```
  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// A subclass of Gen that defines a second
// type parameter, called V.
class Gen2<T, V> extends Gen<T> {
  V ob2;

  Gen2(T o, V o2) {
    super(o);
    ob2 = o2;
  }

  V getob2() {
    return ob2;
  }
}

// Create an object of type Gen2.
class HierDemo {
  public static void main(String args[]) {

    // Create a Gen2 object for String and Integer.
    Gen2<String, Integer> x =
      new Gen2<String, Integer>("Value is: ", 99);

    System.out.print(x.getob());
    System.out.println(x.getob2());
  }
}
```

Notice the declaration of this version of **Gen2**, which is shown here:

```
class Gen2<T, V> extends Gen<T> {
```

Here, **T** is the type passed to **Gen**, and **V** is the type that is specific to **Gen2**. **V** is used to declare an object called **ob2**, and as a return type for the method **getob2( )**. In **main( )**, a **Gen2** object is created in which type parameter **T** is **String**, and type parameter **V** is **Integer**. The program displays the following, expected, result:

```
Value is: 99
```

A Generic Subclass

It is perfectly acceptable for a non-generic class to be the superclass of a generic subclass. For example, consider this program:

```
// A non-generic class can be the superclass
// of a generic subclass.

// A non-generic class.
class NonGen {
  int num;

  NonGen(int i) {
    num = i;
  }

  int getnum() {
    return num;
  }
}

// A generic subclass.
class Gen<T> extends NonGen {
  T ob; // declare an object of type T

  // Pass the constructor a reference to
  // an object of type T.
  Gen(T o, int i) {
    super(i);
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// Create a Gen object.
class HierDemo2 {
  public static void main(String args[]) {

    // Create a Gen object for String.
    Gen<String> w = new Gen<String>("Hello", 47);

    System.out.print(w.getob() + " ");
    System.out.println(w.getnum());
  }
}
```

The output from the program is shown here:

```
Hello 47
```

In the program, notice how **Gen** inherits **NonGen** in the following declaration:

```
class Gen<T> extends NonGen {
```

Because **NonGen** is not generic, no type argument is specified. Thus, even though **Gen** declares the type parameter **T**, it is not needed by (nor can it be used by) **NonGen**. Thus, **NonGen** is inherited by **Gen** in the normal way. No special conditions apply.

Run-Time Type Comparisons Within a Generic Hierarchy

Recall the run-time type information operator **instanceof** that was described in Chapter 13. As explained, **instanceof** determines if an object is an instance of a class. It returns true if an object is of the specified type or can be cast to the specified type. The **instanceof** operator can be applied to objects of generic classes. The following class demonstrates some of the type compatibility implications of a generic hierarchy:

```
// Use the instanceof operator with a generic class hierarchy.
class Gen<T> {
  T ob;

  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// A subclass of Gen.
class Gen2<T> extends Gen<T> {
  Gen2(T o) {
    super(o);
  }
}

// Demonstrate run-time type ID implications of generic
// class hierarchy.
class HierDemo3 {
  public static void main(String args[]) {

    // Create a Gen object for Integers.
    Gen<Integer> iOb = new Gen<Integer>(88);

    // Create a Gen2 object for Integers.
    Gen2<Integer> iOb2 = new Gen2<Integer>(99);
```

```
    // Create a Gen2 object for Strings.
    Gen2<String> strOb2 = new Gen2<String>("Generics Test");

    // See if iOb2 is some form of Gen2.
    if(iOb2 instanceof Gen2<?>)
      System.out.println("iOb2 is instance of Gen2");

    // See if iOb2 is some form of Gen.
    if(iOb2 instanceof Gen<?>)
      System.out.println("iOb2 is instance of Gen");

    System.out.println();

    // See if strOb2 is a Gen2.
    if(strOb2 instanceof Gen2<?>)
      System.out.println("strOb2 is instance of Gen2");

    // See if strOb2 is a Gen.
    if(strOb2 instanceof Gen<?>)
      System.out.println("strOb2 is instance of Gen");

    System.out.println();

    // See if iOb is an instance of Gen2, which it is not.
    if(iOb instanceof Gen2<?>)
      System.out.println("iOb is instance of Gen2");

    // See if iOb is an instance of Gen, which it is.
    if(iOb instanceof Gen<?>)
      System.out.println("iOb is instance of Gen");

    // The following can't be compiled because
    // generic type info does not exist at run time.
//    if(iOb2 instanceof Gen2<Integer>)
//      System.out.println("iOb2 is instance of Gen2<Integer>");
  }
}
```

The output from the program is shown here:

```
iOb2 is instance of Gen2
iOb2 is instance of Gen

strOb2 is instance of Gen2
strOb2 is instance of Gen

iOb is instance of Gen
```

In this program, **Gen2** is a subclass of **Gen**, which is generic on type parameter **T**. In **main( )**, three objects are created. The first is **iOb**, which is an object of type **Gen<Integer>**. The second is **iOb2**, which is an instance of **Gen2<Integer>**. Finally, **strOb2** is an object of type **Gen2<String>**.

Then, the program performs these **instanceof** tests on the type of **iOb2**:

```
// See if iOb2 is some form of Gen2.
if(iOb2 instanceof Gen2<?>)
  System.out.println("iOb2 is instance of Gen2");

// See if iOb2 is some form of Gen.
if(iOb2 instanceof Gen<?>)
  System.out.println("iOb2 is instance of Gen");
```

As the output shows, both succeed. In the first test, **iOb2** is checked against **Gen2<?>**. This test succeeds because it simply confirms that **iOb2** is an object of some type of **Gen2** object. The use of the wildcard enables **instanceof** to determine if **iOb2** is an object of any type of **Gen2**. Next, **iOb2** is tested against **Gen<?>**, the superclass type. This is also true because **iOb2** is some form of **Gen**, the superclass. The next few lines in **main( )** show the same sequence (and same results) for **strOb2**.

Next, **iOb**, which is an instance of **Gen<Integer>** (the superclass), is tested by these lines:

```
// See if iOb is an instance of Gen2, which it is not.
if(iOb instanceof Gen2<?>)
  System.out.println("iOb is instance of Gen2");

// See if iOb is an instance of Gen, which it is.
if(iOb instanceof Gen<?>)
  System.out.println("iOb is instance of Gen");
```

The first **if** fails because **iOb** is not some type of **Gen2** object. The second test succeeds because **iOb** is some type of **Gen** object.

Now, look closely at these commented-out lines:

```
    // The following can't be compiled because
    // generic type info does not exist at run time.
//    if(iOb2 instanceof Gen2<Integer>)
//      System.out.println("iOb2 is instance of Gen2<Integer>");
```

As the comments indicate, these lines can't be compiled because they attempt to compare **iOb2** with a specific type of **Gen2**, in this case, **Gen2<Integer>**. Remember, there is no generic type information available at run time. Therefore, there is no way for **instanceof** to know if **iOb2** is an instance of **Gen2<Integer>** or not.

Casting

You can cast one instance of a generic class into another only if the two are otherwise compatible and their type arguments are the same. For example, assuming the foregoing program, this cast is legal:

```
(Gen<Integer>) iOb2 // legal
```

because **iOb2** includes an instance of **Gen<Integer>**. But, this cast:

```
(Gen<Long>) iOb2 // illegal
```

is not legal because **iOb2** is not an instance of **Gen<Long>**.

Overriding Methods in a Generic Class

A method in a generic class can be overridden just like any other method. For example, consider this program in which the method **getob( )** is overridden:

```
// Overriding a generic method in a generic class.
class Gen<T> {
  T ob; // declare an object of type T

  // Pass the constructor a reference to
  // an object of type T.
  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    System.out.print("Gen's getob(): " );
    return ob;
  }
}

// A subclass of Gen that overrides getob().
class Gen2<T> extends Gen<T> {

  Gen2(T o) {
    super(o);
  }

  // Override getob().
  T getob() {
    System.out.print("Gen2's getob(): ");
    return ob;
  }
}

// Demonstrate generic method override.
class OverrideDemo {
  public static void main(String args[]) {

    // Create a Gen object for Integers.
    Gen<Integer> iOb = new Gen<Integer>(88);

    // Create a Gen2 object for Integers.
    Gen2<Integer> iOb2 = new Gen2<Integer>(99);

    // Create a Gen2 object for Strings.
    Gen2<String> strOb2 = new Gen2<String> ("Generics Test");

    System.out.println(iOb.getob());
    System.out.println(iOb2.getob());
    System.out.println(strOb2.getob());
  }
}
```

The output is shown here:

```
Gen's getob(): 88
Gen2's getob(): 99
Gen2's getob(): Generics Test
```

As the output confirms, the overridden version of **getob( )** is called for objects of type **Gen2**, but the superclass version is called for objects of type **Gen**.

Type Inference with Generics

Beginning with JDK 7, it is possible to shorten the syntax used to create an instance of a generic type. To begin, consider the following generic class:

```
class MyClass<T, V> {
  T ob1;
  V ob2;

  MyClass(T o1, V o2) {
    ob1 = o1;
    ob2 = o2;
  }
  // ...
}
```

Prior to JDK 7, to create an instance of **MyClass**, you would have needed to use a statement similar to the following:

```
MyClass<Integer, String> mcOb =
  new MyClass<Integer, String>(98, "A String");
```

Here, the type arguments (which are **Integer** and **String**) are specified twice: first, when **mcOb** is declared, and second, when a **MyClass** instance is created via **new**. Since generics were introduced by JDK 5, this is the form required by all versions of Java prior to JDK 7. Although there is nothing wrong, per se, with this form, it is a bit more verbose than it needs to be. In the **new** clause, the type of the type arguments can be readily inferred from the type of **mcOb**; therefore, there is really no reason that they need to be specified a second time. To address this situation, JDK 7 added a syntactic element that lets you avoid the second specification.

Today the preceding declaration can be rewritten as shown here:

```
MyClass<Integer, String> mcOb = new MyClass<>(98, "A String");
```

Notice that the instance creation portion simply uses <>, which is an empty type argument list. This is referred to as the *diamond* operator. It tells the compiler to infer the type arguments needed by the constructor in the **new** expression. The principal advantage of this type-inference syntax is that it shortens what are sometimes quite long declaration statements.

The preceding can be generalized. When type inference is used, the declaration syntax for a generic reference and instance creation has this general form:

class-name<*type-arg-list*> *var-name* = new *class-name*<>(*cons-arg-list*);

Here, the type argument list of the constructor in the **new** clause is empty.

Type inference can also be applied to parameter passing. For example, if the following method is added to **MyClass**,

```
boolean isSame(MyClass<T, V> o) {
  if(ob1 == o.ob1 && ob2 == o.ob2) return true;
  else return false;
}
```

then the following call is legal:

```
if(mcOb.isSame(new MyClass<>(1, "test"))) System.out.println("Same");
```

In this case, the type arguments for the argument passed to **isSame( )** can be inferred from the parameter's type.

Because the type-inference syntax was added by JDK 7 and won't work with older compilers, most of the examples in this book will continue to use the full syntax when declaring instances of generic classes. This way, the examples will work with any Java compiler that supports generics. Using the full-length syntax also makes it very clear precisely what is being created, which is important in example code shown in a book. However, in your own code, the use of the type-inference syntax will streamline your declarations.

Erasure

Usually, it is not necessary to know the details about how the Java compiler transforms your source code into object code. However, in the case of generics, some general understanding of the process is important because it explains why the generic features work as they do—and why their behavior is sometimes a bit surprising. For this reason, a brief discussion of how generics are implemented in Java is in order.

An important constraint that governed the way that generics were added to Java was the need for compatibility with previous versions of Java. Simply put, generic code had to be compatible with preexisting, non-generic code. Thus, any changes to the syntax of the Java language, or to the JVM, had to avoid breaking older code. The way Java implements generics while satisfying this constraint is through the use of *erasure*.

In general, here is how erasure works. When your Java code is compiled, all generic type information is removed (erased). This means replacing type parameters with their bound type, which is **Object** if no explicit bound is specified, and then applying the appropriate casts (as determined by the type arguments) to maintain type compatibility with the types specified by the type arguments. The compiler also enforces this type compatibility. This approach to generics means that no type parameters exist at run time. They are simply a source-code mechanism.

Bridge Methods

Occasionally, the compiler will need to add a *bridge method* to a class to handle situations in which the type erasure of an overriding method in a subclass does not produce the same erasure as the method in the superclass. In this case, a method is generated that uses the type erasure of the superclass, and this method calls the method that has the type erasure specified by the subclass. Of course, bridge methods only occur at the bytecode level, are not seen by you, and are not available for your use.

Although bridge methods are not something that you will normally need to be concerned with, it is still instructive to see a situation in which one is generated. Consider the following program:

```
// A situation that creates a bridge method.
class Gen<T> {
  T ob; // declare an object of type T

  // Pass the constructor a reference to
  // an object of type T.
  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// A subclass of Gen.
class Gen2 extends Gen<String> {

  Gen2(String o) {
    super(o);
  }

  // A String-specific override of getob().
  String getob() {
    System.out.print("You called String getob(): ");
    return ob;
  }
}

// Demonstrate a situation that requires a bridge method.
class BridgeDemo {
  public static void main(String args[]) {

    // Create a Gen2 object for Strings.
    Gen2 strOb2 = new Gen2("Generics Test");

    System.out.println(strOb2.getob());
  }
}
```

In the program, the subclass **Gen2** extends **Gen**, but does so using a **String**-specific version of **Gen**, as its declaration shows:

```
class Gen2 extends Gen<String> {
```

Furthermore, inside **Gen2**, **getob( )** is overridden with **String** specified as the return type:

```
// A String-specific override of getob().
String getob() {
  System.out.print("You called String getob(): ");
  return ob;
}
```

All of this is perfectly acceptable. The only trouble is that because of type erasure, the expected form of **getob( )** will be

```
Object getob() { // ...
```

To handle this problem, the compiler generates a bridge method with the preceding signature that calls the **String** version. Thus, if you examine the class file for **Gen2** by using **javap**, you will see the following methods:

```
class Gen2 extends Gen<java.lang.String> {
  Gen2(java.lang.String);
  java.lang.String getob();
  java.lang.Object getob(); // bridge method
}
```

As you can see, the bridge method has been included. (The comment was added by the author and not by **javap**, and the precise output you see may vary based on the version of Java that you are using.)

There is one last point to make about this example. Notice that the only difference between the two **getob( )** methods is their return type. Normally, this would cause an error, but because this does not occur in your source code, it does not cause a problem and is handled correctly by the JVM.

Ambiguity Errors

The inclusion of generics gives rise to a new type of error that you must guard against: *ambiguity*. Ambiguity errors occur when erasure causes two seemingly distinct generic declarations to resolve to the same erased type, causing a conflict. Here is an example that involves method overloading:

```
// Ambiguity caused by erasure on
// overloaded methods.
class MyGenClass<T, V> {
  T ob1;
  V ob2;

  // ...
```

```
  // These two overloaded methods are ambiguous
  // and will not compile.
  void set(T o) {
    ob1 = o;
  }

  void set(V o) {
    ob2 = o;
  }
}
```

Notice that **MyGenClass** declares two generic types: **T** and **V**. Inside **MyGenClass**, an attempt is made to overload **set( )** based on parameters of type **T** and **V**. This looks reasonable because **T** and **V** appear to be different types. However, there are two ambiguity problems here.

First, as **MyGenClass** is written, there is no requirement that **T** and **V** actually be different types. For example, it is perfectly correct (in principle) to construct a **MyGenClass** object as shown here:

```
MyGenClass<String, String> obj = new MyGenClass<String, String>()
```

In this case, both **T** and **V** will be replaced by **String**. This makes both versions of **set( )** identical, which is, of course, an error.

The second and more fundamental problem is that the type erasure of **set( )** reduces both versions to the following:

```
void set(Object o) { // ...
```

Thus, the overloading of **set( )** as attempted in **MyGenClass** is inherently ambiguous.

Ambiguity errors can be tricky to fix. For example, if you know that **V** will always be some type of **Number**, you might try to fix **MyGenClass** by rewriting its declaration as shown here:

```
class MyGenClass<T, V extends Number> { // almost OK!
```

This change causes **MyGenClass** to compile, and you can even instantiate objects like the one shown here:

```
MyGenClass<String, Number> x = new MyGenClass<String, Number>();
```

This works because Java can accurately determine which method to call. However, ambiguity returns when you try this line:

```
MyGenClass<Number, Number> x = new MyGenClass<Number, Number>();
```

In this case, since both **T** and **V** are **Number**, which version of **set( )** is to be called? The call to **set( )** is now ambiguous.

Frankly, in the preceding example, it would be much better to use two separate method names, rather than trying to overload **set( )**. Often, the solution to ambiguity involves the restructuring of the code, because ambiguity frequently means that you have a conceptual error in your design.

Some Generic Restrictions

There are a few restrictions that you need to keep in mind when using generics. They involve creating objects of a type parameter, static members, exceptions, and arrays. Each is examined here.

Type Parameters Can't Be Instantiated

It is not possible to create an instance of a type parameter. For example, consider this class:

```
// Can't create an instance of T.
class Gen<T> {
  T ob;

  Gen() {
    ob = new T(); // Illegal!!!
  }
}
```

Here, it is illegal to attempt to create an instance of **T**. The reason should be easy to understand: the compiler does not know what type of object to create. **T** is simply a placeholder.

Restrictions on Static Members

No **static** member can use a type parameter declared by the enclosing class. For example, both of the **static** members of this class are illegal:

```
class Wrong<T> {
  // Wrong, no static variables of type T.
  static T ob;

  // Wrong, no static method can use T.
  static T getob() {
    return ob;
  }
}
```

Although you can't declare **static** members that use a type parameter declared by the enclosing class, you can declare **static** generic methods, which define their own type parameters, as was done earlier in this chapter.

Generic Array Restrictions

There are two important generics restrictions that apply to arrays. First, you cannot instantiate an array whose element type is a type parameter. Second, you cannot create an array of type-specific generic references. The following short program shows both situations:

```
// Generics and arrays.
class Gen<T extends Number> {
  T ob;
```

```
  T vals[]; // OK

  Gen(T o, T[] nums) {
    ob = o;

    // This statement is illegal.
    // vals = new T[10]; // can't create an array of T

    // But, this statement is OK.
    vals = nums; // OK to assign reference to existent array
  }
}

class GenArrays {
  public static void main(String args[]) {
    Integer n[] = { 1, 2, 3, 4, 5 };

    Gen<Integer> iOb = new Gen<Integer>(50, n);

    // Can't create an array of type-specific generic references.
    // Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!

    // This is OK.
    Gen<?> gens[] = new Gen<?>[10]; // OK
  }
}
```

As the program shows, it's valid to declare a reference to an array of type **T**, as this line does:

```
T vals[]; // OK
```

But, you cannot instantiate an array of **T**, as this commented-out line attempts:

```
// vals = new T[10]; // can't create an array of T
```

The reason you can't create an array of **T** is that there is no way for the compiler to know what type of array to actually create.

However, you can pass a reference to a type-compatible array to **Gen( )** when an object is created and assign that reference to **vals**, as the program does in this line:

```
vals = nums; // OK to assign reference to existent array
```

This works because the array passed to **Gen** has a known type, which will be the same type as **T** at the time of object creation.

Inside **main( )**, notice that you can't declare an array of references to a specific generic type. That is, this line

```
// Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!
```

won't compile.

You *can* create an array of references to a generic type if you use a wildcard, however, as shown here:

```
Gen<?> gens[] = new Gen<?>[10]; // OK
```

This approach is better than using an array of raw types, because at least some type checking will still be enforced.

Generic Exception Restriction

A generic class cannot extend **Throwable**. This means that you cannot create generic exception classes.

This page has been intentionally left blank

CHAPTER

15 Lambda Expressions

During Java's ongoing development and evolution, many features have been added since its original 1.0 release. However, two stand out because they have profoundly reshaped the language, fundamentally changing the way that code is written. The first was the addition of generics, added by JDK 5. (See Chapter 14.) The second is the *lambda expression,* which is the subject of this chapter.

Added by JDK 8, lambda expressions (and their related features) significantly enhance Java because of two primary reasons. First, they add new syntax elements that increase the expressive power of the language. In the process, they streamline the way that certain common constructs are implemented. Second, the addition of lambda expressions resulted in new capabilities being incorporated into the API library. Among these new capabilities are the ability to more easily take advantage of the parallel processing capabilities of multi-core environments, especially as it relates to the handling of for-each style operations, and the new stream API, which supports pipeline operations on data. The addition of lambda expressions also provided the catalyst for other new Java features, including the default method (described in Chapter 9), which lets you define default behavior for an interface method, and the method reference (described here), which lets you refer to a method without executing it.

Beyond the benefits that lambda expressions bring to the language, there is another reason why they constitute an important addition to Java. Over the past few years, lambda expressions have become a major focus of computer language design. For example, they have been added to languages such as C# and C++. Their inclusion in JDK 8 helps Java remain the vibrant, innovative language that programmers have come to expect.

In the final analysis, in much the same way that generics reshaped Java several years ago, lambda expressions are reshaping Java today. Simply put, lambda expressions will impact virtually all Java programmers. They truly are that important.

Introducing Lambda Expressions

Key to understanding Java's implementation of lambda expressions are two constructs. The first is the lambda expression, itself. The second is the functional interface. Let's begin with a simple definition of each.

A *lambda expression* is, essentially, an anonymous (that is, unnamed) method. However, this method is not executed on its own. Instead, it is used to implement a method defined by a functional interface. Thus, a lambda expression results in a form of anonymous class. Lambda expressions are also commonly referred to as *closures*.

A *functional interface* is an interface that contains one and only one abstract method. Normally, this method specifies the intended purpose of the interface. Thus, a functional interface typically represents a single action. For example, the standard interface **Runnable** is a functional interface because it defines only one method: **run( )**. Therefore, **run( )** defines the action of **Runnable**. Furthermore, a functional interface defines the *target type* of a lambda expression. Here is a key point: a lambda expression can be used only in a context in which its target type is specified. One other thing: a functional interface is sometimes referred to as a *SAM type*, where SAM stands for Single Abstract Method.

NOTE A functional interface may specify any public method defined by **Object**, such as **equals( )**, without affecting its "functional interface" status. The public **Object** methods are considered implicit members of a functional interface because they are automatically implemented by an instance of a functional interface.

Let's now look more closely at both lambda expressions and functional interfaces.

Lambda Expression Fundamentals

The lambda expression introduces a new syntax element and operator into the Java language. The new operator, sometimes referred to as the *lambda operator* or the *arrow operator*, is **->**. It divides a lambda expression into two parts. The left side specifies any parameters required by the lambda expression. (If no parameters are needed, an empty parameter list is used.) On the right side is the *lambda body*, which specifies the actions of the lambda expression. The **->** can be verbalized as "becomes" or "goes to."

Java defines two types of lambda bodies. One consists of a single expression, and the other type consists of a block of code. We will begin with lambdas that define a single expression. Lambdas with block bodies are discussed later in this chapter.

At this point, it will be helpful to look a few examples of lambda expressions before continuing. Let's begin with what is probably the simplest type of lambda expression you can write. It evaluates to a constant value and is shown here:

```
() -> 123.45
```

This lambda expression takes no parameters, thus the parameter list is empty. It returns the constant value 123.45. Therefore, it is similar to the following method:

```
double myMeth() { return 123.45; }
```

Of course, the method defined by a lambda expression does not have a name.

A slightly more interesting lambda expression is shown here:

```
() -> Math.random() * 100
```

This lambda expression obtains a pseudo-random value from **Math.random( )**, multiplies it by 100, and returns the result. It, too, does not require a parameter.

When a lambda expression requires a parameter, it is specified in the parameter list on the left side of the lambda operator. Here is a simple example:

```
(n) -> (n % 2)==0
```

This lambda expression returns **true** if the value of parameter **n** is even. Although it is possible to explicitly specify the type of a parameter, such as **n** in this case, often you won't need to do so because in many cases its type can be inferred. Like a named method, a lambda expression can specify as many parameters as needed.

Functional Interfaces

As stated, a functional interface is an interface that specifies only one abstract method. If you have been programming in Java for some time, you might at first think that all interface methods are implicitly abstract. Although this was true prior to JDK 8, the situation has changed. As explained in Chapter 9, beginning with JDK 8, it is possible to specify default behavior for a method declared in an interface. This is called a *default method*. Today, an interface method is abstract only if it does not specify a default implementation. Because nondefault interface methods are implicitly abstract, there is no need to use the **abstract** modifier (although you can specify it, if you like).

Here is an example of a functional interface:

```
interface MyNumber {
  double getValue();
}
```

In this case, the method **getValue( )** is implicitly abstract, and it is the only method defined by **MyNumber**. Thus, **MyNumber** is a functional interface, and its function is defined by **getValue( )**.

As mentioned earlier, a lambda expression is not executed on its own. Rather, it forms the implementation of the abstract method defined by the functional interface that specifies its target type. As a result, a lambda expression can be specified only in a context in which a target type is defined. One of these contexts is created when a lambda expression is assigned to a functional interface reference. Other target type contexts include variable initialization, **return** statements, and method arguments, to name a few.

Let's work through an example that shows how a lambda expression can be used in an assignment context. First, a reference to the functional interface **MyNumber** is declared:

```
// Create a reference to a MyNumber instance.
MyNumber myNum;
```

Next, a lambda expression is assigned to that interface reference:

```
// Use a lambda in an assignment context.
myNum = () -> 123.45;
```

When a lambda expression occurs in a target type context, an instance of a class is automatically created that implements the functional interface, with the lambda expression defining the behavior of the abstract method declared by the functional interface. When that method is called through the target, the lambda expression is executed. Thus, a lambda expression gives us a way to transform a code segment into an object.

In the preceding example, the lambda expression becomes the implementation for the **getValue( )** method. As a result, the following displays the value 123.45:

```
// Call getValue(), which is implemented by the previously assigned
// lambda expression.
System.out.println("myNum.getValue());
```

Because the lambda expression assigned to **myNum** returns the value 123.45, that is the value obtained when **getValue( )** is called.

In order for a lambda expression to be used in a target type context, the type of the abstract method and the type of the lambda expression must be compatible. For example, if the abstract method specifies two **int** parameters, then the lambda must specify two parameters whose type either is explicitly **int** or can be implicitly inferred as **int** by the context. In general, the type and number of the lambda expression's parameters must be compatible with the method's parameters; the return types must be compatible; and any exceptions thrown by the lambda expression must be acceptable to the method.

Some Lambda Expression Examples

With the preceding discussion in mind, let's look at some simple examples that illustrate the basic lambda expression concepts. The first example puts together the pieces shown in the foregoing section.

```
// Demonstrate a simple lambda expression.

// A functional interface.
interface MyNumber {
  double getValue();
}

class LambdaDemo {
  public static void main(String args[])
  {
    MyNumber myNum;  // declare an interface reference

    // Here, the lambda expression is simply a constant expression.
    // When it is assigned to myNum, a class instance is
    // constructed in which the lambda expression implements
    // the getValue() method in MyNumber.
    myNum = () -> 123.45;
```

```
    // Call getValue(), which is provided by the previously assigned
    // lambda expression.
    System.out.println("A fixed value: " + myNum.getValue());

    // Here, a more complex expression is used.
    myNum = () -> Math.random() * 100;

    // These call the lambda expression in the previous line.
    System.out.println("A random value: " + myNum.getValue());
    System.out.println("Another random value: " + myNum.getValue());

    // A lambda expression must be compatible with the method
    // defined by the functional interface. Therefore, this won't work:
//  myNum = () -> "123.03"; // Error!
  }
}
```

Sample output from the program is shown here:

```
A fixed value: 123.45
A random value: 88.90663650412304
Another random value: 53.00582701784129
```

As mentioned, the lambda expression must be compatible with the abstract method that it is intended to implement. For this reason, the commented-out line at the end of the preceding program is illegal because a value of type **String** is not compatible with **double**, which is the return type required by **getValue( )**.

The next example shows the use of a parameter with a lambda expression:

```
// Demonstrate a lambda expression that takes a parameter.

// Another functional interface.
interface NumericTest {
 boolean test(int n);
}

class LambdaDemo2 {
  public static void main(String args[])
  {
    // A lambda expression that tests if a number is even.
    NumericTest isEven = (n) -> (n % 2)==0;

    if(isEven.test(10)) System.out.println("10 is even");
    if(!isEven.test(9)) System.out.println("9 is not even");

    // Now, use a lambda expression that tests if a number
    // is non-negative.
    NumericTest isNonNeg = (n) -> n >= 0;

    if(isNonNeg.test(1)) System.out.println("1 is non-negative");
    if(!isNonNeg.test(-1)) System.out.println("-1 is negative");
  }
}
```

The output from this program is shown here:

```
10 is even
9 is not even
1 is non-negative
-1 is negative
```

This program demonstrates a key fact about lambda expressions that warrants close examination. Pay special attention to the lambda expression that performs the test for evenness. It is shown again here:

```
(n) -> (n % 2)==0
```

Notice that the type of **n** is not specified. Rather, its type is inferred from the context. In this case, its type is inferred from the parameter type of **test( )** as defined by the **NumericTest** interface, which is **int**. It is also possible to explicitly specify the type of a parameter in a lambda expression. For example, this is also a valid way to write the preceding:

```
(int n) -> (n % 2)==0
```

Here, **n** is explicitly specified as **int**. Usually it is not necessary to explicitly specify the type, but you can in those situations that require it.

This program demonstrates another important point about lambda expressions: A functional interface reference can be used to execute any lambda expression that is compatible with it. Notice that the program defines two different lambda expressions that are compatible with the **test( )** method of the functional interface **NumericTest**. The first, called **isEven**, determines if a value is even. The second, called **isNonNeg**, checks if a value is non-negative. In each case, the value of the parameter **n** is tested. Because each lambda expression is compatible with **test( )**, each can be executed through a **NumericTest** reference.

One other point before moving on. When a lambda expression has only one parameter, it is not necessary to surround the parameter name with parentheses when it is specified on the left side of the lambda operator. For example, this is also a valid way to write the lambda expression used in the program:

```
n -> (n % 2)==0
```

For consistency, this book will surround all lambda expression parameter lists with parentheses, even those containing only one parameter. Of course, you are free to adopt a different style.

The next program demonstrates a lambda expression that takes two parameters. In this case, the lambda expression tests if one number is a factor of another.

```
// Demonstrate a lambda expression that takes two parameters.

interface NumericTest2 {
  boolean test(int n, int d);
}

class LambdaDemo3 {
```

```
  public static void main(String args[])
  {
    // This lambda expression determines if one number is
    // a factor of another.
    NumericTest2 isFactor = (n, d) -> (n % d) == 0;

    if(isFactor.test(10, 2))
      System.out.println("2 is a factor of 10");

    if(!isFactor.test(10, 3))
      System.out.println("3 is not a factor of 10");
  }
}
```

The output is shown here:

```
2 is a factor of 10
3 is not a factor of 10
```

In this program, the functional interface **NumericTest2** defines the **test( )** method:

```
boolean test(int n, int d);
```

In this version, **test( )** specifies two parameters. Thus, for a lambda expression to be compatible with **test( )**, the lambda expression must also specify two parameters. Notice how they are specified:

```
(n, d) -> (n % d) == 0
```

The two parameters, **n** and **d**, are specified in the parameter list, separated by commas. This example can be generalized. Whenever more than one parameter is required, the parameters are specified, separated by commas, in a parenthesized list on the left side of the lambda operator.

Here is an important point about multiple parameters in a lambda expression: If you need to explicitly declare the type of a parameter, then all of the parameters must have declared types. For example, this is legal:

```
(int n, int d) -> (n % d) == 0
```

But this is not:

```
(int n, d) -> (n % d) == 0
```

Block Lambda Expressions

The body of the lambdas shown in the preceding examples consist of a single expression. These types of lambda bodies are referred to as *expression bodies*, and lambdas that have expression bodies are sometimes called *expression lambdas*. In an expression body, the code on the right side of the lambda operator must consist of a single expression. While

expression lambdas are quite useful, sometimes the situation will require more than a single expression. To handle such cases, Java supports a second type of lambda expression in which the code on the right side of the lambda operator consists of a block of code that can contain more than one statement. This type of lambda body is called a *block body*. Lambdas that have block bodies are sometimes referred to as *block lambdas*.

A block lambda expands the types of operations that can be handled within a lambda expression because it allows the body of the lambda to contain multiple statements. For example, in a block lambda you can declare variables, use loops, specify **if** and **switch** statements, create nested blocks, and so on. A block lambda is easy to create. Simply enclose the body within braces as you would any other block of statements.

Aside from allowing multiple statements, block lambdas are used much like the expression lambdas just discussed. One key difference, however, is that you must explicitly use a **return** statement to return a value. This is necessary because a block lambda body does not represent a single expression.

Here is an example that uses a block lambda to compute and return the factorial of an **int** value:

```
// A block lambda that computes the factorial of an int value.

interface NumericFunc {
  int func(int n);
}

class BlockLambdaDemo {
  public static void main(String args[])
  {

    // This block lambda computes the factorial of an int value.
    NumericFunc factorial = (n) ->  {
      int result = 1;

      for(int i=1; i <= n; i++)
        result = i * result;

      return result;
    };

    System.out.println("The factoral of 3 is " + factorial.func(3));
    System.out.println("The factoral of 5 is " + factorial.func(5));
  }
}
```

The output is shown here:

```
The factorial of 3 is 6
The factorial of 5 is 120
```

In the program, notice that the block lambda declares a variable called **result**, uses a **for** loop, and has a **return** statement. These are legal inside a block lambda body. In essence, the block body of a lambda is similar to a method body. One other point. When a **return**

statement occurs within a lambda expression, it simply causes a return from the lambda. It does not cause an enclosing method to return.

Another example of a block lambda is shown in the following program. It reverses the characters in a string.

```
// A block lambda that reverses the characters in a string.

interface StringFunc {
  String func(String n);
}

class BlockLambdaDemo2 {
  public static void main(String args[])
  {

    // This block lambda reverses the characters in a string.
    StringFunc reverse = (str) ->  {
      String result = "";
      int i;

      for(i = str.length()-1; i >= 0; i--)
        result += str.charAt(i);

      return result;
    };

    System.out.println("Lambda reversed is " +
                       reverse.func("Lambda"));
    System.out.println("Expression reversed is " +
                       reverse.func("Expression"));
  }
}
```

The output is shown here:

```
Lambda reversed is adbmaL
Expression reversed is noisserpxE
```

In this example, the functional interface **StringFunc** declares the **func( )** method. This method takes a parameter of type **String** and has a return type of **String**. Thus, in the **reverse** lambda expression, the type of **str** is inferred to be **String**. Notice that the **charAt( )** method is called on **str**. This is legal because of the inference that **str** is of type **String**.

Generic Functional Interfaces

A lambda expression, itself, cannot specify type parameters. Thus, a lambda expression cannot be generic. (Of course, because of type inference, all lambda expressions exhibit some "generic-like" qualities.) However, the functional interface associated with a lambda expression can be generic. In this case, the target type of the lambda expression is

determined, in part, by the type argument or arguments specified when a functional interface reference is declared.

To understand the value of generic functional interfaces, consider this. The two examples in the previous section used two different functional interfaces, one called **NumericFunc** and the other called **StringFunc**. However, both defined a method called **func( )** that took one parameter and returned a result. In the first case, the type of the parameter and return type was **int**. In the second case, the parameter and return type was **String**. Thus, the only difference between the two methods was the type of data they required. Instead of having two functional interfaces whose methods differ only in their data types, it is possible to declare one generic interface that can be used to handle both circumstances. The following program shows this approach:

```
// Use a generic functional interface with lambda expressions.

// A generic functional interface.
interface SomeFunc<T> {
  T func(T t);
}

class GenericFunctionalInterfaceDemo {
  public static void main(String args[])
  {

    // Use a String-based version of SomeFunc.
    SomeFunc<String> reverse = (str) ->  {
      String result = "";
      int i;

      for(i = str.length()-1; i >= 0; i--)
        result += str.charAt(i);

      return result;
    };

    System.out.println("Lambda reversed is " +
                       reverse.func("Lambda"));
    System.out.println("Expression reversed is " +
                       reverse.func("Expression"));

    // Now, use an Integer-based version of SomeFunc.
    SomeFunc<Integer> factorial = (n) ->  {
      int result = 1;

      for(int i=1; i <= n; i++)
        result = i * result;

      return result;
    };

    System.out.println("The factoral of 3 is " + factorial.func(3));
    System.out.println("The factoral of 5 is " + factorial.func(5));
  }
}
```

The output is shown here:

```
Lambda reversed is adbmaL
Expression reversed is noisserpxE
The factoral of 3 is 6
The factoral of 5 is 120
```

In the program, the generic functional interface **SomeFunc** is declared as shown here:

```
interface SomeFunc<T> {
  T func(T t);
}
```

Here, **T** specifies both the return type and the parameter type of **func( )**. This means that it is compatible with any lambda expression that takes one parameter and returns a value of the same type.

The **SomeFunc** interface is used to provide a reference to two different types of lambdas. The first uses type **String**. The second uses type **Integer**. Thus, the same functional interface can be used to refer to the **reverse** lambda and the **factorial** lambda. Only the type argument passed to **SomeFunc** differs.

Passing Lambda Expressions as Arguments

As explained earlier, a lambda expression can be used in any context that provides a target type. One of these is when a lambda expression is passed as an argument. In fact, passing a lambda expression as an argument is a common use of lambdas. Moreover, it is a very powerful use because it gives you a way to pass executable code as an argument to a method. This greatly enhances the expressive power of Java.

To pass a lambda expression as an argument, the type of the parameter receiving the lambda expression argument must be of a functional interface type compatible with the lambda. Although using a lambda expression as an argument is straightforward, it is still helpful to see it in action. The following program demonstrates the process:

```
// Use lambda expressions as an argument to a method.

interface StringFunc {
  String func(String n);
}

class LambdasAsArgumentsDemo {

  // This method has a functional interface as the type of
  // its first parameter. Thus, it can be passed a reference to
  // any instance of that interface, including the instance created
  // by a lambda expression.
  // The second parameter specifies the string to operate on.
  static String stringOp(StringFunc sf, String s) {
    return sf.func(s);
  }

  public static void main(String args[])
```

```
  {
    String inStr = "Lambdas add power to Java";
    String outStr;

    System.out.println("Here is input string: " + inStr);

    // Here, a simple expression lambda that uppercases a string
    // is passed to stringOp( ).
    outStr = stringOp((str) -> str.toUpperCase(), inStr);
    System.out.println("The string in uppercase: " + outStr);

    // This passes a block lambda that removes spaces.
    outStr = stringOp((str) ->  {
                      String result = "";
                      int i;

                      for(i = 0; i < str.length(); i++)
                      if(str.charAt(i) != ' ')
                        result += str.charAt(i);

                      return result;
                    }, inStr);

    System.out.println("The string with spaces removed: " + outStr);


    // Of course, it is also possible to pass a StringFunc instance
    // created by an earlier lambda expression. For example,
    // after this declaration executes, reverse refers to an
    // instance of StringFunc.
    StringFunc reverse = (str) ->  {
      String result = "";
      int i;

      for(i = str.length()-1; i >= 0; i--)
        result += str.charAt(i);

      return result;
    };

    // Now, reverse can be passed as the first parameter to stringOp()
    // since it refers to a StringFunc object.
    System.out.println("The string reversed: " +
                       stringOp(reverse, inStr));
  }
}
```

The output is shown here:

```
Here is input string: Lambdas add power to Java
The string in uppercase: LAMBDAS ADD POWER TO JAVA
The string with spaces removed: LambdasaddpowertoJava
The string reversed: avaJ ot rewop dda sadbmaL
```

In the program, first notice the **stringOp( )** method. It has two parameters. The first is of type **StringFunc**, which is a functional interface. Thus, this parameter can receive a reference to any instance of **StringFunc**, including one created by a lambda expression. The second argument of **stringOp( )** is of type **String**, and this is the string operated on.

Next, notice the first call to **stringOp( )**, shown again here:

```
outStr = stringOp((str) -> str.toUpperCase(), inStr);
```

Here, a simple expression lambda is passed as an argument. When this occurs, an instance of the functional interface **StringFunc** is created and a reference to that object is passed to the first parameter of **stringOp( )**. Thus, the lambda code, embedded in a class instance, is passed to the method. The target type context is determined by the type of parameter. Because the lambda expression is compatible with that type, the call is valid. Embedding simple lambdas, such as the one just shown, inside a method call is often a convenient technique—especially when the lambda expression is intended for a single use.

Next, the program passes a block lambda to **stringOp( )**. This lambda removes spaces from a string. It is shown again here:

```
outStr = stringOp((str) ->  {
                    String result = "";
                    int i;

                    for(i = 0; i < str.length(); i++)
                    if(str.charAt(i) != ' ')
                      result += str.charAt(i);

                    return result;
                  }, inStr);
```

Although this uses a block lambda, the process of passing the lambda expression is the same as just described for the simple expression lambda. In this case, however, some programmers will find the syntax a bit awkward.

When a block lambda seems overly long to embed in a method call, it is an easy matter to assign that lambda to a functional interface variable, as the previous examples have done. Then, you can simply pass that reference to the method. This technique is shown at the end of the program. There, a block lambda is defined that reverses a string. This lambda is assigned to **reverse**, which is a reference to a **StringFunc** instance. Thus, **reverse** can be used as an argument to the first parameter of **stringOp( )**. The program then calls **stringOp( )**, passing in **reverse** and the string on which to operate. Because the instance obtained by the evaluation of each lambda expression is an implementation of **StringFunc**, each can be used as the first parameter to **stringOp( )**.

One last point: In addition to variable initialization, assignment, and argument passing, the following also constitute target type contexts: casts, the **?** operator, array initializers, **return** statements, and lambda expressions, themselves.

Lambda Expressions and Exceptions

A lambda expression can throw an exception. However, it if throws a checked exception, then that exception must be compatible with the exception(s) listed in the **throws** clause of the abstract method in the functional interface. Here is an example that illustrates this fact. It computes the average of an array of **double** values. If a zero-length array is passed, however, it throws the custom exception **EmptyArrayException**. As the example shows, this exception is listed in the **throws** clause of **func( )** declared inside the **DoubleNumericArrayFunc** functional interface.

```
// Throw an exception from a lambda expression.

interface DoubleNumericArrayFunc {
  double func(double[] n) throws EmptyArrayException;
}

class EmptyArrayException extends Exception {
  EmptyArrayException() {
    super("Array Empty");
  }
}

class LambdaExceptionDemo {

  public static void main(String args[]) throws EmptyArrayException
  {
    double[] values  = { 1.0, 2.0, 3.0, 4.0 };

    // This block lambda computes the average of an array of doubles.
    DoubleNumericArrayFunc average = (n) ->  {
      double sum = 0;

      if(n.length == 0)
        throw new EmptyArrayException();

      for(int i=0; i < n.length; i++)
        sum += n[i];

      return sum / n.length;
    };

    System.out.println("The average is " + average.func(values));

    // This causes an exception to be thrown.
    System.out.println("The average is " + average.func(new double[0]));
  }
}
```

The first call to **average.func( )** returns the value 2.5. The second call, which passes a zero-length array, causes an **EmptyArrayException** to be thrown. Remember, the inclusion of the **throws** clause in **func( )** is necessary. Without it, the program will not compile because the lambda expression will no longer be compatible with **func( )**.

This example demonstrates another important point about lambda expressions. Notice that the parameter specified by **func( )** in the functional interface **DoubleNumericArrayFunc** is an array. However, the parameter to the lambda expression is simply **n**, rather than **n[ ]**. Remember, the type of a lambda expression parameter will be inferred from the target context. In this case, the target context is **double[ ]**, thus the type of **n** will be **double[ ]**. It is not necessary (or legal) to specify it as **n[ ]**. It would be legal to explicitly declare it as **double[ ] n**, but doing so gains nothing in this case.

Lambda Expressions and Variable Capture

Variables defined by the enclosing scope of a lambda expression are accessible within the lambda expression. For example, a lambda expression can use an instance or **static** variable defined by its enclosing class. A lambda expression also has access to **this** (both explicitly and implicitly), which refers to the invoking instance of the lambda expression's enclosing class. Thus, a lambda expression can obtain or set the value of an instance or **static** variable and call a method defined by its enclosing class.

However, when a lambda expression uses a local variable from its enclosing scope, a special situation is created that is referred to as a *variable capture*. In this case, a lambda expression may only use local variables that are *effectively final*. An effectively final variable is one whose value does not change after it is first assigned. There is no need to explicitly declare such a variable as **final**, although doing so would not be an error. (The **this** parameter of an enclosing scope is automatically effectively final, and lambda expressions do not have a **this** of their own.)

It is important to understand that a local variable of the enclosing scope cannot be modified by the lambda expression. Doing so would remove its effectively final status, thus rendering it illegal for capture.

The following program illustrates the difference between effectively final and mutable local variables:

```
// An example of capturing a local variable from the enclosing scope.

interface MyFunc {
  int func(int n);
}

class VarCapture {
  public static void main(String args[])
  {
    // A local variable that can be captured.
    int num = 10;

    MyFunc myLambda = (n) -> {
      // This use of num is OK. It does not modify num.
      int v = num + n;

      // However, the following is illegal because it attempts
      // to modify the value of num.
//    num++;
```

```
      return v;
    };

    // The following line would also cause an error, because
    // it would remove the effectively final status from num.
//  num = 9;
  }
}
```

As the comments indicate, **num** is effectively final and can, therefore, be used inside **myLambda**. However, if **num** were to be modified, either inside the lambda or outside of it, **num** would lose its effectively final status. This would cause an error, and the program would not compile.

It is important to emphasize that a lambda expression can use and modify an instance variable from its invoking class. It just can't use a local variable of its enclosing scope unless that variable is effectively final.

Method References

There is an important feature related to lambda expressions called the *method reference*. A method reference provides a way to refer to a method without executing it. It relates to lambda expressions because it, too, requires a target type context that consists of a compatible functional interface. When evaluated, a method reference also creates an instance of the functional interface.

There are different types of method references. We will begin with method references to **static** methods.

Method References to static Methods

To create a **static** method reference, use this general syntax:

ClassName::*methodName*

Notice that the class name is separated from the method name by a double colon. The **::** is a new separator that has been added to Java by JDK 8 expressly for this purpose. This method reference can be used anywhere in which it is compatible with its target type.

The following program demonstrates a **static** method reference:

```
// Demonstrate a method reference for a static method.

// A functional interface for string operations.
interface StringFunc {
  String func(String n);
}

// This class defines a static method called strReverse().
class MyStringOps {
  // A static method that reverses a string.
  static String strReverse(String str) {
      String result = "";
```

```
      int i;

      for(i = str.length()-1; i >= 0; i--)
        result += str.charAt(i);

      return result;
  }
}

class MethodRefDemo {

  // This method has a functional interface as the type of
  // its first parameter. Thus, it can be passed any instance
  // of that interface, including a method reference.
  static String stringOp(StringFunc sf, String s) {
    return sf.func(s);
  }

  public static void main(String args[])
  {
    String inStr = "Lambdas add power to Java";
    String outStr;

    // Here, a method reference to strReverse is passed to stringOp().
    outStr = stringOp(MyStringOps::strReverse, inStr);

    System.out.println("Original string: " + inStr);
    System.out.println("String reversed: " + outStr);
  }
}
```

The output is shown here:

```
Original string: Lambdas add power to Java
String reversed: avaJ ot rewop dda sadbmaL
```

In the program, pay special attention to this line:

```
outStr = stringOp(MyStringOps::strReverse, inStr);
```

Here, a reference to the **static** method **strReverse( )**, declared inside **MyStringOps**, is passed as the first argument to **stringOp( )**. This works because **strReverse** is compatible with the **StringFunc** functional interface. Thus, the expression **MyStringOps::strReverse** evaluates to a reference to an object in which **strReverse** provides the implementation of **func( )** in **StringFunc**.

Method References to Instance Methods

To pass a reference to an instance method on a specific object, use this basic syntax:

objRef::*methodName*

As you can see, the syntax is similar to that used for a **static** method, except that an object reference is used instead of a class name. Here is the previous program rewritten to use an instance method reference:

```
// Demonstrate a method reference to an instance method

// A functional interface for string operations.
interface StringFunc {
  String func(String n);
}

// Now, this class defines an instance method called strReverse().
class MyStringOps {
  String strReverse(String str) {
      String result = "";
      int i;

      for(i = str.length()-1; i >= 0; i--)
        result += str.charAt(i);

      return result;
  }
}

class MethodRefDemo2 {

  // This method has a functional interface as the type of
  // its first parameter. Thus, it can be passed any instance
  // of that interface, including method references.
  static String stringOp(StringFunc sf, String s) {
    return sf.func(s);
  }

  public static void main(String args[])
  {
    String inStr = "Lambdas add power to Java";
    String outStr;

    // Create a MyStringOps object.
    MyStringOps strOps = new MyStringOps( );

    // Now, a method reference to the instance method strReverse
    // is passed to stringOp().
    outStr = stringOp(strOps::strReverse, inStr);

    System.out.println("Original string: " + inStr);
    System.out.println("String reversed: " + outStr);
  }
}
```

This program produces the same output as the previous version.

In the program, notice that **strReverse( )** is now an instance method of **MyStringOps**. Inside **main( )**, an instance of **MyStringOps** called **strOps** is created. This instance is used to create the method reference to **strReverse** in the call to **stringOp**, as shown again, here:

```
outStr = stringOp(strOps::strReverse, inStr);
```

In this example, **strReverse( )** is called on the **strOps** object.

It is also possible to handle a situation in which you want to specify an instance method that can be used with any object of a given class—not just a specified object. In this case, you will create a method reference as shown here:

ClassName::*instanceMethodName*

Here, the name of the class is used instead of a specific object, even though an instance method is specified. With this form, the first parameter of the functional interface matches the invoking object and the second parameter matches the parameter specified by the method. Here is an example. It defines a method called **counter( )** that counts the number of objects in an array that satisfy the condition defined by the **func( )** method of the **MyFunc** functional interface. In this case, it counts instances of the **HighTemp** class.

```
// Use an instance method reference with different objects.

// A functional interface that takes two reference arguments
// and returns a boolean result.
interface MyFunc<T> {
  boolean func(T v1, T v2);
}

// A class that stores the temperature high for a day.
class HighTemp {
  private int hTemp;

  HighTemp(int ht) { hTemp = ht; }

  // Return true if the invoking HighTemp object has the same
  // temperature as ht2.
  boolean sameTemp(HighTemp ht2) {
    return hTemp == ht2.hTemp;
  }

  // Return true if the invoking HighTemp object has a temperature
  // that is less than ht2.
  boolean lessThanTemp(HighTemp ht2) {
    return hTemp < ht2.hTemp;
  }
}

class InstanceMethWithObjectRefDemo {

  // A method that returns the number of occurrences
  // of an object for which some criteria, as specified by
  // the MyFunc parameter, is true.
  static <T> int counter(T[] vals, MyFunc<T> f, T v) {
```

```
    int count = 0;

    for(int i=0; i < vals.length; i++)
      if(f.func(vals[i], v)) count++;

    return count;
  }

  public static void main(String args[])
  {
    int count;

    // Create an array of HighTemp objects.
    HighTemp[] weekDayHighs = { new HighTemp(89), new HighTemp(82),
                                new HighTemp(90), new HighTemp(89),
                                new HighTemp(89), new HighTemp(91),
                                new HighTemp(84), new HighTemp(83) };

    // Use counter() with arrays of the class HighTemp.
    // Notice that a reference to the instance method
    // sameTemp() is passed as the second argument.
    count = counter(weekDayHighs, HighTemp::sameTemp,
                    new HighTemp(89));
    System.out.println(count + " days had a high of 89");

    // Now, create and use another array of HighTemp objects.
    HighTemp[] weekDayHighs2 = { new HighTemp(32), new HighTemp(12),
                                 new HighTemp(24), new HighTemp(19),
                                 new HighTemp(18), new HighTemp(12),
                                 new HighTemp(-1), new HighTemp(13) };

    count = counter(weekDayHighs2, HighTemp::sameTemp,
                    new HighTemp(12));
    System.out.println(count + " days had a high of 12");


    // Now, use lessThanTemp() to find days when temperature was less
    // than a specified value.
    count = counter(weekDayHighs, HighTemp::lessThanTemp,
                    new HighTemp(89));
    System.out.println(count + " days had a high less than 89");

    count = counter(weekDayHighs2, HighTemp::lessThanTemp,
                    new HighTemp(19));
    System.out.println(count + " days had a high of less than 19");
  }
}
```

The output is shown here:

```
3 days had a high of 89
2 days had a high of 12
3 days had a high less than 89
5 days had a high of less than 19
```

In the program, notice that **HighTemp** has two instance methods: **sameTemp( )** and **lessThanTemp( )**. The first returns **true** if two **HighTemp** objects contain the same temperature. The second returns **true** if the temperature of the invoking object is less than that of the passed object. Each method has a parameter of type **HighTemp** and each method returns a **boolean** result. Thus, each is compatible with the **MyFunc** functional interface because the invoking object type can be mapped to the first parameter of **func( )** and the argument mapped to **func( )**'s second parameter. Thus, when the expression

```
HighTemp::sameTemp
```

is passed to the **counter( )** method, an instance of the functional interface **MyFunc** is created in which the parameter type of the first parameter is that of the invoking object of the instance method, which is **HighTemp**. The type of the second parameter is also **HighTemp** because that is the type of the parameter to **sameTemp( )**. The same is true for the **lessThanTemp( )** method.

One other point: you can refer to the superclass version of a method by use of **super**, as shown here:

super::*name*

The name of the method is specified by *name*.

Method References with Generics

You can use method references with generic classes and/or generic methods. For example, consider the following program:

```
// Demonstrate a method reference to a generic method
// declared inside a non-generic class.

// A functional interface that operates on an array
// and a value, and returns an int result.
interface MyFunc<T> {
  int func(T[] vals, T v);
}

// This class defines a method called countMatching() that
// returns the number of items in an array that are equal
// to a specified value. Notice that countMatching()
// is generic, but MyArrayOps is not.
class MyArrayOps {
  static <T> int countMatching(T[] vals, T v) {
    int count = 0;

    for(int i=0; i < vals.length; i++)
      if(vals[i] == v) count++;

      return count;
  }
}
```

```
class GenericMethodRefDemo {

  // This method has the MyFunc functional interface as the
  // type of its first parameter. The other two parameters
  // receive an array and a value, both of type T.
  static <T> int myOp(MyFunc<T> f, T[] vals, T v) {
    return f.func(vals, v);
  }

  public static void main(String args[])
  {
    Integer[] vals = { 1, 2, 3, 4, 2, 3, 4, 4, 5 };
    String[] strs = { "One", "Two", "Three", "Two" };
    int count;

    count = myOp(MyArrayOps::<Integer>countMatching, vals, 4);
    System.out.println("vals contains " + count + " 4s");

    count = myOp(MyArrayOps::<String>countMatching, strs, "Two");
    System.out.println("strs contains " + count + " Twos");
  }
}
```

The output is shown here:

```
vals contains 3 4s
strs contains 2 Twos
```

In the program, **MyArrayOps** is a non-generic class that contains a generic method called **countMatching( )**. The method returns a count of the elements in an array that match a specified value. Notice how the generic type argument is specified. For example, its first call in **main( )**, shown here:

```
count = myOp(MyArrayOps::<Integer>countMatching, vals, 4);
```

passes the type argument **Integer**. Notice that it occurs after the **::**. This syntax can be generalized: When a generic method is specified as a method reference, its type argument comes after the **::** and before the method name. It is important to point out, however, that explicitly specifying the type argument is not required in this situation (and many others) because the type argument would have been automatically inferred. In cases in which a generic class is specified, the type argument follows the class name and precedes the **::**.

Although the preceding examples show the mechanics of using method references, they don't show their real benefits. One place method references can be quite useful is in conjunction with the Collections Framework, which is described later in Chapter 18. However, for completeness, a short, but effective, example that uses a method reference to help determine the largest element in a collection is included here. (If you are unfamiliar with the Collections Framework, return to this example after you have worked through Chapter 18.)

One way to find the largest element in a collection is to use the **max( )** method defined by the **Collections** class. For the version of **max( )** used here, you must pass a reference to the collection and an instance of an object that implements the **Comparator<T>** interface. This interface specifies how two objects are compared. It defines only one abstract method, called **compare( )**, that takes two arguments, each the type of the objects being compared. It must return greater than zero if the first argument is greater than the second, zero if the two arguments are equal, and less than zero if the first object is less than the second.

In the past, to use **max( )** with user-defined objects, an instance of **Comparator<T>** had to be obtained by first explicitly implementing it by a class, and then creating an instance of that class. This instance was then passed as the comparator to **max( )**. With JDK 8, it is now possible to simply pass a reference to a comparison method to **max( )** because doing so automatically implements the comparator. The following simple example shows the process by creating an **ArrayList** of **MyClass** objects and then finding the one in the list that has the highest value (as defined by the comparison method).

```
// Use a method reference to help find the maximum value in a collection.
import java.util.*;

class MyClass {
  private int val;

  MyClass(int v) { val = v; }

  int getVal() { return val; }
}

class UseMethodRef {
  // A compare() method compatible with the one defined by Comparator<T>.
  static int compareMC(MyClass a, MyClass b) {
    return a.getVal() - b.getVal();
  }

  public static void main(String args[])
  {
    ArrayList<MyClass> al = new ArrayList<MyClass>();

    al.add(new MyClass(1));
    al.add(new MyClass(4));
    al.add(new MyClass(2));
    al.add(new MyClass(9));
    al.add(new MyClass(3));
    al.add(new MyClass(7));

    // Find the maximum value in al using the compareMC() method.
    MyClass maxValObj = Collections.max(al, UseMethodRef::compareMC);

    System.out.println("Maximum value is: " + maxValObj.getVal());
  }
}
```

The output is shown here:

```
Maximum value is: 9
```

In the program, notice that **MyClass** neither defines any comparison method of its own, nor does it implement **Comparator**. However, the maximum value of a list of **MyClass** items can still be obtained by calling **max( )** because **UseMethodRef** defines the static method **compareMC( )**, which is compatible with the **compare( )** method defined by **Comparator**. Therefore, there is no need to explicitly implement and create an instance of **Comparator**.

Constructor References

Similar to the way that you can create references to methods, you can create references to constructors. Here is the general form of the syntax that you will use:

classname::new

This reference can be assigned to any functional interface reference that defines a method compatible with the constructor. Here is a simple example:

```
// Demonstrate a Constructor reference.

// MyFunc is a functional interface whose method returns
// a MyClass reference.
interface MyFunc {
   MyClass func(int n);
}

class MyClass {
  private int val;

  // This constructor takes an argument.
  MyClass(int v) { val = v; }

  // This is the default constructor.
  MyClass() { val = 0; }

  // ...

  int getVal() { return val; };
}

class ConstructorRefDemo {
  public static void main(String args[])
  {
    // Create a reference to the MyClass constructor.
    // Because func() in MyFunc takes an argument, new
    // refers to the parameterized constructor in MyClass,
    // not the default constructor.
    MyFunc myClassCons = MyClass::new;

    // Create an instance of MyClass via that constructor reference.
```

```
    MyClass mc = myClassCons.func(100);

    // Use the instance of MyClass just created.
    System.out.println("val in mc is " + mc.getVal( ));
  }
}
```

The output is shown here:

```
val in mc is 100
```

In the program, notice that the **func( )** method of **MyFunc** returns a reference of type **MyClass** and has an **int** parameter. Next, notice that **MyClass** defines two constructors. The first specifies a parameter of type **int**. The second is the default, parameterless constructor. Now, examine the following line:

```
MyFunc myClassCons = MyClass::new;
```

Here, the expression **MyClass::new** creates a constructor reference to a **MyClass** constructor. In this case, because **MyFunc**'s **func( )** method takes an **int** parameter, the constructor being referred to is **MyClass(int v)** because it is the one that matches. Also notice that the reference to this constructor is assigned to a **MyFunc** reference called **myClassCons**. After this statement executes, **myClassCons** can be used to create an instance of **MyClass**, as this line shows:

```
MyClass mc = myClassCons.func(100);
```

In essence, **myClassCons** has become another way to call **MyClass(int v)**.

Constructor references to generic classes are created in the same fashion. The only difference is that the type argument can be specified. This works the same as it does for using a generic class to create a method reference: simply specify the type argument after the class name. The following illustrates this by modifying the previous example so that **MyFunc** and **MyClass** are generic.

```
// Demonstrate a constructor reference with a generic class.

// MyFunc is now a generic functional interface.
interface MyFunc<T> {
   MyClass<T> func(T n);
}

class MyClass<T> {
  private T val;

  // A constructor that takes an argument.
  MyClass(T v) { val = v; }

  // This is the default constructor.
  MyClass( ) { val = null;  }

  // ...
```

```
  T getVal() { return val; };
}

class ConstructorRefDemo2 {

  public static void main(String args[])
  {
    // Create a reference to the MyClass<T> constructor.
    MyFunc<Integer> myClassCons = MyClass<Integer>::new;

    // Create an instance of MyClass<T> via that constructor reference.
    MyClass<Integer> mc = myClassCons.func(100);

    // Use the instance of MyClass<T> just created.
    System.out.println("val in mc is " + mc.getVal( ));
  }
}
```

This program produces the same output as the previous version. The difference is that now both **MyFunc** and **MyClass** are generic. Thus, the sequence that creates a constructor reference can include a type argument (although one is not always needed), as shown here:

```
MyFunc<Integer> myClassCons = MyClass<Integer>::new;
```

Because the type argument **Integer** has already been specified when **myClassCons** is created, it can be used to create a **MyClass<Integer>** object, as the next line shows:

```
MyClass<Integer> mc = myClassCons.func(100);
```

Although the preceding examples demonstrate the mechanics of using a constructor reference, no one would use a constructor reference as just shown because nothing is gained. Furthermore, having what amounts to two names for the same constructor creates a confusing situation (to say the least). However, to give you the flavor of a more practical usage, the following program uses a **static** method, called **myClassFactory( )**, that is a factory for objects of any type of **MyFunc** objects. It can be used to create any type of object that has a constructor compatible with its first parameter.

```
// Implement a simple class factory using a constructor reference.

interface MyFunc<R, T> {
   R func(T n);
}

// A simple generic class.
class MyClass<T> {
  private T val;

  // A constructor that takes an argument.
  MyClass(T v) { val = v; }
```

```
  // The default constructor. This constructor
  // is NOT used by this program.
  MyClass() { val = null; }
  // ...

  T getVal() { return val; };
}

// A simple, non-generic class.
class MyClass2 {
  String  str;

  // A constructor that takes an argument.
  MyClass2(String s) { str = s; }

  // The default constructor. This
  // constructor is NOT used by this program.
  MyClass2() { str = ""; }

  // ...

  String getVal() { return str; };
}

class ConstructorRefDemo3 {

  // A factory method for class objects. The class must
  // have a constructor that takes one parameter of type T.
  // R specifies the type of object being created.
  static <R,T> R myClassFactory(MyFunc<R, T> cons, T v) {
    return cons.func(v);
  }

  public static void main(String args[])
  {
    // Create a reference to a MyClass constructor.
    // In this case, new refers to the constructor that
    // takes an argument.
    MyFunc<MyClass<Double>, Double> myClassCons = MyClass<Double>::new;

    // Create an instance of MyClass by use of the factory method.
    MyClass<Double> mc = myClassFactory(myClassCons, 100.1);

    // Use the instance of MyClass just created.
    System.out.println("val in mc is " + mc.getVal( ));

    // Now, create a different class by use of myClassFactory().
    MyFunc<MyClass2, String> myClassCons2 = MyClass2::new;

    // Create an instance of MyClass2 by use of the factory method.
    MyClass2 mc2 = myClassFactory(myClassCons2, "Lambda");
```

```
    // Use the instance of MyClass just created.
    System.out.println("str in mc2 is " + mc2.getVal( ));
  }
}
```

The output is shown here:

```
val in mc is 100.1
str in mc2 is Lambda
```

As you can see, **myClassFactory( )** is used to create objects of type **MyClass<Double>** and **MyClass2**. Although both classes differ, for example **MyClass** is generic and **MyClass2** is not, both can be created by **myClassFactory( )** because they both have constructors that are compatible with **func( )** in **MyFunc**. This works because **myClassFactory( )** is passed the constructor for the object that it builds. You might want to experiment with this program a bit, trying different classes that you create. Also try creating instances of different types of **MyClass** objects. As you will see, **myClassFactory( )** can create any type of object whose class has a constructor that is compatible with **func( )** in **MyFunc**. Although this example is quite simple, it hints at the power that constructor references bring to Java.

Before moving on, it is important to mention a second form of the constructor reference syntax that is used for arrays. To create a constructor reference for an array, use this construct:

type[]::new

Here, *type* specifies the type of object being created. For example, assuming the form of **MyClass** as shown in the first constructor reference example (**ConstructorRefDemo**) and given the **MyArrayCreator** interface shown here:

```
interface MyArrayCreator<T> {
   T func(int n);
}
```

the following creates a two-element array of **MyClass** objects and gives each element an initial value:

```
MyArrayCreator<MyClass[]> mcArrayCons = MyClass[]::new;
MyClass[] a = mcArrayCons.func(2);
a[0] = new MyClass(1);
a[1] = new MyClass(2);
```

Here, the call to **func(2)** causes a two-element array to be created. In general, a functional interface must contain a method that takes a single **int** parameter if it is to be used to refer to an array constructor.

Predefined Functional Interfaces

Up to this point, the examples in this chapter have defined their own functional interfaces so that the fundamental concepts behind lambda expressions and functional interfaces could be clearly illustrated. However, in many cases, you won't need to define your own functional interface because JDK 8 adds a new package called **java.util.function** that

provides several predefined ones. Although we will look at them more closely in Part II, here is a sampling:

Interface	Purpose
UnaryOperator<T>	Apply a unary operation to an object of type **T** and return the result, which is also of type **T**. Its method is called **apply**( ).
BinaryOperator<T>	Apply an operation to two objects of type **T** and return the result, which is also of type **T**. Its method is called **apply**( ).
Consumer<T>	Apply an operation on an object of type **T**. Its method is called **accept**( ).
Supplier<T>	Return an object of type **T**. Its method is called **get**( ).
Function<T, R>	Apply an operation to an object of type **T** and return the result as an object of type **R**. Its method is called **apply**( ).
Predicate<T>	Determine if an object of type **T** fulfills some constraint. Return a **boolean** value that indicates the outcome. Its method is called **test**( ).

The following program shows the **Function** interface in action by using it to rework the earlier example called **BlockLambdaDemo** that demonstrated block lambdas by implementing a factorial example. That example created its own functional interface called **NumericFunc**, but the built-in **Function** interface could have been used, as this version of the program illustrates:

```
// Use the Function built-in functional interface.

// Import the Function interface.
import java.util.function.Function;

class UseFunctionInterfaceDemo {
  public static void main(String args[])
  {

    // This block lambda computes the factorial of an int value.
    // This time, Function is the functional interface.
    Function<Integer, Integer> factorial = (n) ->  {
      int result = 1;
      for(int i=1; i <= n; i++)
        result = i * result;
      return result;
    };

    System.out.println("The factoral of 3 is " + factorial.apply(3));
    System.out.println("The factoral of 5 is " + factorial.apply(5));
  }
}
```

It produces the same output as previous versions of the program.

This page has been intentionally left blank

PART II

The Java Library

CHAPTER

16 String Handling

A brief overview of Java's string handling was presented in Chapter 7. In this chapter, it is described in detail. As is the case in most other programming languages, in Java a string is a sequence of characters. But, unlike some other languages that implement strings as character arrays, Java implements strings as objects of type **String**.

Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, **String** objects can be constructed a number of ways, making it easy to obtain a string when needed.

Somewhat unexpectedly, when you create a **String** object, you are creating a string that cannot be changed. That is, once a **String** object has been created, you cannot change the characters that comprise that string. At first, this may seem to be a serious restriction. However, such is not the case. You can still perform all types of string operations. The difference is that each time you need an altered version of an existing string, a new **String** object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, Java provides two options: **StringBuffer** and **StringBuilder**. Both hold strings that can be modified after they are created.

The **String**, **StringBuffer**, and **StringBuilder** classes are defined in **java.lang**. Thus, they are available to all programs automatically. All are declared **final**, which means that none of these classes may be subclassed. This allows certain optimizations that increase performance to take place on common string operations. All three implement the **CharSequence** interface.

One last point: To say that the strings within objects of type **String** are unchangeable means that the contents of the **String** instance cannot be changed after it has been created.

However, a variable declared as a **String** reference can be changed to point at some other **String** object at any time.

The String Constructors

The **String** class supports several constructors. To create an empty **String**, call the default constructor. For example,

```
String s = new String();
```

will create an instance of **String** with no characters in it.

Frequently, you will want to create strings that have initial values. The **String** class provides a variety of constructors to handle this. To create a **String** initialized by an array of characters, use the constructor shown here:

String(char *chars*[ ])

Here is an example:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

This constructor initializes **s** with the string "abc".

You can specify a subrange of a character array as an initializer using the following constructor:

String(char *chars*[ ], int *startIndex*, int *numChars*)

Here, *startIndex* specifies the index at which the subrange begins, and *numChars* specifies the number of characters to use. Here is an example:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
```

This initializes **s** with the characters **cde**.

You can construct a **String** object that contains the same character sequence as another **String** object using this constructor:

String(String *strObj*)

Here, *strObj* is a **String** object. Consider this example:

```
// Construct one String from another.
class MakeString {
  public static void main(String args[]) {
    char c[] = {'J', 'a', 'v', 'a'};
    String s1 = new String(c);
    String s2 = new String(s1);

    System.out.println(s1);
    System.out.println(s2);
  }
}
```

The output from this program is as follows:

```
Java
Java
```

As you can see, **s1** and **s2** contain the same string.

Even though Java's **char** type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set. Because 8-bit ASCII strings are common, the **String** class provides constructors that initialize a string when given a **byte** array. Two forms are shown here:

String(byte *chrs*[ ])
String(byte *chrs*[ ], int *startIndex*, int *numChars*)

Here, *chrs* specifies the array of bytes. The second form allows you to specify a subrange. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform. The following program illustrates these constructors:

```
// Construct string from subset of char array.
class SubStringCons {
  public static void main(String args[]) {
    byte ascii[] = {65, 66, 67, 68, 69, 70 };

    String s1 = new String(ascii);
    System.out.println(s1);

    String s2 = new String(ascii, 2, 3);
    System.out.println(s2);
  }
}
```

This program generates the following output:

```
ABCDEF
CDE
```

Extended versions of the byte-to-string constructors are also defined in which you can specify the character encoding that determines how bytes are converted to characters. However, you will often want to use the default encoding provided by the platform.

NOTE The contents of the array are copied whenever you create a **String** object from an array. If you modify the contents of the array after you have created the string, the **String** will be unchanged.

You can construct a **String** from a **StringBuffer** by using the constructor shown here:

String(StringBuffer *strBufObj*)

You can construct a **String** from a **StringBuilder** by using this constructor:

String(StringBuilder *strBuildObj*)