Generics and the ArrayList Class

Introduction to Generics

- Beginning with version 5.0, Java allows class and method definitions that include parameters for types
- Such definitions are called *generics*
    - Generic programming with a type parameter enables code to be written that applies to any class

The **`ArrayList`** Class

- **`ArrayList`** is a class in the standard Java libraries
  - Unlike arrays, which have a fixed length once they have been created, an **`ArrayList`** is an object that can grow and shrink while your program is running
- In general, an **`ArrayList`** serves the same purpose as an array, except that an **`ArrayList`** can change length while the program is running

The **`ArrayList`** Class

- The class **`ArrayList`** is implemented using an array as a private instance variable
  - When this hidden array is full, a new larger hidden array is created and the data is transferred to this new array

The **`ArrayList`** Class

- Why not always use an **`ArrayList`** instead of an array?
  1. An **`ArrayList`** is less efficient than an array
  2. It does not have the convenient square bracket notation
  3. The base type of an **`ArrayList`** must be a class type (or other reference type): it cannot be a primitive type

  – This last point is less of a problem now that Java provides automatic boxing and unboxing of primitives

Using the **`ArrayList`** Class

- In order to make use of the **`ArrayList`** class, it must first be imported from the package **`java.util`**
- An **`ArrayList`** is created and named in the same way as object of any class, except that you specify the base type as follows:

```
ArrayList<BaseType> aList =
     new ArrayList<BaseType>();
```

Using the **`ArrayList`** Class

- An initial capacity can be specified when creating an **`ArrayList`** as well
  - The following code creates an **`ArrayList`** that stores objects of the base type **`String`** with an initial capacity of 20 items

    ```
    ArrayList<String> list =
       new ArrayList<String>(20);
    ```

  - Specifying an initial capacity does not limit the size to which an **`ArrayList`** can eventually grow
- Note that the base type of an ArrayList is specified as a *type parameter*

Using the **`ArrayList`** Class

- The **`add`** method is used to set an element for the first time in an **`ArrayList`**

```
list.add("something");
```

  - The method name **`add`** is overloaded
  - There is also a two argument version that allows an item to be added at any currently used index position or at the first unused position

Using the **`ArrayList`** Class

- The **`size`** method is used to find out how many indices already have elements in the **`ArrayList`**

```
    int howMany = list.size();
```

- The **`set`** method is used to replace any existing element, and the **`get`** method is used to access the value of any existing element

```
    list.set(index, "something else");
    String thing = list.get(index);
```

Tip: Summary of Adding to an **`ArrayList`**

- The **`add`** method is usually used to place an element in an **`ArrayList`** position for the first time (at an **`ArrayList`** index)
- The simplest **`add`** method has a single parameter for the element to be added, and adds an element at the next unused index, in order

Tip: Summary of Adding to an **`ArrayList`**

- An element can be added at an already occupied list position by using the two-parameter version of **`add`**
- This causes the new element to be placed at the index specified, and every other member of the **`ArrayList`** to be moved up by one position

Tip: Summary of Adding to an `ArrayList`

- The two-argument version of `add` can also be used to add an element at the first unused position (if that position is known)
- Any individual element can be changed using the `set` method
  - However, `set` can only reset an element at an index that already contains an element
- In addition, the method `size` can be used to determine how many elements are stored in an `ArrayList`

Constructor Summary

- **ArrayList**()

  Constructs an empty list with an initial capacity of ten.

- **ArrayList**(Collection <? extends E> c)

  Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

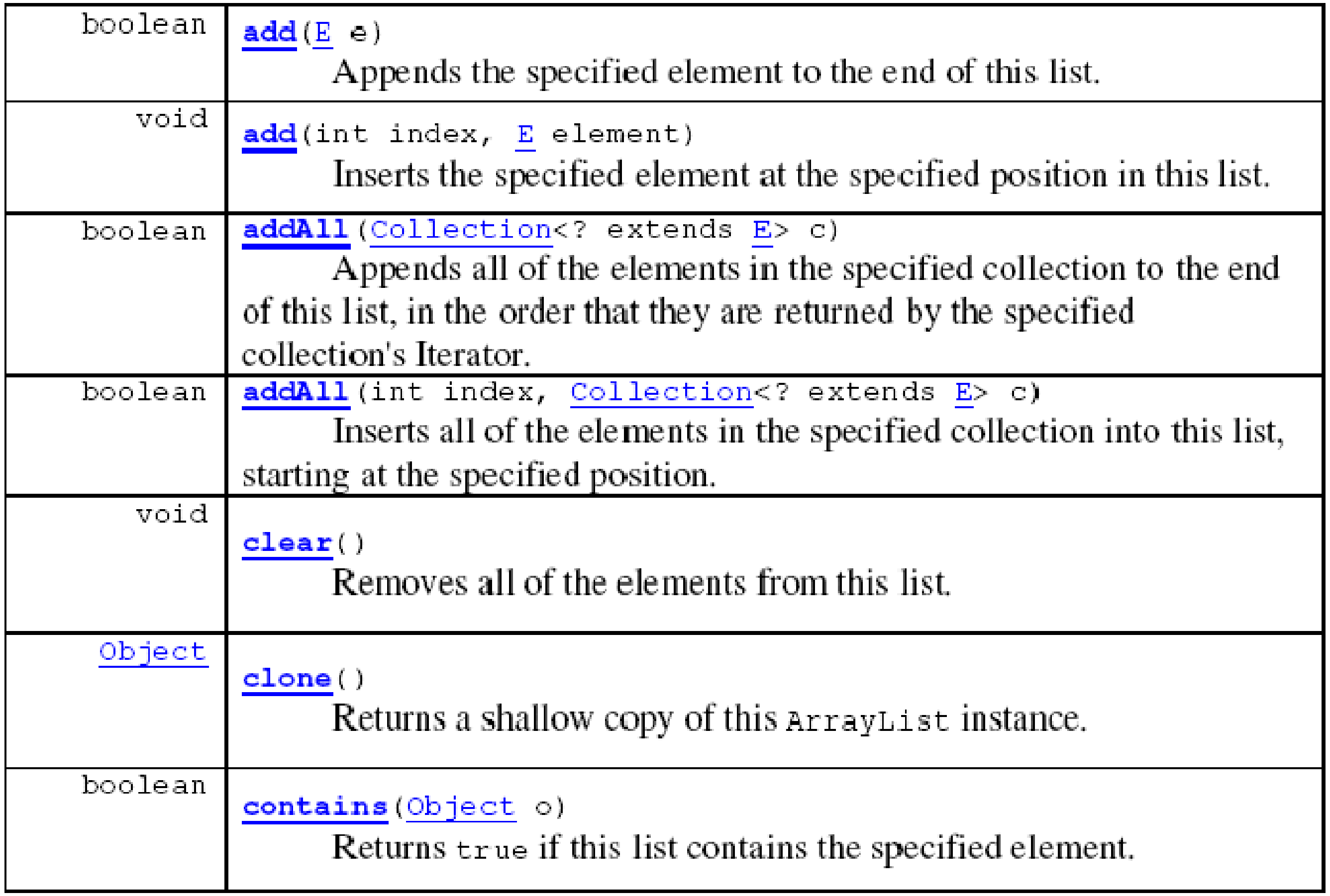
- **ArrayList**(int initialCapacity)

  Constructs an empty list with the specified initial capacity.

Methods in the Class `ArrayList`

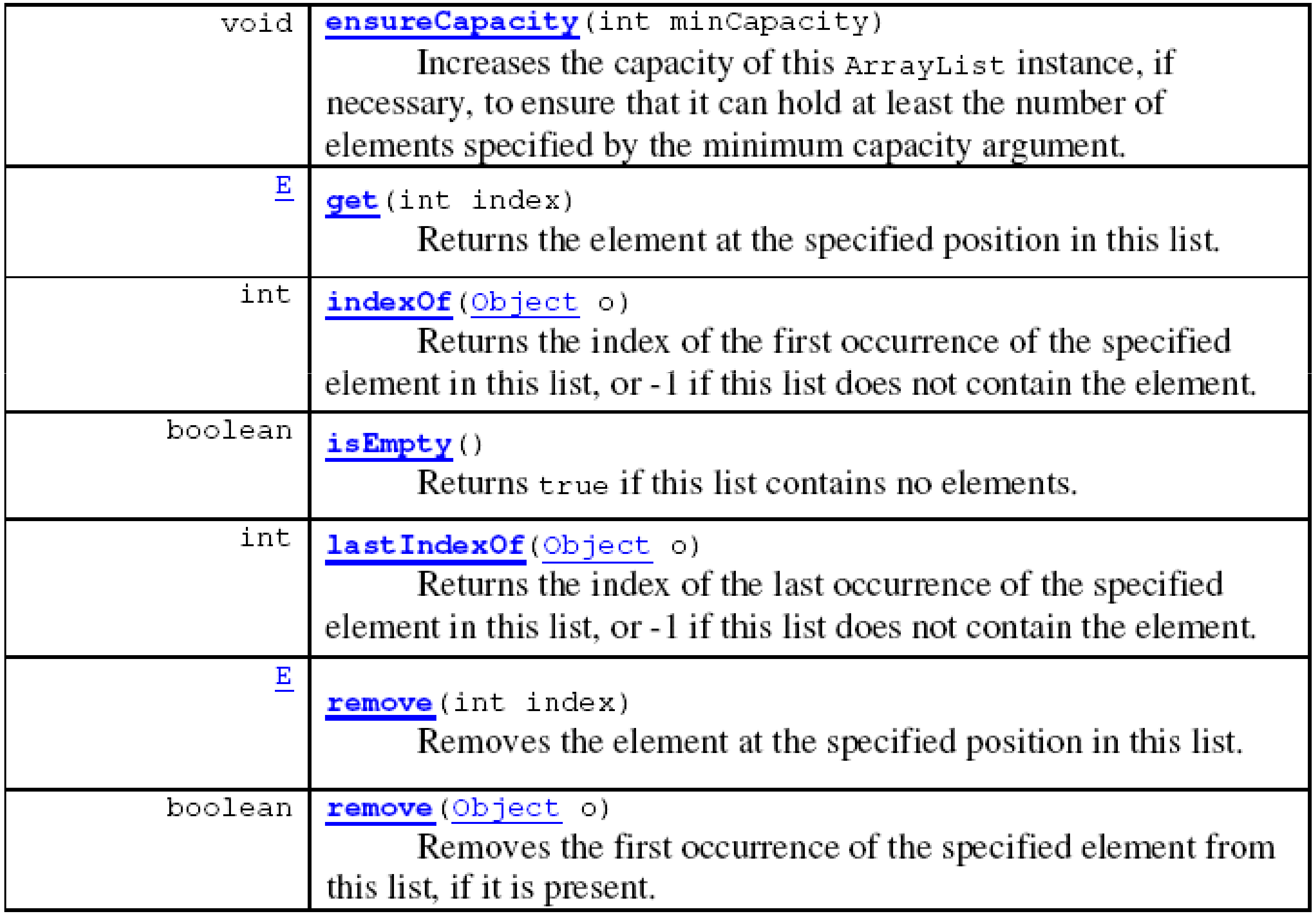
- The tools for manipulating arrays consist only of the square brackets and the instance variable `length`
- `ArrayLists`, however, come with a selection of powerful methods that can do many of the things for which code would have to be written in order to do them using arrays

Some Methods in the Class **ArrayList**



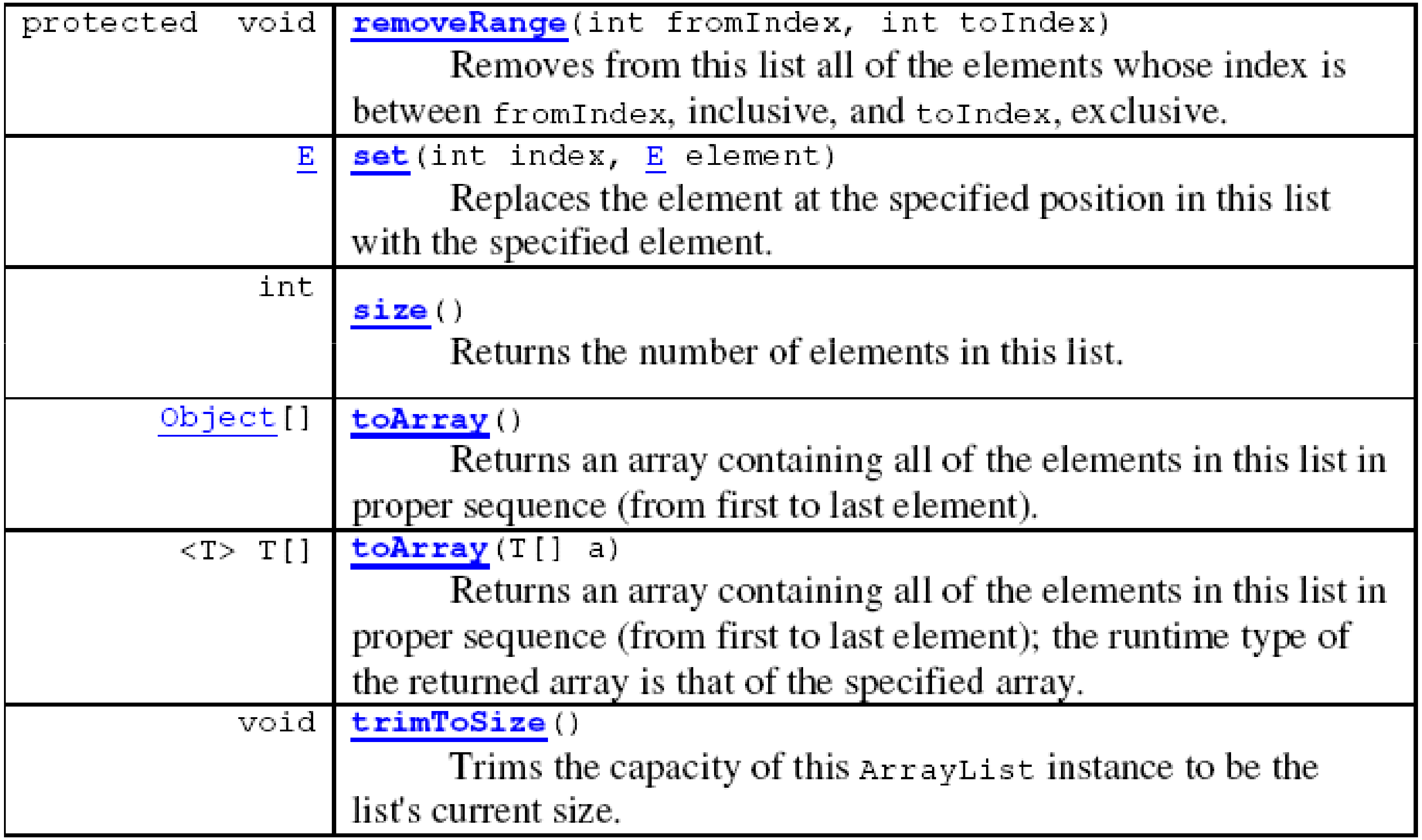
boolean	add(E e) Appends the specified element to the end of this list.
void	add(int index, E element) Inserts the specified element at the specified position in this list.
boolean	addAll(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
boolean	addAll(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	clear() Removes all of the elements from this list.
Object	clone() Returns a shallow copy of this ArrayList instance.
boolean	contains(Object o) Returns true if this list contains the specified element.

Some Methods in the Class **`ArrayList`**



`void`	**`ensureCapacity`**`(int minCapacity)` Increases the capacity of this `ArrayList` instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
`E`	**`get`**`(int index)` Returns the element at the specified position in this list.
`int`	**`indexOf`**`(Object o)` Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`boolean`	**`isEmpty`**`()` Returns `true` if this list contains no elements.
`int`	**`lastIndexOf`**`(Object o)` Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
`E`	**`remove`**`(int index)` Removes the element at the specified position in this list.
`boolean`	**`remove`**`(Object o)` Removes the first occurrence of the specified element from this list, if it is present.

Some Methods in the Class **`ArrayList`**



`protected void`	`removeRange(int fromIndex, int toIndex)` Removes from this list all of the elements whose index is between `fromIndex`, inclusive, and `toIndex`, exclusive.
`E`	`set(int index, E element)` Replaces the element at the specified position in this list with the specified element.
`int`	`size()` Returns the number of elements in this list.
`Object[]`	`toArray()` Returns an array containing all of the elements in this list in proper sequence (from first to last element).
`<T> T[]`	`toArray(T[] a)` Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.
`void`	`trimToSize()` Trims the capacity of this `ArrayList` instance to be the list's current size.

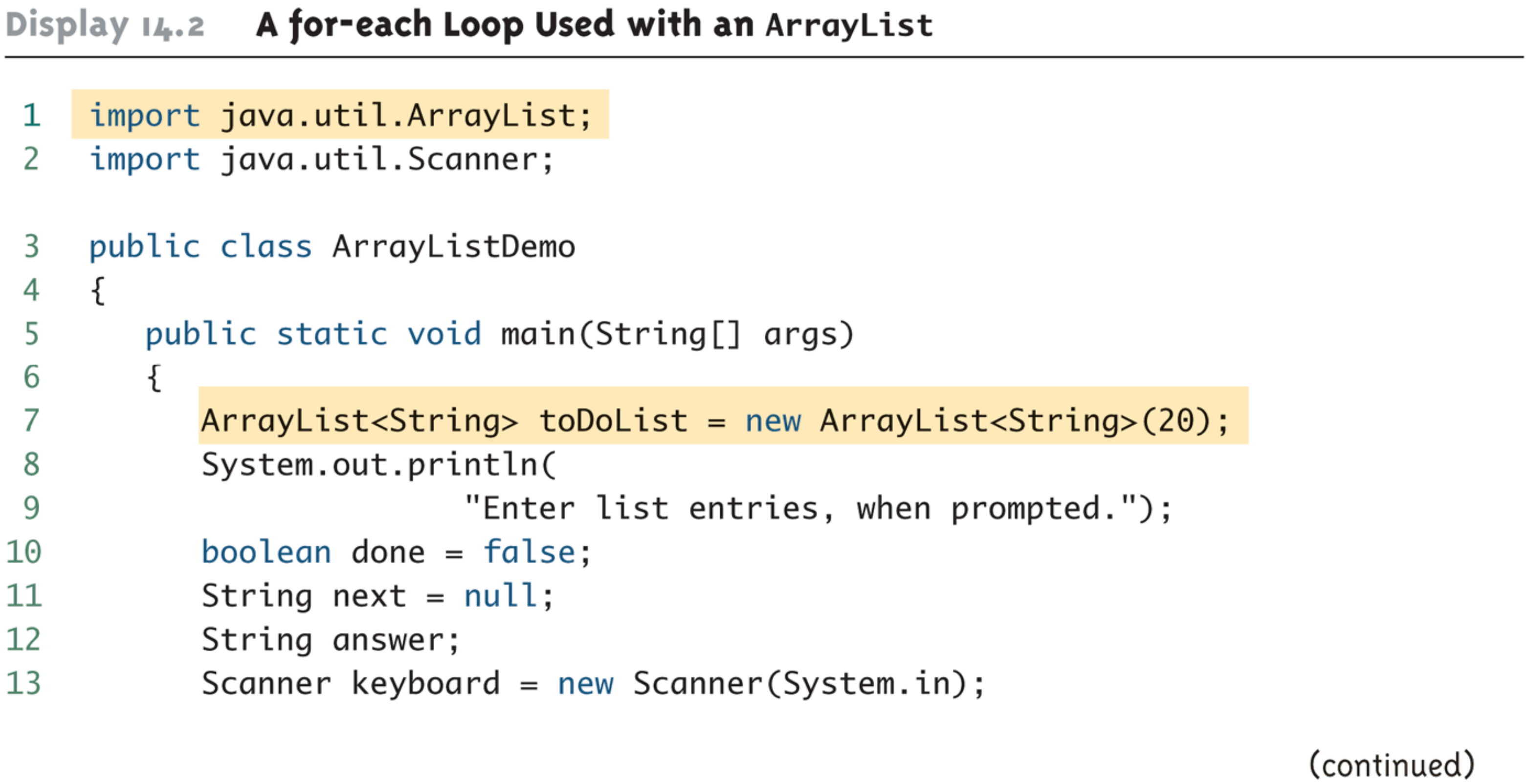
Why are Some Parameters of Type **`Base_Type`** and Others of type **`Object`**

- When looking at the methods available in the **`ArrayList`** class, there appears to be some inconsistency
  - In some cases, when a parameter is naturally an object of the base type, the parameter type is the base type
  - However, in other cases, it is the type **`Object`**
- This is because the **`ArrayList`** class implements a number of interfaces, and inherits methods from various ancestor classes
  - These interfaces and ancestor classes specify that certain parameters have type **`Object`**

The "For Each" Loop

- The `ArrayList` class is an example of a *collection* class
- Starting with version 5.0, Java has added a new kind of for loop called a *for-each* or *enhanced for* loop
  - This kind of loop has been designed to cycle through all the elements in a collection (like an `ArrayList`)

A for-each Loop Used with an **ArrayList** (Part 1 of 3)

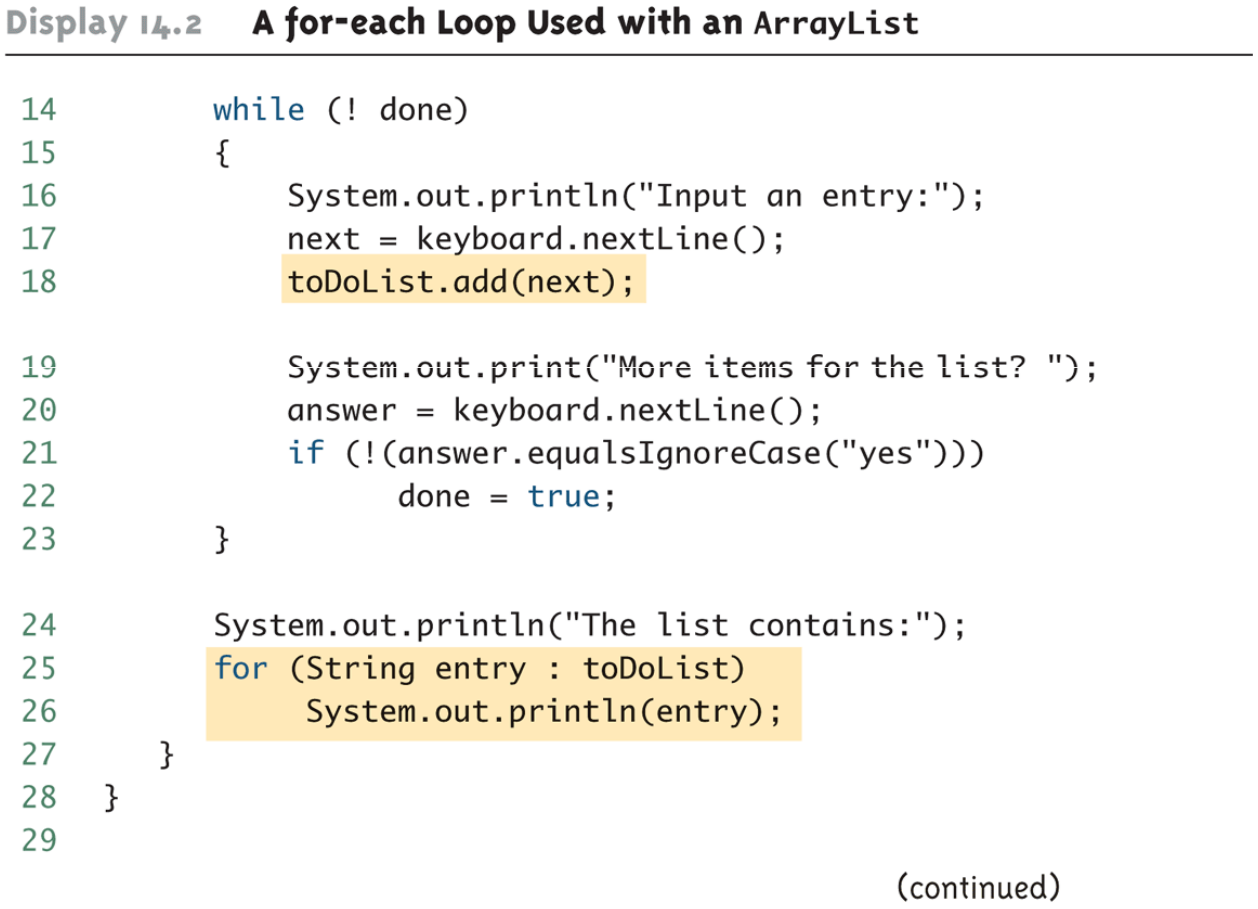


Display 14.2 **A for-each Loop Used with an ArrayList**

```
import java.util.ArrayList;
import java.util.Scanner;

public class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList<String> toDoList = new ArrayList<String>(20);
        System.out.println(
                      "Enter list entries, when prompted.");
        boolean done = false;
        String next = null;
        String answer;
        Scanner keyboard = new Scanner(System.in);
```

(continued)

A for-each Loop Used with an **`ArrayList`** (Part 2 of 3)

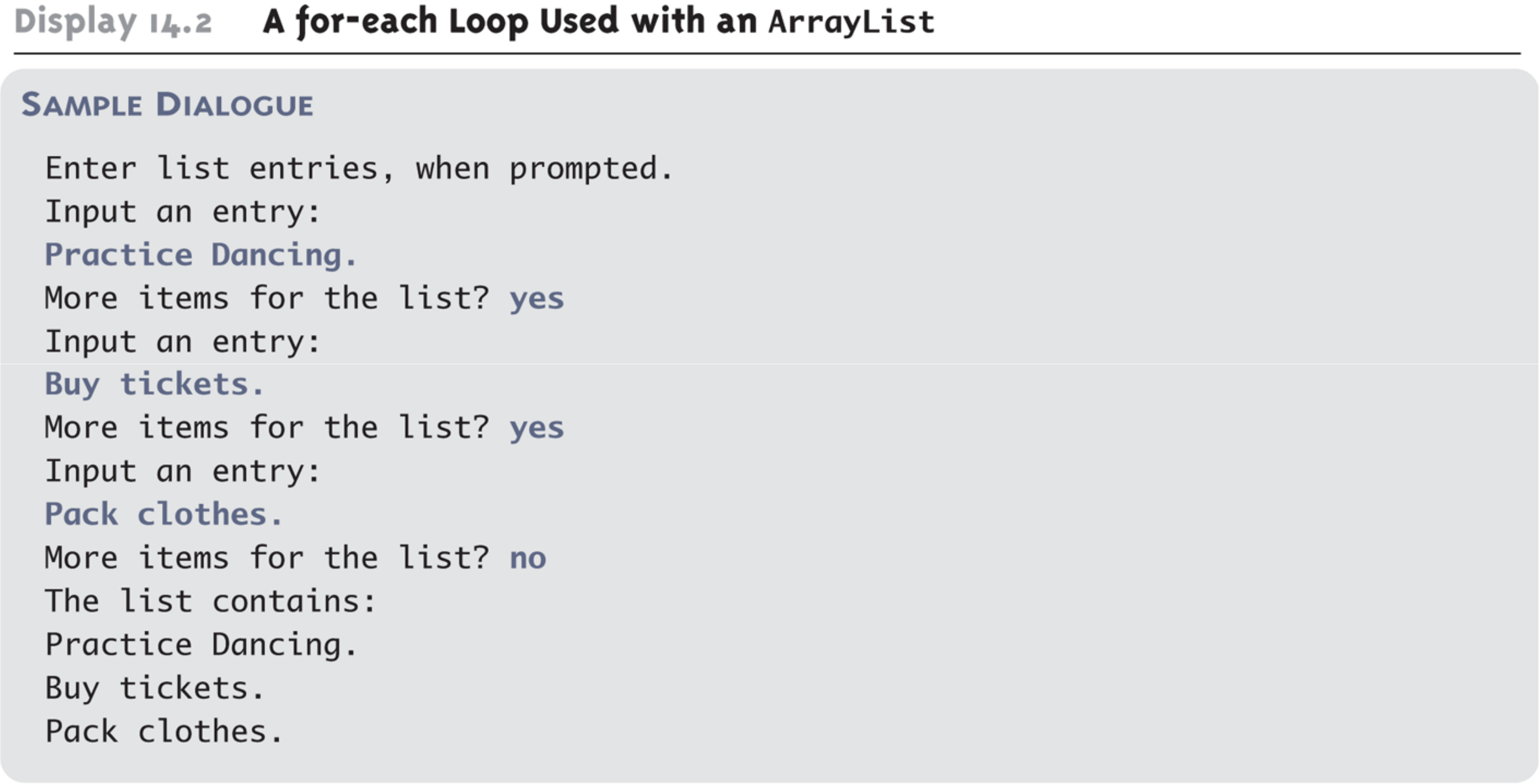


Display 14.2 **A for-each Loop Used with an `ArrayList`**

```
        while (! done)
        {
            System.out.println("Input an entry:");
            next = keyboard.nextLine();
            toDoList.add(next);

            System.out.print("More items for the list? ");
            answer = keyboard.nextLine();
            if (!(answer.equalsIgnoreCase("yes")))
                done = true;
        }

        System.out.println("The list contains:");
        for (String entry : toDoList)
            System.out.println(entry);
    }
}

```

(continued)

A for-each Loop Used with an **`ArrayList`** (Part 3 of 3)

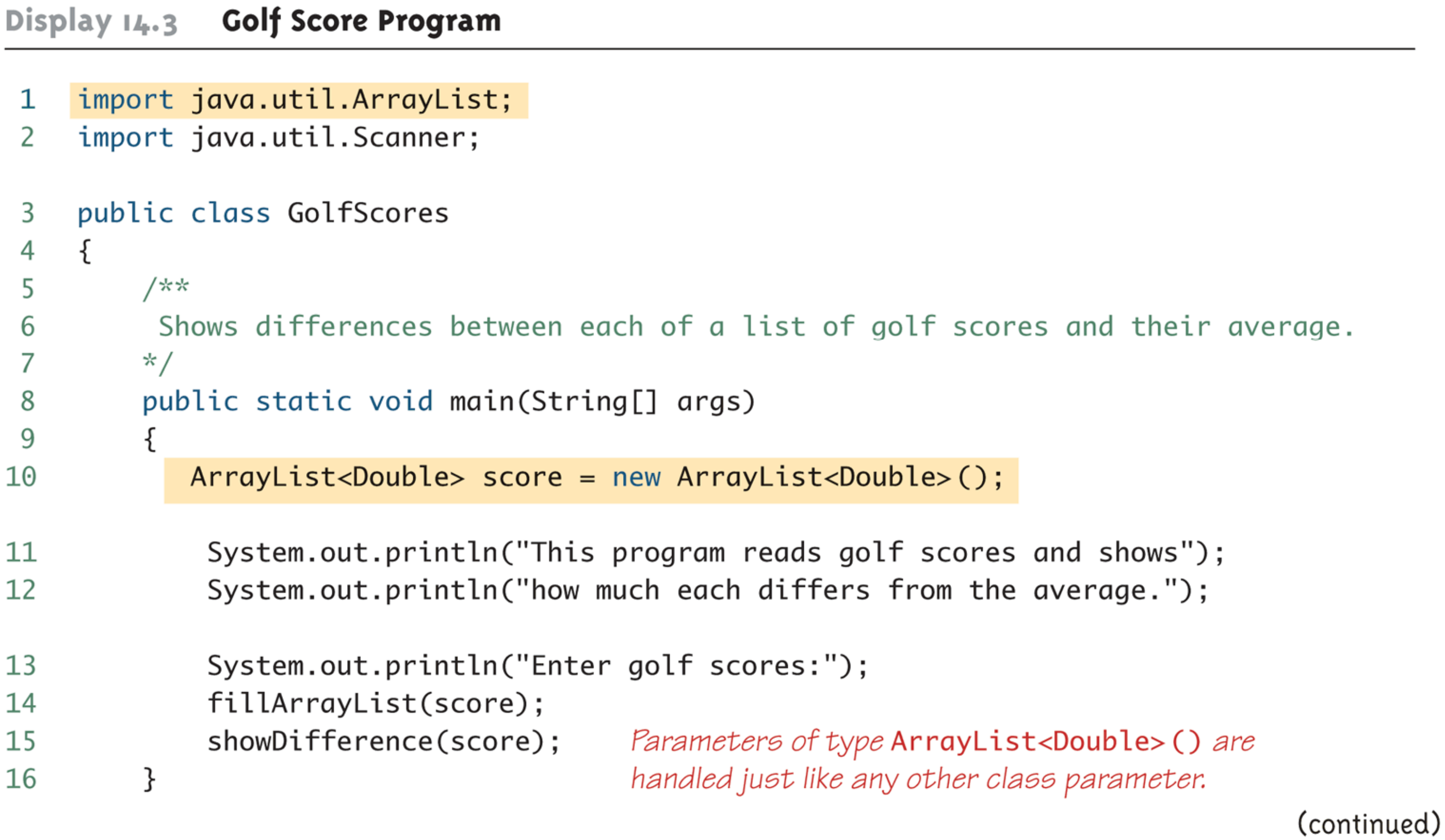


Display 14.2 **A for-each Loop Used with an** `ArrayList`

SAMPLE DIALOGUE

```
Enter list entries, when prompted.
Input an entry:
Practice Dancing.
More items for the list? yes
Input an entry:
Buy tickets.
More items for the list? yes
Input an entry:
Pack clothes.
More items for the list? no
The list contains:
Practice Dancing.
Buy tickets.
Pack clothes.
```

Golf Score Program (Part 1 of 6)



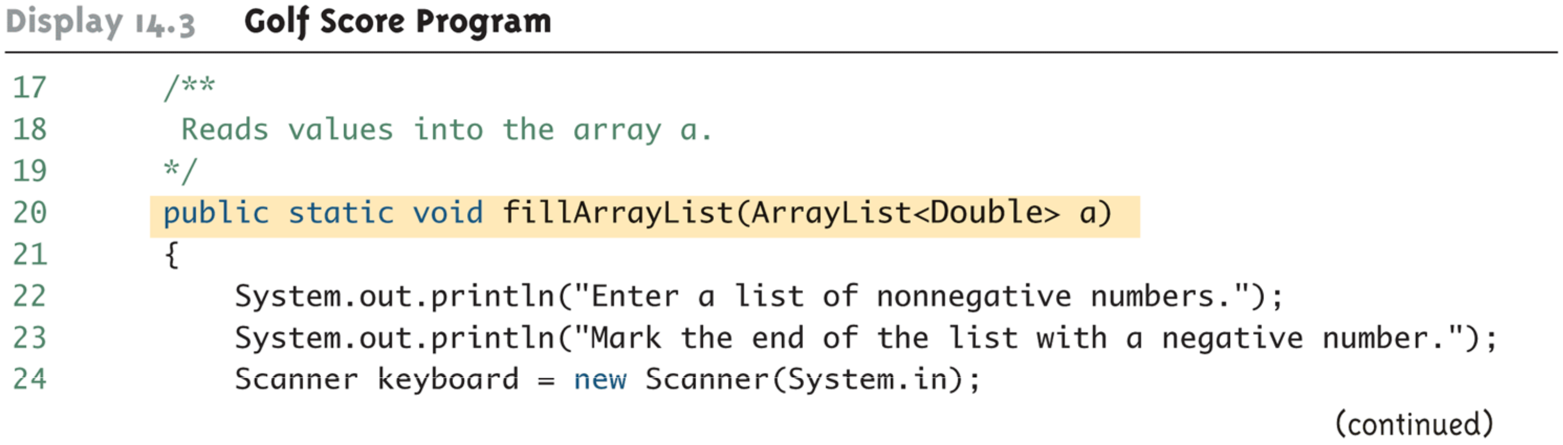
Display 14.3 **Golf Score Program**

```
import java.util.ArrayList;
import java.util.Scanner;

public class GolfScores
{
    /**
     Shows differences between each of a list of golf scores and their average.
    */
    public static void main(String[] args)
    {
       ArrayList<Double> score = new ArrayList<Double>();

        System.out.println("This program reads golf scores and shows");
        System.out.println("how much each differs from the average.");

        System.out.println("Enter golf scores:");
        fillArrayList(score);
        showDifference(score);
    }
```

Parameters of type `ArrayList<Double>()` *are handled just like any other class parameter.*

(continued)

Golf Score Program (Part 2 of 6)

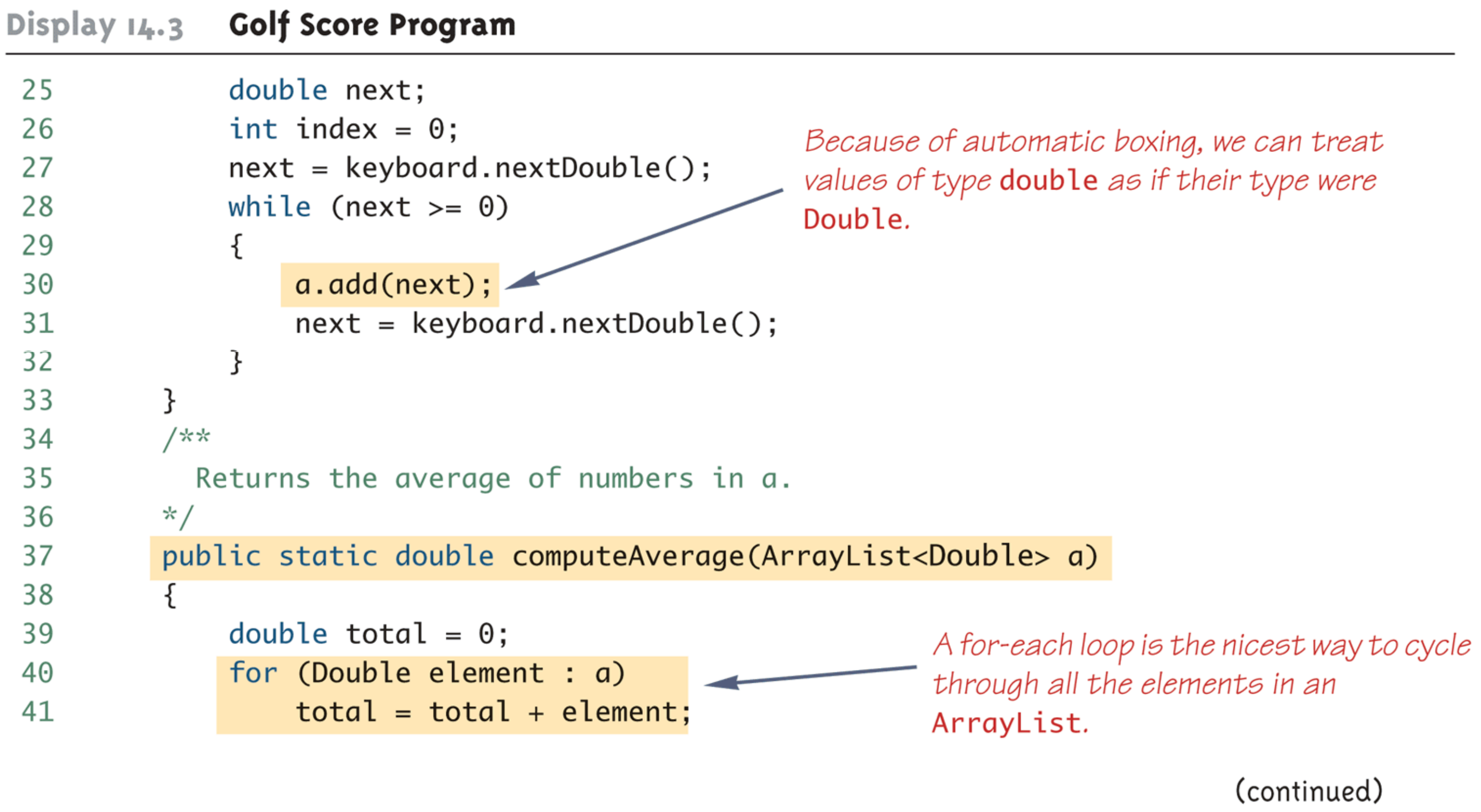


Display 14.3 **Golf Score Program**

```
/**
 Reads values into the array a.
*/
public static void fillArrayList(ArrayList<Double> a)
{
    System.out.println("Enter a list of nonnegative numbers.");
    System.out.println("Mark the end of the list with a negative number.");
    Scanner keyboard = new Scanner(System.in);
```

(continued)

Golf Score Program (Part 3 of 6)



Display 14.3 Golf Score Program

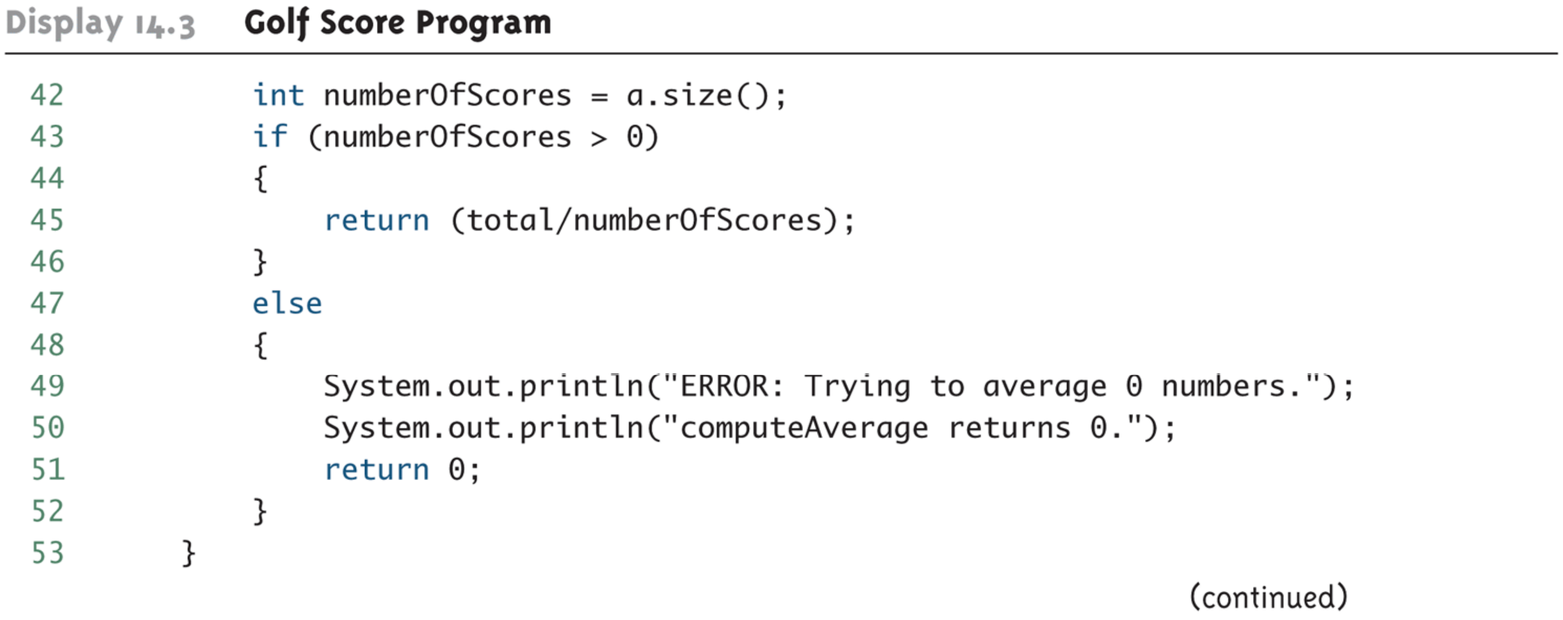
```
        double next;
        int index = 0;
        next = keyboard.nextDouble();
        while (next >= 0)
        {
            a.add(next);
            next = keyboard.nextDouble();
        }
    }
    /**
      Returns the average of numbers in a.
    */
    public static double computeAverage(ArrayList<Double> a)
    {
        double total = 0;
        for (Double element : a)
            total = total + element;
```

Because of automatic boxing, we can treat values of type **double** *as if their type were* Double.

A for-each loop is the nicest way to cycle through all the elements in an ArrayList.

(continued)

Golf Score Program (Part 4 of 6)

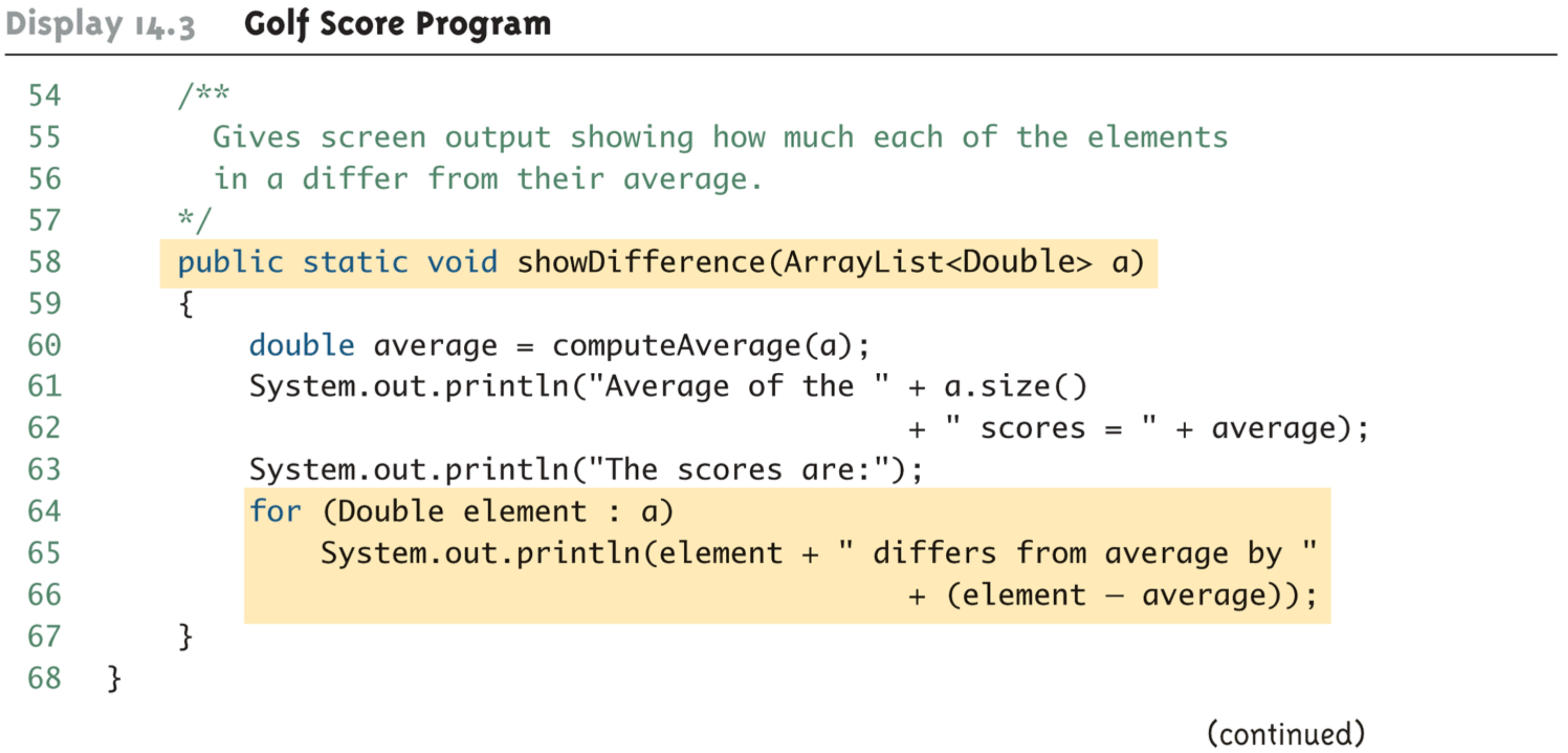


Display 14.3 **Golf Score Program**

```
        int numberOfScores = a.size();
        if (numberOfScores > 0)
        {
            return (total/numberOfScores);
        }
        else
        {
            System.out.println("ERROR: Trying to average 0 numbers.");
            System.out.println("computeAverage returns 0.");
            return 0;
        }
    }
```

(continued)

Golf Score Program (Part 5 of 6)

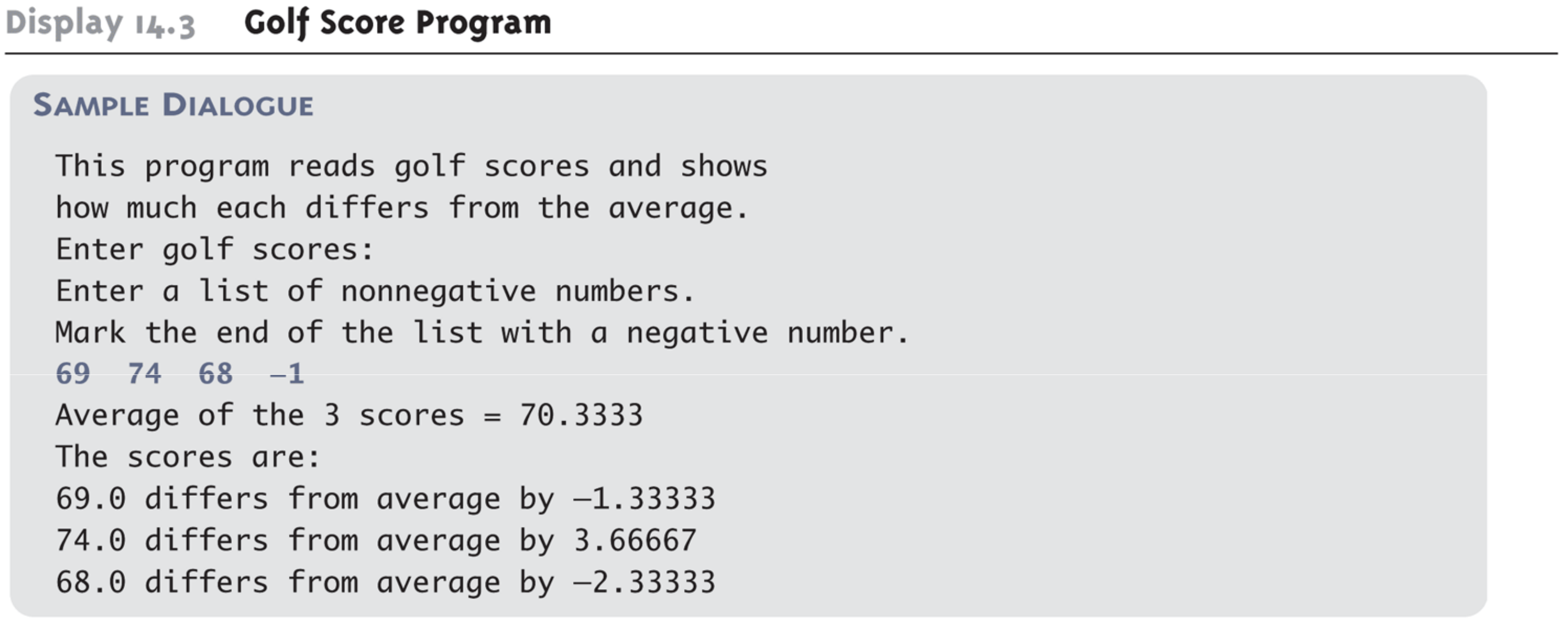


Display 14.3 **Golf Score Program**

```
    /**
      Gives screen output showing how much each of the elements
      in a differ from their average.
    */
    public static void showDifference(ArrayList<Double> a)
    {
        double average = computeAverage(a);
        System.out.println("Average of the " + a.size()
                                             + " scores = " + average);
        System.out.println("The scores are:");
        for (Double element : a)
            System.out.println(element + " differs from average by "
                                             + (element – average));
    }
}
```

(continued)

Golf Score Program (Part 6 of 6)



Display 14.3 **Golf Score Program**

Sample Dialogue

```
This program reads golf scores and shows
how much each differs from the average.
Enter golf scores:
Enter a list of nonnegative numbers.
Mark the end of the list with a negative number.
69  74  68  –1
Average of the 3 scores = 70.3333
The scores are:
69.0 differs from average by –1.33333
74.0 differs from average by 3.66667
68.0 differs from average by –2.33333
```

Tip: Use **`trimToSize`** to Save Memory

- An **`ArrayList`** automatically increases its capacity when needed
  - However, the capacity may increase beyond what a program requires
  - In addition, although an **`ArrayList`** grows automatically when needed, it does not shrink automatically
- If an **`ArrayList`** has a large amount of excess capacity, an invocation of the method **`trimToSize`** will shrink the capacity of the **`ArrayList`** down to the size needed

Pitfall: The `clone` method Makes a Shallow Copy

- When a deep copy of an `ArrayList` is needed, using the clone method is not sufficient
  - Invoking `clone` on an `ArrayList` object produces a shallow copy, not a deep copy
- In order to make a deep copy, it must be possible to make a deep copy of objects of the base type
  - Then a deep copy of each element in the `ArrayList` can be created and placed into a new `ArrayList` object

The **`Vector`** Class

- The Java standard libraries have a class named **`Vector`** that behaves almost exactly the same as the class **`ArrayList`**
- In most situations, either class could be used
  - However the **`ArrayList`** class is newer, and is becoming the preferred class

Parameterized Classes and Generics

- The class **`ArrayList`** is a *parameterized class*
- It has a parameter, denoted by **`Base_Type`**, that can be replaced by any reference type to obtain a class for **`ArrayLists`** with the specified base type
- Starting with version 5.0, Java allows class definitions with parameters for types
  - These classes that have type parameters are called *parameterized class* or *generic definitions*, or, simply, *generics*

Nonparameterized **`ArrayList`** and **`Vector`** Classes

- The **`ArrayList`** and **`Vector`** classes discussed here have a type parameter for the base type
- There are also **`ArrayList`** and **`Vector`** classes with no parameter whose base type is **`Object`**
  - These classes are left over from earlier versions of Java

Generics

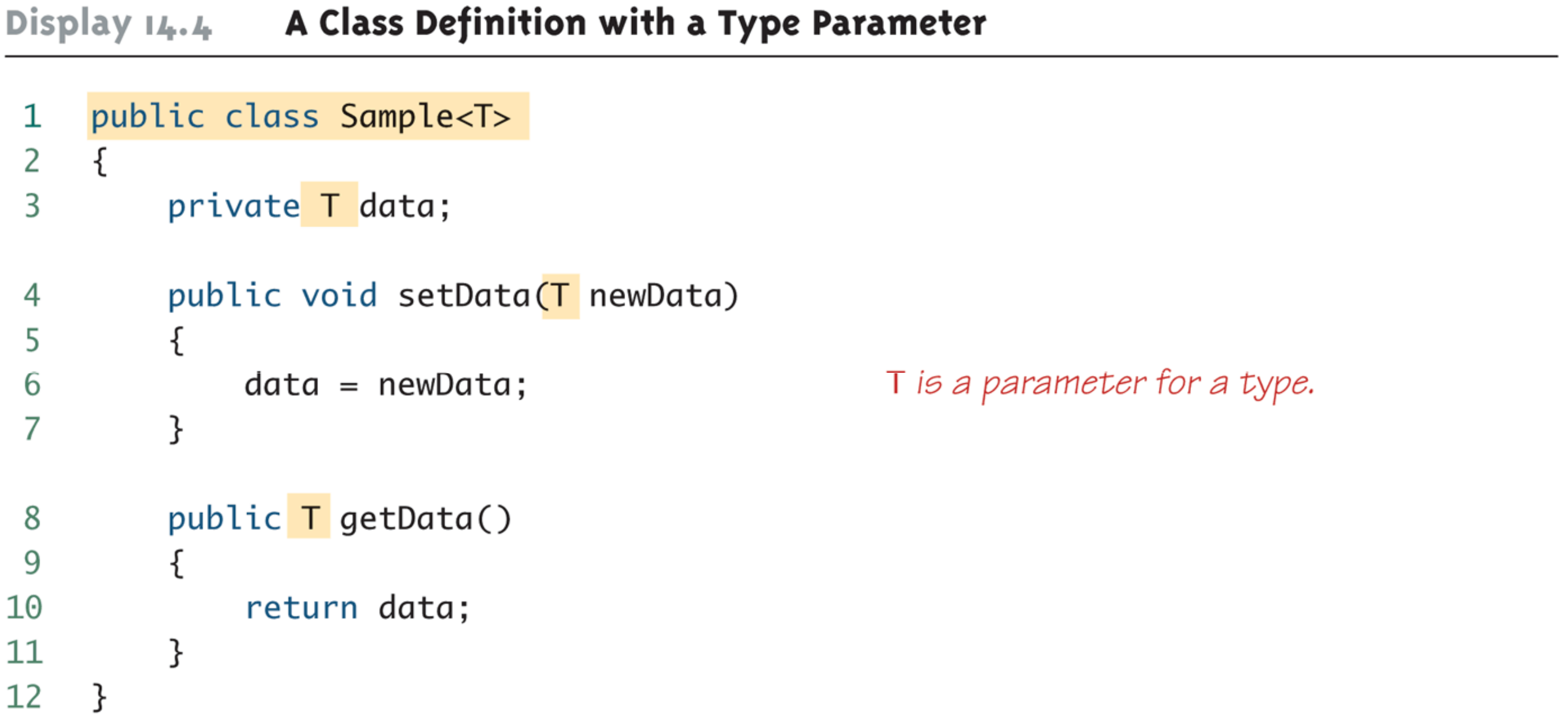
- Classes and methods can have a type parameter
  - A type parameter can have any reference type (i.e., any class type) plugged in for the type parameter
  - When a specific type is plugged in, this produces a specific class type or method
  - Traditionally, a single uppercase letter is used for a type parameter, but any non-keyword identifier may be used

Generics

- A class definition with a type parameter is stored in a file and compiled just like any other class
- Once a parameterized class is compiled, it can be used like any other class
  - However, the class type plugged in for the type parameter must be specified before it can be used in a program
  - Doing this is said to *instantiate* the generic class

```
Sample<String> object =
    new Sample<String>();
```

A Class Definition with a Type Parameter



Display 14.4 **A Class Definition with a Type Parameter**

```java
public class Sample<T>
{
    private T data;

    public void setData(T newData)
    {
        data = newData;
    }

    public T getData()
    {
        return data;
    }
}
```

T is a parameter for a type.

Class Definition with a Type Parameter

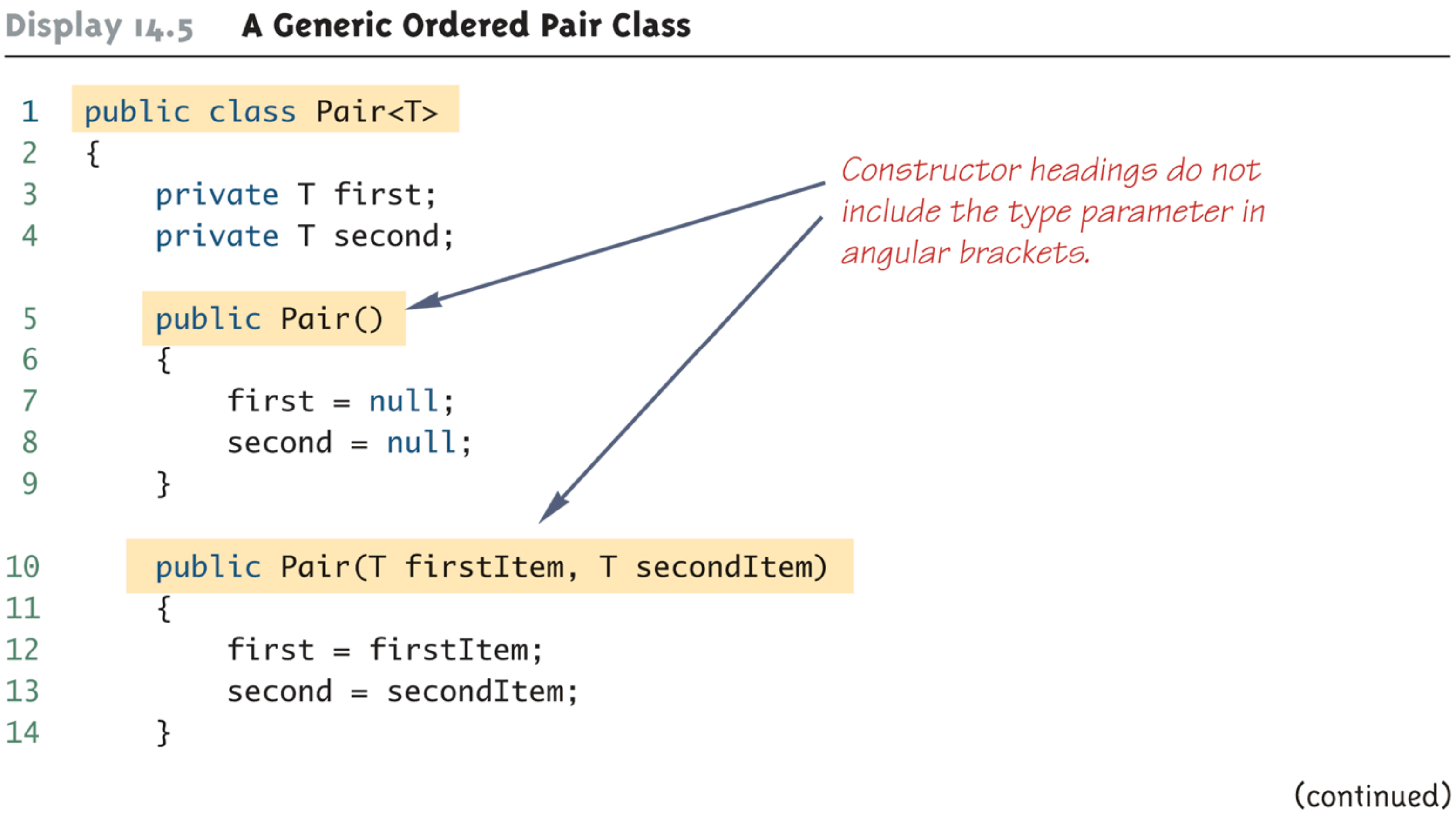
- A class that is defined with a parameter for a type is called a generic class or a parameterized class
  - The type parameter is included in angular brackets after the class name in the class definition heading
  - Any non-keyword identifier can be used for the type parameter, but by convention, the parameter starts with an uppercase letter
  - The type parameter can be used like other types used in the definition of a class

Tip: Compile with the **`-Xlint`** Option

- There are many pitfalls that can be encountered when using type parameters
- Compiling with the **`-Xlint`** option will provide more informative diagnostics of any problems or potential problems in the code

```
 javac –Xlint Sample.java
```

A Generic Ordered Pair Class (Part 1 of 4)



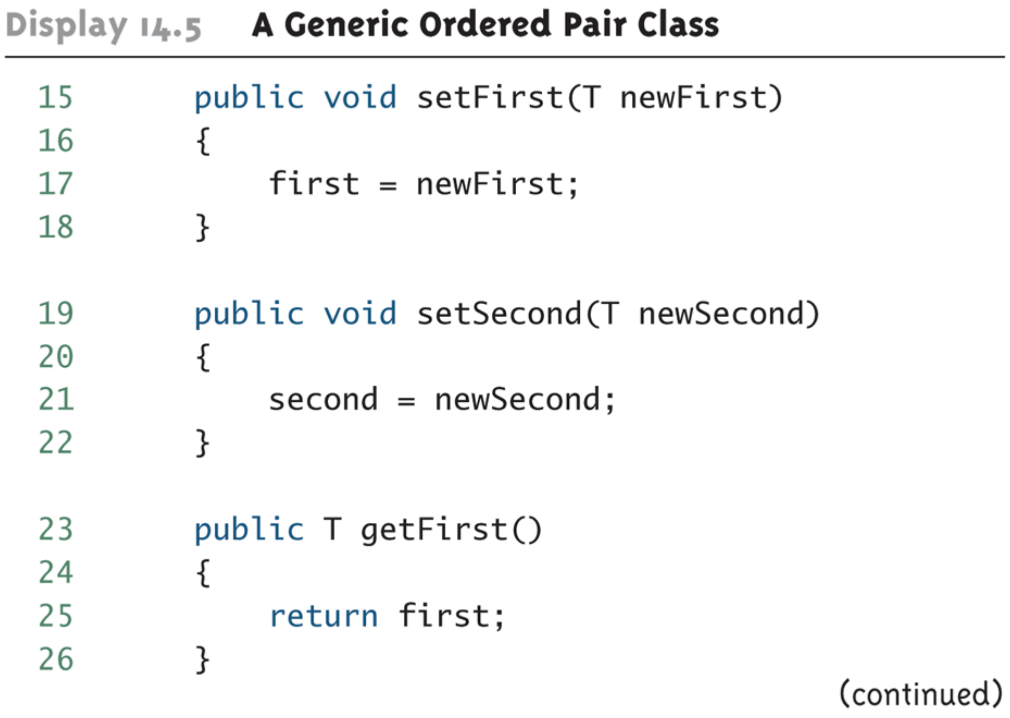
Display 14.5 A Generic Ordered Pair Class

```java
public class Pair<T>
{
    private T first;
    private T second;

    public Pair()
    {
        first = null;
        second = null;
    }

    public Pair(T firstItem, T secondItem)
    {
        first = firstItem;
        second = secondItem;
    }
```

Constructor headings do not include the type parameter in angular brackets.

(continued)

A Generic Ordered Pair Class (Part 2 of 4)

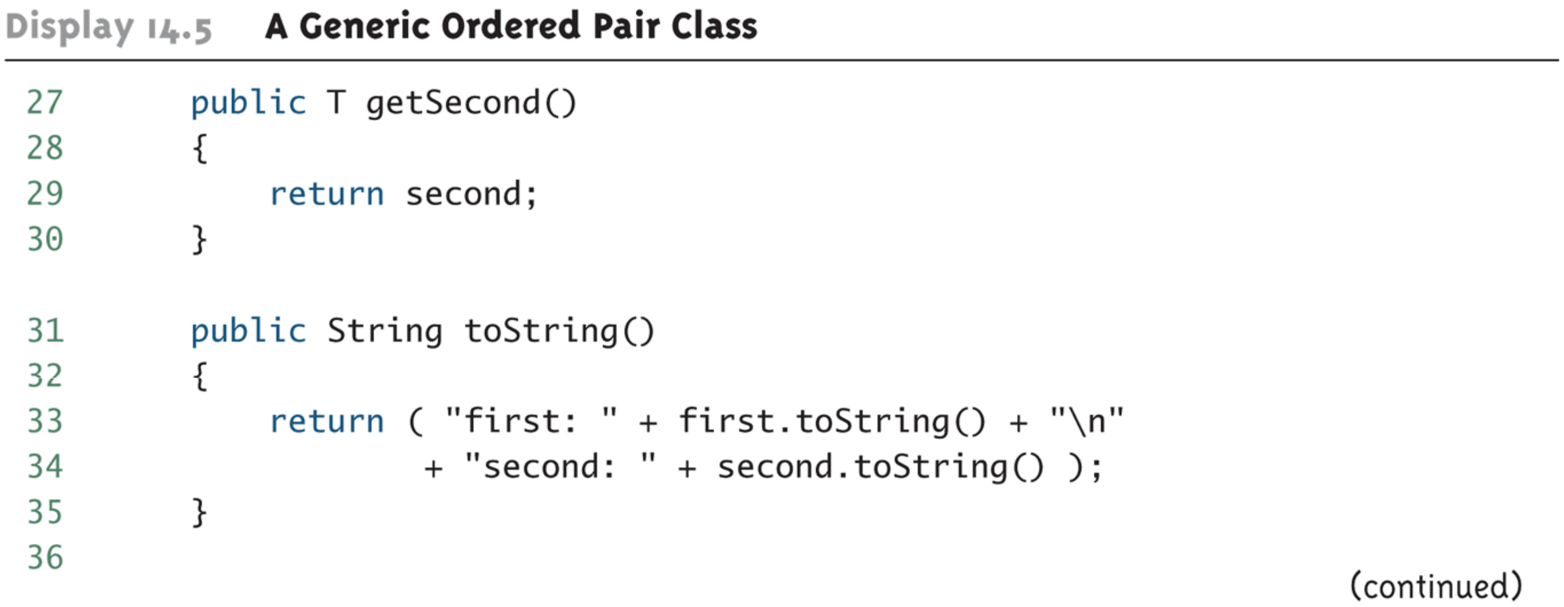


Display 14.5 A Generic Ordered Pair Class

```
    public void setFirst(T newFirst)
    {
        first = newFirst;
    }

    public void setSecond(T newSecond)
    {
        second = newSecond;
    }

    public T getFirst()
    {
        return first;
    }
```

(continued)

A Generic Ordered Pair Class (Part 3 of 4)

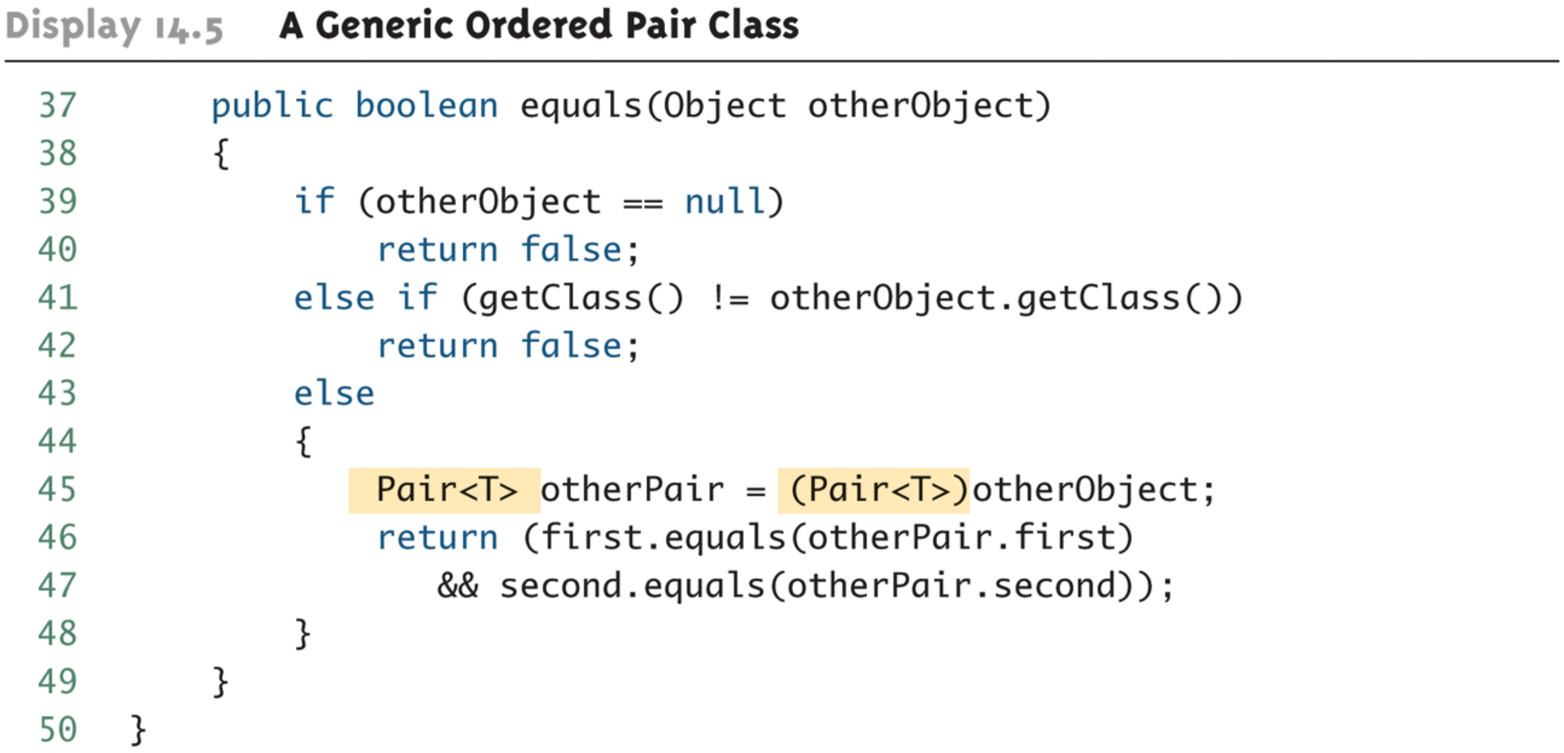


Display 14.5 **A Generic Ordered Pair Class**

```
    public T getSecond()
    {
        return second;
    }

    public String toString()
    {
        return ( "first: " + first.toString() + "\n"
                    + "second: " + second.toString() );
    }

```

(continued)

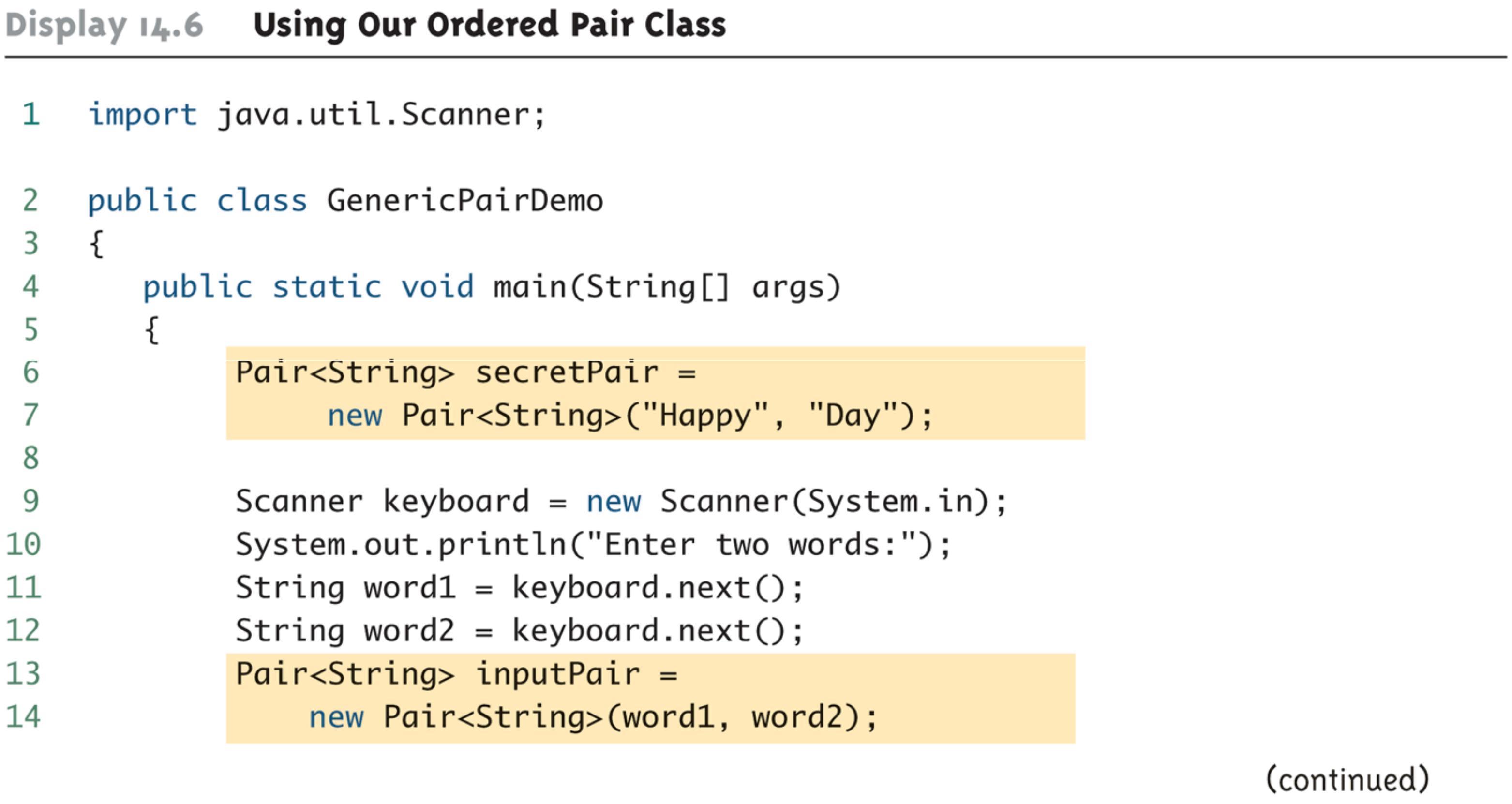
A Generic Ordered Pair Class (Part 4 of 4)



Display 14.5 **A Generic Ordered Pair Class**

```
    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass() != otherObject.getClass())
            return false;
        else
        {
            Pair<T> otherPair = (Pair<T>)otherObject;
            return (first.equals(otherPair.first)
               && second.equals(otherPair.second));
        }
    }
}
```

Using Our Ordered Pair Class (Part 1 of 3)

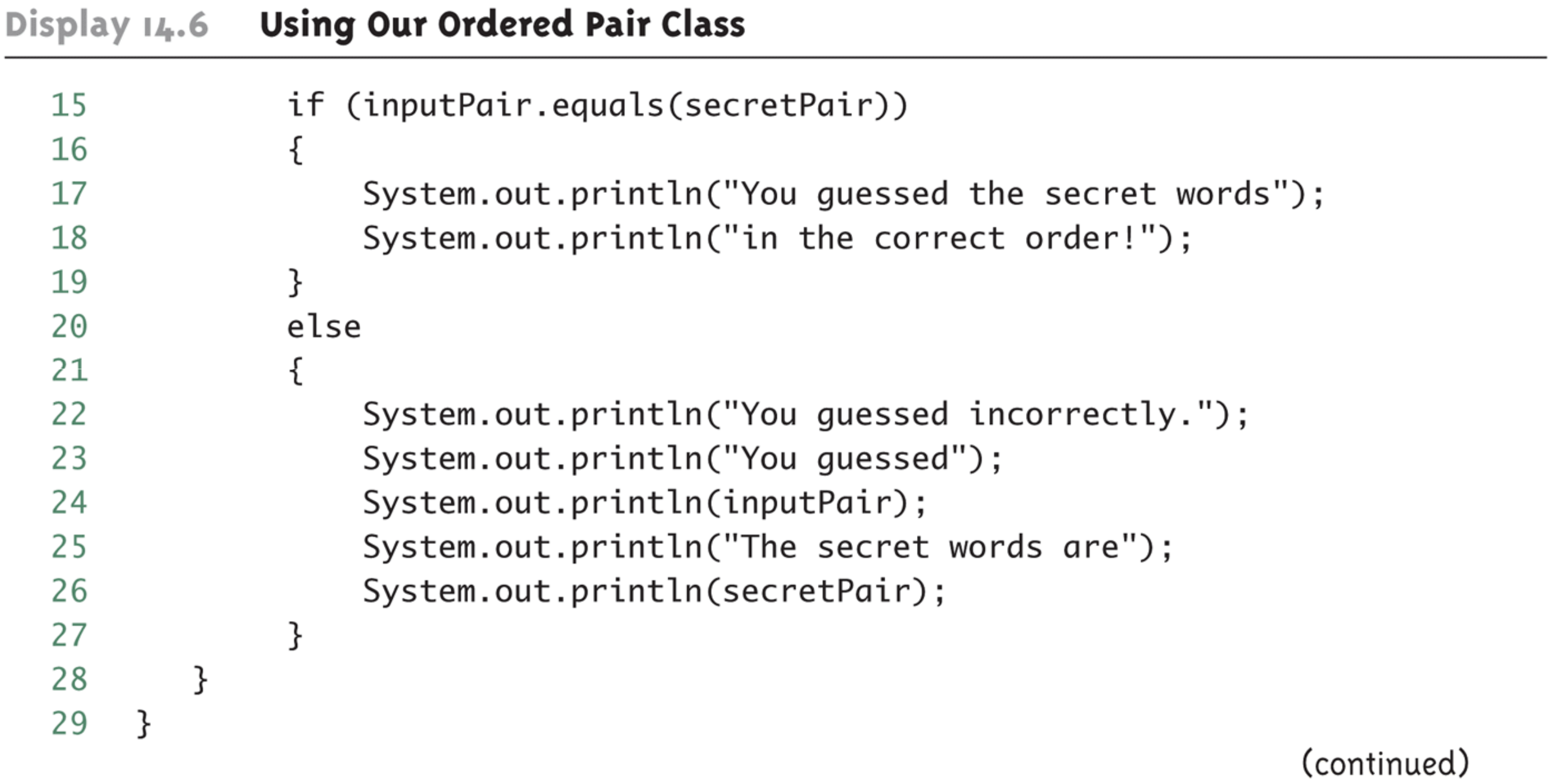


Display 14.6 Using Our Ordered Pair Class

```
import java.util.Scanner;

public class GenericPairDemo
{
   public static void main(String[] args)
   {
         Pair<String> secretPair =
              new Pair<String>("Happy", "Day");

         Scanner keyboard = new Scanner(System.in);
         System.out.println("Enter two words:");
         String word1 = keyboard.next();
         String word2 = keyboard.next();
         Pair<String> inputPair =
             new Pair<String>(word1, word2);
```

(continued)

Using Our Ordered Pair Class (Part 2 of 3)

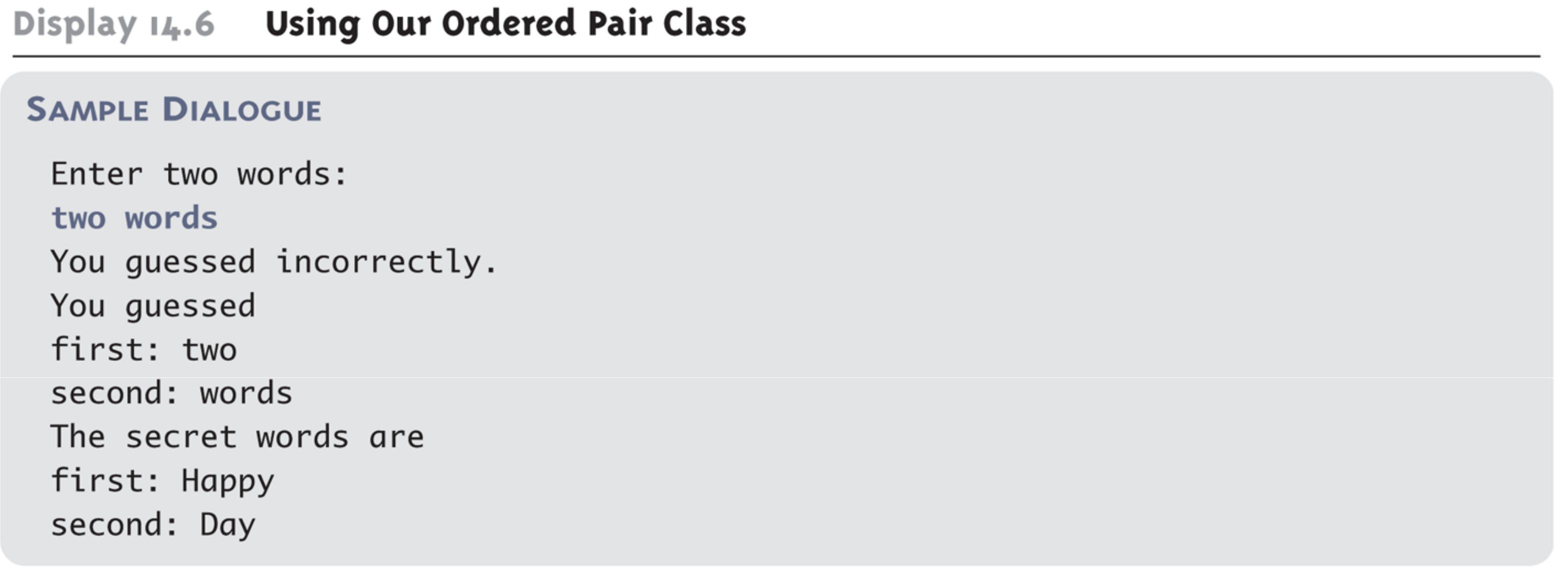


Display 14.6 Using Our Ordered Pair Class

```
        if (inputPair.equals(secretPair))
        {
            System.out.println("You guessed the secret words");
            System.out.println("in the correct order!");
        }
        else
        {
            System.out.println("You guessed incorrectly.");
            System.out.println("You guessed");
            System.out.println(inputPair);
            System.out.println("The secret words are");
            System.out.println(secretPair);
        }
    }
}
```

(continued)

Using Our Ordered Pair Class (Part 3 of 3)



Display 14.6 **Using Our Ordered Pair Class**

Sample Dialogue

```
Enter two words:
two words
You guessed incorrectly.
You guessed
first: two
second: words
The secret words are
first: Happy
second: Day
```

Pitfall: A Generic Constructor Name Has No Type Parameter

- Although the class name in a parameterized class definition has a type parameter attached, the type parameter is not used in the heading of the constructor definition

```
public Pair<T>()
```

- A constructor can use the type parameter as the type for a parameter of the constructor, but in this case, the angular brackets are not used

```
public Pair(T first, T second)
```

- However, when a generic class is instantiated, the angular brackets are used

```
Pair<String> pair =
    new Pair<STring>("Happy", "Day");
```

Pitfall: A Primitive Type Cannot be Plugged in for a Type Parameter

- The type plugged in for a type parameter must always be a reference type
  - It cannot be a primitive type such as **`int`**, **`double`**, or **`char`**
  - However, now that Java has automatic boxing, this is not a big restriction
  - Note: reference types can include arrays

Pitfall: A Type Parameter Cannot Be Used Everywhere a Type Name Can Be Used

- Within the definition of a parameterized class definition, there are places where an ordinary class name would be allowed, but a type parameter is not allowed
- In particular, the type parameter cannot be used in simple expressions using new to create a new object
  - For instance, the type parameter cannot be used as a constructor name or like a constructor:

```
T object = new T();
T[] a = new T[10];
```

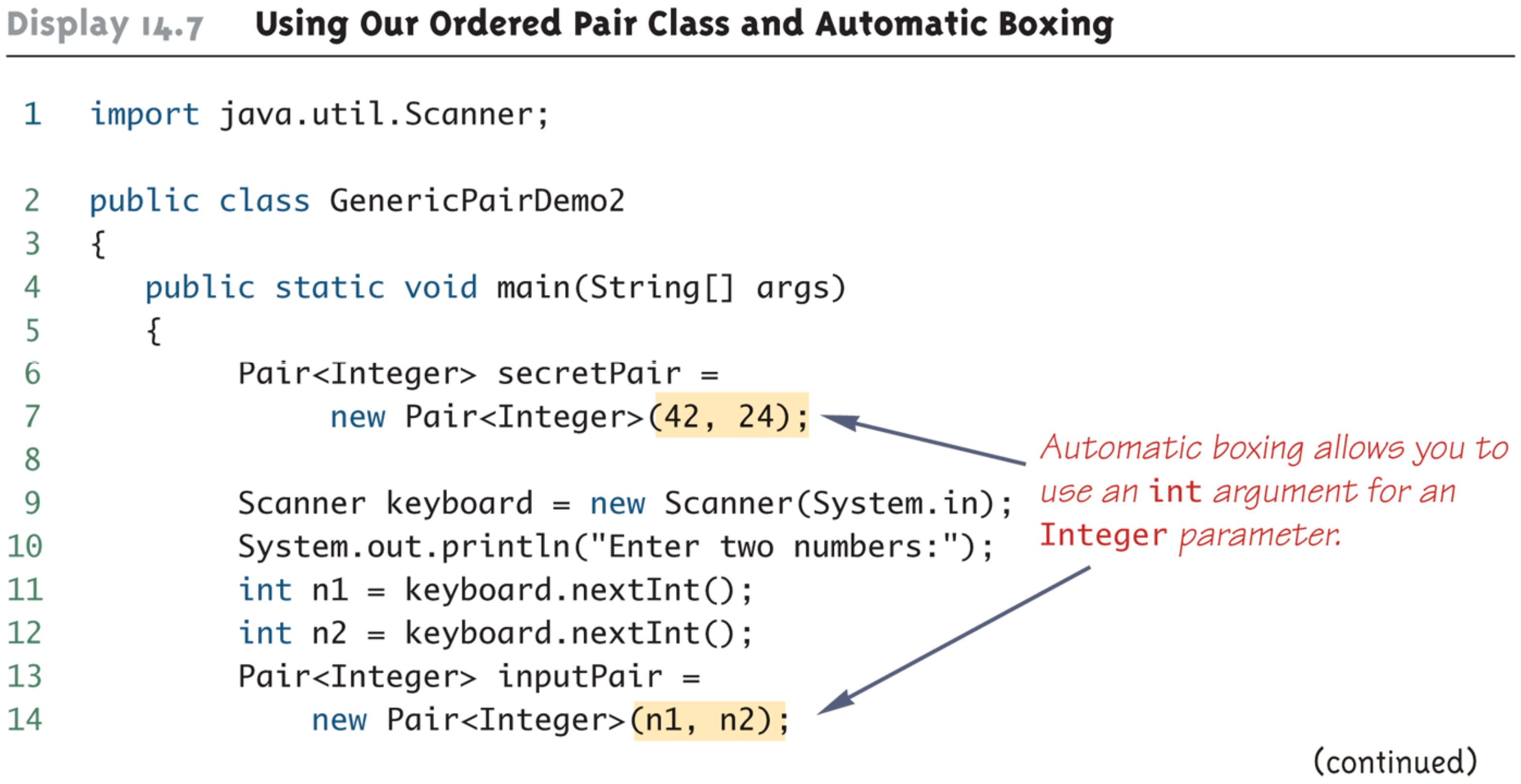
Pitfall: An Instantiation of a Generic Class Cannot be an Array Base Type

- Arrays such as the following are illegal:

```
Pair<String>[] a =
   new Pair<String>[10];
```

  - Although this is a reasonable thing to want to do, it is not allowed given the way that Java implements generic classes

Using Our Ordered Pair Class and Automatic Boxing (Part 1 of 3)



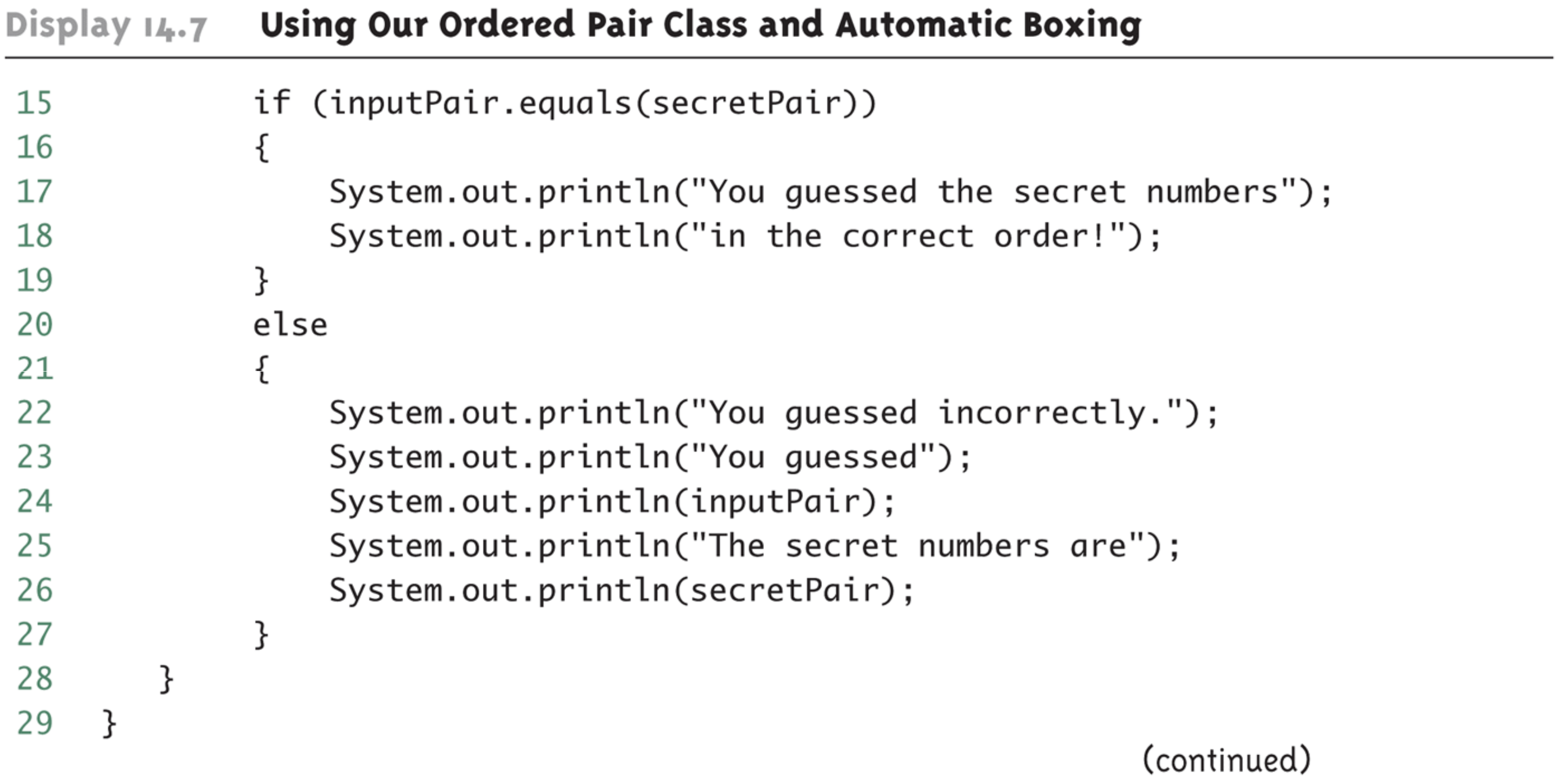
Display 14.7 **Using Our Ordered Pair Class and Automatic Boxing**

```
import java.util.Scanner;

public class GenericPairDemo2
{
   public static void main(String[] args)
   {
        Pair<Integer> secretPair =
             new Pair<Integer>(42, 24);

        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter two numbers:");
        int n1 = keyboard.nextInt();
        int n2 = keyboard.nextInt();
        Pair<Integer> inputPair =
            new Pair<Integer>(n1, n2);
```

Automatic boxing allows you to use an `int` *argument for an* `Integer` *parameter.*

(continued)

Using Our Ordered Pair Class and Automatic Boxing (Part 2 of 3)

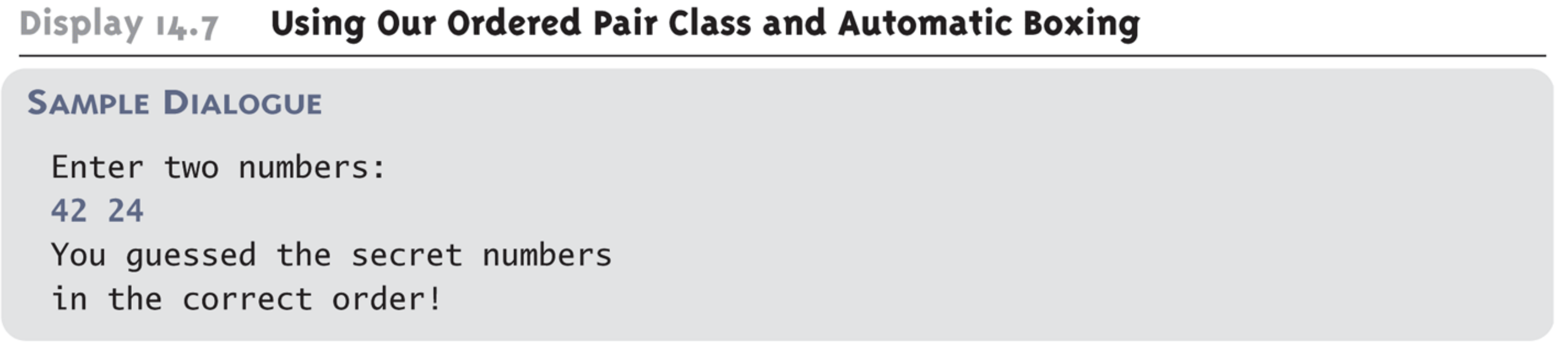


Display 14.7 Using Our Ordered Pair Class and Automatic Boxing

```
        if (inputPair.equals(secretPair))
        {
            System.out.println("You guessed the secret numbers");
            System.out.println("in the correct order!");
        }
        else
        {
            System.out.println("You guessed incorrectly.");
            System.out.println("You guessed");
            System.out.println(inputPair);
            System.out.println("The secret numbers are");
            System.out.println(secretPair);
        }
   }
}
```

(continued)

Using Our Ordered Pair Class and Automatic Boxing (Part 3 of 3)



Display 14.7 **Using Our Ordered Pair Class and Automatic Boxing**

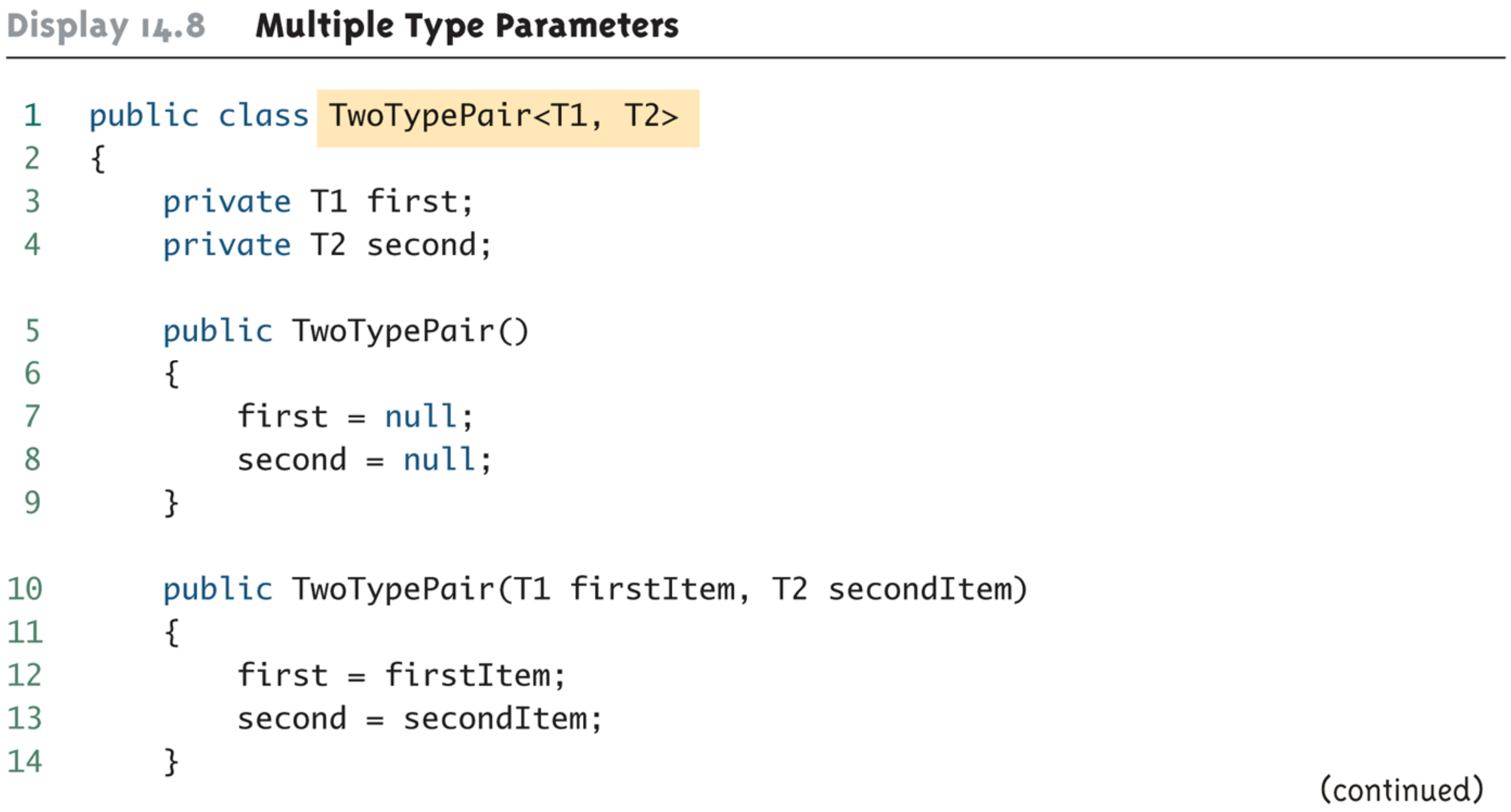
Sample Dialogue

```
Enter two numbers:
42 24
You guessed the secret numbers
in the correct order!
```

Pitfall: A Class Definition Can Have More Than One Type Parameter

- A generic class definition can have any number of type parameters
  - Multiple type parameters are listed in angular brackets just as in the single type parameter case, but are separated by commas

Multiple Type Parameters (Part 1 of 4)

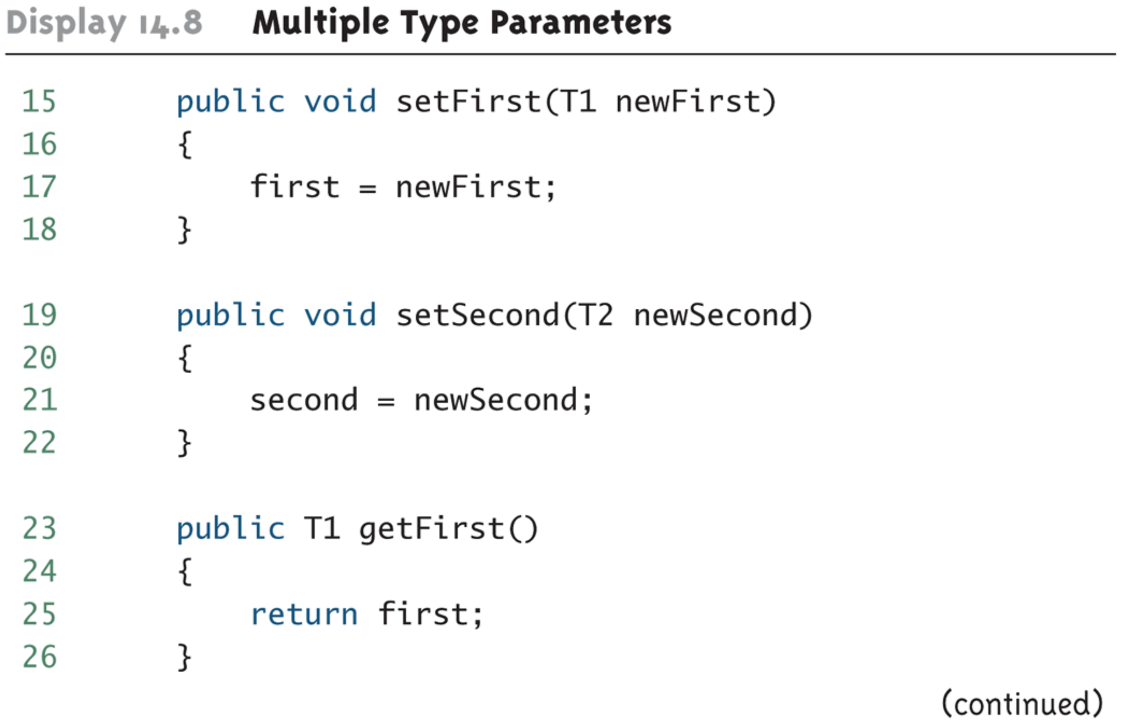


Display 14.8 **Multiple Type Parameters**

```
public class TwoTypePair<T1, T2>
{
    private T1 first;
    private T2 second;

    public TwoTypePair()
    {
        first = null;
        second = null;
    }

    public TwoTypePair(T1 firstItem, T2 secondItem)
    {
        first = firstItem;
        second = secondItem;
    }
```

(continued)

Multiple Type Parameters (Part 2 of 4)

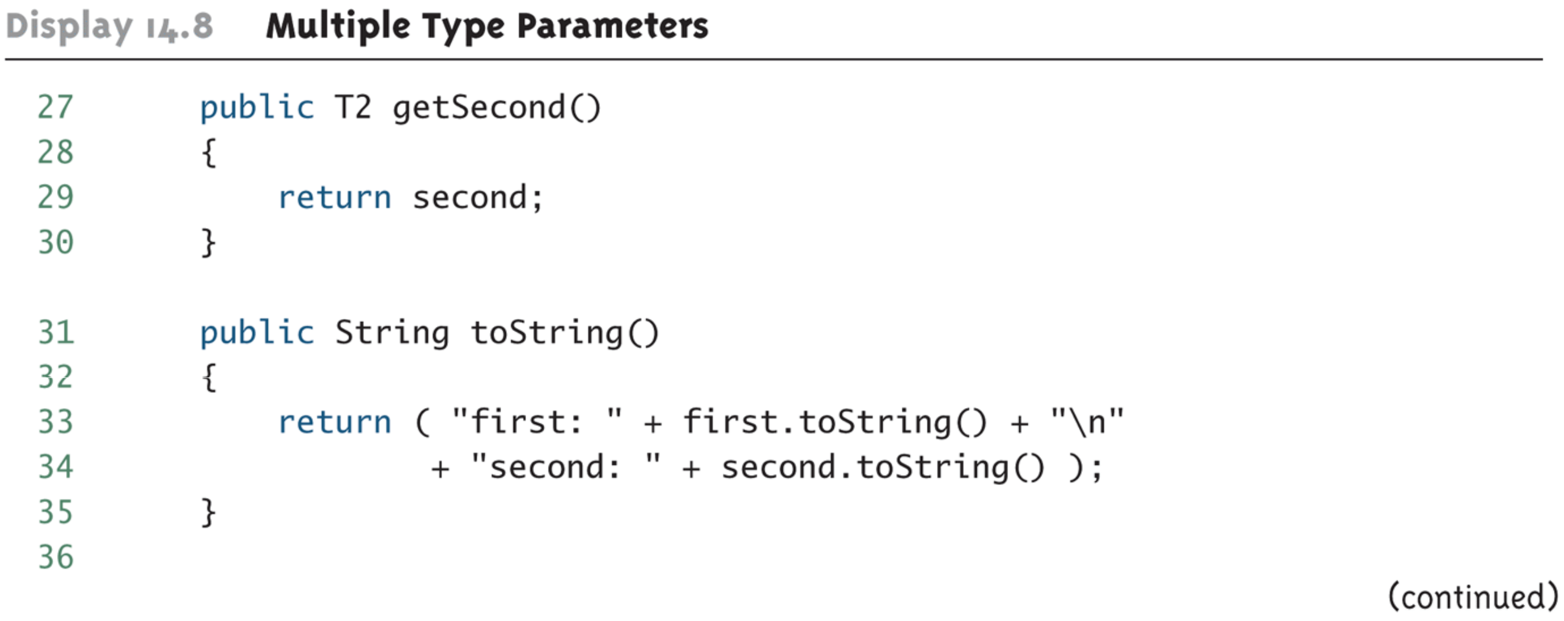


Display 14.8 **Multiple Type Parameters**

```
    public void setFirst(T1 newFirst)
    {
        first = newFirst;
    }

    public void setSecond(T2 newSecond)
    {
        second = newSecond;
    }

    public T1 getFirst()
    {
        return first;
    }
```

(continued)

Multiple Type Parameters (Part 3 of 4)

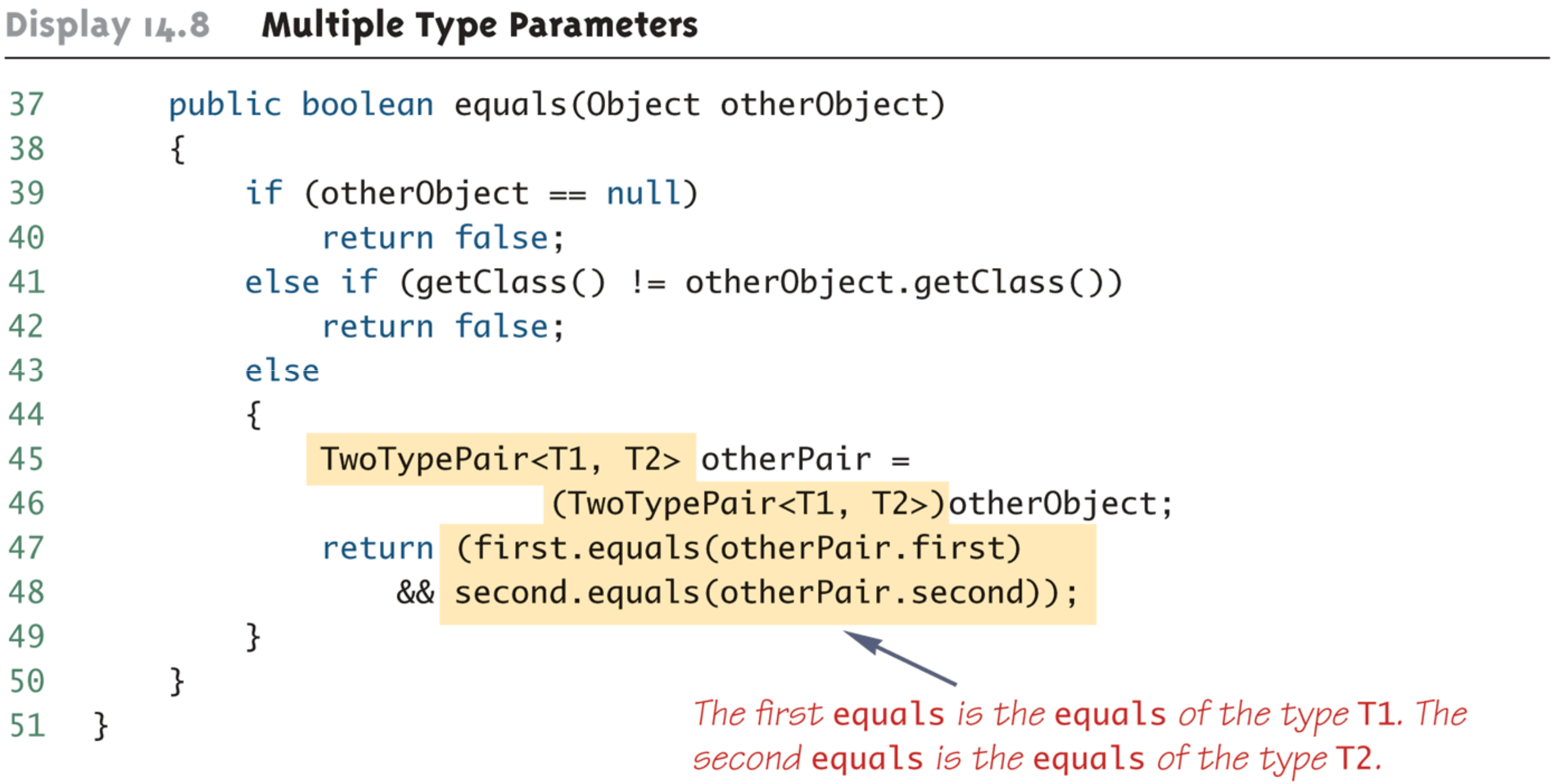


Display 14.8 **Multiple Type Parameters**

```
    public T2 getSecond()
    {
        return second;
    }

    public String toString()
    {
        return ( "first: " + first.toString() + "\n"
                     + "second: " + second.toString() );
    }

```

(continued)

Multiple Type Parameters (Part 4 of 4)



Display 14.8 Multiple Type Parameters

```
    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass() != otherObject.getClass())
            return false;
        else
        {
            TwoTypePair<T1, T2> otherPair =
                        (TwoTypePair<T1, T2>)otherObject;
            return (first.equals(otherPair.first)
                && second.equals(otherPair.second));
        }
    }
}
```

The first equals *is the* equals *of the type* T1. *The second* equals *is the* equals *of the type* T2.

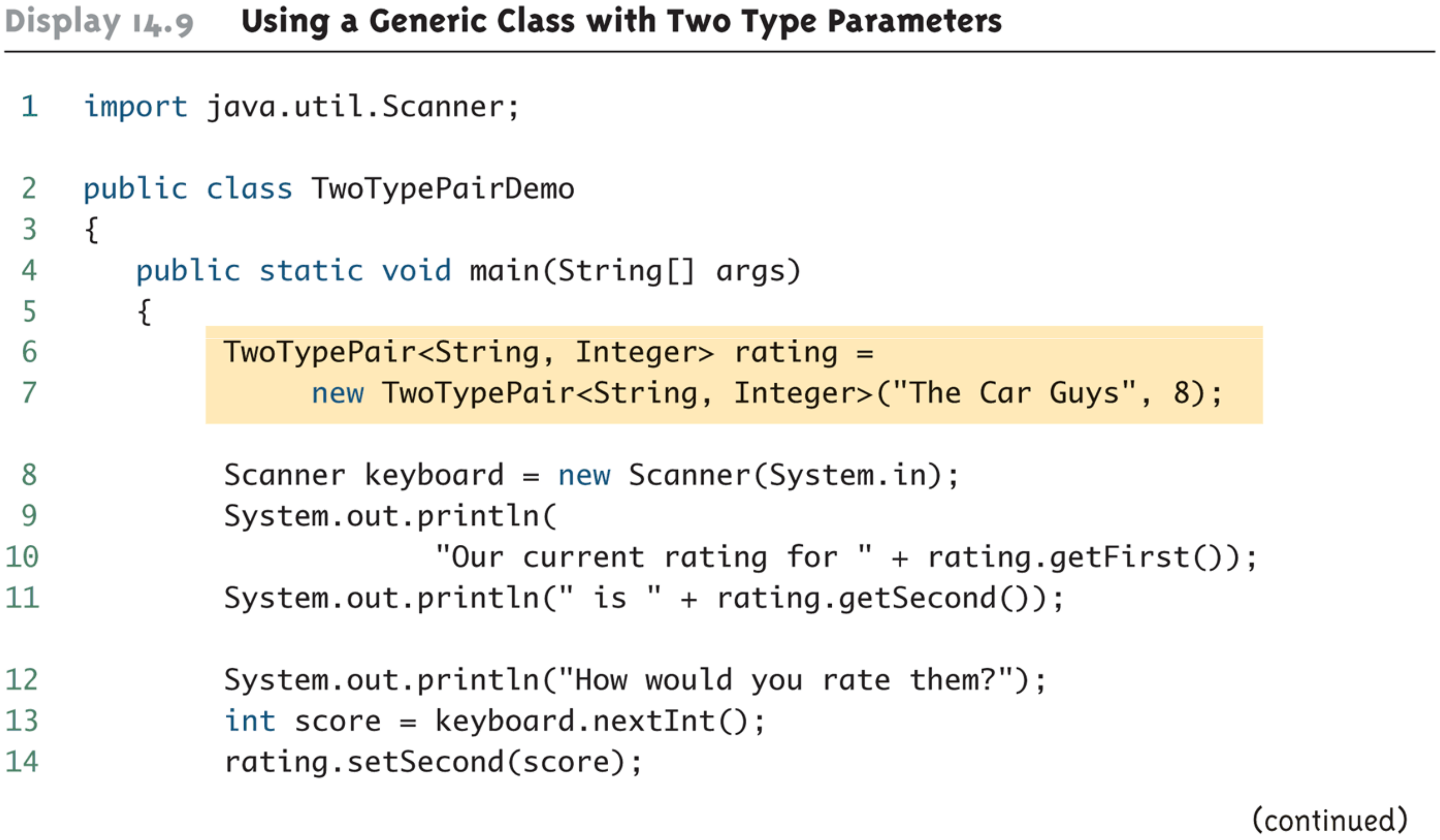
Pitfall: A Generic Class Cannot Be an Exception Class

- It is not permitted to create a generic class with `Exception`, `Error`, `Throwable`, or any descendent class of `Throwable`
  - A generic class cannot be created whose objects are throwable

    ```
    public class GEx<T> extends Exception
    ```

  - The above example will generate a compiler error message

Using a Generic Class with Two Type Parameters (Part 1 of 2)

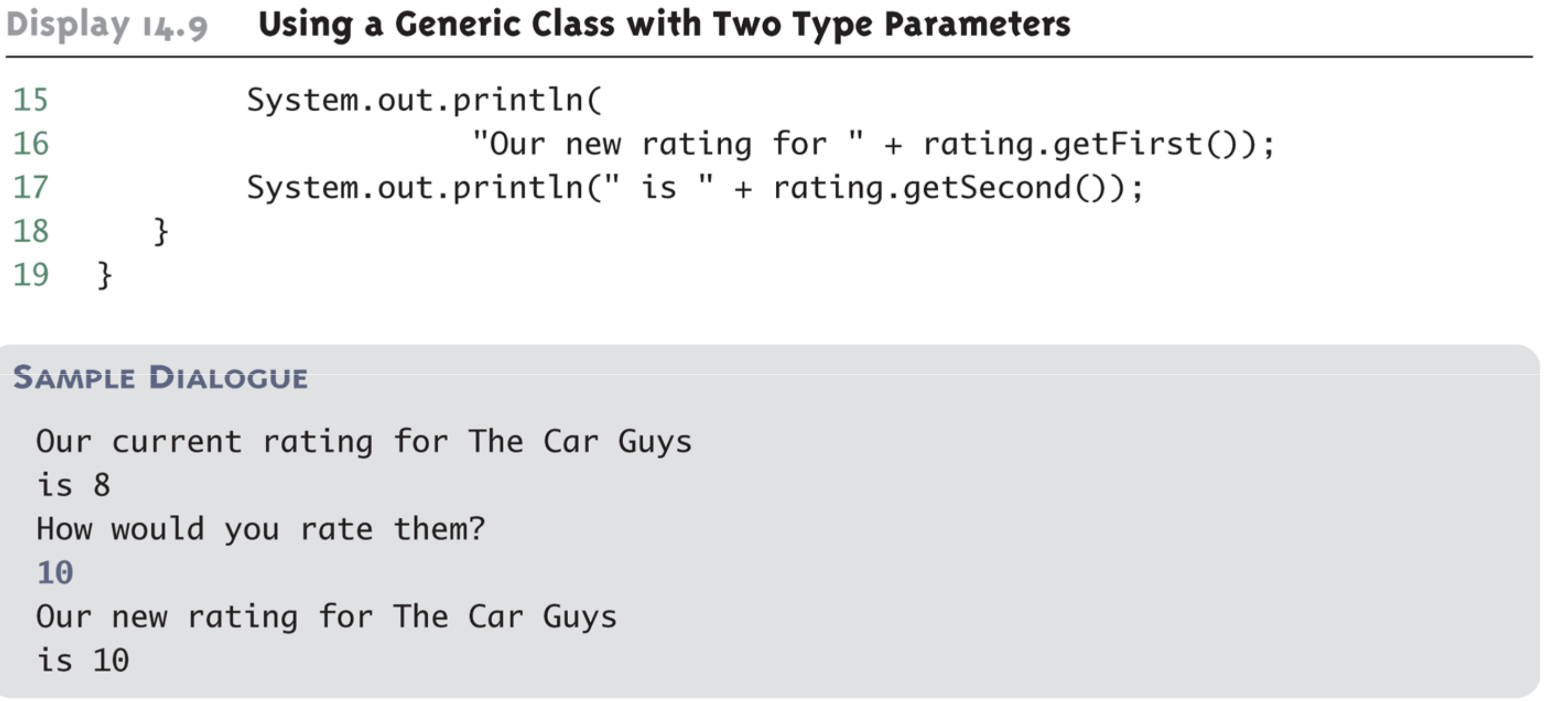


Display 14.9 **Using a Generic Class with Two Type Parameters**

```
import java.util.Scanner;

public class TwoTypePairDemo
{
   public static void main(String[] args)
   {
        TwoTypePair<String, Integer> rating =
             new TwoTypePair<String, Integer>("The Car Guys", 8);

        Scanner keyboard = new Scanner(System.in);
        System.out.println(
                    "Our current rating for " + rating.getFirst());
        System.out.println(" is " + rating.getSecond());

        System.out.println("How would you rate them?");
        int score = keyboard.nextInt();
        rating.setSecond(score);
```

(continued)

Using a Generic Class with Two Type Parameters (Part 2 of 2)



Display 14.9 **Using a Generic Class with Two Type Parameters**

```
        System.out.println(
                    "Our new rating for " + rating.getFirst());
        System.out.println(" is " + rating.getSecond());
    }
}
```

Sample Dialogue

```
Our current rating for The Car Guys
is 8
How would you rate them?
10
Our new rating for The Car Guys
is 10
```

Bounds for Type Parameters

- Sometimes it makes sense to restrict the possible types that can be plugged in for a type parameter **`T`**
  - For instance, to ensure that only classes that implement the **`Comparable`** interface are plugged in for **`T`**, define a class as follows:

    ```
    public class RClass<T extends Comparable>
    ```

  - "**`extends Comparable`**" serves as a *bound* on the type parameter **`T`**
  - Any attempt to plug in a type for **`T`** which does not implement the **`Comparable`** interface will result in a compiler error message

Bounds for Type Parameters

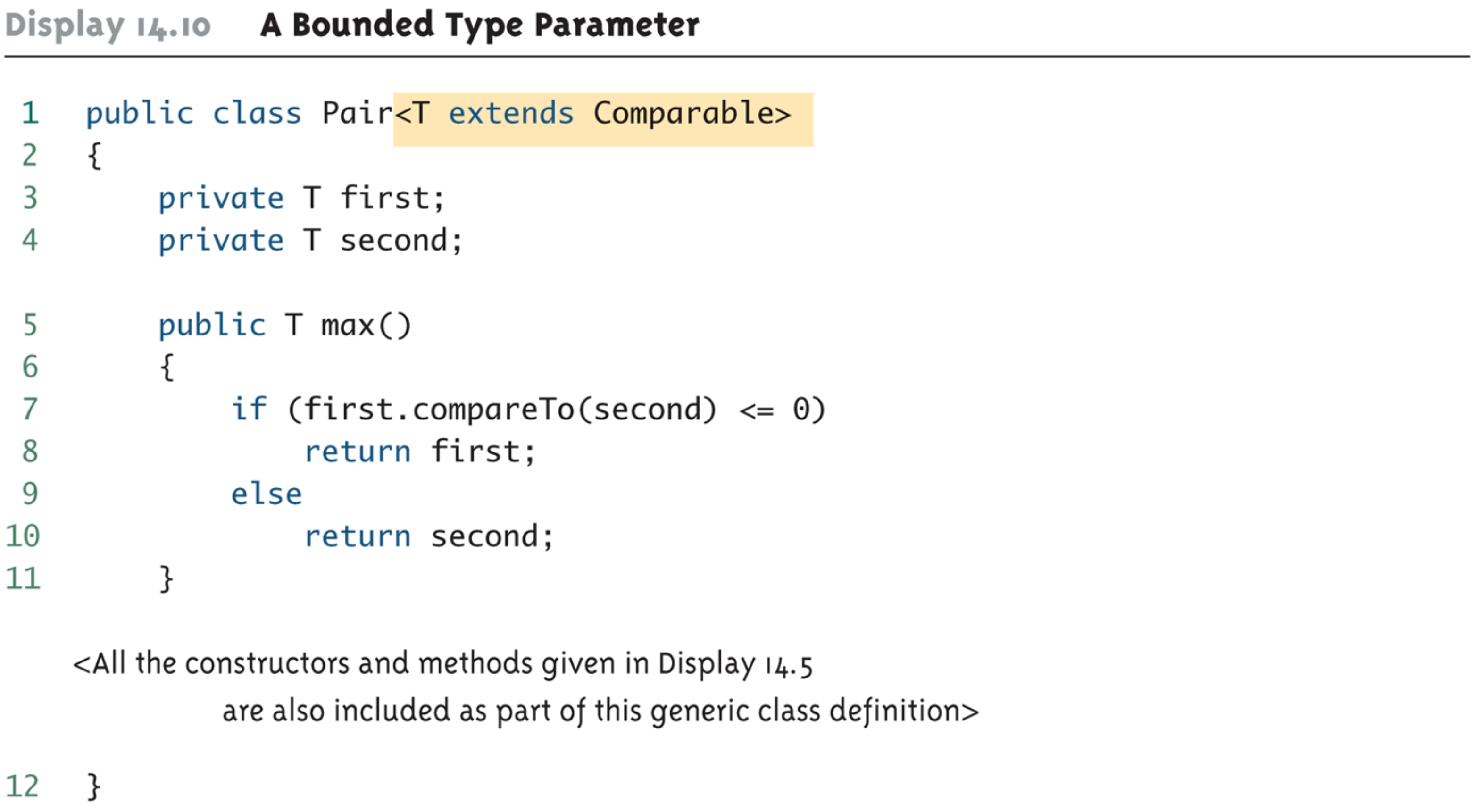
- A bound on a type may be a class name (rather than an interface name)
  - Then only descendent classes of the bounding class may be plugged in for the type parameters

```
public class ExClass<T extends Class1>
```

- A bounds expression may contain multiple interfaces and up to one class
- If there is more than one type parameter, the syntax is as follows:

```
public class Two<T1 extends Class1, T2 extends
  Class2 & Comparable>
```

A Bounded Type Parameter



Display 14.10 A Bounded Type Parameter

```
public class Pair<T extends Comparable>
{
    private T first;
    private T second;

    public T max()
    {
        if (first.compareTo(second) <= 0)
            return first;
        else
            return second;
    }

<All the constructors and methods given in Display 14.5
            are also included as part of this generic class definition>

}
```

Tip: Generic Interfaces

- An interface can have one or more type parameters
- The details and notation are the same as they are for classes with type parameters

Generic Methods

- When a generic class is defined, the type parameter can be used in the definitions of the methods for that generic class
- In addition, a generic method can be defined that has its own type parameter that is not the type parameter of any class
  - A generic method can be a member of an ordinary class or a member of a generic class that has some other type parameter
  - The type parameter of a generic method is local to that method, not to the class

Generic Methods

- The type parameter must be placed (in angular brackets) after all the modifiers, and before the returned type

```
public static <T> T genMethod(T[] a)
```

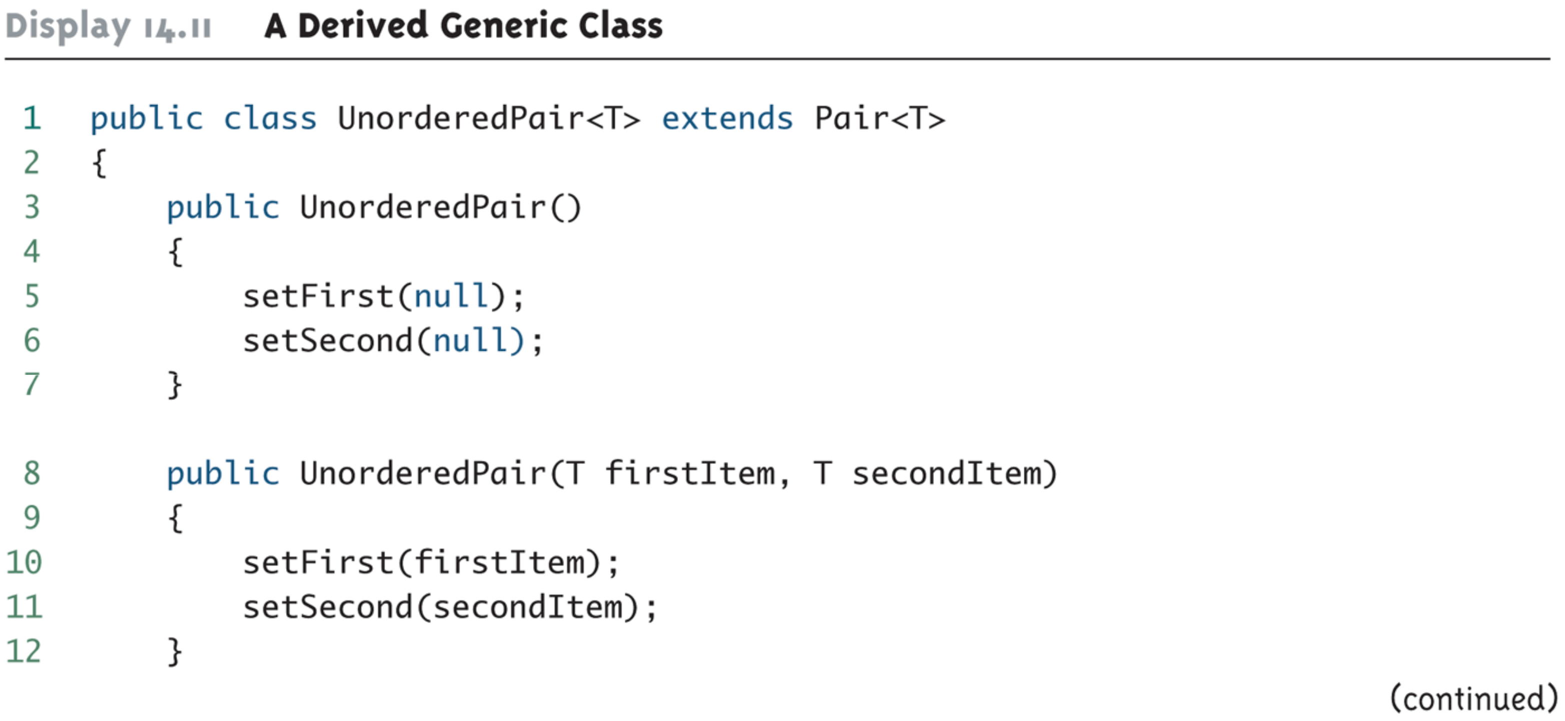
- When one of these generic methods is invoked, the method name is prefaced with the type to be plugged in, enclosed in angular brackets

```
String s = NonG.<String>genMethod(c);
```

Inheritance with Generic Classes

- A generic class can be defined as a derived class of an ordinary class or of another generic class
  - As in ordinary classes, an object of the subclass type would also be of the superclass type
- Given two classes: **A** and **B**, and given **G**: a generic class, there is no relationship between **`G<A>`** and **`G<B>`**
  - This is true regardless of the relationship between class **A** and **B**, e.g., if class **B** is a subclass of class **A**

A Derived Generic Class (Part 1 of 2)

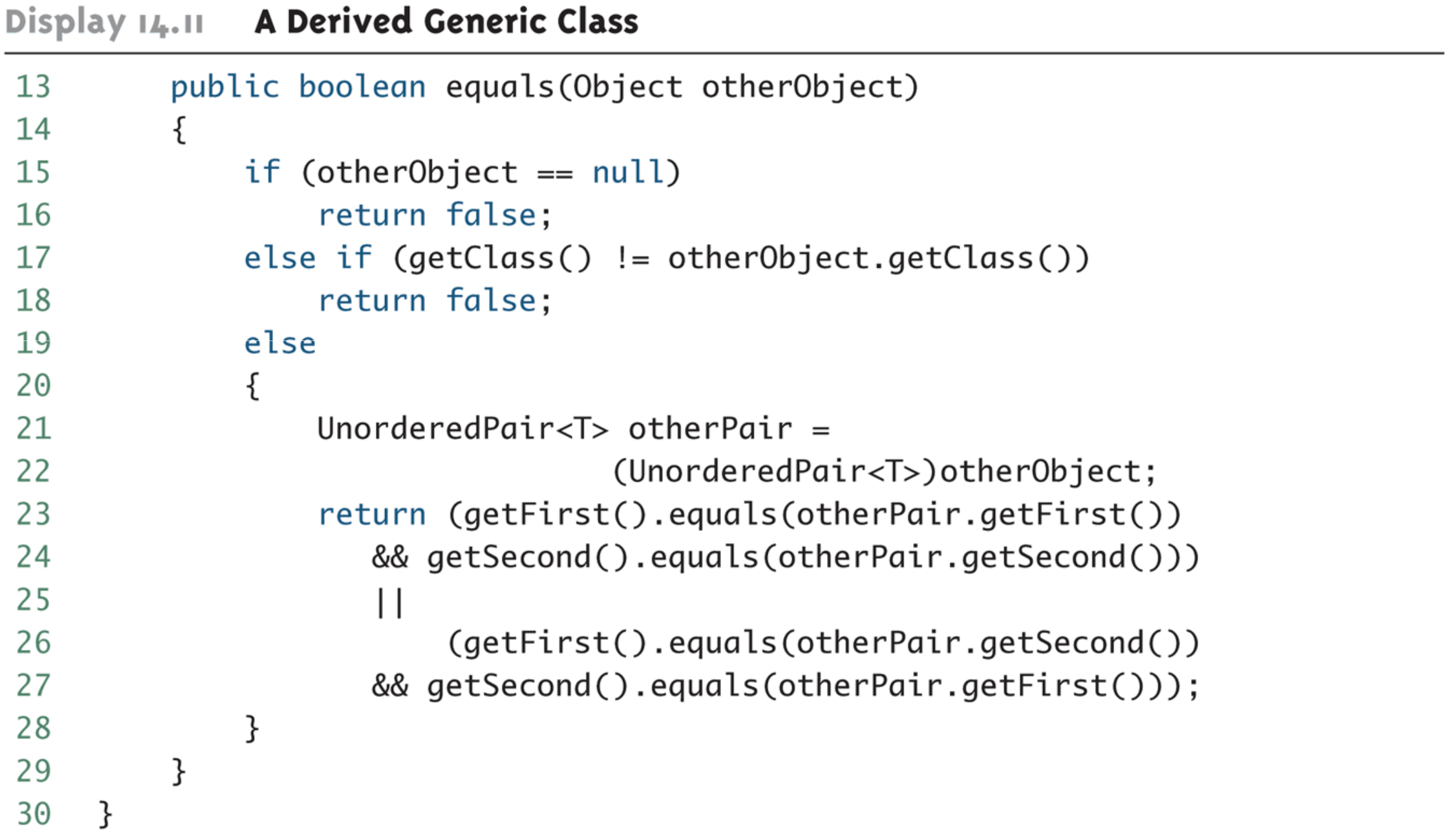


Display 14.11 **A Derived Generic Class**

```
public class UnorderedPair<T> extends Pair<T>
{
    public UnorderedPair()
    {
        setFirst(null);
        setSecond(null);
    }

    public UnorderedPair(T firstItem, T secondItem)
    {
        setFirst(firstItem);
        setSecond(secondItem);
    }
```

(continued)

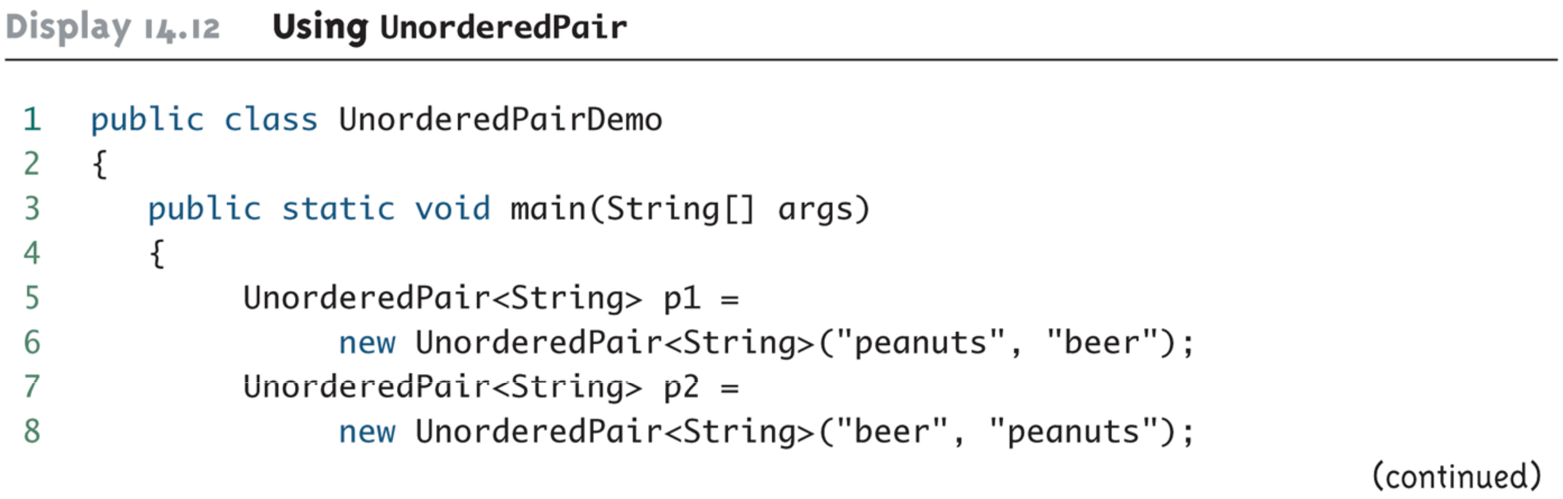
A Derived Generic Class (Part 2 of 2)



Display 14.11 **A Derived Generic Class**

```
    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass() != otherObject.getClass())
            return false;
        else
        {
            UnorderedPair<T> otherPair =
                            (UnorderedPair<T>)otherObject;
            return (getFirst().equals(otherPair.getFirst())
               && getSecond().equals(otherPair.getSecond()))
               ||
                   (getFirst().equals(otherPair.getSecond())
               && getSecond().equals(otherPair.getFirst()));
        }
    }
}
```

Using UnorderedPair (Part 1 of 2)

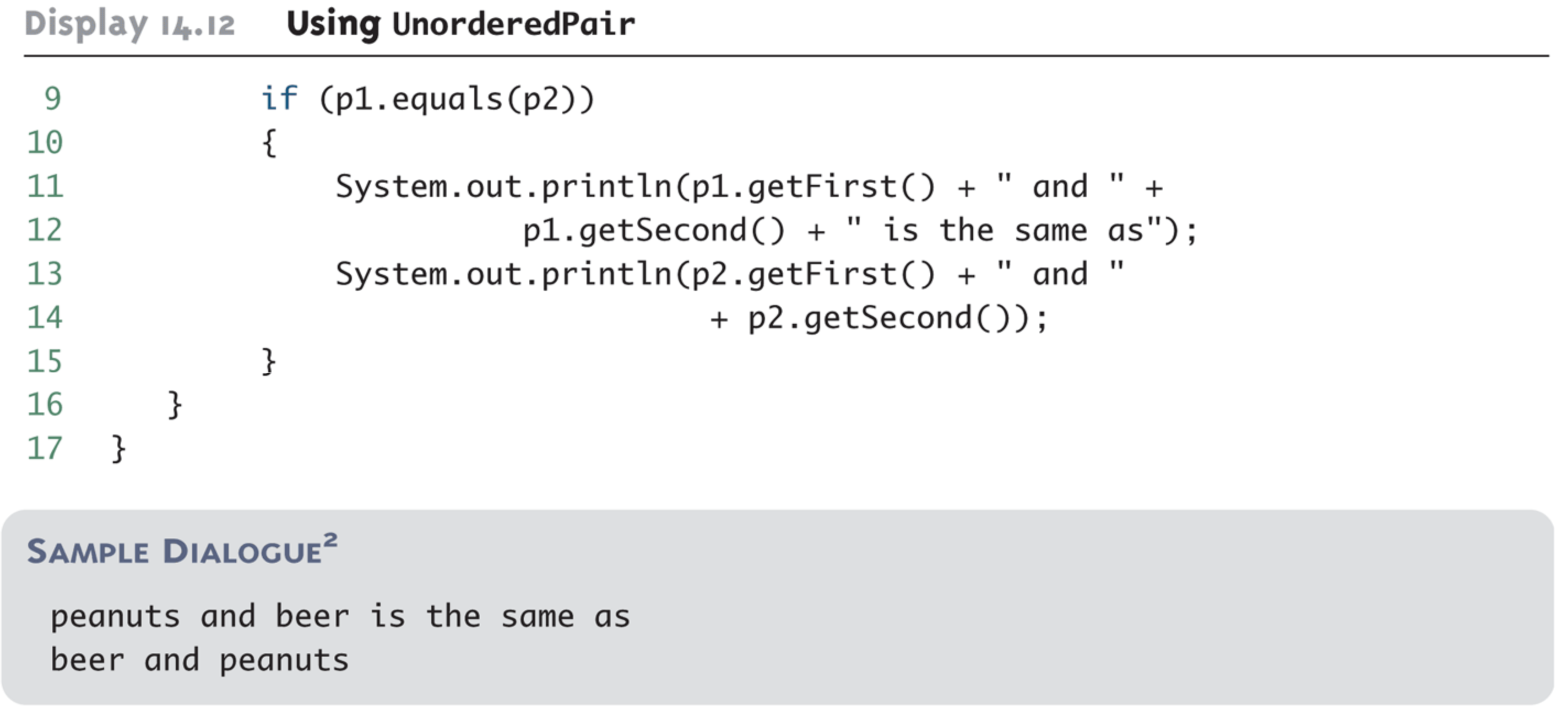


Display 14.12 **Using UnorderedPair**

```
public class UnorderedPairDemo
{
   public static void main(String[] args)
   {
        UnorderedPair<String> p1 =
             new UnorderedPair<String>("peanuts", "beer");
        UnorderedPair<String> p2 =
             new UnorderedPair<String>("beer", "peanuts");
```

(continued)

Using UnorderedPair (Part 2 of 2)



Display 14.12 **Using UnorderedPair**

```
        if (p1.equals(p2))
        {
            System.out.println(p1.getFirst() + " and " +
                      p1.getSecond() + " is the same as");
            System.out.println(p2.getFirst() + " and "
                                + p2.getSecond());
        }
    }
}
```

Sample Dialogue[2]

```
peanuts and beer is the same as
beer and peanuts
```