1

8.8 Composition

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

2

Software Engineering Observation 8.9

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

```
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
```

Outline

Date.java

(1 of 3)


```
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)

5

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

6

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
```

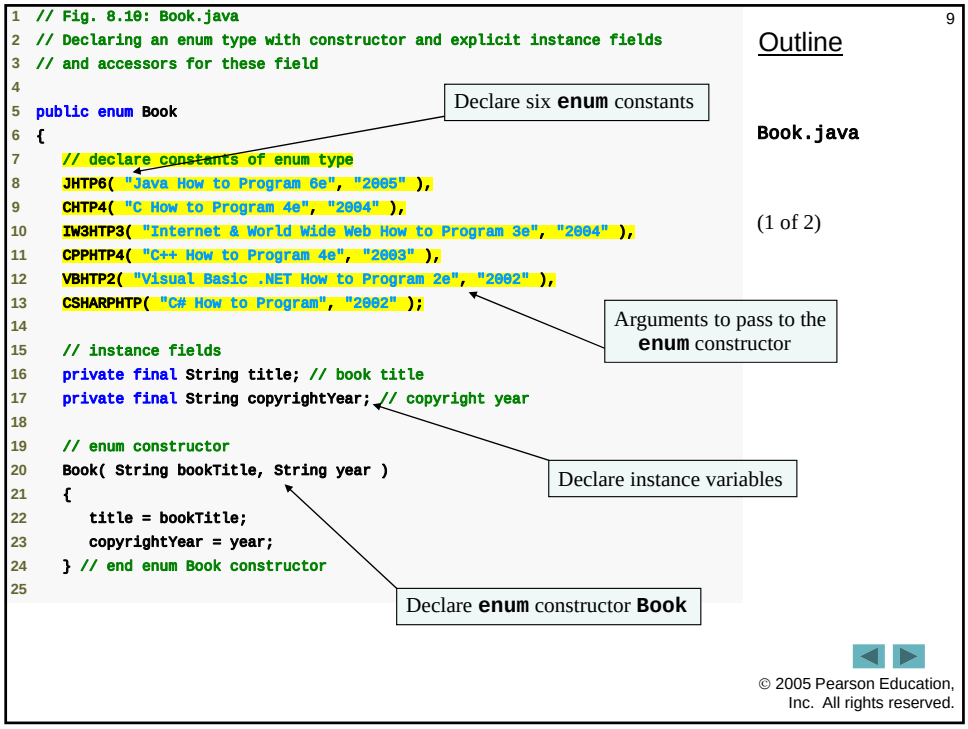
Create an **Employee** object

Display the **Employee** object

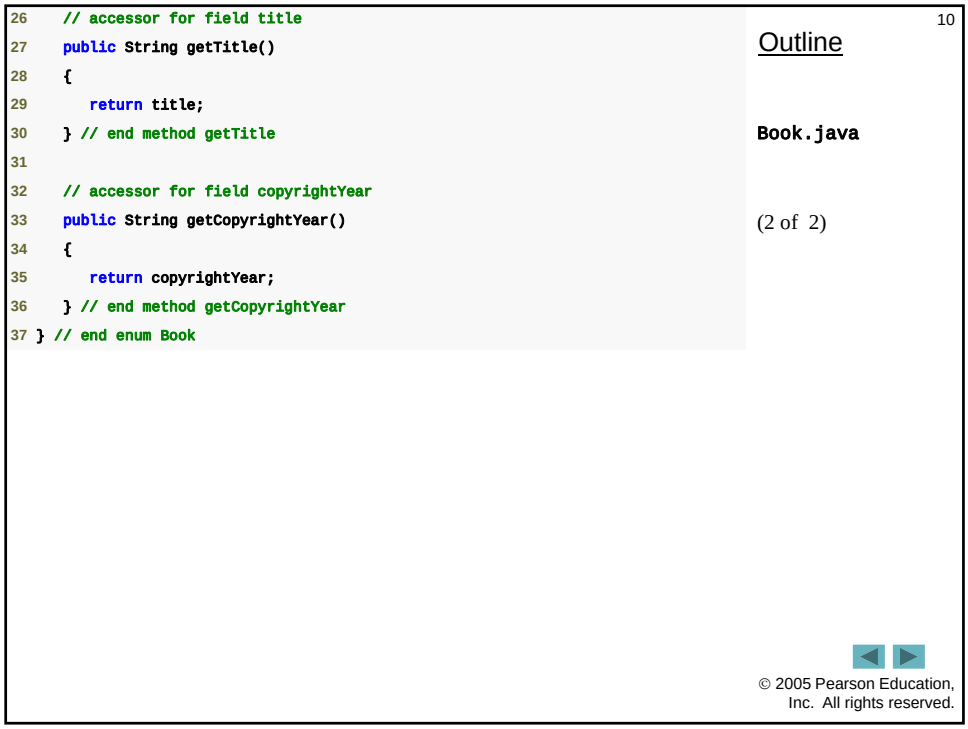
```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
```

8.9 Enumerations

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




```
1  // Fig. 8.10: Book.java
2  // Declaring an enum type with constructor and explicit instance fields
3  // and accessors for these field
4
5  public enum Book
6  {
7     // declare constants of enum type
8     JHTP6( "Java How to Program 6e", "2005" ),
9     CHTP4( "C How to Program 4e", "2004" ),
10    IW3HTP3( "Internet & World Wide Web How to Program 3e", "2004" ),
11    CPPHTP4( "C++ How to Program 4e", "2003" ),
12    VBHTP2( "Visual Basic .NET How to Program 2e", "2002" ),
13    CSHARPHTP( "C# How to Program", "2002" );
14
15    // instance fields
16    private final String title; // book title
17    private final String copyrightYear; // copyright year
18
19    // enum constructor
20    Book( String bookTitle, String year )
21    {
22       title = bookTitle;
23       copyrightYear = year;
24    } // end enum Book constructor
25
```




```
26    // accessor for field title
27    public String getTitle()
28    {
29       return title;
30    } // end method getTitle
31
32    // accessor for field copyrightYear
33    public String getCopyrightYear()
34    {
35       return copyrightYear;
36    } // end method getCopyrightYear
37 } // end enum Book
```

11

8.9 Enumerations (Cont.)

- **`static` method `values`**
  - Generated by the compiler for every `enum`
  - Returns an array of the `enum`'s constants in the order in which they were declared
- **`static` method `range` of class `EnumSet`**
  - Takes two parameters, the first and last `enum` constants in the desired range
  - Returns an `EnumSet` containing the constants in that range, inclusive
  - An enhanced `for` statement can iterate over an `EnumSet` as it can over an array

12

Outline

`EnumTest.java`

```
1  // Fig. 8.11: EnumTest.java
2  // Testing enum type Book.
3  import java.util.EnumSet;
4
5  public class EnumTest
6  {
7     public static void main( String args[] )
8     {
9        System.out.println( "All books:\n" );
10
11       // print all books in enum Book
12       for ( Book book : Book.values() )
13          System.out.printf( "%-10s%-45s%s\n", book,
14             book.getTitle(), book.getCopyrightYear() );
15
16       System.out.println( "\nDisplay a range of enum constants:\n" );
17
18       // print first four books
19       for ( Book book : EnumSet.range( Book.JHTP6, Book.CPPHTP4 ) )
20          System.out.printf( "%-10s%-45s%s\n", book,
21             book.getTitle(), book.getCopyrightYear() );
22    } // end main
23 } // end class EnumTest
```

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

Enhanced `for` loop iterates for each `enum` constant in the `EnumSet` returned by method `range`

```
All books:

JHTP6     Java How to Program 6e                        2005
CHTP4     C How to Program 4e                           2004
IW3HTP3   Internet & World Wide Web How to Program 3e   2004
CPPHTP4   C++ How to Program 4e                         2003
VBHTP2    Visual Basic .NET How to Program 2e           2002
CSHARPHTP C# How to Program                             2002

Display a range of enum constants:

JHTP6     Java How to Program 6e                        2005
CHTP4     C How to Program 4e                           2004
IW3HTP3   Internet & World Wide Web How to Program 3e   2004
CPPHTP4   C++ How to Program 4e                         2003
```

13

Outline

EnumTest.java

(2 of 2)

14

Common Programming Error 8.6

In an `enum` declaration, it is a syntax error to declare `enum` constants after the `enum` type's constructors, fields and methods in the `enum` declaration.

15

8.11 `static` Class Members

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

16

Software Engineering Observation 8.11

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

Software Engineering Observation 8.12

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

Outline

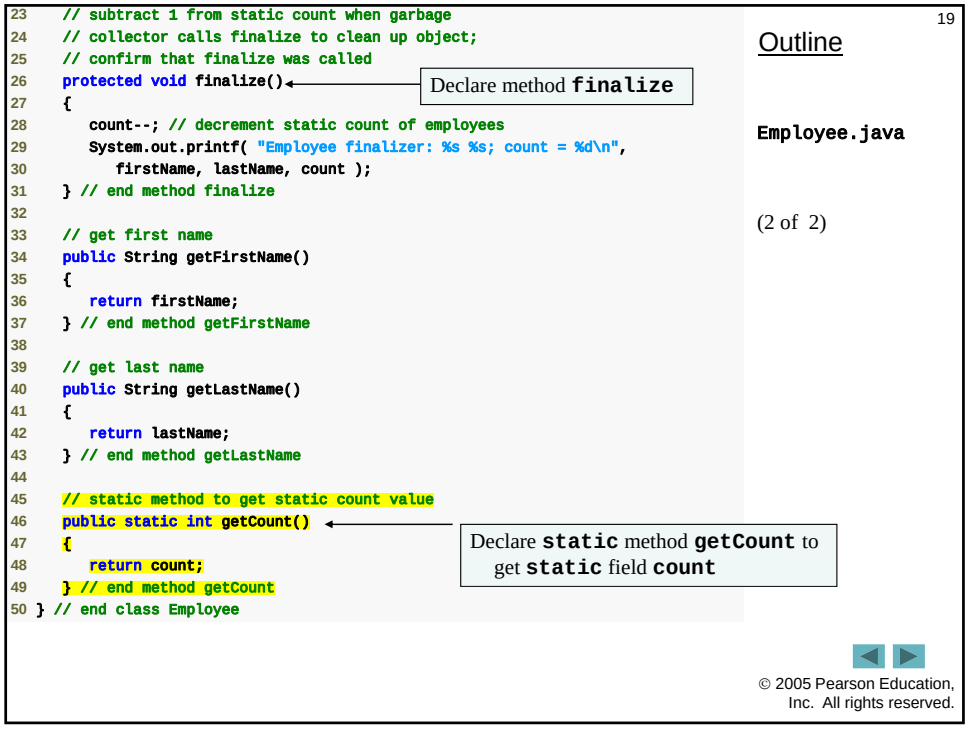
Employee.java

(1 of 2)

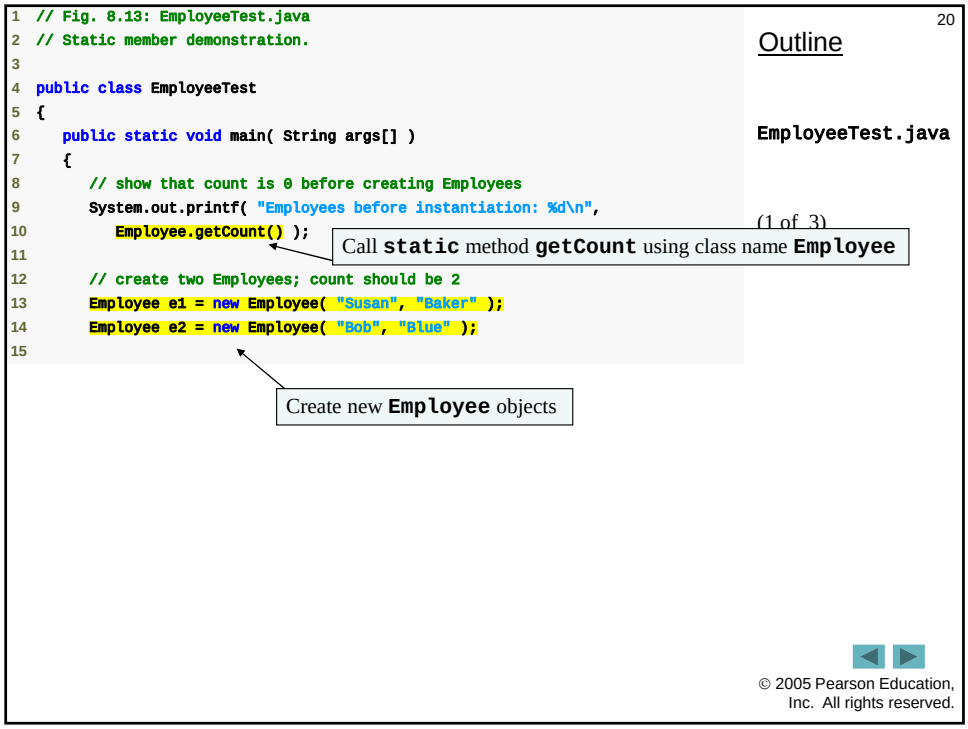
```
1  // Fig. 8.12: Employee.java
2  // Static variable used to maintain a count of the number of
3  // Employee objects in memory.
4
5  public class Employee
6  {
7     private String firstName;
8     private String lastName;
9     private static int count = 0; // number of objects in memory
10
11    // 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



```
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
```

```
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
```

Outline

EmployeeTest.java

(2 of 3)

```
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
34       System.gc(); // ask for garbage collection to occur now
35
```

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

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

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

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

Outline

EmployeeTest.java

(3 of 3)

```
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**

```
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
```

23

Good Programming Practice 8.1

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

24

8.11 `static` Class Members (Cont.)

- **`static` methods cannot access non-`static` class members**
  - **Also cannot use the `this` reference**

25

Common Programming Error 8.7

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

26

Common Programming Error 8.8

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

27

8.12 `static` Import

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

28

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
```

29

Common Programming Error 8.9

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.

30

8.13 `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 8.13

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

33

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
```

34

Common Programming Error 8.10

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

35

Software Engineering Observation 8.14

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

36

Common Programming Error 8.11

Not initializing a `final` instance variable in its declaration or in every constructor of the class yields a compilation error indicating that the variable might not have been initialized. The same error occurs if the class initializes the variable in some, but not all, of the class's constructors.