Classes and other Concepts

Java™ How to Program, 10/e
Late Objects Version

References & Reading

- The content is mainly selected (sometimes modified) from the original slides provided by the authors of the textbook

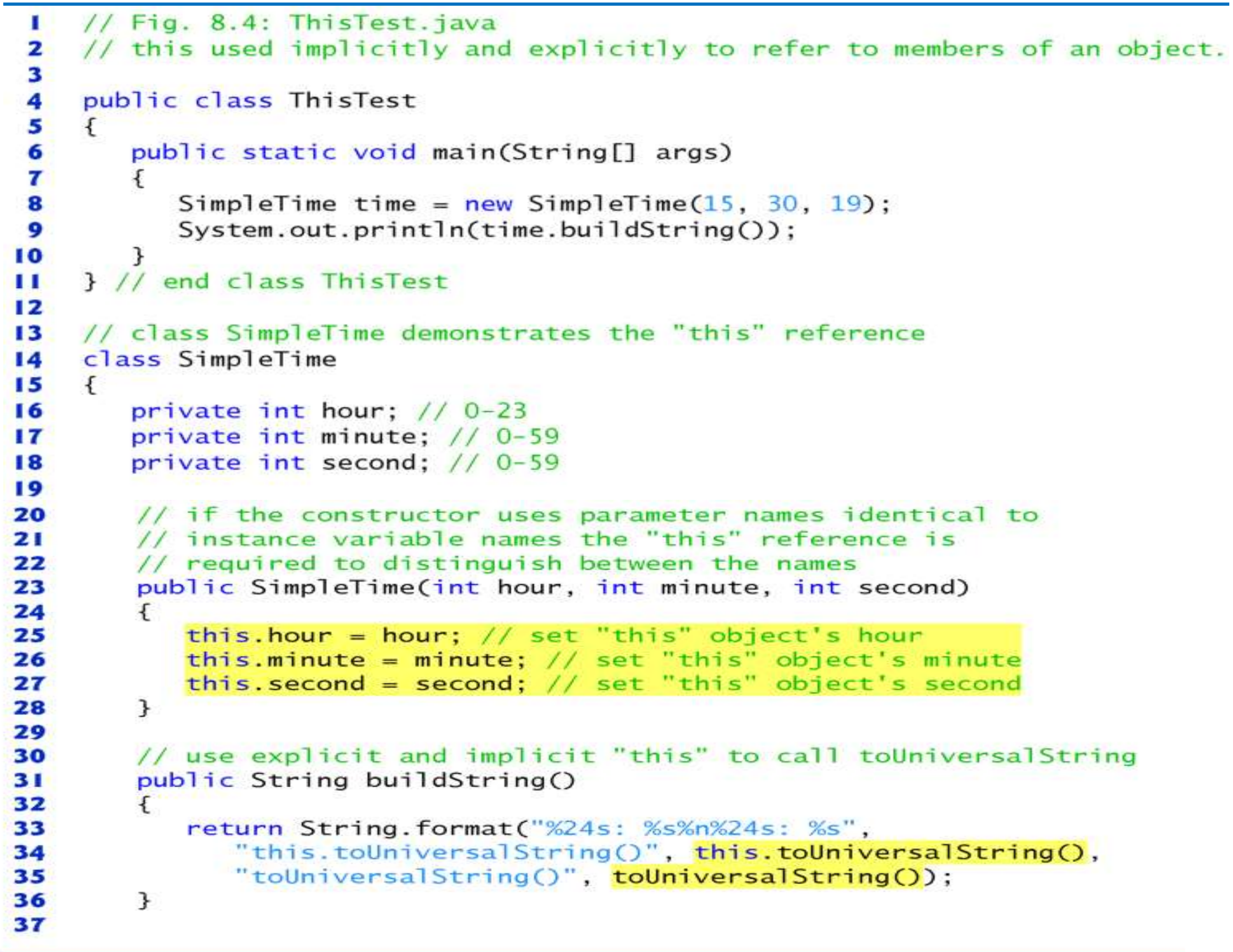
- Readings
  - Chapter 7: Introduction to Classes and Objects
  - Chapter 8: Classes and Objects: A Deeper Look

Outline

- **Referring to the Current Object's Members with the `this` Reference**
- **Garbage Collection**
- **`static` Class Members**
- **Additional Notes on This Example (*Notes on `static` Methods*)**
- **`static` Import**
- **`final` Instance Variables**
- **`Package Access`**

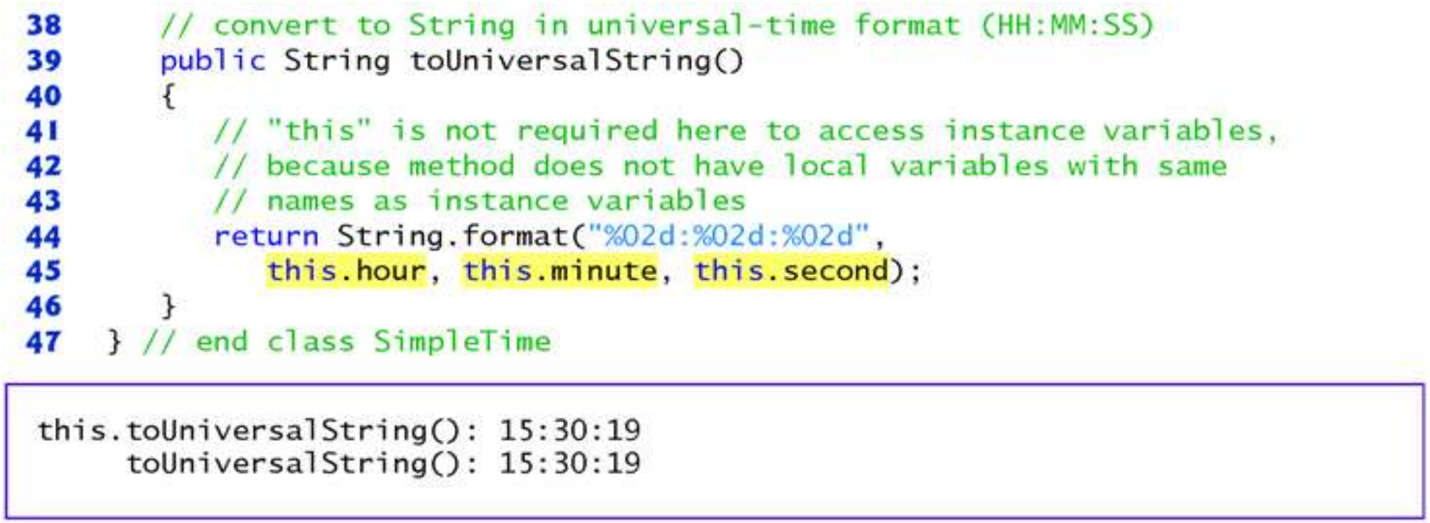
Referring to the Current Object's Members using `this` keyword

- Every object can access a reference to itself with keyword **`this`**.
- **`Commonly,`** **`this`** reference keyword can be used implicitly or explicitly.
- When an instance method is called for a particular object, the method's body *implicitly* uses keyword **`this`** to refer to the object's instance variables and other methods.
- We can also use keyword **`this`** *explicitly* in an instance method's body.
- When you compile a `.java` file containing more than one class, the compiler produces a separate class file with the `.class` extension for every compiled class and put them in the same directory.
- A source-code file can contain only *one* `public` class—otherwise, a compilation error occurs.
- Non-`public` classes can be used only by other classes in the *same package*.



```java
// Fig. 8.4: ThisTest.java
// this used implicitly and explicitly to refer to members of an object.

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

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

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

   // use explicit and implicit "this" to call toUniversalString
   public String buildString()
   {
      return String.format("%24s: %s%n%24s: %s",
         "this.toUniversalString()", this.toUniversalString(),
         "toUniversalString()", toUniversalString());
   }

```

Fig. 8.4 | this used implicitly and explicitly to refer to members of an object. (Part 2 of 3.)



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

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

- Notes:
  - SimpleTime declares three private instance variables— hour, minute and second.
  - **If** parameter names for the constructor are *identical* to the class's instance-variable names **then** this reference is used to refer to the instance variables.

static Class Members

- In certain cases, only one copy of a particular variable should be *shared* by all objects of a class. A **`static`** field—called a class variable— **is used** in such cases.
- A **`static variable`** represents classwide information—all objects of the class share the *same* piece of data.
- The declaration of a **`static`** variable begins with the keyword `static`.
- **`Static variables`** have *class scope* —they can be used in all of the class's methods.
- To access a **`public static`** member (variable or method) when no objects of the class exist (and even when they do), prefix the class name and a dot (`.`) to the **`static`** member, as in `Math.PI`.
- Can access a class's **`public static`** members through a reference to any object of the class
- **`private static`** class members can be accessed by client code only through methods of the class.

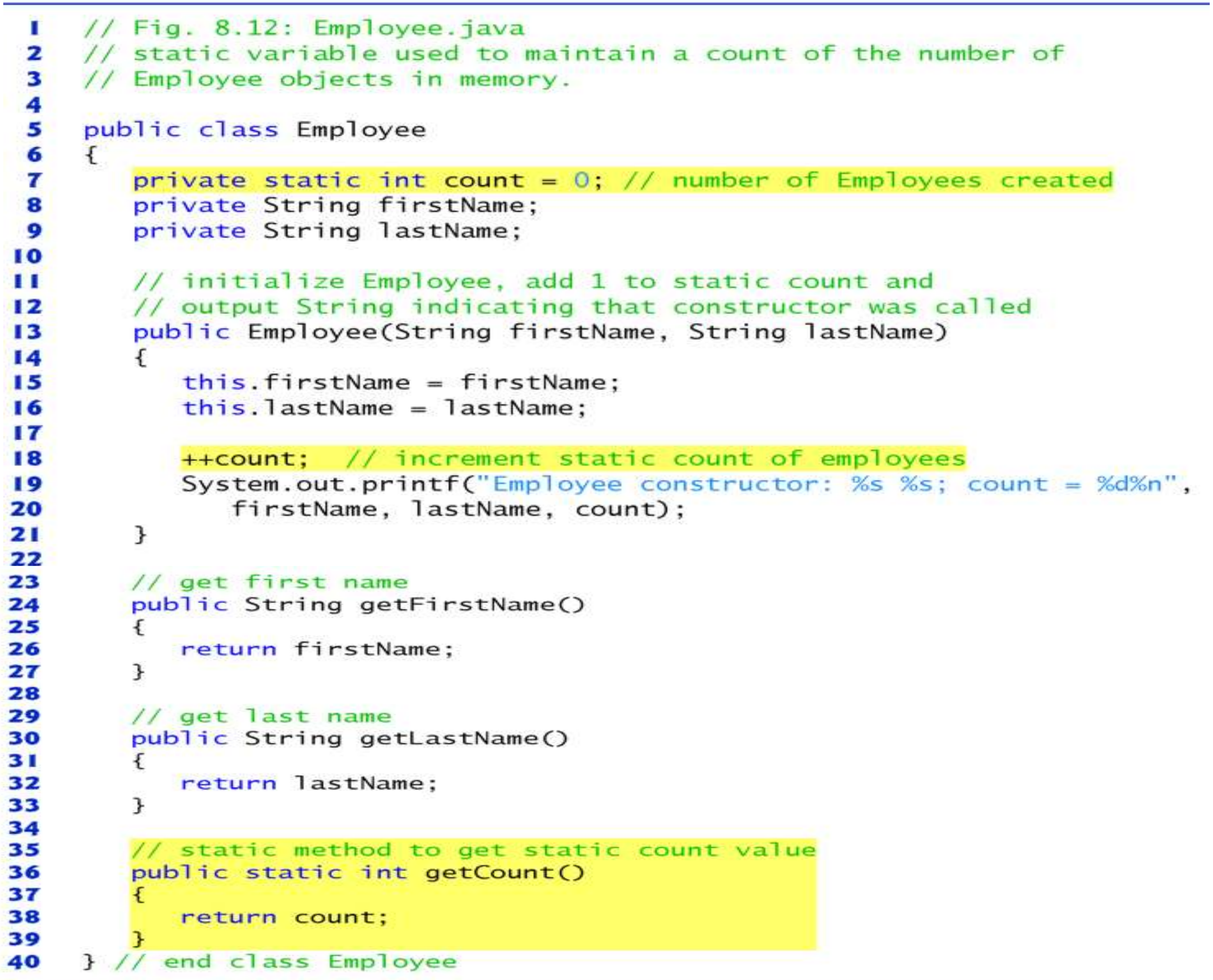
static Class Members (Cont.)

- static class members are available as soon as the class is loaded into memory at execution time.

- A static method *cannot* access a class's instance variables and instance methods, because a static method can be called even when no objects of the class have been instantiated.
  - For the same reason, the this reference *cannot* be used in a static method.
  - The this reference must refer to a specific object of the class, and **when** a static method is called, there might not be any objects of its class in memory.

- If a static variable is not initialized, the compiler assigns it a default value— 0 in case of variable of type int.
- A static method **can (1)** call other static methods of the same class directly (i.e., using the method name by itself) and **(2) can** manipulate static variables in the same class directly.

- Instance methods can access all fields (static variables and instance variables) and methods of the class.

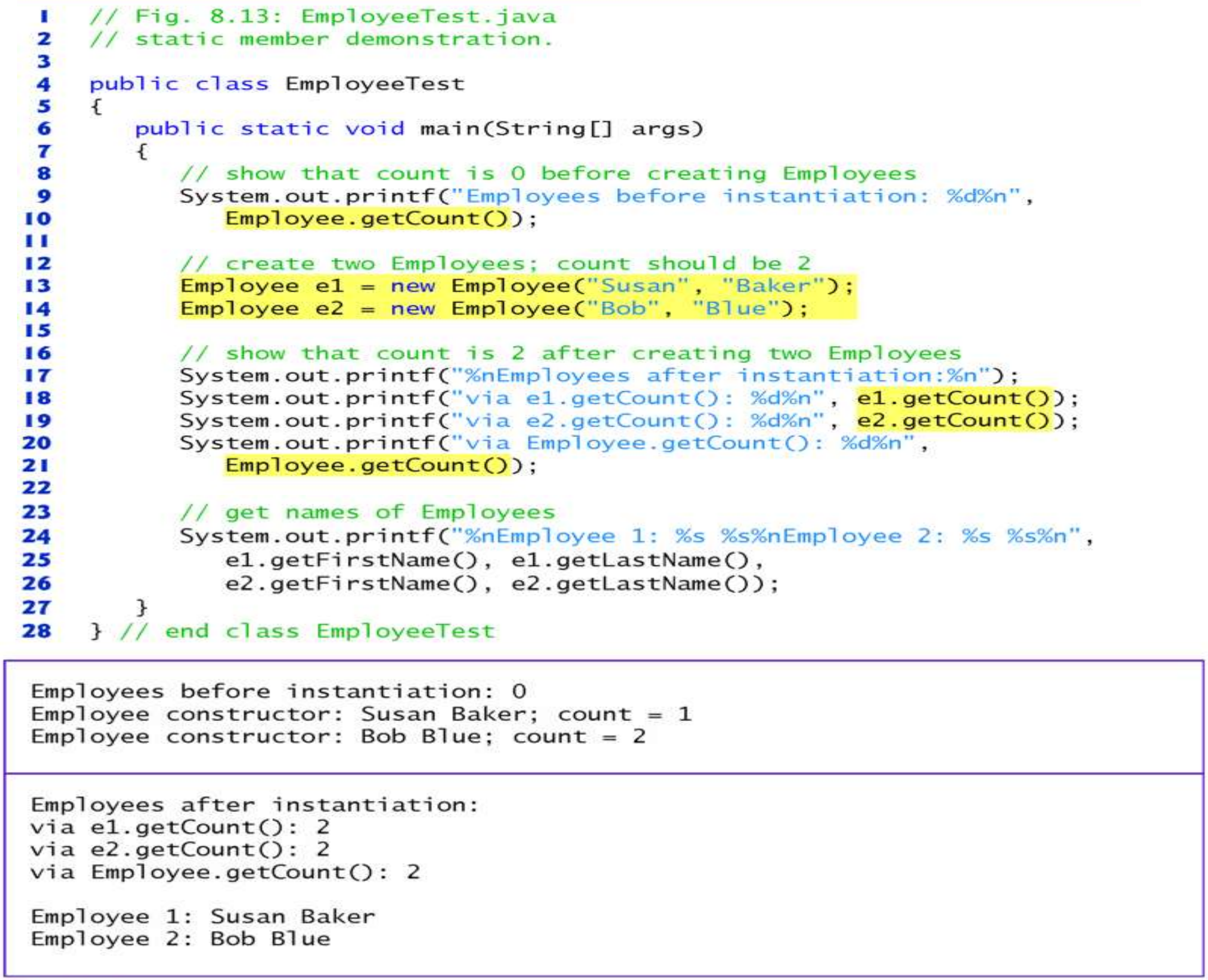


```java
// Fig. 8.12: Employee.java
// static variable used to maintain a count of the number of
// Employee objects in memory.

public class Employee
{
   private static int count = 0; // number of Employees created
   private String firstName;
   private String lastName;

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

      ++count;  // increment static count of employees
      System.out.printf("Employee constructor: %s %s; count = %d%n",
         firstName, lastName, count);
   }

   // get first name
   public String getFirstName()
   {
      return firstName;
   }

   // get last name
   public String getLastName()
   {
      return lastName;
   }

   // static method to get static count value
   public static int getCount()
   {
      return count;
   }
} // end class Employee
```

Fig. 8.12 | static variable used to maintain a count of the number of Employee objects in memory. (Part 2 of 2.)

static Class Members (Cont.)

- String objects in Java are immutable—they cannot be modified after they are created.
  - Therefore, it’s safe to have many references to one String object.
  - This is not normally the case for objects of most other classes in Java.
- If String objects are immutable, you might wonder why are we able to use operators + and += to concatenate String objects.
- String-concatenation actually results in a *new* String object containing the concatenated values—the original String objects are *not* modified.



```java
// Fig. 8.13: EmployeeTest.java
// static member demonstration.

public class EmployeeTest
{
   public static void main(String[] args)
   {
      // show that count is 0 before creating Employees
      System.out.printf("Employees before instantiation: %d%n",
         Employee.getCount());

      // create two Employees; count should be 2
      Employee e1 = new Employee("Susan", "Baker");
      Employee e2 = new Employee("Bob", "Blue");

      // show that count is 2 after creating two Employees
      System.out.printf("%nEmployees after instantiation:%n");
      System.out.printf("via e1.getCount(): %d%n", e1.getCount());
      System.out.printf("via e2.getCount(): %d%n", e2.getCount());
      System.out.printf("via Employee.getCount(): %d%n",
         Employee.getCount());

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

```
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
```

Fig. 8.13 | static member demonstration. (Part 2 of 2.)

static Import

- A **static** import declaration enables you to import only the static members of a class or interface so you can access them via their *unqualified names* in your class— **that is**, *the class name and a dot (.) are not required* when *using an imported* **static** *member*.

- Two forms of importing static members:
  - single **static** import: imports a particular static member
  - **static** import on demand: imports all static members of a class.

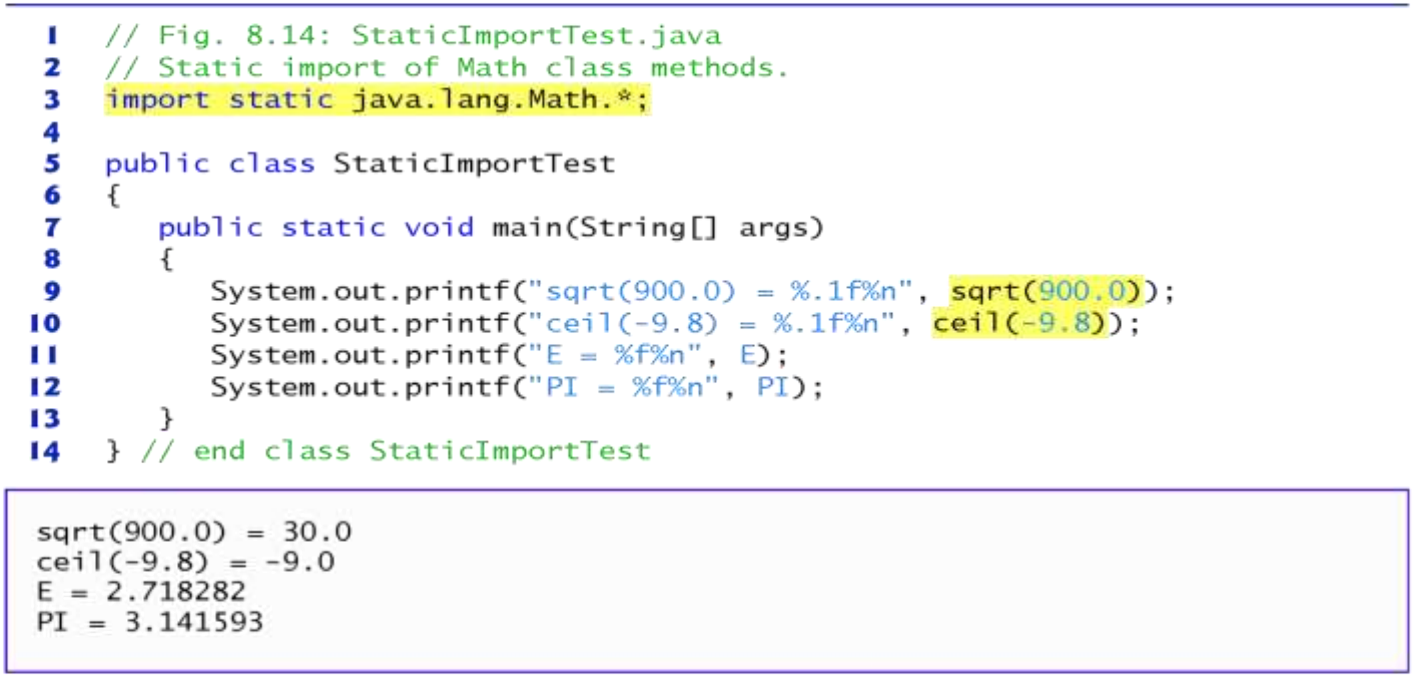
- The following syntax imports a particular static member:

```
    import static packageName.ClassName.staticMemberName;
```

- The following syntax imports all static members of a class:

```
    import static packageName.ClassName.*;
```

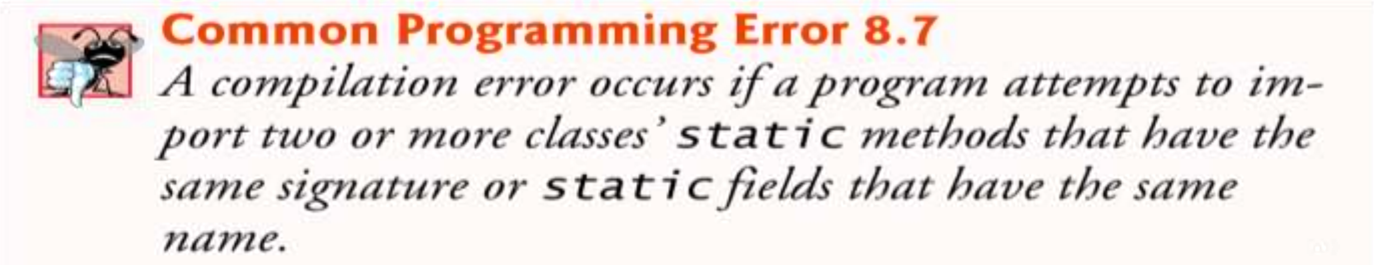
  - * indicates that *all* static members of the specified class should be available for use in the class(es) declared in the file.



```java
// Fig. 8.14: StaticImportTest.java
// Static import of Math class methods.
import static java.lang.Math.*;

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

```
sqrt(900.0) = 30.0
ceil(-9.8) = -9.0
E = 2.718282
PI = 3.141593
```

Fig. 8.14 | static import of Math class methods.



Common Programming Error 8.7

A compilation error occurs if a program attempts to import two or more classes' `static` methods that have the same signature or `static` fields that have the same name.

`final` Instance Variables

- The principle of least privilege is fundamental to good software engineering.
  - Code should be granted only the amount of privilege and access that it needs to accomplish its designated task, but no more.
  - Makes your programs more robust by preventing code from accidentally (or maliciously) modifying variable values and calling methods that should not be accessible.

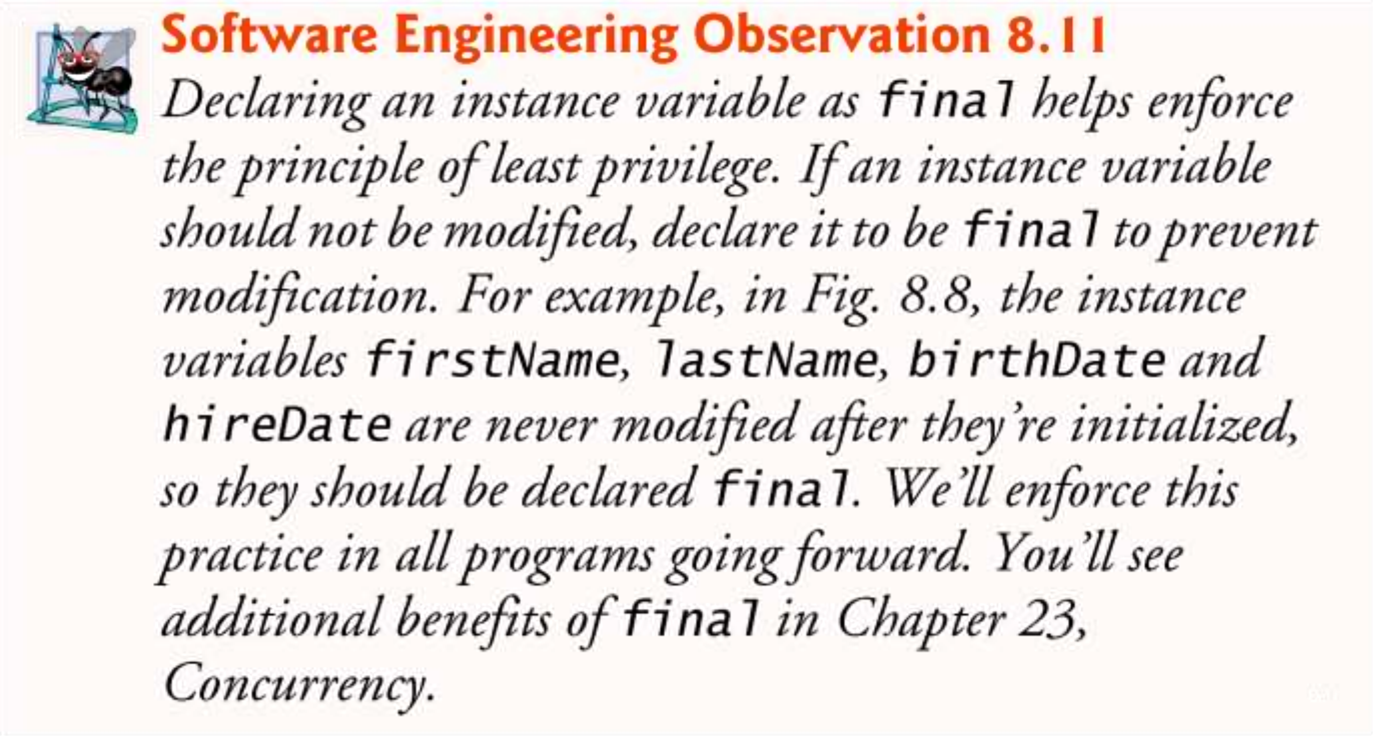
- Keyword `final` specifies that a variable is not modifiable (i.e., it's a constant) and any attempt to modify it by assignment after it's initialized is an error.

EX:

```
    private final int INCREMENT;
```

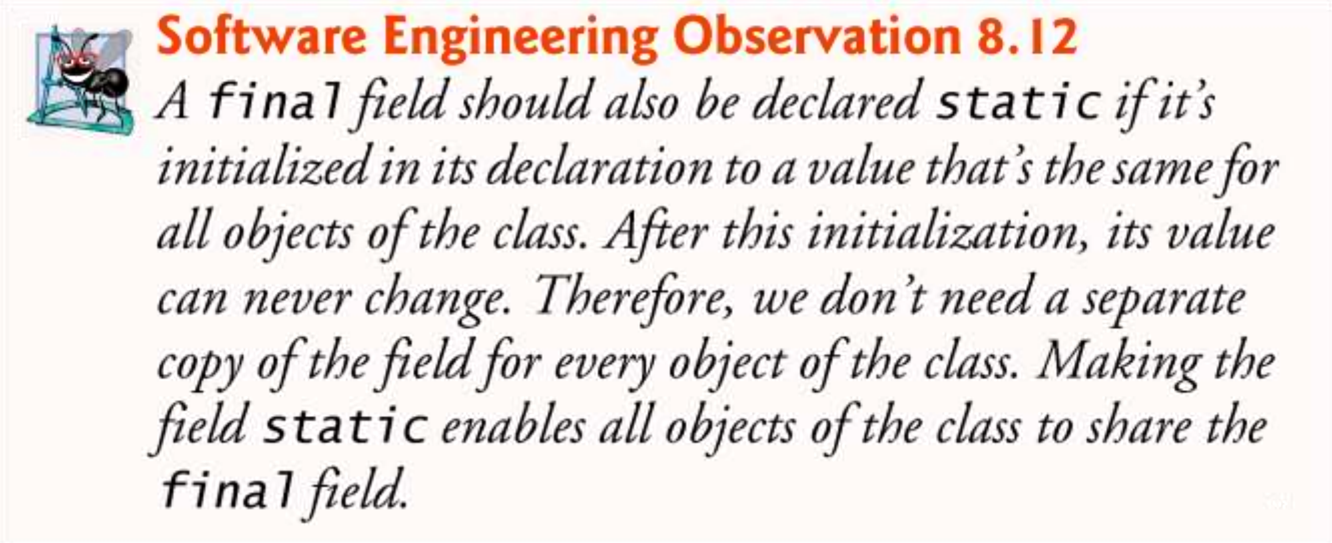
// Declares a `final` (constant) instance variable `INCREMENT` of type `int`.

- `final` variables can be initialized when (1) they are declared or (2) by each of the class's constructors **so that :** each object of the class has a different value.
- If a class provides multiple constructors, every one would be required to initialize each `final` variable.
- If a `final` variable is not initialized, then a compilation error occurs.



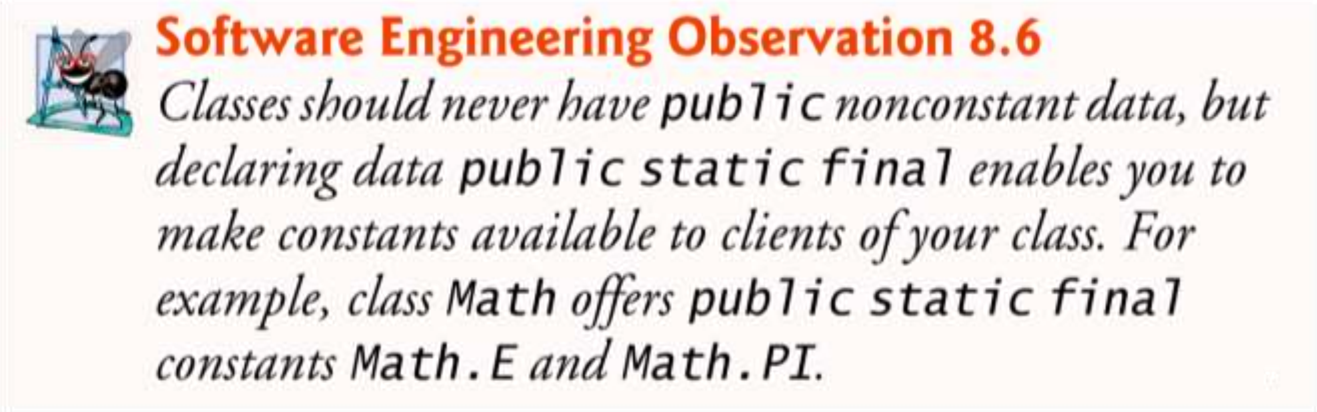
Software Engineering Observation 8.11

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. For example, in Fig. 8.8, the instance variables* `firstName`, `lastName`, `birthDate` *and* `hireDate` *are never modified after they're initialized, so they should be declared* `final`. *We'll enforce this practice in all programs going forward. You'll see additional benefits of* `final` *in Chapter 23, Concurrency.*



Software Engineering Observation 8.12

A `final` field should also be declared `static` if it's initialized in its declaration to a value that's the same for all objects of the class. After this initialization, its value can never change. Therefore, we don't need 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.



Software Engineering Observation 8.6

Classes should never have `public` nonconstant data, but declaring data `public static final` enables you to make constants available to clients of your class. For example, class `Math` offers `public static final` constants `Math.E` and `Math.PI`.

Self-Reading

Garbage Collection

- Every object uses system resources, such as memory.
  - Need a disciplined way to give resources back to the system when they're no longer needed; otherwise, "resource leaks" might occur.
- The JVM performs automatic garbage collection to reclaim the *memory* occupied by objects that are no longer used.
  - When there are *no more references* to an object, the object is *eligible* to be collected.
  - Collection typically occurs when the JVM executes its garbage collector, which may not happen for a while, or even at all before a program terminates.
- Memory leaks that are common in other languages like C and C++ (because memory is *not* automatically reclaimed in those languages) are *less* likely in Java, but some can still happen in subtle ways.

Garbage Collection (Cont.)

Notes about Class `Object`'s `finalize` Method:

- Every class in Java has the methods of class `Object` (package `java.lang`), one of which is method finalize.
- You should *never* use method `finalize`, because it can cause many problems.
- The original intent of `finalize` was to allow the garbage collector to perform termination housekeeping on an object just before reclaiming the object's memory.