Chapter 7 Generics

What is Generics?

- **Genrics** is one of the core feature of Java programming and it was introduced in Java 5

- *Generics* is the capability to parameterize types. With this capability, you can define a class or a method with generic types that can be substituted using *concrete types* by the compiler.

Example

- ☞ A generic stack class that stores the elements of a *generic type*. From this generic class, you may create a stack object for holding <u>strings</u> and a stack object for holding <u>numbers</u>. Here, strings and numbers are *concrete types* that replace the generic type.

Why Generics?

- The key benefit of generics is to enable errors to be detected at *compile time* rather than at *runtime*.
- A generic class or method permits you to specify allowable types of objects that the class or method may work with. If you attempt to use the class or method with an incompatible object, a compile error occurs.
- Improve software reliability and readability.

- ☞ Generic programming **mostly** involves the design and implementation of data structures (e.g. ArrayList) and algorithms that work for multiple types.

`ArrayList`s, then and now

- Starting in Java 5, `ArrayList`s have been generics generalized
  - That means, every place you used to say `ArrayList`, you now have to say what kind of objects it holds; like this: `ArrayList<String>`
  - If you don’t do this, you will get a warning message, but your program will still run

`ArrayList`s and arrays

- A `ArrayList` is like an array of `Object`s, but...
  - Arrays use `[ ]` syntax; `ArrayList`s use object syntax
  - An `ArrayList` expands as you add things to it
  - Arrays can hold primitives or objects, but `ArrayList`s can only hold objects
- To create an ArrayList:
  - `ArrayList myList = new ArrayList();`
  - Or, since an `ArrayList` is a kind of `List`,
    `List myList = new ArrayList();`
- To use an ArrayList,
  - `boolean add(Object` ***obj***`)`
  - `Object set(int` ***index***`, Object` ***obj***`)`
  - `Object get(int` ***index***`)`

Auto boxing and unboxing

- Java won't let you use a primitive value where an object is required--you need a "wrapper"
  - `ArrayList myList = new ArrayList();`
  - `myList.add(new Integer(5));`
- Similarly, you can't use an object where a primitive is required--you need to "unwrap" it
  - `int n = ((Integer) myList.get(2)).intValue();`
- Java 1.5 makes this automatic:

  -
    ```
    myArrayList<Integer> myList = new myArrayList<Integer>();
    myList.add(5);
    int n = myList.get(2);
    ```
- Other extensions make this as transparent as possible
  - For example, control statements that previously required a `boolean` (`if`, `while`, `do-while`) can now take a `Boolean`
  - There are some subtle issues with equality tests, though

Generics

- A generic is a method that is recompiled with different types as the need arises
- The bad news:
  - Instead of saying: `List words = new ArrayList();`
  - You'll have to say:
    ```
    List<String> words = new ArrayList<String>();
    ```
- The good news:
  - Replaces runtime type checks with compile-time checks
  - No casting; instead of
    ```
    String title = (String) words.get(i);
    ```
    you use
    ```
    String title = words.get(i);
    ```
- Some classes and interfaces that have been "genericized" are: `Vector`, `ArrayList`, `LinkedList`, `Hashtable`, `HashMap`, `Stack`, `Queue`, `PriorityQueue`, `Dictionary`, `TreeMap` and `TreeSet`

Generic Iterators

- An Iterator is an object that will let you step through the elements of a list one at a time

  -
    ```
    List<String> listOfStrings = new ArrayList<String>();
    ...
    for (Iterator i = listOfStrings.iterator(); i.hasNext(); ) {
        String s = (String) i.next();
        System.out.println(s);
    }
    ```
- Iterators have also been genericized:

  -
    ```
    List<String> listOfStrings = new ArrayList<String>();
    ...
    for (Iterator<String> i = listOfStrings.iterator();
    i.hasNext(); ) {
        String s = i.next();
        System.out.println(s);
    }
    ```
- If a class implements `Iterable`, you can use the new for loop to iterate through all its objects

Writing generic methods

-
  ```
  private void printListOfStrings(List<String> list) {
      for (Iterator<String> i = list.iterator();
  i.hasNext(); ) {
          System.out.println(i.next());
      }
  }
  ```
- This method *should* be called with a parameter of type `List<String>`, but it *can* be called with a parameter of type `List`
  - The disadvantage is that the compiler won't catch errors; instead, errors will cause a `ClassCastException`
  - This is necessary for backward compatibility
  - Similarly, the Iterator need not be genericized as an `Iterator<String>`

Type wildcards

- Here’s a simple (no generics) method to print out any list:

  -
    ```
    private void printList(List list) {
        for (Iterator i = list.iterator(); i.hasNext(); ) {
            System.out.println(i.next());
        }
    }
    ```
- The above still works in Java 1.5, but generates warning messages
  - Java 1.5 incorporates `lint` (like C `lint`) to look for possible problems
- You should eliminate *all* errors and warnings in your final code, so you need to *tell* Java that any type is acceptable:

  -
    ```
    private void printListOfStrings(List<?> list) {
        for (Iterator<?> i = list.iterator(); i.hasNext(); ) {
            System.out.println(i.next());
        }
    }
    ```

Writing your own generic types

-
  ```
  public class Box<T> {
      private List<T> contents;

      public Box() {
          contents = new ArrayList<T>();
      }

      public void add(T thing) { contents.add(thing); }

      public T grab() {
          if (contents.size() > 0) return
  contents.remove(0);
          else return null;
  }
  ```
- Sun's recommendation is to use single capital letters (such as `T`) for types
- If you have more than a couple generic types, though, you should use better names

New `for` statement with arrays

- The new `for` statement can also be used with arrays
- Instead of

  ```
  for (int i = 0; i < array.length; i++) {
      System.out.println(array[i]);
  }
  ```

  you can say (assuming `array` is an `int` array):

  ```
  for (int value : array) {
      System.out.println(value);
  }
  ```
- Disadvantage: You don’t know the index of any of your values

Creating a `ArrayList` the old way

- The syntax for creating `ArrayList`s has *changed* between Java 1.4 and Java 5
- For compatibility reasons, the old way still works, but will give you warning messages
- Here are the (old) constructors:
  - `import java.util.ArrayList;`
  - `ArrayList vec1 = new ArrayList();`
    - Constructs an ArrayList with an initial capacity of 10
  - `ArrayList vec2 = new ArrayList(`***initialCapacity***`);`

Creating a `ArrayList` the new way

- Specify, in angle brackets after the name, the type of object that the class will hold
- Examples:
  - `ArrayList<String> vec1 = new ArrayList<String>();`
  - `ArrayList<String> vec2 = new ArrayList<String>(10);`

- To get the old behavior, but without the warning messages, use the `<?>` wildcard
  - Example: `ArrayList<?> vec1 = new ArrayList<?>();`

Accessing with and without generics

- `Object get(int index)`
  - Returns the component at position ***index***
- Using `get` the old way:

  -
    ```
    ArrayList myList = new ArrayList();
    myList.add("Some string");
    String s = (String)myList.get(0);
    ```
- Using `get` the new way:

  -
    ```
    ArrayList<String> myList = new ArrayList<String>();
    myList.add("Some string");
    String s = myList.get(0);
    ```
- Notice that casting is no longer necessary when we retrieve an element from a “genericized” `ArrayList`

Summary

- If you think of a genericized type as a ***type***, you won't go far wrong
  - Use it wherever a type would be used
  - `ArrayList myList` becomes `ArrayList<String> myList`
  - `new ArrayList()` becomes `new ArrayList<String>()`
  - `public ArrayList reverse(ArrayList list)` becomes `public ArrayList<String> reverse(ArrayList<String> list`)
- Advantage: Instead of having collections of "Objects", you can control the type of object
- Disadvantage: more complex, more typing

The End

- For more information
  - Tutorial: Generics in the Java Programming Language
    - http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

  - Generics
    - http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html

just another example

Introduction

- Prior to the JDK 5.0 release, when you created a Collection, you could put any object in it.

```
List myList = new ArrayList(10);
myList.add(new Integer(10));
myList.add("Hello, World");
```

- Getting items out of the collection required you to use a casting operation:

```
Integer myInt = (Integer)myList.iterator().next();
```

Introduction (cont'd)

- If you accidentally cast the wrong type, the program would successfully compile, but an exception would be thrown at runtime.
- Use instanceof to avoid a blind cast

```
Iterator listItor = myList.iterator();
Object myObject = listItor.next();
Integer myInt = null;
if (myObject instanceof Integer) {
        myInt = (Integer)myObject;
}
```

Generics

- J2SE 5.0 provides *compile-time* **type safety** with the Java Collections framework through ***generics***
- Generics allows you to specify, at compile-time, the types of objects you want to store in a Collection. Then when you add and get items from the list, **the list already knows what types of objects are supposed to be acted on**.
- So you don't need to cast anything. The "<>" characters are used to designate what type is to be stored. If the wrong type of data is provided, a compile-time exception is thrown.

Generics (cont'd)

- Example:

```
import java.util.*;
public class First {
    public static void main(String args[]) {
      List<Integer> myList = new ArrayList<Integer>(10);
      myList.add(10);  // OK ???
      myList.add("Hello, World"); // OK ???
    }
}
```

Generics (cont’d)

- Example (compile-time exception):

```
import java.util.*;
public class First {
      public static void main(String args[]) {
         List<Integer> myList = new ArrayList<Integer>(10);
         myList.add(10);  // 🡸 Autoboxing converted the int type to an Integer
         myList.add("Hello, World");
      }
}

First.java:7: cannot find symbol symbol :
      method add(java.lang.String)
      location: interface java.util.List<java.lang.Integer>
      myList.add("Hello, World");
                  ^ 1 error
```

Generics (cont’d)

- Consider this:

```
import java.util.*;
public class Second {
     public static void main(String args[]) {
        List list = new ArrayList();
        list.add(10); // 🡐 the "E" in the "add(E)" call, was never specified.
     }
}
```

- When you compile the program, you get the following warning:

```
Note: Second.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Generics (cont’d)

- If you want to get rid of the warning, you can add <Object> to the List declaration, as follows:

```
import java.util.*;
public class Second {
    public static void main(String args[]) {
      List<Object> list = new ArrayList<Object>();
      list.add(10);
    }
}
```

- Here, Object is the E, and basically says that anything can be stored in the List.

Generics (cont'd)

- Another example:

```
// old way of looping through a List of String objects
// without generics and an enhanced for loop
import java.util.*;
public class Old {
      public static void main(String args[]) {
         List list = Arrays.asList(args);
         Iterator itor = list.iterator();
         while (itor.hasNext()) {
                     String element = (String)itor.next();
                     System.out.println(element + " / " + element.length());
         } } }
```

- If you compile and run the Old class and then run it with a string, like this:
- java Old Hello
- you get: Hello / 5

Generics (cont'd)

- Another example:

```
// old way of looping through a List of String objects
// without generics and an enhanced for loop
import java.util.*;
public class New {
        public static void main(String args[]) {
           List<String> list = Arrays.asList(args);
           for (String element : list) {
              System.out.println(element + " / " + element.length());
           }
        }
  }
```

```
java New Hello
Hello / 5
```