

Chapter 13

Generic Collections

Goals

- Introduce class `Object` and inheritance
 - Show how one collection class can store any type of element using `Object []`
-

13.1 A Generic Collection with `Object []`

The `StringBag` class shown earlier is a collection type that could only store one type of element. If you wanted to store a collection of `Integer` objects you would need a new type perhaps named `IntegerBag`. Rather than implementing a new class for each type, Java provides two approaches to allow for just one collection class to store any type of elements:

1. Store references to `Object` objects rather than just `String` (this section)
2. Use Java generics with a type parameter like `<E>` (in a later section of this chapter)

We'll consider the first option now, which requires knowledge of Java's `Object` class, inheritance, and casting.

Java's `Object` class has one constructor (no arguments) and 11 methods, including `equals` and `toString`. All classes in Java extend the `Object` class or another class that extends `Object`. There is no exception to this. All classes inherit the methods of the `Object` class.

One class inherits the methods and instance variables of another class with the keyword `extends`. For example, the following class heading explicitly states that the `EmptyClass` class inherits all of the method of class `Object`. If a class heading does not have `extends`, the compiler automatically adds `extends Object`.

```
// This class extends Object even if extends Object is omitted
public class EmptyClass extends Object {
}

```

Even though this `EmptyClass` defines no methods, a programmer can construct and send messages to `EmptyClass` objects. This is possible because a class that extends `Object` inherits (obtains) `Object`'s methods. A class that extends another is called a subclass. Here are three of the methods that `EmptyClass` inherits from the `Object` class:

Two Methods of the `Object` Class

- `toString` returns a `String` that is the class name concatenated with the at symbol (`@`) and a hexadecimal (base 16) number related to locating the object at runtime.
- `equals` returns `true` if both the receiver and argument reference the same object.

Additionally, a class that does not declare a constructor is given a default constructor. This is to ensure that constructor for `Object` gets invoked. The following code is equivalent to that shown above

```
// This class extends Object implicitly
public class EmptyClass {

    public EmptyClass() {
        super();    // Explicitly call the constructor of the superclass, which is Object
    }
}
```

This following code shows these two methods used by a class that extends class `Object`.

```
EmptyClass one = new EmptyClass();
EmptyClass two = new EmptyClass();

assertFalse(one.equals(two)); // passes
System.out.println(two.toString());
System.out.println(one.toString());

one = two;    // The variable two will now reference the same object as one
assertTrue(one.equals(two));
System.out.println("after assignment->");
System.out.println(two.toString());
System.out.println(one.toString());
```

Output

```
EmptyClass@8813f2
EmptyClass@1d58aae
after assignment->
EmptyClass@8813f2
EmptyClass@8813f2
```

The `Object` class captures methods that are common to all Java objects. Java makes sure that all classes extend the `Object` class because there are several things that all objects must be capable of in order to work with Java's runtime system. For example, `Object`'s constructor gets invoked for every object construction to help allocate computer memory for the object at runtime. The class also has methods to allow different processes to run at the same time, allowing applications such as Web browsers to be more efficient. Java programs can download several files while browsing elsewhere, while creating another image, and so on.

One-way Assignment Compatibility

Because all classes extend Java's `Object` class, a reference to any type of object can be assigned to an `Object` reference variable. For example, consider the following valid code that assigns a `String` object and an `EmptyClass` object to two different reference variables of type `Object`:

```
String aString = new String("first");
// Assign a String reference to an Object reference
Object obj1 = aString;

EmptyClass one = new EmptyClass();
// Assign an EmptyClass reference to an Object reference
Object obj2 = one;
```

Java's one-way assignment compatibility means that you can assign a reference variable to the class that it extends. However, you cannot directly assign in the other direction. For example, an `Object` reference cannot be directly assigned to a `String` reference.

```
Object obj = new Object();
String str = obj;
Type mismatch: cannot convert from Object to String
```

This compile time error occurs because the compiler recognizes that `obj` is a reference to an `Object` that cannot be assigned down to a `String` reference variable.

This `GenericList` class uses `Object` parameters in the `add` and `remove` methods, an `Object` return type in the `get` method, and an array of `Objects` as the instance variable to store any type element that can be assigned to `Object` (which is any Java type). The `SimpleList` interface was introduced at the end of the previous chapter.

```
public class GenericList implements SimpleList {

    private Object[] elements;
    private int n;

    // Construct an empty list
    public GenericList() {
        elements = new Object[10];
    }

    // Provide access to the number of meaningful list elements
    public int size() {
        return n;
    }

    // Add an element to the end of this list
    public void add(Object elementToAdd) {
        elements[n] = elementToAdd;
        n++;
    }

    // Retrieve a reference to an element
    public Object get(int index) {
        return elements[index];
    }
}
```

This design allows one class to store collections with any type of elements:

```
@Test
public void testGenericity() {
    GenericList names = new GenericList();
    names.add("Kim");
    names.add("Devon");

    GenericList accounts = new GenericList();
    accounts.add(new BankAccount("Speilberg", 1942));

    GenericList numbers = new GenericList();
    numbers.add(12.3);
    numbers.add(4.56);
}
```

In such a class, a method that returns a value would have to return an `Object` reference.

```
public Object get(int index) {
    return elements[index];
}
```

This approach requires a cast. You have to know the type stored.

```
@Test
public void testGet() {
    GenericList strings = new GenericList();
    strings.add("A");
    strings.add("B");
    String firstElement = (String) strings.get(0);
    assertEquals("A", firstElement);
}
```

With this approach, programmers always have to cast, something Java software developers had been complaining about for years (before Java 5). With this approach, you also have to be wary of runtime exceptions. For example, even though the following code compiles, when the test runs, a runtime error occurs.

```
@Test
public void testGetWithLoop() {
    GenericList strings = new GenericList();
    strings.add("A");
    strings.add("B");
    // Using Object objects for a generic collection is NOT type safe.
    // Any type of object can be added accidentally (not usually desirable).
    strings.add(123);

    // The attempt to cast different types of elements fails when index == 2:
    for (int index = 0; index < 3; index++) {
        String theString = (String) strings.get(index);
        System.out.println(theString.toLowerCase());
    }
}
java.lang.ClassCastException: java.util.Integer
```

`strings.get(2)` returns a reference to an integer, which the runtime treats as a `String` in the cast. A `ClassCastException` occurs because a `String` cannot be cast to an integer. In a later section, Java Generics will be shown as a way to have a collection store a specific type. One collection class is all that is needed, but the casting and runtime error will disappear.

Self-Check

13-1 Which statements generate compiletime errors?

```
Object anObject = "3";           // a.
int anInt = anObject;           // b.
String aString = anObject;      // c.
anObject = new Object();        // d.
```

13-2 Which letters represent valid assignment statements that compile?

```
Object obj = "String";          // a.
String str = (String)obj;       // b.
Object obj2 = new Point(3, 4);  // c.
Point p = (String)obj2;         // d.
```

13-3 Which statements generate compile time errors?

```
Object[] elements = new Object[5]; // a.
elements[0] = 12;                  // b.
elements[1] = "Second";            // c.
elements[2] = 4.5;                 // d.
elements[3] = new Point(5, 6);     // e.
```

Collections of Primitive Types

Collections of the primitive types such `int`, `double`, `char` can also be stored in a generic class. The type parameter could be one of Java's "wrapper" classes (or had to be wrapped before Java 5). Java has a "wrapper" class for each primitive type named `Integer`, `Double`, `Character`, `Boolean`, `Long`, `Short`, and `Float`. A wrapper class does little more than allow a primitive value to be viewed as a reference type that can be assigned to an `Object` reference. A `GenericList` of integer values can be stored like this:

```
GenericList tests = new GenericList();
tests.add(new Integer(79));
tests.add(new Integer(88));
```

However, since Java 5, integer values can also be added like this:

```
tests.add(76);
tests.add(100);
```

Java now allows primitive integers to be treated like objects through the process known as autoboxing.

Autoboxing / Unboxing

Before Java 5, to treat primitive types as reference types, programmers were required to "box" primitive values in their respective "wrapper" class. For example, the following code was used to assign an `int` to an `Object` reference.

```
Integer anInt = new Integer(123); // Wrapper class needed for an int
tests.add(anInt);                // to be stored as an Object reference
```

To convert from reference type back to a primitive type, programmers were required to "unbox" by asking the `Integer` object for its `intValue` like this:

```
int primitiveInt = anInt.intValue();
```

Java 5.0 automatically performs this boxing and unboxing.

```
Integer anotherInt = 123;           // autobox 123 as new Integer(123)
int anotherPrimitiveInt = anotherInt; // unboxed automatically
```

This allows primitive literals to be added. The autoboxing occurs automatically when assigning the `int` arguments 79 and 88 to the `Object` parameter of the `add` method.

```
GenericList tests = new GenericList();
tests.add(79);
tests.add(88);
```

However, with the current implementation of `GenericList`, we still have to cast the return value from this `get` method.

```
public Object get(int atIndex) {
    return elements[index];
}
```

The compiler sees the return type `Object` that must be cast to whatever type of value happens to be stored at `elements[index]`:

```
Integer anInt = (Integer)tests.get(0);
```

Self-Check

13-4 Place a check mark \checkmark in the comment after assignment statement that compiles (or leave blank).

```
Object anObject = new Object();
String aString = "abc";
Integer anInteger = new Integer(5);
anObject = aString; // _____
anInteger = aString; // _____
anObject = anInteger; // _____
anInteger = anObject; // _____
```

13-5 Place a check mark \checkmark in the comment after assignment statement that compiles (or leave blank).

```
Object anObject = new String("abc");
Object anotherObject = new Integer(50);
Integer n = (Integer) anObject; // _____
String s = (String) anObject; // _____
anObject = anotherObject; // _____
String another = (String) anotherObject; // _____
Integer anotherInt = (Integer) anObject; // _____
```

13-6 Place a check mark \checkmark in the comment after assignment statement that compiles (or leave blank).

```
Integer num1 = 5; // _____
Integer num2 = 5.0; // _____
Object num3 = 5.0; // _____
int num4 = new Integer(6); // _____
```

13.2 Java Generics with Type Parameters

The manual boxing, the cast code, and the problems associated with collections that can accidentally add the wrong type element are problems that all disappeared in Java 5. Now the programmer can specify the one type that should be stored by passing the type as an argument to the collection class. Type parameters and arguments parameters are enclosed between `<` and `>` rather than `(` and `)`. Now these safer collection classes look like this:

```
public class GenericListWithTypeParameter<E> {
    private Object[] elements;
    private int n;

    public GenericListWithTypeParameter() {
        elements = new Object[10];
        n = 0;
    }

    public int size() {
        return n;
    }

    // Retrieve a reference to an element
    // Precondition: index >= 0 && index < size()
    public void add(E element) {
        elements[n] = element;
        n++;
    }

    // Place the cast code here once so no other casting code is necessary
    public E get(int index) {
        return (E)elements[index];
    }
}
```

Instances of `GenericListWithTypeParameter` would now be constructed by passing the type of element as an argument to the class:

```
GenericListWithTypeParameter<String> strings =
    new GenericListWithTypeParameter<String>();

GenericListWithTypeParameter<Integer> tests =
    new GenericListWithTypeParameter<Integer>();

GenericListWithTypeParameter<BankAccount> accounts =
    new GenericListWithTypeParameter<BankAccount>();
```

In the `add` method, `Object` is replaced with `E` so only elements of the type argument—`String`, `Integer` or `BankAccount` above—can be added to that collection.

The compiler catches accidental calls to `add` with the wrong type of argument. For example, the following three attempts to add the wrong type generate compiletime errors. This is a good thing. It is better than waiting until runtime when the program would otherwise terminate.

```
strings.add(123);
The method add(String) in the type GenericListWithTypeParameter<String> is not applicable for the
arguments (int)

tests.add("a String");
The method add(Integer) in the type GenericListWithTypeParameter<Integer> is not applicable for
the arguments (String)
```

You also don't have to manually box primitives during `add` messages.

```
tests.add(89);
tests.add(77);
tests.add(95);
```

Nor do the return values need to be cast since the cast to the correct type is done in the `get` method.

```
strings.add("First");
strings.add("Second");
String noCastToStringNeeded = strings.get(0);
```

And in the case of `Integer`, the `Integer` object is automatically unboxed to a primitive `int`.

```
int sum = 0;
for (int i = 0; i < tests.size(); i++) {
    sum += tests.get(i); // no cast needed because get already casts to (E)
}
```

Using Java generic type arguments does require extra syntax when constructing a collection (two sets of angle brackets and the type to be stored twice). However the benefits include much less casting syntax, the ability to have collections of primitives where the primitives appear to be objects, and we gain the type safety that comes from allowing the one type of element to be maintained by the collection. The remaining examples in this textbook will use Java Generics with `<` and `>` rather than having parameters and return types of type `Object`.

Self-Check

13-7 Give the following code, print all elements in uppercase. Hint no cast required before sending a message

```
GenericListWithTypeParameter<String> strings =
    new GenericListWithTypeParameter<String>();
strings.add("lower");
strings.add("Some UpPeR");
strings.add("ALL UPPER");
```

Answers to Self-Check Questions

13-1 which have errors? b and c

-b cannot assign an Object to an Integer

-c cannot assign an Object to a String even when it references a String

13-2 a, b, and c. In d, the compiler notices the attempt to store a String into a Point?

```
Point p = (String)obj2;           // d.
```

13-3 None

```
13-4 anObject = aString;           //   x  
    anInteger = aString;           // Can't assign String to Integer
    anObject = anInteger;          //   x  
    anInteger = anObject;          // Can't assign Object to Integer
```

```
13-5 Object anObject = new String("abc");
    Object anotherObject = new Integer(50);
    Integer n = (Integer) anObject; //   x  
    String s = (String) anObject;   //   x  
    anObject = anotherObject;       //   x  
    String another = (String) anotherObject; //   x  
    Integer anotherInt = (Integer) anObject; //   x  
```

```
13-6 Integer num1 = 5;              //   x  
    Integer num2 = 5.0;              //           
    Object num3 = 5.0;               //   x  
    int num4 = new Integer(6);       //   x  
```

```
14-7 for (int i = 0; i < strings.size(); i++) {
    System.out.println(strings.get(i).toUpperCase());
}
```