CHAPTER 5

Generic Programming

LEARNING OBJECTIVES

When you complete Chapter 5, you will be able to...

- declare and use Java Object variables that can refer to any kind of object in a Java program.
- correctly use widening and narrowing to convert back and forth between Java Object variables and other object types.
- correctly use autoboxing and unboxing to convert back and forth between Java's primitive types and the Java wrapper classes.
- design, implement and use generic methods that depend on a generic type parameter that varies from one activation to another.
- design, implement and use generic collection classes where the data type of the underlying elements is specified by a generic type parameter.
- create classes that implement interfaces and generic interfaces from the Java API.
- design, implement, and use Java iterators for a collection class.
- use Java's enhanced for-loop for collections with iterators.
- find and read the API documentation for the Java classes that implement the Collection interface (such as Vector) and the Map interface (such as TreeMap) and use these classes in your programs.

CHAPTER CONTENTS

5.1 Java's Object Type and Wrapper classes and Wrapper Classes
5.2 Object Methods and Generic Methods
5.3 Generic Classes
5.4 Generic Nodes
5.5 Interfaces and Iterators
5.6 A Generic Bag Class That Implements the Iterable Interface (Optional Section)
5.7 Introduction to the Java Collection and Map Interfaces (Optional Section)
Chapter Summary
Solutions to Self-Test Exercises
Programming Projects

I never knew just what it was, and I guess I never will.

TOM PAXTON
"Marvelous Toy"

Professional programmers often write classes that have general applicability in many settings. To some extent, our classes do this already. Certainly, the bag and sequence classes can be used in many different settings. However, both our bag and our sequence suffer from the fact that they require the underlying element type to be fixed. We started with a bag of integers (`IntArrayBag`). If we later need a bag of double numbers, then a second implementation (`DoubleArrayBag`) is required. We can end up with eight different bags—one for each of Java's eight primitive types. But even then we're not done; we could use a bag of locations (with the `Location` class in Figure 2.5 on page 67) or a bag of strings (using Java's built-in `String` class, which is not one of the primitive types). It seems as if the onslaught of bags will never stop.

generic classes

A **generic class** is a new feature of Java that provides a solution to this problem. For example, a single generic bag can be used for a bag of integers, but also used for a bag of double numbers, or strings or whatever else is needed. By using Java's `Object` type, a generic bag can even hold a mixture of objects of different types.

This chapter shows how to build your own generic classes and how to use some of the generic collection classes that Java provides. In addition, you'll learn about related issues such as the effective use of Java's `Object` type and wrapper classes, which are addressed in the first section of the chapter.,

5.1 JAVA'S OBJECT TYPE AND WRAPPER CLASSES

A Java variable can be one of the eight primitive data types. Anything that's not one of the eight primitive types is a reference to an object. For example, if you declare a `Location` variable, that variable is capable of holding a reference to a `Location` object; if you declare a `String` variable, that variable is capable of holding a reference to a `String` object.

Java has an important built-in data type called `Object`. An `Object` variable is capable of holding a reference to any kind of object. To see how this is useful, let's look at how assignment statements can go back and forth between an `Object` variable and other data types.

Eight Primitive Types

`byte`
`short`
`int`
`long`
`float`
`double`
`char`
`boolean`

...and everything else is a reference to an object

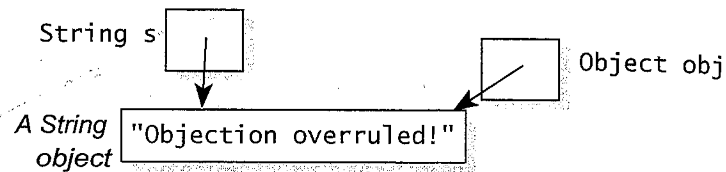
Widening Conversions

Here's some code to show how an assignment can be made to an `Object` variable. The code declares a `String` with the value `"Objection overruled!"` A second `Object` variable is then declared and made to refer to the same string.

```
String s = new String("Objection overruled!");
Object obj;

obj = s;
```

At this point, there is only one string—`"Objection overruled!"`—with both `s` and `obj` referring to this one string, as shown here:



Assigning a specific kind of object to an `Object` variable is an example of a **widening conversion**. The term "widening" means that the `obj` variable has a "wide" ability to refer to different things; in fact, `obj` is very "wide" because it *could* refer to any kind of object. On the other hand, the variable `s` is relatively narrow with regard to the things that it can refer to—`s` can refer to only a `String`.

Widening Conversions with Reference Variables

Suppose `x` and `y` are reference variables. An assignment `x = y` is a **widening conversion** if the data type of `x` is capable of referring to a wider variety of things than the type of `y`.

Example:

```
String s = new String("Objection overruled!");
Object obj;
obj = s;
```

Java permits all widening conversions.

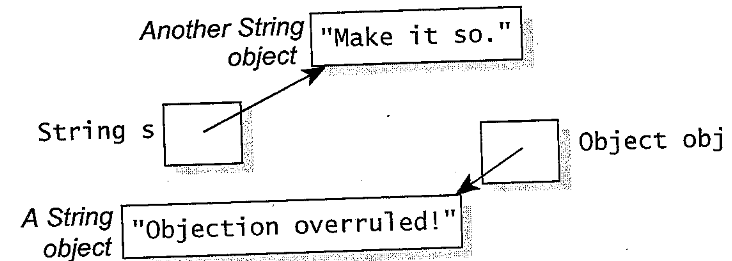
Narrowing Conversions

After the widening conversion `obj = s`, our program can continue to do other things, perhaps even making `s` refer to a new string, as shown here:

```
String s = new String("Objection overruled!");
Object obj;

obj = s;
s = new String("Make it so.");
```

At this point, `s` refers to a new string, and `obj` still refers to the original string, as shown here:



At a later point in the program, we can make `s` refer to the original string once again with an assignment statement, but the assignment needs more than merely `s = obj`. In fact, the Java compiler forbids the assignment `s = obj`. As far as the compiler knows, `obj` could refer to anything—it does not have to refer to a `String`, so the compiler forbids `s = obj`. The way around the restriction is to use the typecast expression shown here:

```
s = (String) obj;
```

The expression `(String) obj` tells the compiler that the reference variable `obj` is really referring to a `String` object. This is a typecast, as discussed in Chapter 2. With the typecast, the compiler allows the assignment statement, though you must still be certain that `obj` actually does refer to a `String` object. Otherwise, when the program runs, there will be a `ClassCastException` (see "Pitfall: Class Cast Exception" on page 82).

The complete assignment `s = (String) obj` is an example of a **narrowing conversion**. The term "narrowing" means that the left side of the assignment has a smaller ability to refer to different things than the right side.

Narrowing Conversion with Reference Variables

Suppose x and y are reference variables, and x has a smaller ability to refer to things than y. A narrowing conversion using a typecast can be made to assign x from y.

Example:

```
String s = new String("Objection overruled!");
Object obj;
obj = s;
...
s = (String) obj;
```

By using a typecast, Java permits all narrowing conversions, though a `ClassCastException` may be thrown at runtime if the object does not satisfy the typecast.

Narrowing conversions also occur when a method returns an `Object`, and the program assigns that `Object` to a variable of a particular type. For example, the `IntLinkedBag` class from Chapter 4 has a `clone` method:

♦ **clone**
`public IntLinkedBag clone( )`
Generate a copy of this bag.

Returns:
The return value is a copy of this bag. Subsequent changes to the copy will not affect the original, nor vice versa. The return value must be typecast to an `IntLinkedBag` before it is used.

In the implementation of this method, we have the assignment:

```
IntLinkedBag answer;
answer = (IntLinkedBag) super.clone( );
```

The super.clone method from Java's `Object` class returns an `Object`. Of course, in this situation, *we* know that the `Object` is really an `IntLinkedBag`, but *the compiler* doesn't know that. Therefore, to use the answer from the `super.clone` method, we must insert the narrowing typecast, `(IntLinkedBag)`. This tells the compiler that the programmer knows that the actual value is an `IntLinkedBag`, so it is safe to assign it to the answer variable.

Methods That Returns an Object

If a method returns an `Object`, then a narrowing conversion is usually needed to actually use the return value.

Wrapper Classes

As a programmer, it's annoying that *almost* everything is a Java object. It's those eight darn primitive types that cause the problem. Java's wrapper classes offer a partial solution to this annoyance. A **wrapper class** is a class in whic each object holds a primitive value. For example, a wrapper class called `Integer` is designed so that each `Integer` object holds one `int` value. The two most important `Integer` methods are the constructor (which creates an `Integer` object from an `int` value) and the `intValue` method (which accesses the `int` value stored in an `Integer` object). Here are the specifications for these two `Integer` methods:

Primitive	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

♦ **Constructor for the Integer class**
`public Integer(int value)`
Construct an `Integer` object from an `int` value.

Parameters:
`value` – the `int` value for this `Integer` to hold

Postcondition:
This `Integer` has been initialized with the specified `int` value.

♦ **intValue**
`public int intValue( )`
Accessor method to retrieve the `int` value of this object.

Returns:
the `int` value of this object

Here's an example to show how an integer is placed into an `Integer` object (called a **boxing conversion**) and taken back out (an **unboxing conversion**):

```
int i = 42;
int j;
Integer example;
example = new Integer(i); // The example has value 42 (boxing)
j = example.intValue( );  // j is now 42 too (unboxing)
```

Autoboxing and Auto-Unboxing Conversions

J2SE 5.0 introduced a new convenience that simplifies the process of putting a primitive value into a wrapper object and taking it back out. In many situations, the Java compiler will recognize that a wrapper object is needed, and automatically convert a primitive value to the wrapper object (an **autoboxing conversion**). Java will also convert from a wrapper object to the corresponding primitive type value in many situations (an **auto-unboxing conversion**). For example, in J2SE 5.0, the assignments shown above can be simplified:

automatically convert from a primitive value to a wrapper object, or vice versa

```
example = i;      // Autoboxing occurs in an assignment statement
j = example;      // Auto-unboxing occurs in an assignment statement
```

In a similar way, if a method expects an `Integer` parameter, the actual argument may be a primitive `int` value, and an autoboxing conversion will occur. If a method expects an int parameter, the actual argument may be an Integer object, and an auto-unboxing conversion will occur.

> **Boxing and Unboxing Conversions**
>
> A **boxing conversion** occurs when a primitive value is converted to a wrapper object of the corresponding type. An **unboxing conversion** occurs when a wrapper object is converted to the corresponding primitive type. These may occur with an explicit conversion or automatically when the compiler detects the need for the conversion.
>
> Examples:
>
> ```
> int i = 42;
> int j;
> Integer example;
> example = new Integer(i); // Boxing
> example = i;              // Autoboxing
> j = example.intValue( );  // Unboxing
> j = example;              // Auto-unboxing
> ```

Advantages and Disadvantages of Wrapper Objects

The advantage of putting a value in a wrapper object is that the wrapper object is a full-blown Java object. For example, it can be put into the generic bag that we will create in Section 5.3. The disadvantage is that the ordinary primitive operations are no longer directly available. For example, suppose x, y, and z are `Integer` objects. Although the statement `z = x + y;` is legal, its evaluation is somewhat slow because the `x` and `y` `Integer` objects must first be auto-unboxed (converted to primitive `int` values) before the "+" operation can be applied. The answer must then undergo an autoboxing conversion before it can be stored back in the `Integer` `z`.

Self-Test Exercises for Section 5.1

1. Can an `Object` variable refer to a `String`? To a `Location`? To an `IntArrayBag`?
2. Write some code that includes both a widening conversion and a narrowing conversion. Draw a circle by the widening and a triangle by the narrowing.
3. Write an example of a ... `ClassCastException` at runtime.
4. Write some code that creates a `Character` wrapper object called `example`, initializing with the char `'w'`.
5. Describe an advantage and a disadvantage of wrapper objects.
6. Suppose that `x`, `y` and `z` are all `Double` objects. Describe all the boxing and unboxing that occurs in the assignment `z = x + y;`

5.2 OBJECT METHODS AND GENERIC METHODS

Sometimes it seems that programmers intentionally make extra work for themselves. For example, suppose we write this method:

```
static Integer middle(Integer[] data)
// If the array is nonempty, then the function returns an element from
// near the middle of the array; otherwise the return value is null.
{
   if (data.length == 0)
   {  // The array has no elements, so return null.
      return null;
   }
   else
   {  // Return an element near the center of the array.
      return data[data.length/2];
   }
}
```

This is a fine method, reliably returning an element from near the middle of any nonempty array. Such a method is useful in certain kinds of searches. For example (with an `Integer` `i` and a non-empty `Integer` array `ia`):

```
i = middle(ia); // Set i to something from the middle of the array ia
```

Now, suppose that tomorrow you have another program that needs to do the same thing with `Character` objects:

```
static Integer middle(Integer[] data)...
```

we write lots of different middle methods

As you start writing the code, you realize that the only difference between this new method and the original version is that we must change every occurrence of the word `Integer` to `Character`. Later, if we want to carry out the same task with values of some other type, we might write another method, and another.... In fact, a single program might use many `middle` methods. When one of the functions is used, the compiler looks at the type of the argument and selects the appropriate version of the `middle` method. This works fine, but with this approach, you do need to write a different method for each different data type.

Object Methods

One way to avoid many different `middle` methods is to write just one method using the `Object` data type:

```
static Object middle(Object[] data)
{
   if (data.length == 0)
   {  // The array has no elements, so return null.
      return null;
   }
   else
   {  // Return an element near the center of the array.
      return data[data.length/2];
   }
}
```

This approach also works fine. We can use this version of occurs with an `Integer` array and a narrowing conversion to convert the return value from an `Object` to an `Integer`:

```
i = (Integer) middle(ia); // i is an Integer; ia is an Integer array.
```

We need only one version of the `Object` method, but there are some potential type errors that can't be caught until the program is running. For example, a programmer might accidentally use the middle method with a `Character` array, but assign the result to an `Integer` variable. Such a mistake would compile, but at runtime there would be a `ClassCastException` from the illegal narrowing conversion. Another approach, called generic methods, offer a safer approach that cannot result in a type mismatch.

Generic Methods

A **generic method** is a method that is written with the types of the parameters not fully specified. Our middle method is written this way as a generic method:

```
static <T> T middle(T[] data)
{
   if (data.length == 0)
   {  // The array has no elements, so return null.
      return null;
   }
   else
   {  // Return an element near the center of the array.
      return data[data.length/2];
   }
}
```

Notice that the name `T` appears in angle brackets, `<T>`, that appears right before the return type in the method header. This name is called the **generic type parameter**. The generic type parameter indicates that the method depends on some particular class, but the programmer is not going to specify exactly what that class is. Perhaps `T` will eventually be an `Integer`, or a `Character`, or who knows what else. The name, `T`, that we used for the generic type parameter may be any legal Java identifier, but the designers of Java recommend single capital letters such as `T` (an abbreviation of "type") or `E` (an abbreviation of "element").

The generic type parameter always appears in angle brackets right before the return type of the method. This is how the compiler knows that the method is a generic method. In the rest of the method's implementation, the generic type may be used just like any other class name, and for most situations, it must be used at least once in the parameter list. In our example, `T` is used twice (once as the return type of the method, and once as the data type of the array in the parameter list):

```
static <T> T middle(T[] data)...
```

When a generic method is activated, the compiler infers the correct class for the generic type parameter. For example:

```
i = middle(ia); // i is an Integer; ia is an Integer array.
```

a generic method allows any class for the argument, and the compiler can detect certain type errors

In this case, the compiler sees the `Integer` array, `ia`, and determines that the data type of `T` must be `Integer`. This allows the compiler to detect potential type errors at compile time, before the program is running. For example:

```
c = middle(ia); // c is a Character; ia is an Integer array--type error!
```

For this assignment statement, the compiler will indicate that the `Integer` return value cannot be assigned to a `Character` variable.

Later, we will see generic methods with more than one generic type parameter. For example, here is a method with two generic type parameters, each of which occurs twice in the parameter list:

```
static <S,T> bool most(S[] sa, S starget, T[] ta, T ttarget)
// Returns true if starget appears in sa more often than ttarget appears in ta.
```

PITFALL

GENERIC METHOD RESTRICTIONS

The data type inferred by the compiler for the generic type must always be a class type (not one of the primitive types). In addition, you may not call a constructor for the generic type, nor may you create a new array of elements of that type.

These restrictions are a consequence of the compilation method that Java uses for generic methods.

Generic Methods

A **generic method** is written by putting a generic type parameter in angle brackets before the return type in the heading of the method. For example:-

```
static <T> T middle(T[] data)...
```

The generic data type is often named with a single capital letter, such as T for "type." The name T can then be used in the rest of the method implementation as if it were a real class name (with a few exceptions).

A programmer can activate a generic method just like any other method. The compiler will look at the data types of the arguments and infer a correct type for each generic type parameter. The inferred types help catch type errors at compile time. For example:

```
// i is an integer; ia is an Integer array; c is a Character

i = middle(ia); // This is fine
c = middle(ia); // Compile-time type error
```

Some additional features of generic methods, such as the ability to restrict the type of the argument to specific kinds of objects, will be covered in Chapter 13.

Self-Test Exercises for Section 5.2

7. Describe the main purpose of a generic method.
8. What is the principle disadvantage of using an object method as compared to a generic method?
9. What is the generic type parameter, and where does it first appear in the method implementation? Where else may it appear in the implementation? In what one other location does it nearly always appear?
10. Name two things that can usually be done with a class type, but which are forbidden for a generic type.
11. Write a generic method for counting the number of occurrences of a target in an array. If the target is non-null, then use `target.equals` to determine equality.
12. Write the `most` method that was specified at the end of this section.