# Generics

## Typecasting

In versions of JDK prior to 1.5, retrieving an item from a data structure required that you typecast the item from `Object` to whatever class the item was. Typecasting was required because in order for data structures to be useful they must accept all objects; hence they can only return instances of the `Object` class. For example:

```
// create an instance of class Student
Student s = new Student();

// add student s to an ArrayList, aList
aList.add(s);

// incorrectly try to retrieve the first item in aList
Student retrievedStudentWrong = aList.get(0);

// correctly retrieve the first item in aList, by typecasting to Student
Student retrievedStudent = (Student)aList.get(0);
```

The method `get(int index)` of `ArrayList` has a return type of `Object`. The first attempt to retrieve the first item from `aList` in the above code tries to store an `Object` in a variable of type `Student`, which causes a `ClassCastException`. To solve this problem, we have to explicitly cast (sometimes called "co-erce") the `Object` returned by the get method to the proper type — in this case, `Student`. This is shown above in the last line of code.

Now, we could have avoided the need for typecasting by simply declaring the variable `s` as `Object` instead of `Student`. However, by so doing `s` would only be considered as being type `Object`, and we would thus only be allowed to invoke the methods found in the `Object` class. In some cases that may be all we need. For example, if all we wanted to do was print out the results of the `toString()` method, it would be perfectly fine to leave `s` as an `Object`. But if we needed to call a method specific for the `Student` class, we would be out of luck unless we typecast `s` as `Student`.

## Generics

While typecasting is not difficult to do, it can be tedious, and it does increase the chances of introducing faults into code. To help alleviate this problem, Sun introduced generics into JDK version 1.5. Using generics eliminates the need for explicit typecasting. Instead, the Java runtime does it automatically. It doesn't come without a price, though. We have to modify the class and method signatures to enable the Java runtime to recognize when to automatically typecast. An abbreviated version of the `ArrayList` class is shown on the next page to illustrate these changes.

```
public class ArrayList<E> implements List<E>
{

      /** array to store items */
      private E[] theItems;


      /**
       * double the size of the underlying array
       */
      private void doubleArray()
      {
            // backup original array
            E[] old = theItems;

            // create new array
            theItems = (E[])new Object[theItems.length * 2 + 1];

            // copy items from old array to new array
            for(int i = 0; i < theSize; i++)
            {
                  theItems[i] = old[i];
            }
      }


      /**
       * return the item at the specified position
       *
       * throws ArrayIndexOutOfBoundsException if position is invalid
       */
      public E get(int index)
      {
            // throw exception if invalid index
            if(index < 0 || index >= theSize)
            {
                  throw new ArrayIndexOutOfBoundsException("Index " + index + ";
                        size " + theSize);
            }

            // return the item at index
            return theItems[index];
      }
}
```

Note that the token "`<E>`" has been appended to the class name `ArrayList` and the interface name `List`. This signifies that a given instance of `ArrayList` may only contain elements of a single type. Any type will do, but all the elements must be of that type. The "`E`" serves as a placeholder for whatever class type is chosen, which will be determined when the `ArrayList` instance is created. For example, the statement:

```
ArrayList<Student> studentList = new ArrayList<Student>();
```

creates an instance of `ArrayList` that may only contain elements of type `Student`. Retrieving an item from this list can be done with a statement like:

```
Student s = studentList.get(0);
```

Note the need for typecasting has been eliminated. In fact, depending on the IDE you use to write your Java code, the IDE may notify you if you should attempt to add an element of the incorrect type, and may even prompt you with the correct argument types as you enter the code. Eclipse will do this for you.

The problems with using generics stem from the fact that you have to make sure you use the placeholder in all the proper locations in your code. For example, the get method of `ArrayList` must return a value of type `E`, not `Object` as before. Additionally, Java does not yet support automatic typecasting for simple arrays, so arrays must still be typecast. See the code for the `doubleArray` method for an example of this.

By the way, you can use any letter you like as a placeholder. You can even use entire alphanumeric strings of characters if you want. It is the angle brackets that define a generic, not the contents of the angle brackets. You just have to make sure that whatever you use for a placeholder is used consistently throughout a class. The letter `E` is used by convention to mean "element". Other letters are used for different data structures. For hashtables, the letters `K` and `V` are used to stand for "key" and "value", respectively. The letter `T` is used on occasion, usually when arrays are involved. For cases where the placeholder itself can be variable, the questions mark (`?`) is used as a wildcard placeholder.

Section 11.2 in the textbook provides a good example of a rather complicated method declaration using generics:

```
public static <T extends Comparable<? super T>> void sort(T[] a, int n){}
```

This is a static method for sorting any array of objects that extend some superclass `T`, and implement the `Comparable` interface. In this method, the return type is a generic, `T`, which extends the `Comparable` interface. In order to allow this method to work for all subclasses of `T`, as well as `T` itself, we use the wildcard after `Comparable` to declare that the method should accept any class which has `T` as its superclass (i.e., any class that extends `T`).