Chapter 14
Generics and the ArrayList Class

Prof. Yongsu Park
Dept. of Computer Science and Engineering
Hanyang University

Contents

- The ArrayList Class

- Generics

The ArrayList Class

- Introduction to Generics
- The ArrayList Class
- Using the ArrayList Class
- Methods in the Class ArrayList
- The "For Each" Loop
- The Vector 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);
    ```

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`** (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)

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)

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)

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

Display 14.1 **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)

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

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

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

```java
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.

```java
public void clear()
```

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

(continued)

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

Display 14.1   **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)

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

Display 14.1   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)

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

```java
public int size()
```

Returns the number of elements in the calling ArrayList.

```java
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.

```java
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)

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)

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)

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 `ArrayList`s 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.)

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

```
 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)

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)

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.
```

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

Generics

- Parameterized Classes and Generics
- Generics
- A Class Definition with a Type Parameter
- A Generic Ordered Pair Class
- Pitfall: A Primitive Type Cannot be Plugged in for a Type Parameter
- Multiple Type Parameters
- Using a Generic Class with Two Type Parameters
- Generic Methods

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*

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

A Class Definition with a Type Parameter

```java
1   public class Sample<T>
2   {
3       private T data;

4       public void setData(T newData)
5       {
6           data = newData;
7       }

8       public T getData()
9       {
10          return data;
11      }
12  }
```

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
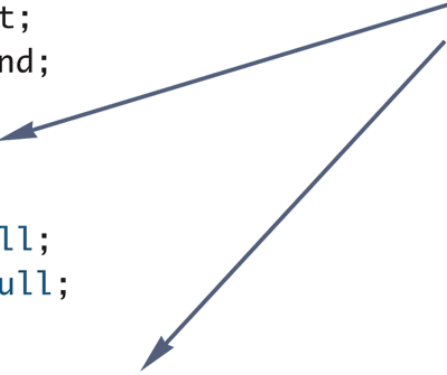
A Generic Ordered Pair Class (Part 1 of 4)

Display 14.5 A Generic Ordered Pair Class

```
1    public class Pair<T>
2    {
3        private T first;
4        private T second;

5        public Pair()
6        {
7            first = null;
8            second = null;
9        }

10       public Pair(T firstItem, T secondItem)
11       {
12           first = firstItem;
13           second = secondItem;
14       }
```

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

```
15      public void setFirst(T newFirst)
16      {
17          first = newFirst;
18      }

19      public void setSecond(T newSecond)
20      {
21          second = newSecond;
22      }

23      public T getFirst()
24      {
25          return first;
26      }
```

(continued)

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      public String toString()
32      {
33          return ( "first: " + first.toString() + "\n"
34                  + "second: " + second.toString() );
35      }
36
```

(continued)

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    }
```

Using Our Ordered Pair Class (Part 1 of 3)

Display 14.6   **Using Our Ordered Pair Class**

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

2   public class GenericPairDemo
3   {
4       public static void main(String[] args)
5       {
6           Pair<String> secretPair =
7               new Pair<String>("Happy", "Day");

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

(continued)

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)

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 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 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**

```
1    public class TwoTypePair<T1, T2>
2    {
3        private T1 first;
4        private T2 second;

5        public TwoTypePair()
6        {
7            first = null;
8            second = null;
9        }

10       public TwoTypePair(T1 firstItem, T2 secondItem)
11       {
12           first = firstItem;
13           second = secondItem;
14       }
```

(continued)

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      public void setSecond(T2 newSecond)
20      {
21          second = newSecond;
22      }

23      public T1 getFirst()
24      {
25          return first;
26      }
```

(continued)

Display 14.8   **Multiple Type Parameters**

```
27        public T2 getSecond()
28        {
29            return second;
30        }

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

(continued)

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    }
```

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

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

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

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

12           System.out.println("How would you rate them?");
13           int score = keyboard.nextInt();
14           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

```
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
```

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);`

Generic Methods

```
public class Utility
{
    …
    public static <T> T getMidpoint(T[] a)
    {
        return a[a.length/2];
    }

    public static <T> T getFirst(T[] a)
    {
        return a[0];
    }
    …
}


String midString = Utility.<String>getMidpoint(b);
double firstNumber = Utility.<Double>getFirst(c);
```