj.foo());
      }
}
```

Quiz- what is the output?

Instead of `NullPointerException`, when we invoke a method on object that is `null`, this program will work and prints "`Test foo called`".

The reason for this is the Java compiler code optimization. When the Java code is compiled to produced bytecode, it figures out that `foo()` is a `static` method and should be called using a `class name`. So it changes the method call `obj.foo()` to `Test.foo()` and hence there is no `NullPointerException`.

Outline

Employee.java

(1 of 2)

```
1  // Fig. 8.12: Employee.java
2  // Static variable used to maintain a count of the number of
3  // Employee objects in memory.
4
5  public class Employee
6  {
7     private String firstName;
8     private String lastName;
9     private static int count = 0; // number of objects in memory
10
11    // initialize employee, add 1 to static count and
12    // output String indicating that constructor was called
13    public Employee( String first, String last )
14    {
15       firstName = first;
16       lastName = last;
17
18       count++;  // increment static count of employees
19       System.out.printf( "Employee constructor: %s %s; count = %d\n",
20          firstName, lastName, count );
21    } // end Employee constructor
22
```

Declare a **`static`** field

Increment **`static`** field

Outline

Employee.java

(2 of 2)

```
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    {
28       count--; // decrement static count of employees
29       System.out.printf( "Employee finalizer: %s %s; count = %d\n",
30          firstName, lastName, count );
31    } // end method finalize
32
33    // get first name
34    public String getFirstName()
35    {
36       return firstName;
37    } // end method getFirstName
38
39    // get last name
40    public String getLastName()
41    {
42       return lastName;
43    } // end method getLastName
44
45    // static method to get static count value
46    public static int getCount()
47    {
48       return count;
49    } // end method getCount
50 } // end class Employee
```

Declare method **`finalize`**

Declare **`static`** method **`getCount`** to get **`static`** field **`count`**

Outline

EmployeeTest.java

(1 of 3)

```
1  // Fig. 8.13: EmployeeTest.java
2  // Static member demonstration.
3
4  public class EmployeeTest
5  {
6     public static void main( String args[] )
7     {
8        // show that count is 0 before creating Employees
9        System.out.printf( "Employees before instantiation: %d\n",
10          Employee.getCount() );
11
12       // create two Employees; count should be 2
13       Employee e1 = new Employee( "Susan", "Baker" );
14       Employee e2 = new Employee( "Bob", "Blue" );
15
```

Call **`static`** method **`getCount`** using class name **`Employee`**

Create new **`Employee`** objects

Outline

EmployeeTest.java

```
16      // show that count is 2 after creating two Employees
17      System.out.println( "\nEmployees after instantiation: " );
18      System.out.printf( "via e1.getCount(): %d\n", e1.getCount() );
19      System.out.printf( "via e2.getCount(): %d\n", e2.getCount() );
20      System.out.printf( "via Employee.getCount(): %d\n",
21         Employee.getCount() );
22
23      // get names of Employees
24      System.out.printf( "\nEmployee 1: %s %s\nEmployee 2: %s %s\n\n",
25         e1.getFirstName(), e1.getLastName(),
26         e2.getFirstName(), e2.getLastName() );
27
28      // in this example, there is only one reference to each Employee,
29      // so the following two statements cause the JVM to mark each
30      // Employee object for garbage collection
31      e1 = null;
32      e2 = null;
33       System.gc(); // ask for garbage collection to occur now
34      Runtime.getRuntime().runFinalization();
35
```

Call **static** method **getCount** outside objects

Call **static** method **getCount** inside objects

(2 of 3)

Remove references to objects, JVM will mark them for garbage collection

Wait garbage collection to complete

Call **static** method **gc** of class **System** to indicate that garbage collection should be attempted

Outline

```
36       // show Employee count after calling garbage collector; count
37       // displayed may be 0, 1 or 2 based on whether garbage collector
38       // executes immediately and number of Employee objects collected
39       System.out.printf( "\nEmployees after System.gc(): %d\n",
40          Employee.getCount() );
41    } // end main
42 } // end class EmployeeTest
```

Call **`static`** method **`getCount`**

`EmployeeTest.java`

(3 of 3)

```
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

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

7.10 `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

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

Referring to `this` in a `static` method is a syntax error.

7.11 `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

Outline

StaticImportTest.java

```
1  // Fig. 8.14: StaticImportTest.java
2  // Using static import to import static methods of class Math.
3  import static java.lang.Math.*;
4
5  public class StaticImportTest
6  {
7     public static void main( String args[] )
8     {
9        System.out.printf( "sqrt( 900.0 ) = %.1f\n", sqrt( 900.0 ) );
10       System.out.printf( "ceil( -9.8 ) = %.1f\n", ceil( -9.8 ) );
11       System.out.printf( "log( E ) = %.1f\n", log( E ) );
12       System.out.printf( "cos( 0.0 ) = %.1f\n", cos( 0.0 ) );
13    } // end main
14 } // end class StaticImportTest
```

static import on demand

Use **Math**'s **static** methods and instance variable without preceding them with **Math.**

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

Common Programming Error

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.

7.12 `final` Instance Variables

Principle of least privilege

- Code should have only the privilege ad 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

Software Engineering Observation

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.

Outline

Increment.java

```
1  // Fig. 8.15: Increment.java
2  // final instance variable in a class.
3
4  public class Increment
5  {
6     private int total = 0; // total of all increments
7     private final int INCREMENT; // constant variable (uninitialized)
8
9     // constructor initializes final instance variable INCREMENT
10    public Increment( int incrementValue )
11    {
12       INCREMENT = incrementValue; // initialize constant variable (once)
13    } // end Increment constructor
14
15    // add INCREMENT to total
16    public void addIncrementToTotal()
17    {
18       total += INCREMENT;
19    } // end method addIncrementToTotal
20
21    // return String representation of an Increment object's data
22    public String toString()
23    {
24       return String.format( "total = %d", total );
25    } // end method toIncrementString
26 } // end class Increment
```

Declare **`final`** instance variable

Initialize **`final`** instance variable inside a constructor

Outline

IncrementTest.java

```
1  // Fig. 8.16: IncrementTest.java
2  // final variable initialized with a constructor argument.
3
4  public class IncrementTest
5  {
6     public static void main( String args[] )
7     {
8        Increment value = new Increment( 5 );
9
10       System.out.printf( "Before incrementing: %s\n\n", value );
11
12       for ( int i = 1; i <= 3; i++ )
13       {
14          value.addIncrementToTotal();
15          System.out.printf( "After increment %d: %s\n", i, value );
16       } // end for
17    } // end main
18 } // end class IncrementTest
```

Create an **Increment** object

Call method **addIncrementToTotal**

```
Before incrementing: total = 0

After increment 1: total = 5
After increment 2: total = 10
After increment 3: total = 15
```

Common Programming Error

Attempting to modify a `final` instance variable after it is initialized is a compilation error.

Error-Prevention Tip

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

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

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
```

Outline

Increment.java

7.13 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:
  - https://docs.oracle.com/javase/8/
  - Or click to download

7.14 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
      - `int`s are only approximations of integers, can produce arithmetic overflow
  - Operations that can be performed on data

Good Programming Practice

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.

7.14 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

Software Engineering Observation

Programmers create types through the class mechanism. New types can be designed to be 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.

7.15 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 very 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.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

Outline

```
1  // Fig. 8.18: Time1.java
2  // Time1 class declaration maintains the time in 24-hour format.
3  package com.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
```

`package` declaration

`Time1` is a **`public`** class so it can be used by importers of this package

`Time1.java`

(1 of 2)

Outline

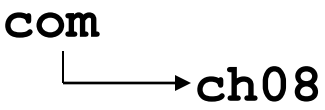
Time1.java

(2 of 2)

```
20    // convert to String in universal-time format (HH:MM:SS)
21    public String toUniversalString()
22    {
23       return String.format( "%02d:%02d:%02d", hour, minute, second );
24    } // end method toUniversalString
25
26    // convert to String in standard-time format (H:MM:SS AM or PM)
27    public String toString()
28    {
29       return String.format( "%d:%02d:%02d %s",
30          ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
31          minute, second, ( hour < 12 ? "AM" : "PM" ) );
32    } // end method toString
33 } // end class Time1
```

7.15 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



  - `javac` command-line option `-d`
    - `javac` creates appropriate directories based on the class's `package` declaration
    - A period (.) after `-d` represents the current directory

7.15 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

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.

Outline

```
1  // Fig. 8.19: Time1PackageTest.java
2  // Time1 object used in an application.
3  import com.ch08.Time1; // import class Time1
4
5  public class Time1PackageTest
6  {
7     public static void main( String args[] )
8     {
9        // create and initialize a Time1 object
10       Time1 time = new Time1(); // calls Time1 constructor
11
12       // output string representations of the time
13       System.out.print( "The initial universal time is: " );
14       System.out.println( time.toUniversalString() );
15       System.out.print( "The initial standard time is: " );
16       System.out.println( time.toString() );
17       System.out.println(); // output a blank line
18
```

Single-type **`import`** declaration

Refer to the **`Time1`** class by its simple name

`Time1PackageTest`

`.java`

Outline

Time1PackageTest.java

(2 of 2)

```
19       // change time and output updated time
20       time.setTime( 13, 27, 6 );
21       System.out.print( "Universal time after setTime is: " );
22       System.out.println( time.toUniversalString() );
23       System.out.print( "Standard time after setTime is: " );
24       System.out.println( time.toString() );
25       System.out.println(); // output a blank line
26
27       // set time with invalid values; output updated time
28       time.setTime( 99, 99, 99 );
29       System.out.println( "After attempting invalid settings:" );
30       System.out.print( "Universal time: " );
31       System.out.println( time.toUniversalString() );
32       System.out.print( "Standard time: " );
33       System.out.println( time.toString() );
34    } // end main
35 } // 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
```

7.15 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`

7.15 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 classpathes by using the same techniques that the `javac` command uses**

Common Programming Error

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

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

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.

7.16 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

Outline

PackageDataTest

.java

(1 of 2)

```
1  // Fig. 8.20: PackageDataTest.java
2  // Package-access members of a class are accessible by other classes
3  // in the same package.
4
5  public class PackageDataTest
6  {
7     public static void main( String args[] )
8     {
9        PackageData packageData = new PackageData();
10
11       // output String representation of packageData
12       System.out.printf( "After instantiation:\n%s\n", packageData );
13
14       // change package access data in packageData object
15       packageData.number = 77;
16       packageData.string = "Goodbye";
17
18       // output String representation of packageData
19       System.out.printf( "\nAfter changing values:\n%s\n", packageData );
20    } // end main
21 } // end class PackageDataTest
22
```

Can directly access package-access members

Outline

PackageDataTest.java

(2 of 2)

```
23 // class with package access instance variables
24 class PackageData
25 {
26    int number; // package-access instance variable
27    String string; // package-access instance variable
28
29    // constructor
30    public PackageData()
31    {
32       number = 0;
33       string = "Hello";
34    } // end PackageData constructor
35
36    // return PackageData object String representation
37    public String toString()
38    {
39       return String.format( "number: %d; string: %s", number, string );
40    } // end method toString
41 } // end class PackageData
```

Package-access instance variables

```
After instantiation:
number: 0; string: Hello

After changing values:
number: 77; string: Goodbye
```

7.17 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**

```
1  // MyLine.java
2  // Declaration of class MyLine.
3  import javafx.scene.Group;
4  import javafx.scene.paint.Color;
5  import javafx.scene.shape.Line;
6
7  public class MyLine
8  {
9     private final double x1; // x coordinate of first endpoint
10    private final double y1; // y coordinate of first endpoint
11    private final double x2; // x coordinate of second endpoint
12    private final double y2; // y coordinate of second endpoint
13    private final Color myColor; // color of this shape
14    private final double thickness;
15
16    // constructor with input values
17    public MyLine( double x1, double y1, double x2, double y2, Color color,
18                                                                double thickness)
19    {
20       this.x1 = x1; // set x coordinate of first endpoint
21       this.y1 = y1; // set y coordinate of first endpoint
22       this.x2 = x2; // set x coordinate of second endpoint
23       this.y2 = y2; // set y coordinate of second endpoint
24       this.thickness = thickness;
25       myColor = color; // set the color
26    } // end MyLine constructor
27
```

Instance variables to store coordinates and color for a line

Initialize instance variables

Myline.java

```
28    // Actually draws the line
29    public void draw( Group pane )
30    {
31       Line line = new Line(x1,  y1,  x2,  y2 );
32       line.setStroke(myColor);
33       line.setStrokeWidth(thickness);
34       pane.getChildren().add(line);
35    } // end method draw
36 } // end class MyLine
```

Draw a line in the specified Parent node

Create a Line and adjust its properties

Draw the Line in the specified Parent node

Myline.java

```
1  import java.util.Random;
2  import javafx.application.Application;
3  import javafx.scene.Group;
4  import javafx.scene.Scene;
5  import javafx.scene.paint.Color;
6  import javafx.stage.Stage;
7
8  public class DrawRandomLinesJfx extends Application {
9     private Random randomNumbers = new Random();
10    private MyLine lines[]; // array on lines
11
12    @Override
13    public void start(Stage primaryStage) {
14        Group root = new Group();
15        Scene scene = new Scene(root, 500, 300);
16        // create lines
17        lines = new MyLine[5 + randomNumbers.nextInt(5)];
18        for (int count = 0; count < lines.length; count++) {
19            // generate random coordinates
20            double x1 = randomNumbers.nextInt((int) scene.getWidth());
21            double y1 = randomNumbers.nextInt((int) scene.getHeight());
22            double x2 = randomNumbers.nextInt((int) scene.getWidth());
23            double y2 = randomNumbers.nextInt((int) scene.getHeight());
24            double penThickness = 10 * randomNumbers.nextDouble() + 0.5;
25            // generate a random color
26            Color color = Color.rgb(randomNumbers.nextInt(256),
27                    randomNumbers.nextInt(256), randomNumbers.nextInt(256));
```

Declare a **MyLine** array

Create the **MyLine** array

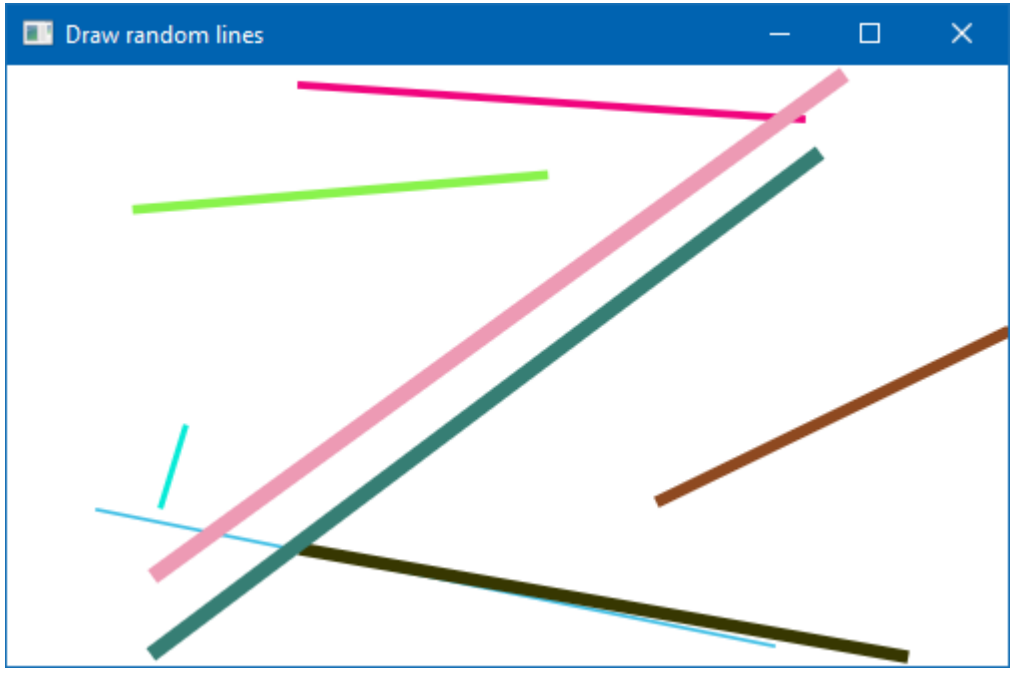
Generate coordinates for this line

Generate a color and thickness for this line

```
28
29              // add the line to the list of lines to be displayed
30              lines[count] = new MyLine(x1, y1, x2, y2, color, penThickness);
31              lines[count].draw(root);
32          } // end for
33          primaryStage.setTitle("Draw random lines");
34          primaryStage.setScene(scene);
35          primaryStage.show();
36      }
37
38      public static void main(String[] args) {
39          launch(args);
40      }
41  }
```

Create the new **MyLine** object with the generated attributes

Draw each **MyLine**

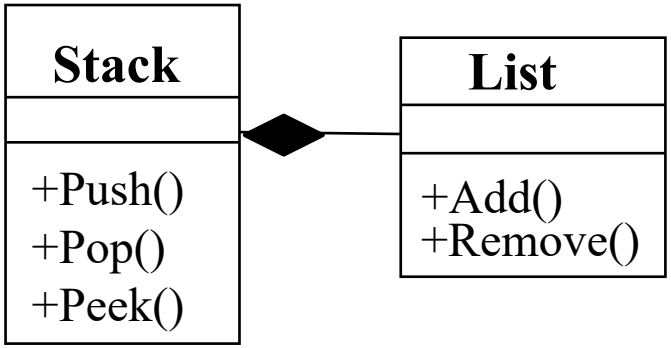


7.18 Building classes with delegation

Technique: **Delegation** is a software engineering technique, **allowing an object to pass the execution of its behavior to another object referenced by this object**.

Delegation: **Catching an operation** and **sending it to another object**

Which of the following models is better?

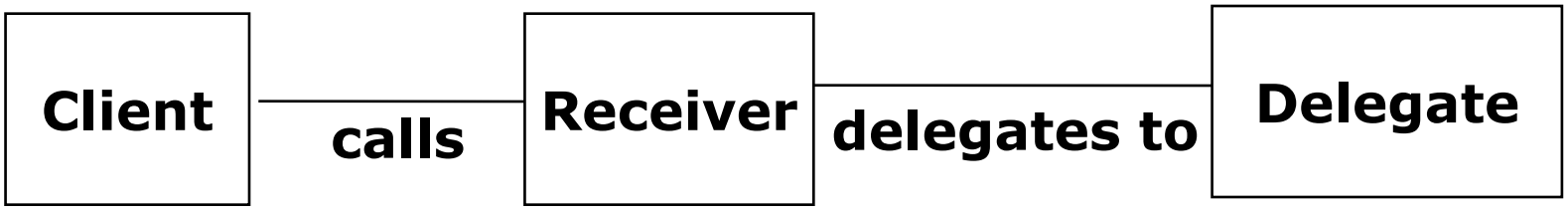


7.18 Building classes with delegation

Delegation is a way of **making composition a powerful technique for reuse**

In delegation two objects are involved in handling a **request** from a **Client**

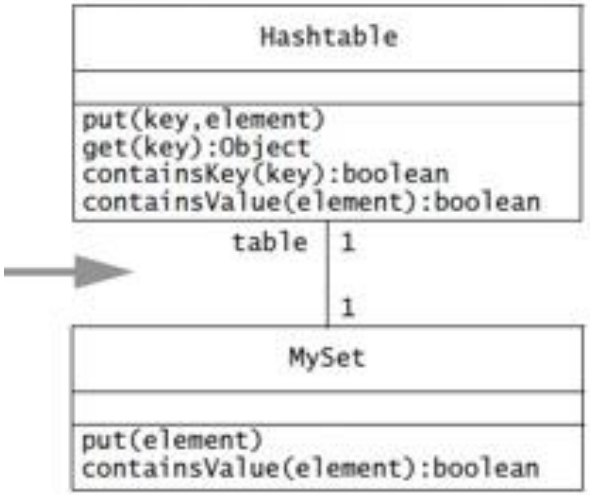
The **Receiver object** delegates operations to the **Delegate object**
The **Receiver** object makes sure, that the **Client does not misuse** the **Delegate** object.



7.18a Implementation approach

```
class A {
    void f() {  System.out.println("A: doing f()");     }
    void g() {  System.out.println("A: doing g()");     }
    }
}
class C {
    // delegation
    A a = new A();
    void f() {  a.f();   }
    void g() {  a.g();   }    // normal attributes
    X x = new X();
    void y() { /* do stuff */ }
}
class X {} // some class
public class MainClass {
    public static void main(String[] args) {
        C c = new C();
        c.f();
        c.g();
    }
}
```


7.18a Implementation approach




```
/* Implementation of MySet using
delegation */
class MySet {
    private Hashtable table;
    MySet() {
        table = Hashtable();
    }
    void put(Object element) {
        if (!containsValue(element)){
            table.put(element,this);
        }
    }
    boolean containsValue(Object
            element) {
        return
        (table.containsKey(element));
    }
    /* Other methods omitted */
}
```

7.18a Implementation approach

Delegation advantages

- ☺ **Flexibility**: Any object can be replaced at run time by another one (as long as it has the same type)
- ☹ **Inefficiency**: Objects are encapsulated

Задачи

Задача 1.

Write class Stack making use of delegation to an instance of ArrayList for the purpose of implementing methods:

- ✓ **push**
- ✓ **pop**
- ✓ **peek**
- ✓ **size**
- ✓ **isEmpty**
- ✓ **toString**

Задачи

Задача 2.

Напишете *class Computer*.

- **Нека class *Computer* *има* *type* (име на производител) , *procSpeed* (тактова честота на процесора в MHz) и *files* (масив с имена на файлове).**
- **Напишете SET и GET методи за *type* , *procSpeed* и *files*. SET методите да валидират по подходящ начин тези клас данни, съобразен с контекста на задачата**
- **Напишете пълен списък с конструктори- за общо ползване, по подразбиране и копиране като спазвате концепцията за скриване на информация (*encapsulation, information hiding*) и изискване за избягване на повторно използване на код (*software reuse*- избягване на дублиране на код!)**
- **Нека да има и *toString()* метод за извеждане на текущите стойности за клас данните, всяка на отделен ред със съответен промпт.**

Напишете приложение за тестване *class Computer* :

- **- създаване на обекти с всеки от трите конструктора**
- **- промяна на данните с използване на SET методите и извеждане на данните с *toString()* метода**