Classes and Objects

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

1.5 Introduction to Object Technology

- Objects, or more precisely, the *classes* objects come from, are essentially *reusable* software components.
  - There are date objects, time objects, audio objects, video objects, automobile objects, people objects, etc.
  - Almost any *noun* can be reasonably represented as a software object in terms of *attributes* (e.g., name, color and size) and *behaviors* (e.g., calculating, moving and communicating).
- Software development groups can use a modular, object-oriented design-and-implementation approach to be much more productive than with earlier popular techniques like "structured programming"—object-oriented programs are often easier to understand, correct and modify.

Methods and Classes

- In Java, we create a program unit called a class to house the set of methods that perform the class's tasks.
- Performing a task in a program requires a method.
- The method houses the program statements that actually perform its tasks and hides these statements from its user
- Example of class: `Account`
- Example of methods: `withdraw()` and `deposit()`

Instantiation

- Just as someone has to *build* a car from its engineering drawings before you can actually drive a car, you must *build an object* of a class before a program can perform the tasks that the class's methods define.
- An object is then referred to as an instance of its class.
- Example of instantiating an object:

```
Account account1 = new Account();
```

`account1` is an instance or object of the `Account` class.

Reusability

- Just as a car's engineering drawings can be *reused* many times to build many cars, you can reuse a class many times to build many objects.
- Reuse of existing classes when building new classes and programs saves time and effort.
- Reuse also helps you build more reliable and effective systems, because existing classes and components often have undergone extensive *testing*, *debugging* and *performance* tuning.

Attributes and Instance Variables

- A car has *attributes like :* Color, its number of doors, the amount of gas in its tank, its current speed and its record of total miles driven (i.e., its odometer reading).
- The car's attributes are represented as part of its design in its engineering diagrams.
- Every car maintains its own attributes.
- Each car knows how much gas is in its own gas tank, but not how much is in the tanks of other cars.

Attributes and Instance Variables (Cont.)

- An object, has attributes that it carries along as it's used in a program.
- An `Account` object has a `balance` *attribute* that represents the amount of money in the account.
- Each `Account` object knows the balance in the account it represents, but *not* the balances of the *other* accounts in the bank.
- Attributes are specified by the class's instance variables.

Encapsulation

- Classes (and their objects) encapsulate, i.e., encase, their attributes and methods.
- Objects may communicate with one another, but they're normally not allowed to know how other objects are implemented—implementation details are *hidden* within the objects themselves.
- Information hiding, as we'll see, is crucial to good software engineering.

Inheritance

- A new class of objects can be created conveniently by inheritance—the new class (called the subclass) starts with the characteristics of an existing class (called the superclass), possibly customizing them and adding unique characteristics of its own.

- Example: an object of class "convertible" certainly *is an* object of the more *general* class "automobile", but more *specifically*, the roof can be raised or lowered.

Interfaces

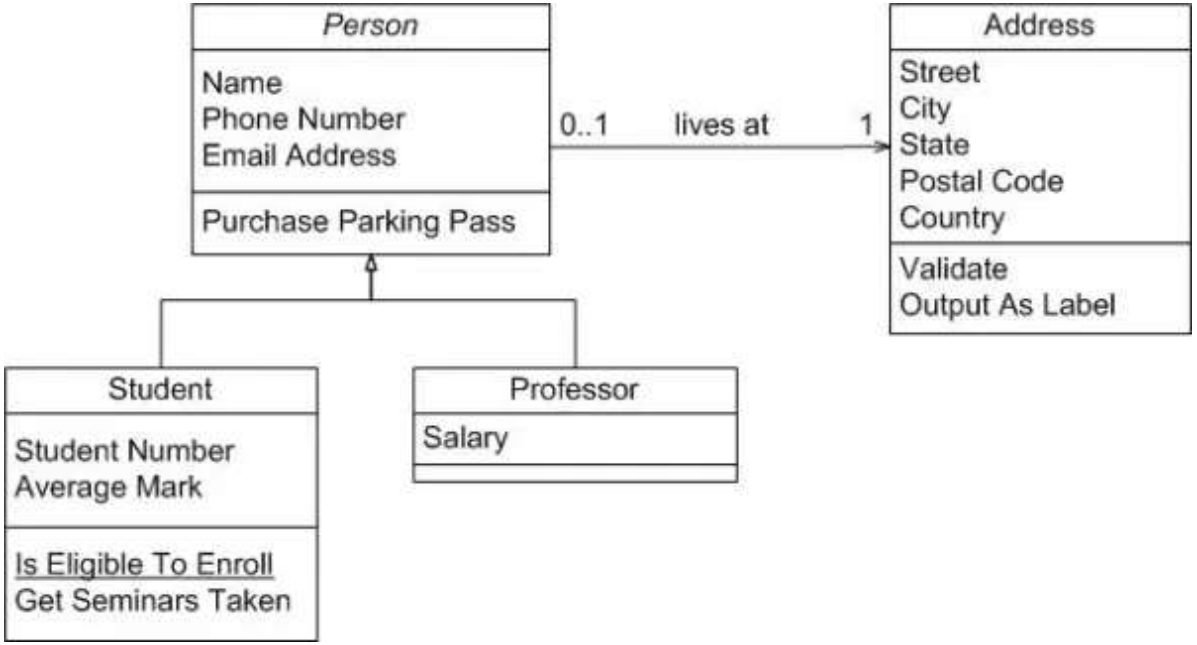
- Interfaces are collections of related methods that typically enable you to tell objects *what* to do, but not *how* to do it
- This feature allows programmers to work similarly with different APIs since they implement the same interface(s).
- A class implements zero or more interfaces, each of which can have one or more methods, just as a car implements separate interfaces for basic driving functions, controlling the radio, controlling the heating and air conditioning systems, and the like.

Object-Oriented Analysis and Design (OOAD)

- How will you create the code (i.e., the program instructions) for your programs?
- Follow a detailed analysis process for determining your project's requirements (i.e., defining *what* the system is supposed to do)
- Develop a design that satisfies them (i.e., specifying *how* the system should do it).
- Carefully review the design (and have your design reviewed by other software professionals) before writing any code.
- Analyzing and designing your system from an object-oriented point of view is called an object-oriented-analysis-and-design (OOAD) process.
- Languages like Java are object oriented.
- Object-oriented programming (OOP) allows you to implement an object-oriented design as a working system.

The UML (Unified Modeling Language)

- The Unified Modeling Language (UML) is the most widely used graphical scheme for modeling object-oriented systems.



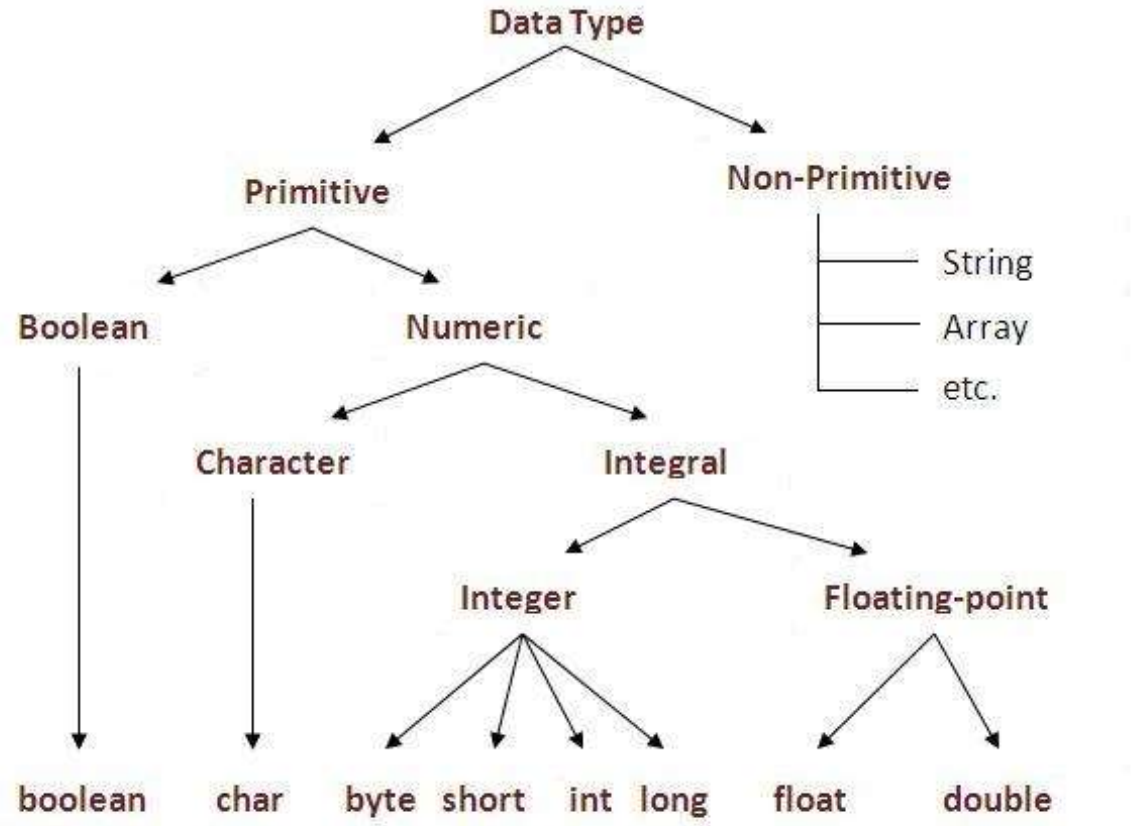
Primitive Types vs. Reference Types

- Java's types are divided into primitive types and reference types.
- Primitive types: `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`.
  - Appendix D lists the eight primitive types in Java.
- All nonprimitive types (classes) are *reference types*.

Software Engineering Observation 6.1

A variable's declared type (e.g., `int`, `double` or `Scanner`) indicates whether the variable is of a primitive or a reference type. If a variable's type is not one of the eight primitive types, then it's a reference type.

Primitive Types vs. Reference Types (cont.)



Primitive Types vs. Reference Types (cont.)

- A primitive-type variable can hold exactly *one* value of its declared type at a time.
- Programs use variables of reference types (normally called references) to store the *addresses* of objects in the computer's memory.
  - Such a variable is said to refer to an object in the program.
- To call an object's methods, you need a reference to the object.
- If an object's method requires additional data to perform its task, then you'd pass arguments in the method call.
- Primitive-type variables do *not* refer to objects, so such variables cannot be used to invoke methods.

Instance Variables, *set* Methods and *get* Methods

- Each class you create becomes a new type that can be used to declare variables and create objects.
- You can declare new classes as needed.
- Declaring instance variables `private` is known as **data hiding or information hiding**.

```
1  // Fig. 7.1: Account.java
2  // Account class that contains a name instance variable
3  // and methods to set and get its value.
4
5  public class Account
6  {
7     private String name; // instance variable
8
9     // method to set the name in the object
10    public void setName(String name)
11    {
12       this.name = name; // store the name
13    }
14
15    // method to retrieve the name from the object
16    public String getName()
17    {
18       return name; // return value of name to caller
19    }
20 } // end class Account
```

Fig. 7.1 | `Account` class that contains a `name` instance variable and methods to *set* and *get* its value.

Class Declaration

- Each class declaration that begins with the access modifier `public` must be stored in a file that has the same name as the class and ends with the `.java` filename extension.
- Every class declaration starts with the keyword **`class`** followed by the class name.
- Class, method and variable names are identifiers.
- By convention all use camel case names.
- Class names begin with an uppercase letter, and method and variable names begin with a lowercase letter.

Instance Variable name

- An object has attributes that are implemented as instance variables and carried with it throughout its lifetime.
- Instance variables exist before methods are called on an object, while the methods are executing and after the methods complete execution.
- A class normally contains one or more methods that manipulate the instance variables that belong to particular objects of the class.
- Instance variables are declared inside a class declaration but outside the bodies of the class's method declarations.
- Each object (instance) of the class has its own copy of each of the class's instance variables.

Time1 class declaration maintains the time in 24-hour format

```
// Fig. 8.1: Time1 class declaration maintains the time in 24-hour format.
package pkg8_02;
public class Time1
{
    private int hour; // 0 - 23
    private int minute; // 0 - 59
    private int second; // 0 - 59
     // set a time value using universal time; throw an exception if the hour, minute or second is invalid
     public void setTime(int hour, int minute, int second)
     {     // validate hour, minute and second
          if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 || second < 0 || second >= 60)
          {       throw new IllegalArgumentException( "hour, minute and/or second was out of range");    }
           this.hour = hour;
           this.minute = minute;
           this.second = second;
      }  // end method setTime

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

      public String toString() // convert to String in standard-time format (H:MM:SS AM or PM)
      {     return String.format("%d:%02d:%02d %s", ((hour == 0 || hour == 12) ? 12: hour % 12), minute, second, (hour < 12
? "AM" : "PM"));   }
} // end class Time1
```

Controlling Access: `public` and `private` Access Modifiers

- Most instance-variable declarations are preceded with the keyword **`private`**, which is an access modifier.
- Variables or methods declared with access modifier `private` are accessible only to methods of the class in which they are declared.
- Access modifiers `public` and `private` control access to a class's variables and methods.
  - Chapter 9 introduces access modifier `protected`.
- public methods present to the class's clients a view of the services the class provides (the class's public interface).
- Clients are not concerned with how the class accomplishes its tasks.
  - Accordingly, the class's private variables and private methods (i.e., its implementation details) are not accessible to its clients.
- Note: private class members are not accessible outside the class.

```
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    }
13 } // end class MemberAccessTest
```

Fig. 8.3 | Private members of class `Time1` are not accessible. (Part 1 of 2.)

Software Engineering Observation 8.2

Recall from Chapter 3 that methods declared with access modifier `private` *can be called* only *by other methods of the class in which the* `private` *methods are declared. Such methods are commonly referred to as* ***utility methods*** *or* ***helper methods*** *because they're typically used to support the operation of the class's other methods.*

```
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
```

Fig. 8.3 | Private members of class `Time1` are not accessible. (Part 2 of 2.)

Common Programming Error 8.1

An attempt by a method that's not a member of a class to access a `private` *member of that class generates a compilation error.*

Software Engineering Observation 8.5

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`*).*

Software Engineering Observation 8.4

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.

Package Access

- If no access modifier is specified for a method or variable when it's declared in a class, the method or variable is considered to have package access.
- In a program that uses *multiple* classes from the *same* package, these classes can access each other's package-access members directly through references to objects of the appropriate classes, or in the case of `static` members through the class name.
- Package access is rarely used.

```
1  // Fig. 8.15: 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    }
21 } // end class PackageDataTest
22
```

Fig. 8.15 | Package-access members of a class are accessible by other classes in the same package. (Part 1 of 3.)

```
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     }
35
36     // return PackageData object String representation
37     public String toString()
38     {
39        return String.format("number: %d; string: %s", number, string);
40     }
41  } // end class PackageData
```

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

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

Fig. 8.15 | Package-access members of a class are accessible by other classes in the same package. (Part 3 of 3.)

7.2.1 `Account` Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

Instance Method `setName` of Class `Account`

- In the preceding chapters (in CS140), you've declared only `static` methods in each class.
- A class's `non-static` methods are known as instance methods.
- Method `setName`'s declaration indicates that `setName` receives parameter name of type `String`—which represents the name that will be passed to the method as an argument.
- If a method contains a local variable with the *same* name as an instance variable, that method's body will refer to the local variable rather than the instance variable.
- In this case, the local variable is said to ***shadow*** the instance variable in the method's body.
- The method's body can use the keyword `this` to refer to the shadowed instance variable explicitly, as shown on the left side of the assignment in line 12.

`Account` Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

- Recall that variables declared in the body of a particular method are local variables and can be used only in that method and that a method's parameters also are local variables of the method.
- The Method setName's body contains a single statement that **assigns** the value of the name *parameter* (a String) to the class's name *instance variable*, thus storing the account name in the object.

- The Method getName returns a particular `Account` object's `name` to the caller.
- The method has an empty parameter list, so it does *not* require additional information to perform its task.

Driver Class `AccountTest`

- A class that creates an object of another class, then calls the object's methods, is a driver class.

```
1  // Fig. 7.2: AccountTest.java
2  // Creating and manipulating an Account object.
3  import java.util.Scanner;
4
5  public class AccountTest
6  {
7     public static void main(String[] args)
8     {
9        // create a Scanner object to obtain input from the command window
10       Scanner input = new Scanner(System.in);
11
12       // create an Account object and assign it to myAccount
13       Account myAccount = new Account();
14
15       // display initial value of name (null)
16       System.out.printf("Initial name is: %s%n%n", myAccount.getName());
17
18       // prompt for and read name
19       System.out.println("Please enter the name:");
20       String theName = input.nextLine(); // read a line of text
21       myAccount.setName(theName); // put theName in myAccount
22       System.out.println(); // outputs a blank line
23
```

Fig. 7.2 | Creating and manipulating an **Account** object. (Part 1 of 2.)

```
24         // display the name stored in object myAccount
25         System.out.printf("Name in object myAccount is:%n%s%n",
26            myAccount.getName());
27      }
28   } // end class AccountTest
```

```
Initial name is: null

Please enter the name:
Jane Green

Name in object myAccount is:
Jane Green
```

Fig. 7.2 | Creating and manipulating an **Account** object. (Part 2 of 2.)

Instantiating an Object—Keyword `new` and Constructors

- A class instance creation expression begins with keyword `new` and creates a new object.
- A constructor is similar to a method but is called implicitly by the `new` operator to initialize an object's instance variables at the time the object is *created*.

Calling Class Account's getName & setName Methods

- To call a method of an object, follow the object name with a dot separator, the method name and a set of parentheses containing the method's arguments.

Compiling and Executing an App with Multiple Classes

- The **javac** command can compile *multiple* classes at once.
- Simply list the **source-code** filenames after the command with each filename separated by a space from the next.
- If the directory containing the app includes **only one app's files**, you can compile all of its classes with the command

  javac *.java.

- The asterisk (*) in *.java indicates that all files in the current directory ending with the filename extension ".java" should be compiled.

Account UML Class Diagram with an Instance Variable and set and get Methods

- We'll often use UML class diagrams to summarize a class's *attributes* and *operations*.
- UML diagrams help systems designers specify a system in a concise, graphical, programming-language-independent manner, before programmers implement the system in a specific programming language.
- Figure 7.3 presents a UML class diagram for class `Account` of Fig. 7.1.

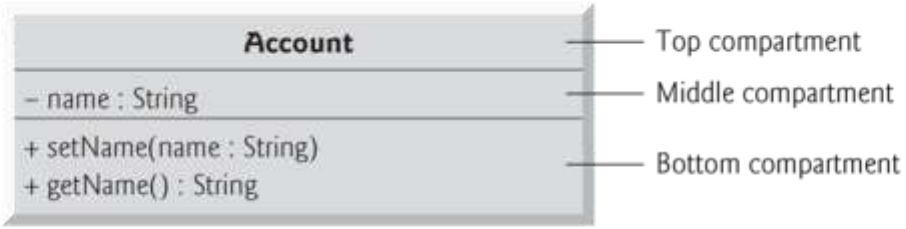


Fig. 7.3 | UML class diagram for class `Account` of Fig. 7.1.

Account UML Class Diagram with an Instance Variable and set & get Methods (Cont.)

Top Compartment

- In the UML, each class is modeled in a class diagram as a rectangle with three compartments.
- The top one contains the class's name centered horizontally in boldface.

Middle Compartment

- The middle compartment contains the class's attributes (attribute name, followed by a colon and the type), which correspond to instance variables in Java.
- Private attributes are preceded by a minus sign (−) in the UML.

Bottom Compartment

- The bottom compartment contains the class's operations, which correspond to methods and constructors in Java.
- A plus sign (+) in front of the operation name indicates that the operation is a public one in the UML.

Account UML Class Diagram with return types and parameters of Methods

Return Types

- The UML indicates an operation's return type by placing a colon and the return type after the parentheses following the operation name.
- UML class diagrams do not specify return types for operations that do not return values.

Parameters

- The UML models a parameter of an operation by listing the parameter name, followed by a colon and the parameter type between the parentheses after the operation name

Additional Notes on Class AccountTest

Notes on** import **Declarations

- Most classes you'll use in Java programs must be imported explicitly.
- Classes that are compiled in the same directory are considered to be in the same package—known as the default package.
- Classes in the same package are implicitly imported into the source-code files of other classes in that package (no need to be explicitly imported) .
- An import- declaration is not required if you always refer to a class with its fully qualified class name, which includes its package name and class name.

Default and Explicit Initialization for Instance Variables

- Recall that local variables are *not* initialized by default.
- Primitive-type instance variables ***are* initialized by default**—instance variables of types **byte, char, short, int, long, float** and **double** are initialized to 0, and variables of type **boolean** are initialized to `false`.
- You can specify your own initial value for a primitive-type instance variable by assigning the variable a value in its declaration, as in
  - `private int numberOfStudents = 10;`
- Reference-type instance variables (such as those of type String), if not explicitly initialized, are initialized by default to the value `null`—which represents a "reference to nothing."

Account Class: Initializing Objects with Constructors

- Every class must have at least one constructor.
- If a class does not define constructors, the compiler creates a default constructor that takes no parameters
- ***Note***: *The default constructor initializes the instance variables to the initial values specified in their declarations or to their default values (zero for primitive numeric types,* `false` *for* `boolean` *values and* `null` *for references).*
- Each declared class can optionally provide a constructor with (or without) parameters
- *Constructors cannot return values.*
- IF you declare a constructor for a class, THEN the compiler will *not* create a *default constructor* for that class.

Notes:

- Local variables are not automatically initialized.
- Every instance variable has a default initial value—a value provided by Java when you do not specify the instance variable's initial value.
- The default value for an instance variable of type `String` is `null`.

Declaring an Account Constructor for Custom Object Initialization

```
1  // Fig. 7.5: Account.java
2  // Account class with a constructor that initializes the name.
3
4  public class Account
5  {
6     private String name; // instance variable
7
8     // constructor initializes name with parameter name
9     public Account(String name) // constructor name is class name
10    {
11       this.name = name;
12    }
13
14    // method to set the name
15    public void setName(String name)
16    {
17       this.name = name;
18    }
19
20    // method to retrieve the name
21    public String getName()
22    {
23       return name;
24    }
25 } // end class Account
```

Fig. 7.5 | `Account` class with a constructor that initializes the `name`.

Class AccountTest: Initializing Account Objects When They're Created

```
1  // Fig. 7.6: AccountTest.java
2  // Using the Account constructor to initialize the name instance
3  // variable at the time each Account object is created.
4
5  public class AccountTest
6  {
7     public static void main(String[] args)
8     {
9        // create two Account objects
10       Account account1 = new Account("Jane Green");
11       Account account2 = new Account("John Blue");
12
13       // display initial value of name for each Account
14       System.out.printf("account1 name is: %s%n", account1.getName());
15       System.out.printf("account2 name is: %s%n", account2.getName());
16    }
17 } // end class AccountTest
```

```
account1 name is: Jane Green
account2 name is: John Blue
```

Fig. 7.6 | Using the `Account` constructor to initialize the `name` instance variable at the time each `Account` object is created.

Software Engineering Observation 7.3

Unless default initialization of your class's instance variables is acceptable, provide a custom constructor to ensure that your instance variables are properly initialized with meaningful values when each new object of your class is created.

Error-Prevention Tip 7.2

Even though it's possible to do so, do not call methods from constructors. We'll explain this in Chapter 10, Object-Oriented Programming: Polymorphism and Interfaces.

A Constructor of Class Account's UML Class Diagram

- The UML models constructors as operations
- To distinguish a constructor from a class's operations, the UML places the word "constructor" between guillemets (« and ») before the constructor's name.

Fig. 7.7 | UML class diagram for **Account** class of Fig. 7.5.

Default and No-Argument Constructors

- If your class declares constructors, the compiler will *not* create a default constructor.
  - In this case, you must declare a no-argument constructor if default initialization is required.
  - Like a default constructor, a no-argument constructor is invoked with empty parentheses.

`Time` Class Case Study: Overloaded Constructors

- Class `Time2` represents the time of day.
- `private int` instance variables `hour`, `minute` and `second` represent the time in universal-time format (24-hour clock format in which hours are in the range 0–23, and minutes and seconds are each in the range 0–59).
- `public` methods `setTime`, `toUniversalString` and `toString` are called the `public` services or the `public` interface that the class provides to its clients.

- Overloaded constructors enable objects of a class to be initialized in different ways.
- To overload constructors, simply provide multiple constructor declarations with different signatures.
- Recall that the compiler differentiates signatures by the *number* of parameters, the *types* of the parameters and the *order* of the parameter types in each signature.
- Class `Time2` (Fig. 8.5) contains five overloaded constructors that provide convenient ways to initialize objects.

```
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:
11    // initializes each instance variable to zero
12    public Time2()
13    {
14       this(0, 0, 0); // invoke constructor with three arguments
15    }
16
17    // Time2 constructor: hour supplied, minute and second defaulted to 0
18    public Time2(int hour)
19    {
20       this(hour, 0, 0); // invoke constructor with three arguments
21    }
22
```

Fig. 8.5 | Time2 class with overloaded constructors. (Part 1 of 6.)

```
23      // Time2 constructor: hour and minute supplied, second defaulted to 0
24      public Time2(int hour, int minute)
25      {
26         this(hour, minute, 0); // invoke constructor with three arguments
27      }
28
29      // Time2 constructor: hour, minute and second supplied
30      public Time2(int hour, int minute, int second)
31      {
32         if (hour < 0 || hour >= 24)
33            throw new IllegalArgumentException("hour must be 0-23");
34
35         if (minute < 0 || minute >= 60)
36            throw new IllegalArgumentException("minute must be 0-59");
37
38         if (second < 0 || second >= 60)
39            throw new IllegalArgumentException("second must be 0-59");
40
41         this.hour = hour;
42         this.minute = minute;
43         this.second = second;
44      }
45
```

Fig. 8.5 | Time2 class with overloaded constructors. (Part 2 of 6.)

```
46      // Time2 constructor: another Time2 object supplied
47      public Time2(Time2 time)
48      {
49         // invoke constructor with three arguments
50         this(time.getHour(), time.getMinute(), time.getSecond());
51      }
52
53      // Set Methods
54      // set a new time value using universal time;
55      // validate the data
56      public void setTime(int hour, int minute, int second)
57      {
58         if (hour < 0 || hour >= 24)
59            throw new IllegalArgumentException("hour must be 0-23");
60
61         if (minute < 0 || minute >= 60)
62            throw new IllegalArgumentException("minute must be 0-59");
63
64         if (second < 0 || second >= 60)
65            throw new IllegalArgumentException("second must be 0-59");
66
67         this.hour = hour;
68         this.minute = minute;
69         this.second = second;
70      }
```

Fig. 8.5 | Time2 class with overloaded constructors. (Part 3 of 6.)

```
71
72      // validate and set hour
73      public void setHour(int hour)
74      {
75         if (hour < 0 || hour >= 24)
76            throw new IllegalArgumentException("hour must be 0-23");
77
78         this.hour = hour;
79      }
80
81      // validate and set minute
82      public void setMinute(int minute)
83      {
84         if (minute < 0 && minute >= 60)
85            throw new IllegalArgumentException("minute must be 0-59");
86
87         this.minute = minute;
88      }
89
```

Fig. 8.5 | Time2 class with overloaded constructors. (Part 4 of 6.)

```
90      // validate and set second
91      public void setSecond(int second)
92      {
93         if (second >= 0 && second < 60)
94            throw new IllegalArgumentException("second must be 0-59");
95
96         this.second = second;
97      }
98
99      // Get Methods
100     // get hour value
101     public int getHour()
102     {
103        return hour;
104     }
105
106     // get minute value
107     public int getMinute()
108     {
109        return minute;
110     }
```

Fig. 8.5 | Time2 class with overloaded constructors. (Part 5 of 6.)

```
111
112    // get second value
113    public int getSecond()
114    {
115       return second;
116    }
117
118    // convert to String in universal-time format (HH:MM:SS)
119    public String toUniversalString()
120    {
121       return String.format(
122          "%02d:%02d:%02d", getHour(), getMinute(), getSecond());
123    }
124
125    // convert to String in standard-time format (H:MM:SS AM or PM)
126    public String toString()
127    {
128       return String.format("%d:%02d:%02d %s",
129          ((getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12),
130          getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
131    }
132 } // end class Time2
```

Fig. 8.5 | `Time2` class with overloaded constructors. (Part 6 of 6.)

Software Engineering Observation 8.1

For a method like `setTime` *in Fig. 8.1, validate all of the method's arguments before using them to set instance variable values to ensure that the object's data is modified only if all the arguments are valid.*

Common Programming Error 8.2

It's a compilation error when `this` *is used in a constructor's body to call another of the class's constructors if that call is not the first statement in the constructor. It's also a compilation error when a method attempts to invoke a constructor directly via* `this`.

```
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(t4); // 12:25:42
13
14       System.out.println("Constructed with:");
15       displayTime("t1: all default arguments", t1);
16       displayTime("t2: hour specified; default minute and second", t2);
17       displayTime("t3: hour and minute specified; default second", t3);
18       displayTime("t4: hour, minute and second specified", t4);
19       displayTime("t5: Time2 object t4 specified", t5);
20
```

Fig. 8.6 | Overloaded constructors used to initialize `Time2` objects. (Part 1 of 3.)

```
21          // attempt to initialize t6 with invalid values
22          try
23          {
24             Time2 t6 = new Time2(27, 74, 99); // invalid values
25          }
26          catch (IllegalArgumentException e)
27          {
28             System.out.printf("%nException while initializing t6: %s%n",
29                e.getMessage());
30          }
31       }
32
33       // displays a Time2 object in 24-hour and 12-hour formats
34       private static void displayTime(String header, Time2 t)
35       {
36          System.out.printf("%s%n   %s%n   %s%n",
37             header, t.toUniversalString(), t.toString());
38       }
39    } // end class Time2Test
```

Fig. 8.6 | Overloaded constructors used to initialize `Time2` objects. (Part 2 of 3.)

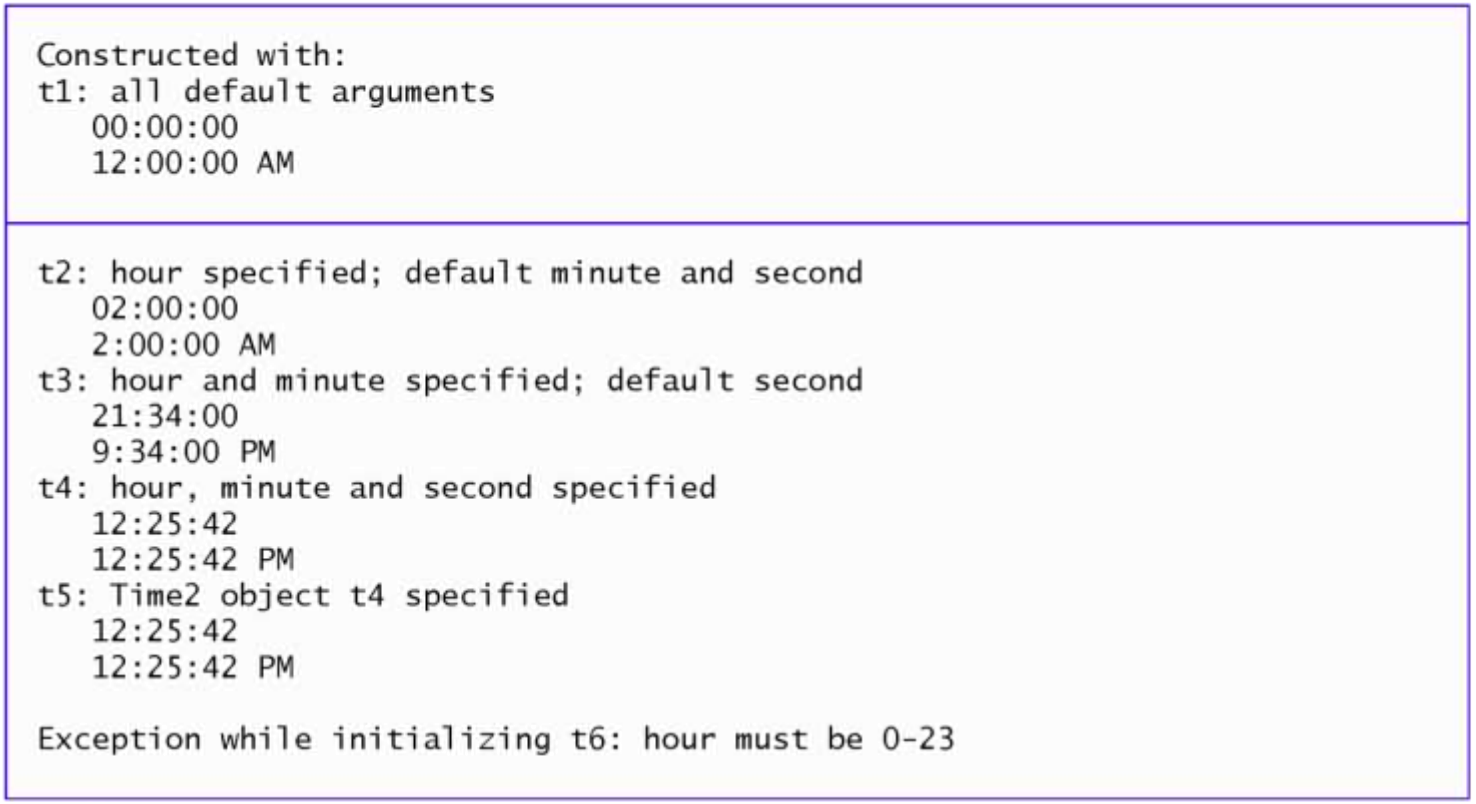


Fig. 8.6 | Overloaded constructors used to initialize `Time2` objects. (Part 3 of 3.)

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

- A program can declare a so-called no-argument constructor that is invoked without arguments.
- Such a constructor simply initializes the object as specified in the constructor's body.
- Using `this` in method-call syntax as the first statement in a constructor's body invokes another constructor of the same class.
  - Popular way to *reuse* initialization code provided by another of the class's constructors rather than defining similar code in the no-argument constructor's body.

Notes Regarding Class Time2's set and get Methods and Constructors

- Methods can access a class's private data directly without calling the *get* methods.
- However, consider changing the representation of the time from three *int* values (requiring *12 bytes* of memory) to a *single int* value representing the *total number of seconds* that have elapsed since midnight (requiring only four bytes of memory).
  - If we made such a change, only the bodies of the methods that access the private data directly would need to change—in particular, the three-argument constructor, the *setTime* method and the individual *set* and *get* methods for the hour, minute and second.
  - There would be no need to modify the bodies of methods *toUniversalString* or *toString* because they do *not* access the data directly.
- Designing the class in this manner reduces the likelihood of programming errors when altering the class's implementation.