Classes 2/5

Outline

- Overloading
- Constructors
- Default Variable Initializations
- The methods **`equals`** and **`toString`**
- Example

- Overloading

- *Overloading* is when two or more methods *in the same class* have the same method name
- To be valid, any two definitions of the method name must have different *signatures*
  - A signature consists of the name of a method together with its parameter list
  - Differing signatures must have different numbers and/or types of parameters

-- Overloading and Automatic Type Conversion

- If Java cannot find a method signature that exactly matches a method invocation, it will try to use automatic type conversion
- The interaction of overloading and automatic type conversion can have unintended results
- In some cases of overloading, because of automatic type conversion, a single method invocation can be resolved in multiple ways
    - Ambiguous method invocations will produce an error in Java

Pitfall: You Can Not Overload Based on the Type Returned

- The signature of a method only includes the method name and its parameter types
    - The signature does **not** include the type returned
- Java does not permit methods with the same name and different return types in the same class

-- You Can Not Overload Operators in Java

- Although many programming languages, such as C++, allow you to overload operators (+, -, etc.), Java does not permit this
  - You may only use a method name and ordinary method syntax to carry out the operations you desire

- Constructors ...

- A *constructor* is a special kind of method that is designed to initialize the instance variables for an object:

  ```
  public ClassName(anyParameters){code}
  ```

  - A constructor must have the same name as the class
  - A constructor has no type returned, not even `void`
  - Constructors are typically overloaded

... - Constructors

- A constructor is called when an object of the class is created using **new**

```
ClassName objectName = new ClassName(anyArgs);
```

  - The name of the constructor and its parenthesized list of arguments (if any) must follow the **new** operator
  - This is the **only** valid way to invoke a constructor: a constructor cannot be invoked like an ordinary method
- If a constructor is invoked again (using **new**), the first object is discarded and an entirely new object is created
  - If you need to change the values of instance variables of the object, use mutator methods instead

-- You Can Invoke Another Method in a Constructor

- The first action taken by a constructor is to create an object with instance variables
- Therefore, it is legal to invoke another method within the definition of a constructor, since it has the newly created object as its calling object
  - For example, mutator methods can be used to set the values of the instance variables
  - It is even possible for one constructor to invoke another

-- Include a No-Argument Constructor

- If you do not include any constructors in your class, Java will automatically create a *default* or *no-argument* constructor that takes no arguments, performs no initializations, but allows the object to be created
- If you include even one constructor in your class, Java will not provide this default constructor
- If you include any constructors in your class, be sure to provide your own no-argument constructor as well

- Default Variable Initializations

- Instance variables are automatically initialized in Java
  - **`boolean`** types are initialized to false
  - Other primitives are initialized to the zero of their type
  - Class types are initialized to **`null`**
- However, it is a better practice to explicitly initialize instance variables in a constructor
- Note: Local variables are not automatically initialized

- The methods `equals` and `toString`

- Java expects certain methods, such as **`equals`** and **`toString`**, to be in all, or almost all, classes

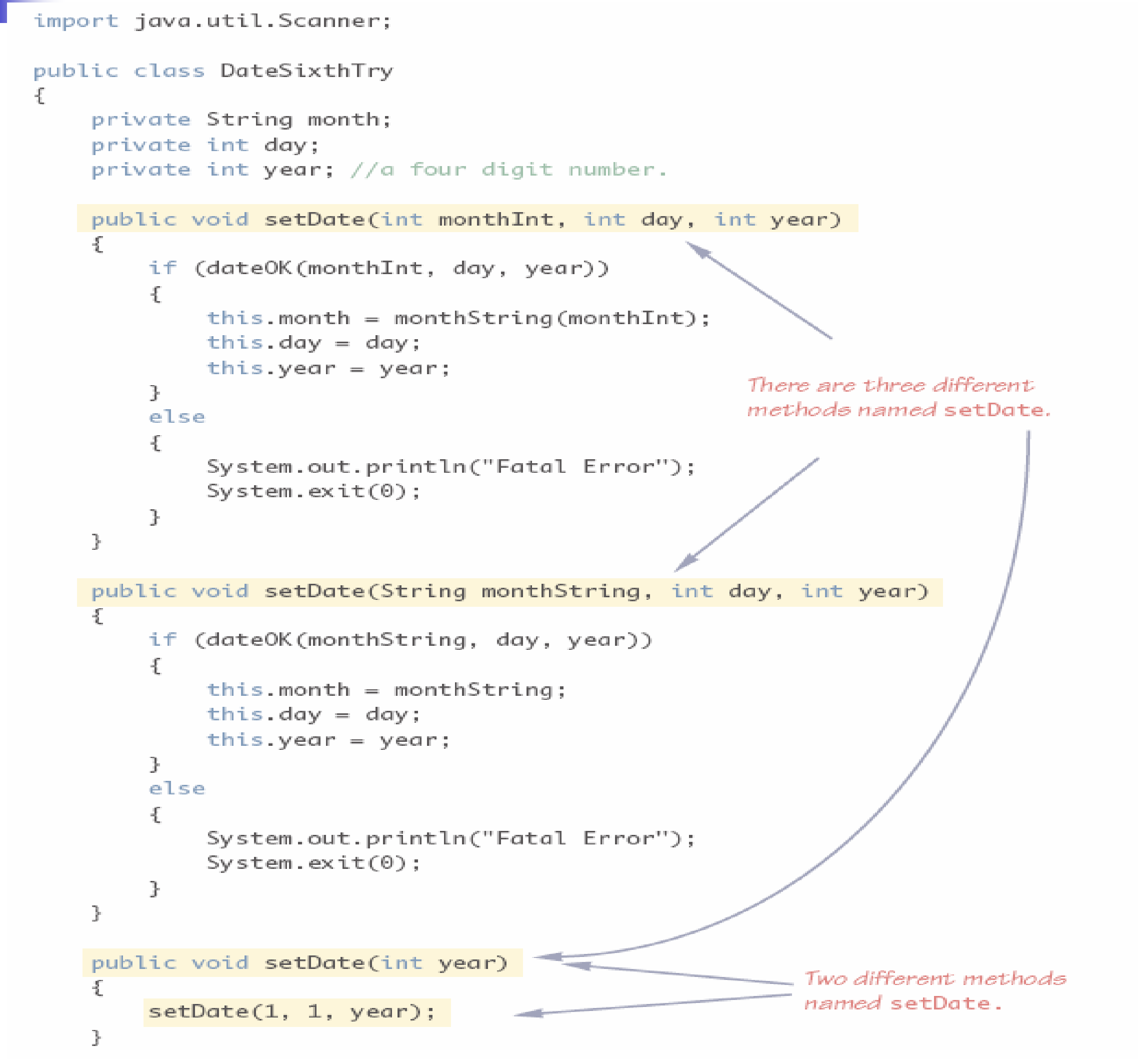
- The purpose of **`equals`**, a **`boolean`** valued method, is to compare two objects of the class to see if they satisfy the notion of "being equal“
  - Note: You cannot use **`==`** to compare objects

```
public boolean equals(ClassName objectName)
```

- The purpose of the **`toString`** method is to return a **`String`** value that represents the data in the object

```
public String toString()
```

- Example ...

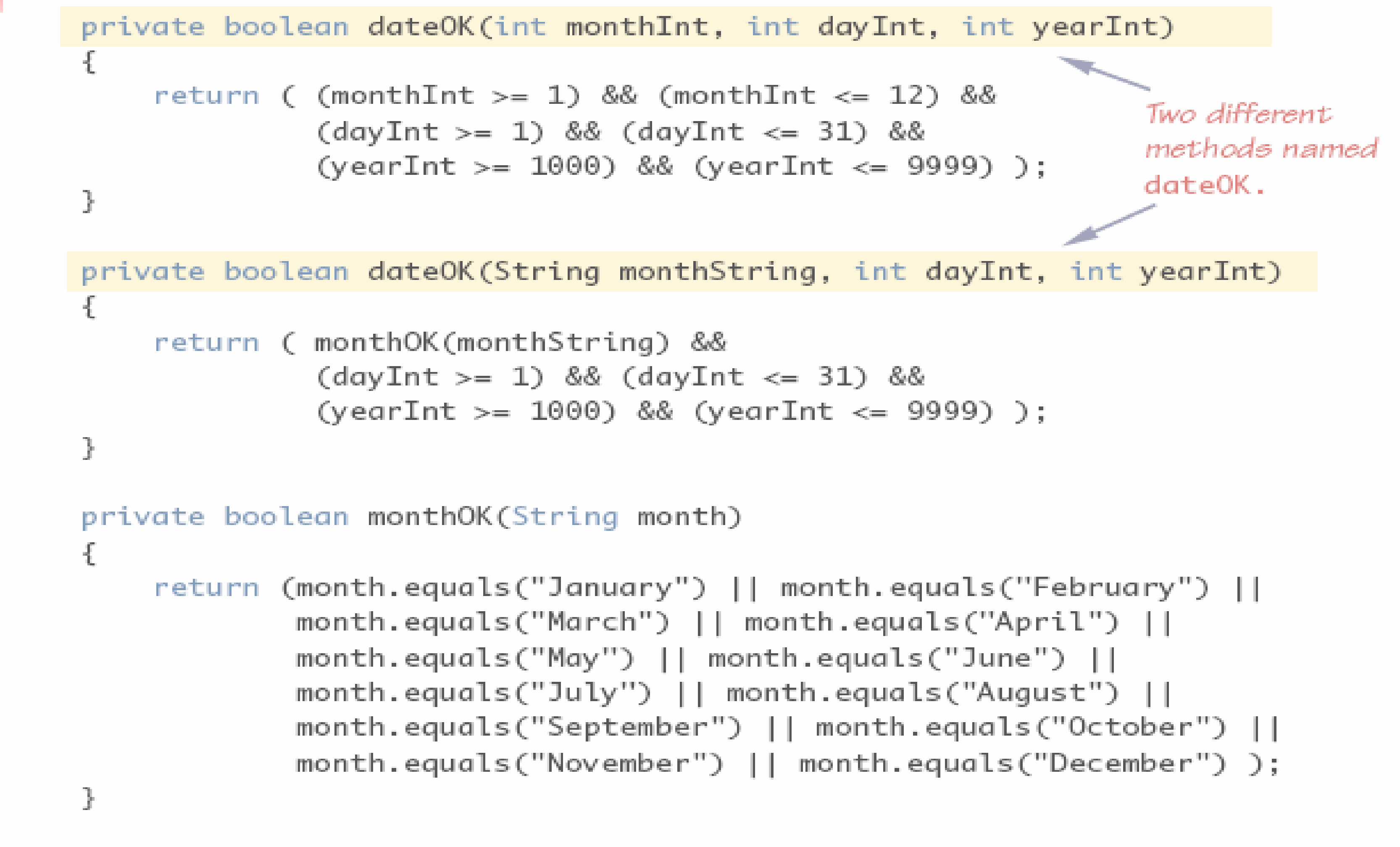


```
import java.util.Scanner;

public class DateSixthTry
{
    private String month;
    private int day;
    private int year; //a four digit number.

    public void setDate(int monthInt, int day, int year)
    {
        if (dateOK(monthInt, day, year))
        {
            this.month = monthString(monthInt);
            this.day = day;
            this.year = year;
        }
        else
        {
            System.out.println("Fatal Error");
            System.exit(0);
        }
    }

    public void setDate(String monthString, int day, int year)
    {
        if (dateOK(monthString, day, year))
        {
            this.month = monthString;
            this.day = day;
            this.year = year;
        }
        else
        {
            System.out.println("Fatal Error");
            System.exit(0);
        }
    }

    public void setDate(int year)
    {
        setDate(1, 1, year);
    }
```

There are three different methods named setDate.

Two different methods named setDate.

... - Example ...



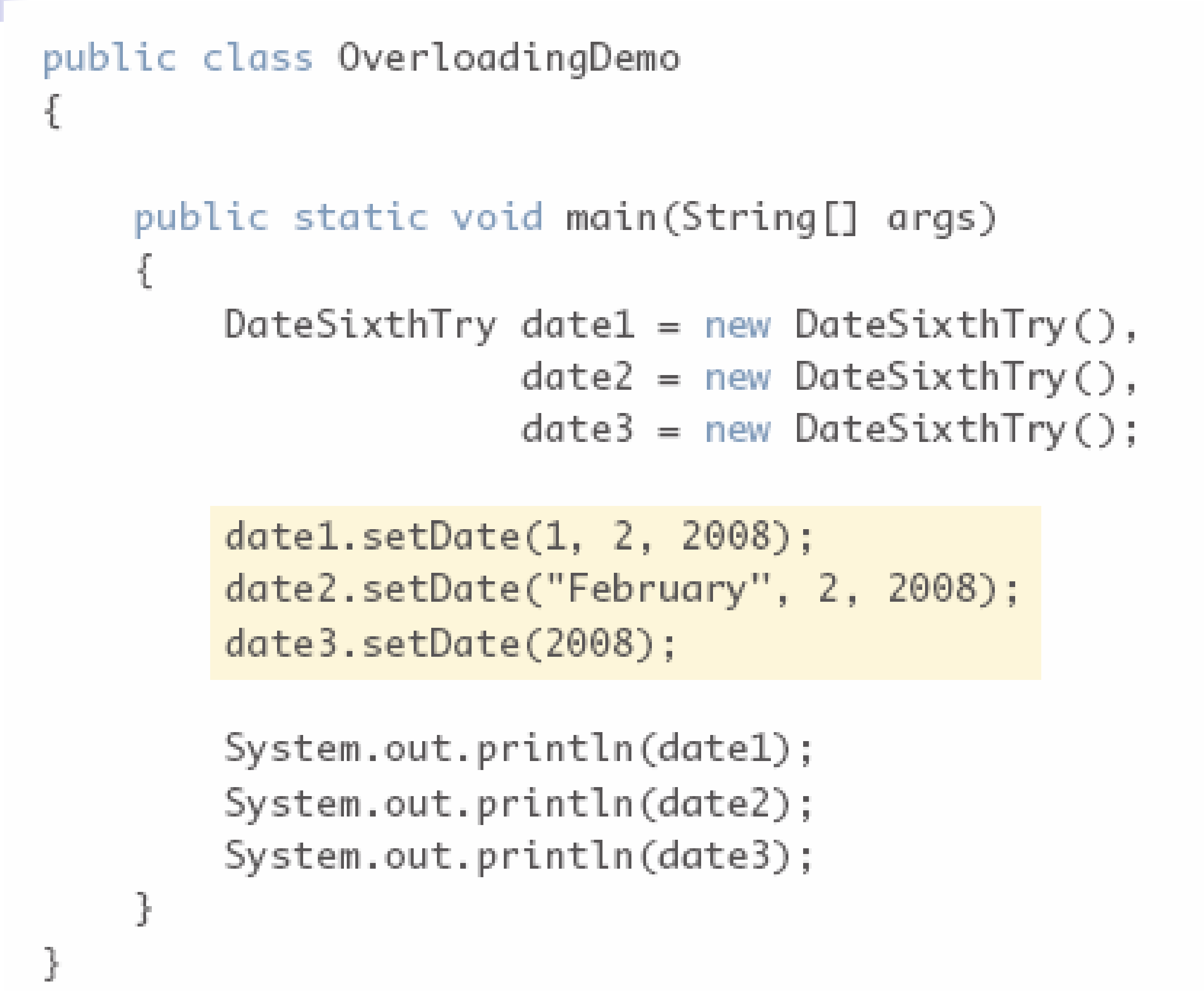
```
private boolean dateOK(int monthInt, int dayInt, int yearInt)
{
    return ( (monthInt >= 1) && (monthInt <= 12) &&
             (dayInt >= 1) && (dayInt <= 31) &&
             (yearInt >= 1000) && (yearInt <= 9999) );
}

private boolean dateOK(String monthString, int dayInt, int yearInt)
{
    return ( monthOK(monthString) &&
             (dayInt >= 1) && (dayInt <= 31) &&
             (yearInt >= 1000) && (yearInt <= 9999) );
}

private boolean monthOK(String month)
{
    return (month.equals("January") || month.equals("February") ||
           month.equals("March") || month.equals("April") ||
           month.equals("May") || month.equals("June") ||
           month.equals("July") || month.equals("August") ||
           month.equals("September") || month.equals("October") ||
           month.equals("November") || month.equals("December") );
}
```

Two different methods named dateOK.

... - Example ...

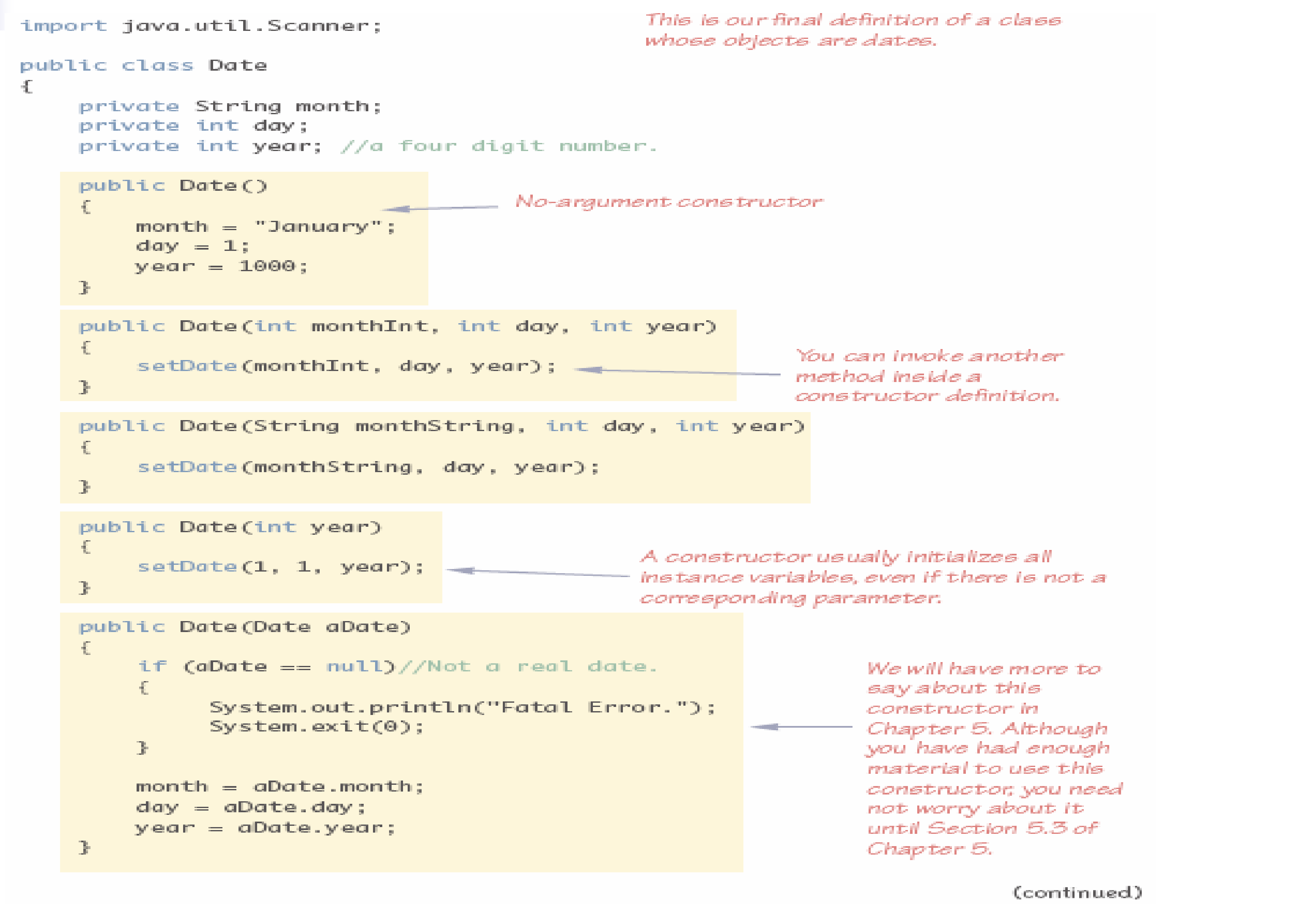
```
    public void readInput()
    {
        boolean tryAgain = true;
        Scanner keyboard = new Scanner(System.in);
        while (tryAgain)
        {
            System.out.println("Enter month, day, and year.");
              System.out.println("Do not use a comma.");
            String monthInput = keyboard.next();
            int dayInput = keyboard.nextInt();
            int yearInput = keyboard.nextInt();
            if (dateOK(monthInput, dayInput, yearInput) )
            {
                setDate(monthInput, dayInput, yearInput);
                tryAgain = false;
            }
            else
                System.out.println("Illegal date. Reenter input.");
         }
    }
<The rest of the methods are the same as in Display 4.9, except that
          the parameter to equals and precedes is, of course, of type DateSixthTry.>
}
```

... - Example ...



```
public class OverloadingDemo
{

    public static void main(String[] args)
    {
        DateSixthTry date1 = new DateSixthTry(),
                     date2 = new DateSixthTry(),
                     date3 = new DateSixthTry();

        date1.setDate(1, 2, 2008);
        date2.setDate("February", 2, 2008);
        date3.setDate(2008);

        System.out.println(date1);
        System.out.println(date2);
        System.out.println(date3);
    }
}
```

... - Example



```
import java.util.Scanner;

public class Date
{
    private String month;
    private int day;
    private int year; //a four digit number.

    public Date()
    {
        month = "January";
        day = 1;
        year = 1000;
    }

    public Date(int monthInt, int day, int year)
    {
        setDate(monthInt, day, year);
    }

    public Date(String monthString, int day, int year)
    {
        setDate(monthString, day, year);
    }

    public Date(int year)
    {
        setDate(1, 1, year);
    }

    public Date(Date aDate)
    {
        if (aDate == null)//Not a real date.
        {
            System.out.println("Fatal Error.");
            System.exit(0);
        }

        month = aDate.month;
        day = aDate.day;
        year = aDate.year;
    }
```

This is our final definition of a class whose objects are dates.

No-argument constructor

You can invoke another method inside a constructor definition.

A constructor usually initializes all instance variables, even if there is not a corresponding parameter.

We will have more to say about this constructor in Chapter 5. Although you have had enough material to use this constructor, you need not worry about it until Section 5.3 of Chapter 5.

(continued)

THE END