Chapter 14

Generic Class

What Are Generics?

- Generics abstract over Types.
- Classes, Interfaces and Methods can be Parameterized by Types.
- Generics provide increased readability and type safety.

Generics (Cont'd)

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

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

A Class Definition with a Type Parameter

Display 14.4 A Class Definition with a Type Parameter

```
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.

A Class Definition with a Type Parameter (Cont'd)

- 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.

Generic Class Definition: An Example

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     }
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     }
```

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

(continued)

Generic Class Definition: An Example (Cont'd)

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
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  }
```

Generic Class Usage: An Example

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);

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 }
```

Generic Class Usage: An Example (Cont'd)

Program Output:

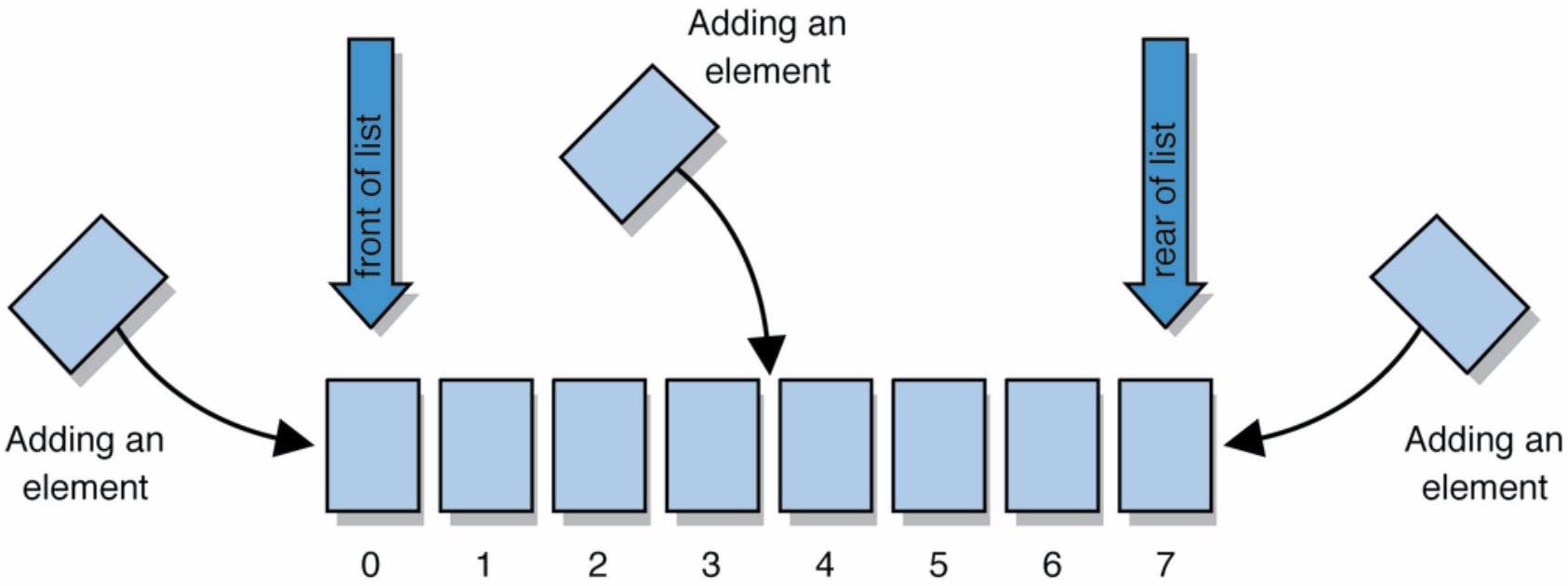
Display 14.6 **Using Our Ordered Pair Class**

SAMPLE DIALOGUE

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

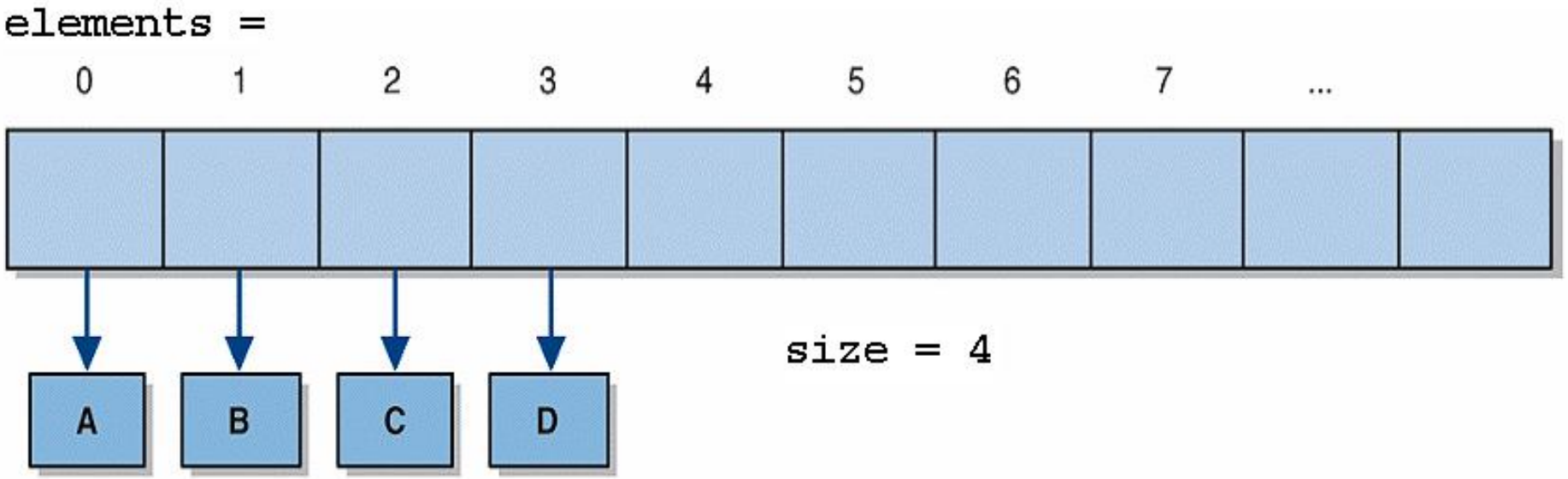
Lists

- **list**: an ordered sequence of elements, each accessible by a 0-based index (array index started with 0)
  - one of the most basic collections of data



The `ArrayList` class

- Class `ArrayList<E>` implements the notion of a list using a partially-filled array
  - when you want to use `ArrayList`, remember to `import java.util.*;`



`ArrayList` features

- Think of it as an auto-resizing array that can hold any type of object, with many convenient methods
- Maintains most of the benefits of arrays, such as fast random access
- Frees us from some tedious operations on arrays, such as sliding elements and resizing
- Can call `toString` on an `ArrayList` to print its elements
  - `[1, 2.65, Marty Stepp, Hello]`

Generic classes

- `ArrayList<E>` is a generic class.
  - The `<E>` is a placeholder in which you write the type of elements you want to store in the `ArrayList`.
- Example:

```
ArrayList<String> words = new ArrayList<String>();
```

  - Now the methods of `words` will manipulate and return `String`s.

ArrayList vs. array

- array

```
String[] names = new String[5];
names[0] = "Jennifer";
String name = names[0];
```

- **ArrayList**

```
ArrayList<String> namesList = new
  ArrayList<String>();
namesList.add("Jennifer");
String name = namesList.get(0);
```

Adding elements

- Elements are added dynamically to the list:

```
ArrayList<String> list = new ArrayList<String>();
System.out.println("list = " + list);
list.add("Tool");
System.out.println("list = " + list);
list.add("Phish");
System.out.println("list = " + list);
list.add("Pink Floyd");
System.out.println("list = " + list);
```

- Output:

```
list = []
list = [Tool]
list = [Tool, Phish]
list = [Tool, Phish, Pink Floyd]
```

Removing elements

- Elements can also be removed by index:

```
System.out.println("before remove list = " + list);
list.remove(0);
list.remove(1);
System.out.println("after remove list = " + list);
```

- Output:

```
before remove list = [Tool, U2, Phish, Pink Floyd]
after remove list = [U2, Pink Floyd]
```

  - Notice that as each element is removed, the others shift downward in position to fill the hole.
  - Therefore, the second `remove` gets rid of Phish, not U2.

index	*0*	*1*
value	U2	Pink Floyd

Searching for elements

- You can search the list for particular elements:

```
if (list.contains("Phish")) {
    int index = list.indexOf("Phish");
    System.out.println(index + " " +
  list.get(index));
}
if (list.contains("Madonna")) {
    System.out.println("Madonna is in the list");
} else {
    System.out.println("Madonna is not found.");
}
```

- Output:

```
2 Phish
Madonna is not found.
```

- `contains` tells you whether an element is in the list or not, and `indexOf` tells you at which index you can find it.

ArrayList methods

Method name	Description
`add(`*value*`)`	adds the given value to the end of the list
`add(`*index*`, `*value*`)`	inserts the given value before the given index
`clear()`	removes all elements
`contains(`*value*`)`	returns `true` if the given element is in the list
`get(`*index*`)`	returns the value at the given index
`indexOf(`*value*`)`	returns the first index at which the given element appears in the list (or -1 if not found)
`lastIndexOf(`*value*`)`	returns the last index at which the given element appears in the list (or -1 if not found)
`remove(`*index*`)`	removes value at given index, sliding others back
`size()`	returns the number of elements in the list

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");
```

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. *Autoboxing* is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on.
  - **Note:** Reference types can include arrays.

Simple Example of Autoboxing

Autoboxing converts primitive data types into their corresponding wrapper classes.

```
class BoxingExample1
{
        public static void main(String args[])
        {
                int a=50;
                Integer a2=new Integer(a);//Boxing
                Integer a3=5;//Boxing
                System.out.println(a2+" "+a3);
        }
}
```

Simple Example of Unboxing

Unboxing in Java refers to the process of converting a wrapper class object (an object that encapsulates a primitive data type) into its corresponding primitive data type.

```
class UnboxingExample1
{
        public static void main(String args[])
        {
                Integer i=new Integer(50);
                int a=i;
                System.out.println(a);
        }
}
```

Limitations on Type Parameter Usage

- 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];
```

Limitations on Generic Class Instantiation

- Arrays such as the following are illegal:

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

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

Using Generic Classes and Automatic Boxing

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);

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  }
```

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

Using Generic Classes and Automatic Boxing (Cont'd)

Program Output:

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!
```

Multiple Type Parameters

- 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 (Cont'd)

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      }
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)

Multiple Type Parameters (Cont'd)

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
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

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

Program Output:

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
```

A Generic Classes and Exceptions

- 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>

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

Bounds for Type Parameters (Cont'd)

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

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

- A bounds expression may contain multiple interfaces and up to one class.

- If there is more than one type parameter, the syntax is as follows:

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

Bounds for Type Parameters (Cont'd)

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  }
```

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 (Cont'd)

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

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

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

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

Inheritance with Generic Classes

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

A Derived Generic Class: An Example

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     }
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 }
```

A Derived Generic Class: An Example (Cont'd)

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");
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 }
```

Program Output:

SAMPLE DIALOGUE

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

End of Chapter 14

Questions?