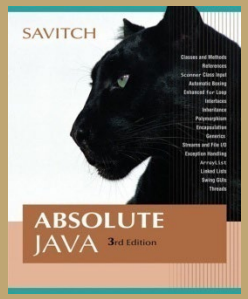
Comp 249
Programming Methodology
Chapter 13
Generics & The ArrayList Class

Dr. Aiman Hanna
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of Absolute Java 3rd Edition by Savitch; which has originally been prepared by Rose Williams of Binghamton University. Absolute Java is published by Pearson Education / Addison-Wesley.



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*

See ArrayList1.java

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`**

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

See ArrayList2.java

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)

Display 14.1 Some Methods in the Class `ArrayList`

```
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.

```
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)

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)

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)

Conversion between ArrayList and arrays

- Conversion from ArrayList to arrays is possible

- Two methods exists:

  `Object[] toArray(),`

  and

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

See ArrayList3.java

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 `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.)

See ArrayList7.java

See ArrayList8.java

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`**)

See ArrayList4.java

Passing ArrayList as Method Parameters

- An **`ArrayList`** can be passed as a parameter to a method in a similar fashion to other types

See ArrayList5.java

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, i.e. T, 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> s1 =
    new Sample<String>();
```

See Generic1.java

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      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

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

```
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

```
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");
8
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 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; i.e. the constructor is not defined as follows:

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

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

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

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

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);
8
9           Scanner keyboard = new Scanner(System.in);
10          System.out.println("Enter two numbers:");
11          int n1 = keyboard.nextInt();
12          int n2 = keyboard.nextInt();
13          Pair<Integer> inputPair =
14              new Pair<Integer>(n1, n2);
```

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

(continued)

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)

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

Example of 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)

Example of 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)

Example of Multiple Type Parameters (Part 3 of 4)

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)

Example of 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`*.*

- See Generic2.java
- See Generic3.java
- See Generic4.java

Notice Carefully

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           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
```

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

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>
```

    - to ensure that only classes that are descendent of the **Vehicle** class are plugged in for **T**, define a class as follows:

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

  - "**extends**" serves as a *bound* on the type parameter **T**
  - Any attempt to plug in a type for **T** which does not follow the restriction would result in a compiler error
    - Notice that "**extends**" is used for both classes and interfaces (i.e. it is **not**

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

Bounds for 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>
```

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       public T max()
6       {
7           if (first.compareTo(second) <= 0)
8               return first;
9           else
10              return second;
11      }

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

12  }
```

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);
```

See Generic5.java

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**

See Generic6.java

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      public UnorderedPair(T firstItem, T secondItem)
9      {
10         setFirst(firstItem);
11         setSecond(secondItem);
12     }
```

(continued)

A Derived Generic Class (Part 2 of 2)

Display 14.11 A Derived Generic Class

```
13      public boolean equals(Object otherObject)
14      {
15          if (otherObject == null)
16              return false;
17          else if (getClass() != otherObject.getClass())
18              return false;
19          else
20          {
21              UnorderedPair<T> otherPair =
22                                  (UnorderedPair<T>)otherObject;
23              return (getFirst().equals(otherPair.getFirst())
24                 && getSecond().equals(otherPair.getSecond()))
25                 ||
26                     (getFirst().equals(otherPair.getSecond())
27                 && getSecond().equals(otherPair.getFirst()));
28          }
29      }
30  }
```

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 DIALOGUE[2]

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

Pitfall: Restrictions with Generics 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: Pitfall: Restrictions with Generics 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 arrays and generic classes differently; for instance,
    - Java, by design, requires arrays at run-time to include information about their contents;
    - This conflicts with Java design of generics, where the generic type is not known at run-time, which directly contradicts the requirements of arrays!

See Generic7.java

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

- However, it is possible that the parameter passed to the generic class is by itself an object of that class
  - For instance: **ArrayList<ArrayList<Car>> aTable = new ArrayList<ArrayList<Car>>(3);**

See ArrayList6.java

Generic Wildcards

- Wildcards are used to cast a collection of a class to:
  - A collection of a subclass; or
  - A collection of a superclass
- **The Unknown Wildcard:**

  List<?> : The list is typed/related to an unknown object
- **The *extends* Wildcard – (Covariance)**

  List<? extends A>
- **The *super* Wildcard –(Contravariance)**

  List<? super A>

Generic Wildcards

- The *extends* Wildcard – (Covariance)

List<? extends A>

? extends T means the generic type can be **T or any subtype of T**.
It is **read-only** in terms of type safety: you can safely read elements from the collection as T (or its supertype).
You **cannot add elements**, because the exact subtype is unknown to the compiler.
Usage:
Use ? extends when you want to **read from a collection** but do not need to modify it.

- The *super* Wildcard –(Contravariance)

List<? super A>

? super T means the generic type can be **T or any supertype of T**.
It is **write-friendly**: you can add elements of type T or its subtypes.
You can only read elements as Object, because the specific type of elements in the collection is unknown to the compiler.
Usage:
Use ? super when you want to **write to a collection** but don't need to read specific types from it.