



Chapter 14

Generics and the ArrayList Class

Slides prepared by Rose Williams,
Binghamton University

Kenrick Mock, *University of Alaska
Anchorage*

ALWAYS LEARNING

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

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-2

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

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-3

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

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-4

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

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-5

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>();
```

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-6

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*

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-7

## 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

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-8

## 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);`

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-9

## 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

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-10

## 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

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-11

## 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**

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-12

## 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

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-13

## Some Methods in the Class **ArrayList** (Part 1 of 11)

Display 14-1 Some Methods in the Class ArrayList

### CONSTRUCTORS

```
public ArrayList<Base_Type>(int initialCapacity)
```

Creates an empty ArrayList with the specified *Base\_Type* and initial capacity.

```
public ArrayList<Base_Type>()
```

Creates an empty ArrayList with the specified *Base\_Type* and an initial capacity of 10.

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-14

## Some Methods in the Class **ArrayList** (Part 2 of 11)

Display 14-1 Some Methods in the Class ArrayList

### ARRAYLIKE METHODS

```
public Base_Type set(int index, Base_Type newElement)
```

Sets the element at the specified *index* to *newElement*. Returns the element previously at that position, but the method is often used as if it were a void method. If you draw an analogy between the *ArrayList* and an array *a*, this statement is analogous to setting *a[index]* to the value *newElement*. The *index* must be a value greater than or equal to 0 and less than the current size of the *ArrayList*. Throws an *IndexOutOfBoundsException* if the *index* is not in this range.

```
public Base_Type get(int index)
```

Returns the element at the specified *index*. This statement is analogous to returning *a[index]* for an array *a*. The *index* must be a value greater than or equal to 0 and less than the current size of the *ArrayList*. Throws *IndexOutOfBoundsException* if the *index* is not in this range.

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-15

## Some Methods in the Class **ArrayList** (Part 3 of 11)

Display 14-1 Some Methods in the Class ArrayList

### METHODS TO ADD ELEMENTS

```
public boolean add(Base_Type newElement)
```

Adds the specified element to the end of the calling *ArrayList* and increases the *ArrayList*'s size by one. The capacity of the *ArrayList* is increased if that is required. Returns true if the add was successful. (The return type is boolean, but the method is typically used as if it were a void method.)

```
public void add(int index, Base_Type newElement)
```

Inserts *newElement* as an element in the calling *ArrayList* at the specified *index*. Each element in the *ArrayList* with an *index* greater or equal to *index* is shifted upward to have an *index* that is one greater than the value it had previously. The *index* must be a value greater than or equal to 0 and less than or equal to the current size of the *ArrayList*. Throws *IndexOutOfBoundsException* if the *index* is not in this range. Note that you can use this method to add an element after the last element. The capacity of the *ArrayList* is increased if that is required.

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-16

## Some Methods in the Class **ArrayList** (Part 4 of 11)

Display I4-1 Some Methods in the Class ArrayList

### METHODS TO REMOVE ELEMENTS

```
public Base_Type remove(int index)
```

Deletes and returns the element at the specified index. Each element in the ArrayList with an index greater than *index* is decreased to have an index that is one less than the value it had previously. The *index* must be a value greater than or equal to 0 and less than the current size of the ArrayList. Throws *IndexOutOfBoundsException* if the *index* is not in this range. Often used as if it were a void method.

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-17

## Some Methods in the Class **ArrayList** (Part 5 of 11)

Display I4-1 Some Methods in the Class ArrayList

```
protected void removeRange(int fromIndex, int toIndex)
```

Deletes all the element with indices *i* such that *fromIndex* ≤ *i* < *toIndex*. Element with indices greater than or equal to *toIndex* are decreased appropriately.

```
public boolean remove(Object theElement)
```

Removes one occurrence of *theElement* from the calling ArrayList. If *theElement* is found in the ArrayList, then each element in the ArrayList with an index greater than the removed element's index is decreased to have an index that is one less than the value it had previously. Returns true if *theElement* was found (and removed). Returns false if *theElement* was not found in the calling ArrayList.

```
public void clear()
```

Removes all elements from the calling ArrayList and sets the ArrayList's size to zero.

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-18

## Some Methods in the Class **ArrayList** (Part 6 of 11)

Display I4-1 Some Methods in the Class ArrayList

### SEARCH METHODS

```
public boolean contains(Object target)
```

Returns true if the calling ArrayList contains *target*; otherwise, returns false. Uses the method *equals* of the object *target* to test for equality with any element in the calling ArrayList.

```
public int indexOf(Object target)
```

Returns the index of the first element that is equal to *target*. Uses the method *equals* of the object *target* to test for equality. Returns -1 if *target* is not found.

```
public int lastIndexOf(Object target)
```

Returns the index of the last element that is equal to *target*. Uses the method *equals* of the object *target* to test for equality. Returns -1 if *target* is not found.

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-19

## Some Methods in the Class **ArrayList** (Part 7 of 11)

Display I4-1 Some Methods in the Class ArrayList

### MEMORY MANAGEMENT (SIZE AND CAPACITY)

```
public boolean isEmpty()
```

Returns true if the calling ArrayList is empty (that is, has size 0); otherwise, returns false.

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-20

## Some Methods in the Class **ArrayList** (Part 8 of 11)

Display 14-1 Some Methods in the Class ArrayList

`public int size()`

Returns the number of elements in the calling ArrayList.

`public void ensureCapacity(int newCapacity)`

Increases the capacity of the calling ArrayList, if necessary, in order to ensure that the ArrayList can hold at least newCapacity elements. Using ensureCapacity can sometimes increase efficiency, but its use is not needed for any other reason.

`public void trimToSize()`

Trims the capacity of the calling ArrayList to the ArrayList's current size. This method is used to save storage space.

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-21

## Some Methods in the Class **ArrayList** (Part 9 of 11)

Display 14-1 Some Methods in the Class ArrayList

MAKE A COPY

`public Object[] toArray()`

Returns an array containing all the elements on the list. Preserves the order of the elements.

`public Type[] toArray(Type[] a)`

Returns an array containing all the elements on the list. Preserves the order of the elements. *Type* can be any class types. If the list will fit in *a*, the elements are copied to *a* and *a* is returned. Any elements of *a* not needed for list elements are set to null. If the list will not fit in *a*, a new array is created.

(As we will discuss in Section 14.2, the correct Java syntax for this method heading is

`public <Type> Type[] toArray(Type[] a)`

However, at this point we have not yet explained this kind of type parameter syntax.)

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-22

## Some Methods in the Class **ArrayList** (Part 10 of 11)

Display 14-1 Some Methods in the Class ArrayList

`public Object clone()`

Returns a shallow copy of the calling ArrayList. Warning: The clone is not an independent copy. Subsequent changes to the clone may affect the calling object and vice versa. (See Chapter 5 for a discussion of shallow copy.)

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-23

## Some Methods in the Class **ArrayList** (Part 11 of 11)

Display 14-1 Some Methods in the Class ArrayList

EQUALITY

`public boolean equals(Object other)`

If *other* is another ArrayList (of any base type), then equals returns true if and only if both ArrayLists are of the same size and contain the same list of elements in the same order. (In fact, if *other* is any kind of list, then equals returns true if and only if both the calling ArrayList and *other* are of the same size and contain the same list of elements in the same order. Lists are discussed in Chapter 16.)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-24

## 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**

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-25

## 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**)

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-26

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

Display 14.2 A for-each Loop Used with an ArrayList

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

3 public class ArrayListDemo
4 {
5 public static void main(String[] args)
6 {
7 ArrayList<String> toDoList = new ArrayList<String>(20);
8 System.out.println(
9 "Enter list entries, when prompted.");
10 boolean done = false;
11 String next = null;
12 String answer;
13 Scanner keyboard = new Scanner(System.in);
```

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-27

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

Display 14.2 A for-each Loop Used with an ArrayList

```
14 while (! done)
15 {
16 System.out.println("Input an entry:");
17 next = keyboard.nextLine();
18 toDoList.add(next);

19 System.out.print("More items for the list? ");
20 answer = keyboard.nextLine();
21 if (!(answer.equalsIgnoreCase("yes")))
22 done = true;
23 }

24 System.out.println("The list contains:");
25 for (String entry : toDoList)
26 System.out.println(entry);
27 }
28}
29
```

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-28

## 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.
```

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-29

## Golf Score Program (Part 1 of 6)

### Display 14.3 Golf Score Program

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

3 public class GolfScores
4 {
5 /**
6 * Shows differences between each of a list of golf scores and their average.
7 */
8 public static void main(String[] args)
9 {
10 ArrayList<Double> score = new ArrayList<Double>();
11
12 System.out.println("This program reads golf scores and shows");
13 System.out.println("how much each differs from the average.");
14
15 System.out.print("Enter golf scores:");
16 fillArrayList(score);
17 showDifference(score);
```

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-30

## Golf Score Program (Part 2 of 6)

### Display 14.3 Golf Score Program

```
17 /**
18 * Reads values into the array a.
19 */
20 public static void fillArrayList(ArrayList<Double> a)
21 {
22 System.out.println("Enter a list of nonnegative numbers.");
23 System.out.println("Mark the end of the list with a negative number.");
24 Scanner keyboard = new Scanner(System.in);
25 }
26
```

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-31

## Golf Score Program (Part 3 of 6)

### Display 14-3 Golf Score Program

```
25 double next;
26 int index = 0;
27 next = keyboard.nextDouble();
28 while (next >= 0)
29 {
30 a.add(next); ← Because of automatic boxing, we can treat
31 next = keyboard.nextDouble(); values of type double as if their type were
32 } Double.
33 }
34 /**
35 * Returns the average of numbers in a.
36 */
37 public static double computeAverage(ArrayList<Double> a)
38 {
39 double total = 0;
40 for (Double element : a) ← A for-each loop is the nicest way to cycle
41 total = total + element; through all the elements in an
 Array list.
}
```

CONTINUOUS

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-32

## Golf Score Program (Part 4 of 6)

Display 14-3 Golf Score Program

```
42 int numberOfScores = a.size();
43 if (numberOfScores > 0)
44 {
45 return (total/numberOfScores);
46 }
47 else
48 {
49 System.out.println("ERROR: Trying to average 0 numbers.");
50 System.out.println("computeAverage returns 0.");
51 return 0;
52 }
```

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-33

## Golf Score Program (Part 5 of 6)

Display 14-3 Golf Score Program

```
54 /**
55 Gives screen output showing how much each of the elements
56 in a differ from their average.
57 */
58 public static void showDifference(ArrayList<Double> o)
59 {
60 double average = computeAverage(o);
61 System.out.println("Average of the " + o.size()
62 + " scores = " + average);
63 System.out.println("The scores are:");
64 for (Double element : o)
65 System.out.println(element + " differs from average by "
66 + (element - average));
67 }
68 }
```

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-34

## 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
```

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-35

### 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

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-36

## 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

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-37

## 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

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-38

## 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*

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-39

## 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

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-40

## 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

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-41

## 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>();
```

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-42

## A Class Definition with a Type Parameter

Display 14-4 A Class Definition with a Type Parameter

```
1 public class Sample<T>
2 {
3 private T data;
4
5 public void setData(T newData)
6 {
7 data = newData;
8 T is a parameter for a type.
9 }
10
11 public T getData()
12 {
13 return data;
14 }
15 }
```

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-43

## 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

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-44

## 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**

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-45

## A Generic Ordered Pair Class (Part 1 of 4)

Display 14-5 A Generic Ordered Pair Class

```
1 public class Pair<T>
2 {
3 private T first;
4 private T second;
5
5 public Pair()
6 {
7 first = null;
8 second = null;
9 }
10
11 public Pair(T firstItem, T secondItem)
12 {
13 first = firstItem;
14 second = secondItem;
15 }
16
17
18
19
20
21
22
23
24
25
26
```

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

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-46

## A Generic Ordered Pair Class (Part 2 of 4)

Display 14-5 A Generic Ordered Pair Class

```
15 public void setFirst(T newFirst)
16 {
17 first = newFirst;
18 }
19
20 public void setSecond(T newSecond)
21 {
22 second = newSecond;
23 }
24
25 public T getFirst()
26 {
27 return first;
28 }
29
30
31
32
33
34
35
36
```

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-47

## A Generic Ordered Pair Class (Part 3 of 4)

Display 14-5 A Generic Ordered Pair Class

```
27 public T getSecond()
28 {
29 return second;
30 }
31
32 public String toString()
33 {
34 return ("first: " + first.toString() + "\n"
35 + "second: " + second.toString());
36 }
```

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-48

## A Generic Ordered Pair Class (Part 4 of 4)

Display 14-5 A Generic Ordered Pair Class

```
37 public boolean equals(Object otherObject)
38 {
39 if (otherObject == null)
40 return false;
41 else if (getClass() != otherObject.getClass())
42 return false;
43 else
44 {
45 Pair<T> otherPair = (Pair<T>)otherObject;
46 return (first.equals(otherPair.first))
47 && second.equals(otherPair.second));
48 }
49 }
50 }
```

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-49

## Using Our Ordered Pair Class (Part 1 of 3)

Display 14-6 Using Our Ordered Pair Class

```
1 import java.util.Scanner;
2
3 public class GenericPairDemo
4 {
5 public static void main(String[] args)
6 {
7 Pair<String> secretPair =
8 new Pair<String>("Happy", "Day");
9
10 Scanner keyboard = new Scanner(System.in);
11 System.out.println("Enter two words:");
12 String word1 = keyboard.next();
13 String word2 = keyboard.next();
14 Pair<String> inputPair =
15 new Pair<String>(word1, word2);
```

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-50

## Using Our Ordered Pair Class (Part 2 of 3)

Display 14-6 Using Our Ordered Pair Class

```
15 if (inputPair.equals(secretPair))
16 {
17 System.out.println("You guessed the secret words");
18 System.out.println("in the correct order!");
19 }
20 else
21 {
22 System.out.println("You guessed incorrectly.");
23 System.out.println("You guessed");
24 System.out.println(inputPair);
25 System.out.println("The secret words are");
26 System.out.println(secretPair);
27 }
28 }
29 }
```

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-51

## 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

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-52

## 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");`

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-53

## 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

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-54

## 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];`

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-55

## 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

Copyright © 2012 Pearson Addison-Wesley.  
All rights reserved.

14-56

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

Display 14-7 Using Our Ordered Pair Class and Automatic Boxing

```
1 import java.util.Scanner;
2 public class GenericPairDemo2
3 {
4 public static void main(String[] args)
5 {
6 Pair<Integer> secretPair =
7 new Pair<Integer>(42, 24); //Automatic boxing allows you to
8 Scanner keyboard = new Scanner(System.in); use an int argument for an
9 System.out.println("Enter two numbers:"); Integer parameter.
10 int n1 = keyboard.nextInt();
11 int n2 = keyboard.nextInt();
12 Pair<Integer> inputPair =
13 new Pair<Integer>(n1, n2); //Automatic boxing allows you to
14
```

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-57

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

Display 14-7 Using Our Ordered Pair Class and Automatic Boxing

```
15 if (inputPair.equals(secretPair))
16 {
17 System.out.println("You guessed the secret numbers");
18 System.out.println("in the correct order!");
19 }
20 else
21 {
22 System.out.println("You guessed incorrectly.");
23 System.out.println("You guessed");
24 System.out.println(inputPair);
25 System.out.println("The secret numbers are");
26 System.out.println(secretPair);
27 }
28 }
29 }
```

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-58

## 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!
```

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-59

## 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

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-60

## Multiple Type Parameters (Part 1 of 4)

Display 14.8 Multiple Type Parameters

```
1 public class TwoTypePair<T1, T2>
2 {
3 private T1 first;
4 private T2 second;
5
6 public TwoTypePair()
7 {
8 first = null;
9 second = null;
10 }
11
12 public TwoTypePair(T1 firstItem, T2 secondItem)
13 {
14 first = firstItem;
15 second = secondItem;
16 }
17
18 public T1 getFirst()
19 {
20 return first;
21 }
22
23 public T2 getSecond()
24 {
25 return second;
26 }
27
28 public String toString()
29 {
30 return "first: " + first.toString() + "\n"
31 + "second: " + second.toString();
32 }
33
34}
```

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-61

## Multiple Type Parameters (Part 2 of 4)

Display 14.8 Multiple Type Parameters

```
15 public void setFirst(T1 newFirst)
16 {
17 first = newFirst;
18 }
19
20 public void setSecond(T2 newSecond)
21 {
22 second = newSecond;
23 }
24
25 public T1 getFirst()
26 {
27 return first;
28 }
29
30 public T2 getSecond()
31 {
32 return second;
33 }
34
35 public boolean equals(Object otherObject)
36 {
37 if (otherObject == null)
38 return false;
39 else if (getClass() != otherObject.getClass())
40 return false;
41 else
42 {
43 TwoTypePair<T1, T2> otherPair =
44 (TwoTypePair<T1, T2>)otherObject;
45 return (first.equals(otherPair.first)
46 && second.equals(otherPair.second));
47 }
48 }
49
50
51 }
```

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-62

## Multiple Type Parameters (Part 3 of 4)

Display 14.8 Multiple Type Parameters

```
27 public T2 getSecond()
28 {
29 return second;
30 }
31
32 public String toString()
33 {
34 return ("first: " + first.toString() + "\n"
35 + "second: " + second.toString());
36 }
37
38 public boolean equals(Object otherObject)
39 {
40 if (otherObject == null)
41 return false;
42 else if (getClass() != otherObject.getClass())
43 return false;
44 else
45 {
46 TwoTypePair<T1, T2> otherPair =
47 (TwoTypePair<T1, T2>)otherObject;
48 return (first.equals(otherPair.first)
49 && second.equals(otherPair.second));
50 }
51 }
52
53
54 }
```

(continued)

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-63

## Multiple Type Parameters (Part 4 of 4)

Display 14.8 Multiple Type Parameters

```
37 public boolean equals(Object otherObject)
38 {
39 if (otherObject == null)
40 return false;
41 else if (getClass() != otherObject.getClass())
42 return false;
43 else
44 {
45 TwoTypePair<T1, T2> otherPair =
46 (TwoTypePair<T1, T2>)otherObject;
47 return (first.equals(otherPair.first)
48 && second.equals(otherPair.second));
49 }
50 }
51
52
53 }
```

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

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

14-64

## 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

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-65

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

Display 14-9 Using a Generic Class with Two Type Parameters

```
1 import java.util.Scanner;
2 public class TwoTypePairDemo
3 {
4     public static void main(String[] args)
5     {
6         TwoTypePair<String, Integer> rating =
7             new TwoTypePair<String, Integer>("The Car Guys", 8);
8
9         Scanner keyboard = new Scanner(System.in);
10        System.out.println(
11            "Our current rating for " + rating.getFirst());
12        System.out.println(" is " + rating.getSecond());
13
14        System.out.println("How would you rate them?");
15        int score = keyboard.nextInt();
16        rating.setSecond(score);
17    }
18}
19
```

(continued)

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-66

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

Display 14-9 Using a Generic Class with Two Type Parameters

```
15     System.out.println(
16         "Our new rating for " + rating.getFirst());
17     System.out.println(" is " + rating.getSecond());
18 }
19 }
```

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
```

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-67

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

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-68

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
- 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>
```

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-69

A Bounded Type Parameter

Display 14.10 A Bounded Type Parameter

```
1 public class Pair<T extends Comparable>
2 {
3     private T first;
4     private T second;
5
6     public T max()
7     {
8         if (first.compareTo(second) <= 0)
9             return first;
10        else
11            return second;
12    }
13 }
```

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

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-70

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

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-71

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

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-72

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);`

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-73

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**
 - This is true regardless of the relationship between class **A** and **B**, e.g., if class **B** is a subclass of class **A**

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-74

A Derived Generic Class (Part 1 of 2)

Display 14.11 A Derived Generic Class

```
1 public class UnorderedPair<T> extends Pair<T>
2 {
3     public UnorderedPair()
4     {
5         setFirst(null);
6         setSecond(null);
7     }
8
9     public UnorderedPair(T firstItem, T secondItem)
10    {
11        setFirst(firstItem);
12        setSecond(secondItem);
13    }
14
15    @Override
16    public String toString()
17    {
18        return "UnorderedPair{" + firstItem + ", " + secondItem + "}";
19    }
20
21    @Override
22    public int hashCode()
23    {
24        return firstItem.hashCode() ^ secondItem.hashCode();
25    }
26
27    @Override
28    public boolean equals(Object otherObject)
29    {
30        if (otherObject == null)
31            return false;
32        else if (getClass() != otherObject.getClass())
33            return false;
34        else
35        {
36            UnorderedPair<T> otherPair =
37                (UnorderedPair<T>)otherObject;
38            return (getFirst().equals(otherPair.getFirst()))
39                  && getSecond().equals(otherPair.getSecond());
40        }
41    }
42}
```

(continued)

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-75

A Derived Generic Class (Part 2 of 2)

Display 14.11 A Derived Generic Class

```
13    @Override
14    public boolean equals(Object otherObject)
15    {
16        if (otherObject == null)
17            return false;
18        else if (getClass() != otherObject.getClass())
19            return false;
20        else
21        {
22            UnorderedPair<T> otherPair =
23                (UnorderedPair<T>)otherObject;
24            return (getFirst().equals(otherPair.getFirst()))
25                  && getSecond().equals(otherPair.getSecond());
26        }
27    }
28}
29}
```

Copyright © 2012 Pearson Addison-Wesley.
All rights reserved.

14-76

Using UnorderedPair (Part 1 of 2)

Display 14.12 Using UnorderedPair

```
1 public class UnorderedPairDemo
2 {
3     public static void main(String[] args)
4     {
5         UnorderedPair<String> p1 =
6             new UnorderedPair<String>("peanuts", "beer");
7         UnorderedPair<String> p2 =
8             new UnorderedPair<String>("beer", "peanuts");
```

(continued)

Using UnorderedPair (Part 2 of 2)

Display 14.12 Using UnorderedPair

```
9     if (p1.equals(p2))
10    {
11        System.out.println(p1.getFirst() + " and " +
12                           p1.getSecond() + " is the same as");
13        System.out.println(p2.getFirst() + " and "
14                           + p2.getSecond());
15    }
16 }
17 }
```

SAMPLE DIALOGUE²

peanuts and beer is the same as
beer and peanuts