Functional-Style Programming 13

Chapter Topics

- Declaring functional interfaces
- Defining lambda expressions
- Implementing functional interfaces using lambda expressions
- Defining and type checking lambda expressions in the context of a functional interface
- Understanding deferred execution of a lambda expression
- Understanding the implications of using class members from the enclosing class, and of using local variables from the enclosing method in a lambda expression
- Comparing lambda expressions and anonymous classes
- Understanding behavior parameterization in functional-style programing
- Using built-in functional interfaces in the `java.util.function` package: suppliers, predicates, consumers, and functions
- Composing compound operators with built-in functional interfaces
- Using specializations of built-in functional interfaces
- Using primitive type specializations of built-in functional interfaces
- Using method and constructor references
- Understanding the contexts in which lambda expressions can be defined

Java SE 17 Developer Exam Objectives	
[6.1] Use Java object and primitive Streams, including lambda expressions implementing functional interfaces, to supply, filter, map, consume, and sort data	§13.1, p. 675 *to* §13.14, p. 733

- ❍ *Lambda expressions to implement functional interfaces are covered in this chapter.*
- ❍ *For streams, see* Chapter 16, p. 879.

Java SE 11 Developer Exam Objectives	
[6.1] Implement functional interfaces using lambda expressions, including interfaces from the java.util.function package	§13.1, p. 675 *to* §13.14, p. 733

In many ways, Java 8 represented a watershed in the history of the language. Before Java 8, the language supported only object-oriented programming. Packing state and behavior into objects that communicate in a procedural manner was the order of the day. Java 8 brought *functional-style programming* into the language, where code representing *functionality* could be passed as values to tailor the *behavior* of methods.

We first look at the two language features that are the basis for functional-style programming in Java—functional interfaces and lambda expressions—before diving into a comparison of lambda expressions and anonymous classes, method and constructor references, and coverage of the built-in general-purpose generic functional interfaces provided by the java.util.function package. Subsequent chapters provide ample examples of this functional-style programming paradigm.

13.1 Functional Interfaces

Functional interfaces and lambda expressions together facilitate *behavior parameterization* (p. 691), a powerful programming paradigm that allows code representing behavior to be passed around as values, and executed when the abstract method of the functional interface is invoked. This approach is scalable, requiring only a lambda expression to implement the functional interface.

Declaring a Functional Interface

A functional interface has exactly one abstract method. This abstract method is called the *functional method* of that interface. Like any other interface, a functional interface can declare other interface members, including `static`, `default`, and `private` methods (§5.6, p. 237). Here we concentrate on the single abstract method of a functional interface. This single abstract method, like all abstract methods in an interface, is implicitly `abstract` and `public`.

The `StrPredicate` interface shown below has only one abstract method and is, by definition, a functional interface.

Click here to view code image

```
@FunctionalInterface                             // Annotation.
interface StrPredicate {
  boolean test(String str);                      // Sole public abstract method.
}
```

The annotation `@FunctionalInterface` (§25.5, p. 1579) is optional when defining functional interfaces. If the annotation is specified, the compiler will issue an error if the declaration violates the definition of a functional interface, and the Javadoc tool will also automatically generate an explanation about the functional nature of the interface. In the absence of this annotation, there is no clue from the compiler to assert that an interface is a functional interface.

The `CharSeqPredicate` interface declaration below has two abstract methods, albeit they are overloaded, but it is not a functional interface.

Click here to view code image

```
@FunctionalInterface
interface CharSeqPredicate {                              // Compile-time error!
  boolean test(String str);                               // Abstract method.
  boolean test(StringBuilder sb);                         // Abstract method.
}
```

The functional interface `NewStrPredicate` below declares only one abstract method at (1). In addition, it provides the implementations of one `default` and one `static` method at (2) and (3), respectively.

Click here to view code image

```
@FunctionalInterface
interface NewStrPredicate {
  boolean test(String str);                               // (1) Abstract method
  default void msg(String str) { System.out.println(str); } // (2) Default method
  static void info() { System.out.println("Testing!"); }    // (3) Static method
}
```

An interface can provide *explicit* `public abstract` method declarations for any of the three *non-* `final public` instance methods in the `Object` class (`equals()`, `hashCode()`, `toString()`). Including these methods explicitly in an interface should not be attempted, unless there are compelling reasons for doing so. These methods are automatically implemented by every class that implements an interface, since every class directly or indirectly inherits from the `Object` class. Therefore, such method declarations are not considered abstract in the definition of a functional interface.

In the `Funky` interface below, the first three abstract method declarations are those of the non- `final public` methods from the `Object` class. As explained above, these methods do not count toward the definition of a functional interface. Effectively, the `Funky` interface declares only one abstract method given by the last abstract method declaration.

Click here to view code image

```
@FunctionalInterface
interface Funky {
  @Override int hashCode();                          // From Object class
  @Override boolean equals(Object obj);              // From Object class
  @Override String toString();                       // From Object class
  boolean doTheFunk(Object obj);                     // Abstract method
}
```

The Java SE 17 platform API provides general-purpose functional interfaces in the `java.util.function` package that are discussed later (p. 695). Partial declaration of one such functional interface is shown below. It is the generic version of the `StrPredicate` functional interface defined earlier. The `test()` method of the `Predicate<T>` functional interface defines a `boolean`-valued predicate and evaluates it on the given object.

Click here to view code image

```
@FunctionalInterface
interface Predicate<T> {
  boolean test(T element);                           // Functional method.

  // ...
}
```

Just like any other interface, a functional interface can be implemented by a class. The following generic method takes an object of type `T` and a `Predicate<T>` and determines whether the object satisfies the predicate.

Click here to view code image

```
public static <T> boolean testPredicate(T object, Predicate<T> predicate) {
  return predicate.test(object);
}
```

Below are two implementations of the parameterized functional interface `Predicate<String>` using a concrete class and an anonymous class, respectively.

Click here to view code image

```
// Class implementation of Predicate<String>.
class PredicateTest implements Predicate<String> {
  public boolean test(String str) {
    return str.startsWith("A");               // (1)
  }
```

```
  }

  // An anonymous class implementation of Predicate<String>.
  static Predicate<String> testLength = new Predicate<>() {
    public boolean test(String str) {
      return str.length() < 4;                  // (2)
    }
  };

  // Client code:
  System.out.println(testPredicate("Anna", new PredicateTest()));   // true
  System.out.println(testPredicate("Anna", testLength));            // false
```

Note that for any combination of an object type and a predicate criteria, the `Predicate<T>` interface has to be implemented by a *new* class (or a *new* anonymous class) in order to utilize the `testPredicate()` method. Looking at the two implementations of the `Predicate<String>` above, it is essentially the expressions in the respective `return` statements at (1) and (2) that are different. This is where lambda expressions come in, allowing more concise implementation of the abstract method of a functional interface, without requiring all the boilerplate code.

Overriding Abstract Methods from Multiple Interfaces, Revisited

A general discussion on overriding abstract methods from multiple superinterfaces can be found in §11.12, p. 621. In general, an interface can inherit multiple abstract methods from its superinterfaces, but a functional interface can only have a single abstract method. Note that superinterfaces need not be functional interfaces. It is even more imperative to use the `@FunctionalInterface` annotation to catch any inadvertent errors when a functional interface inherits abstract methods from multiple superinterfaces.

In the code below, the compiler does not flag any error in the declaration of the sub-interface `ContractZ`. It is perfectly valid for the subinterface to inherit the `doIt()` method from each of its superinterfaces, as the two method declarations will be overloaded.

Click here to view code image

```
  interface ContractX { void doIt(int i); }
  interface ContractY { void doIt(double d); }
  interface ContractZ extends ContractX, ContractY {
    @Override void doIt(int d);                            // Can be omitted.
    @Override void doIt(double d);                         // Can be omitted.
  }
```

Trying to declare the subinterface `ContractZ` as a functional interface, however, does not work. The multiple abstract methods inherited by the `ContractZ` are not override-equivalent—their signatures are different—and there is no single method that can override these methods, thus disqualifying `ContractZ` as a functional interface.

Click here to view code image

```
  interface ContractX { void doIt(int i); }
  interface ContractY { void doIt(double d); }
  @FunctionalInterface interface ContractZ                 // Compile-time error!
                       extends ContractX, ContractY {

    @Override void doIt(int d);                            // Compile-time error!
    @Override void doIt(double d);                         // Compile-time error!
  }
```

In the code below, the subinterface `Features<T>` at (1) is a functional interface, as the declarations of the abstract method `flatten()` in its superinterfaces are override-equivalent. Any declaration of the abstract method `flatten()` can represent the abstract methods from the superinterfaces in the subinterface.

Click here to view code image

```
interface Feature1<R> { void flatten(List<R> plist); }
interface Feature2<S> { void flatten(List<S> plist); }
@FunctionalInterface
interface Features<T> extends Feature1<T>, Feature2<T> {   // (1)
  @Override void flatten(List<T> plist);                   // Can be omitted.
}
```

Selected Interfaces in the Java SE Platform API

A functional interface, like any other interface, can always be implemented by a class. The `@FunctionalInterface` annotation on a functional interface in the Java SE Platform API documentation indicates that the implementation of the functional interface is meant to be implemented by lambda expressions.

In the Java SE Platform API, not all interfaces with one abstract method declaration are marked with the `@FunctionalInterface` annotation. Such single-abstract-method (*SAM*) interfaces usually elicit specific behavior in objects which is best provided by the class of an object, and not by users of the class. Examples of single-abstract-method interfaces that should be implemented by classes include the following interfaces from the `java.lang` package:

- `Comparable<T>` for creating an object that can be compared with another object according to their natural ordering (§14.4, p. 761)
- `Iterable<T>` for creating an object that represents a collection in a `for(:)` loop to iterate over its elements (§15.2, p. 791)
- `AutoCloseable` for creating an object that represents a resource that is automatically closed when exiting a `try`-with-resources statement (§7.7, p. 412)
- `Readable` for creating an object that is a source for reading characters (§20.3, p. 1249)

The Java SE Platform API also has ample examples of functional interfaces that are intended to be implemented by lambda expressions. Built-in general-purpose functional interfaces provided in the `java.util.function` package are discussed in §13.4, p. 695. In addition, the following specialized functional interfaces are worth noting:

- `java.util.Comparator<T>` with the method `compare()` for defining criteria for comparing two objects according to their total ordering (§14.5, p. 769)
- `java.lang.Runnable` with the method `run()` for defining code to be executed in threads (§22.3, p. 1370)
- `java.util.concurrent.Callable<V>` with the method `call()` for defining code to be executed in threads, that can return a result and throw an exception (§23.2, p. 1423)

13.2 Lambda Expressions

Lambda expressions *implement* functional interfaces, and thereby define *anonymous functions* —that is, functions that do not have a name. They can be used as a *value* in a program, without the excess baggage of first being packaged into objects. As we shall see, these two features together facilitate *behavior parameterization*— that is, customizing method behavior by passing code as values.

A lambda expression has the following syntax:

formal_parameter_list `->` *lambda_body*

The parameter list and the body are separated by the `->` operator (a.k.a. the *function arrow*). The arrow operator has the penultimate level of precedence, only higher than the assignment operators which have the lowest precedence, and is right-associative.

The lambda expression syntax resembles a simplified declaration of a method, without many of the bells and whistles of a method declaration—omitting, in particular, any modifiers, return type, or `throws` clause specification. That is important, as it avoids verbosity and provides a simple and succinct notation to write lambda expressions on the fly.

A lambda expression can only occur in specific contexts: for example, as the value on the right-hand side of an assignment, as an argument passed in a method or constructor call, or as the value to cast with the cast operator (p. 733). Such an occurrence *defines* a lambda expression, which is *evaluated* at runtime to produce a new kind of value: *an instance of a functional interface*. The *(deferred) execution* of the lambda body only occurs at a later time when the sole abstract method of this functional interface instance is invoked.

In the rest of this section, we take a closer look at the parameter list, the lambda body, the type checking, and evaluation of lambda expressions.

Lambda Parameters

The parameter list of a lambda expression is a comma-separated list of formal parameters that is enclosed in parentheses, `()`, analogous to the parameter list in a method declaration. There is one special case where the parentheses can be omitted, which we shall get to shortly. The lambda body in the code examples in this subsection is intentionally kept simple—an empty code block, `{}`, that does nothing.

The formal parameters of a lambda expression can be declared as one of the following forms for parameter declarations:

- *Declared-type parameters*
  If the types of the parameters are explicitly specified, they are known as *declared-type parameters*. The syntax of declared-type parameters is analogous to the formal parameters in a method declaration.
  Click here to view code image

  ```
  (Integer a, String y) -> {};          // Multiple declared-type parameters
  ```

- *Inferred-type parameters*
  If the types of the parameters are not explicitly specified, they are known as *inferred-type parameters*. Types of the inferred-type parameters are derived from the context, and if necessary, from the functional type that is the target type of the lambda expression, as explained later in this section.
  Click here to view code image

  ```
  (a, b) -> {};                         // Multiple inferred-type parameters
  ```

  If the types of the parameters are explicitly specified with the reserved type name `var`, their type is inferred by local variable type inference (§3.13, p. 142). Thus the syntax of such a parameter is consistent with the syntax of a local variable declaration. We will refer to such parameters as `var`*-type inferred parameters*.
  Click here to view code image

  ```
  (var a, var b) -> {};                // Multiple var-type inferred parameters
  ```

A lambda expression with inferred parameters is called an *implicitly typed lambda expression*.

All parameters in the parameter list must conform to the same form of parameter declaration—mixing of different forms of parameter declaration is not allowed. Parentheses are mandatory with multiple parameters, whether they are declared-type or inferred-type. For a parameter list with a single inferred-type parameter, the parentheses can be omitted—but not for a declared-type or a `var`-type inferred parameter. Also, only declared-type parameters and `var`-type inferred parameters can have modifiers or annotations, like the `final` modifier or an annotation.

The code examples below illustrate the different forms of formal parameter declarations in a lambda expression.

Click here to view code image

```
() -> {};                         // Empty parameter list

// Single formal parameter:
(String str) -> {};               // Single declared-type parameter
(str)        -> {};               // Single inferred-type parameter
str          -> {};               // Single inferred-type parameter
(var str)    -> {};               // Single var-type inferred parameter

// Multiple formal parameters:
(Integer x, Integer y) -> {};     // Multiple declared-type parameters
(x, y)                 -> {};     // Multiple inferred-type parameters
(var x, var y)         -> {};     // Multiple var-type inferred parameters

// Modifiers and annotations with formal parameters:
(final int i, int j) -> {};       // Modifier with declared-type parameters
(final var i, var j) -> {};       // Modifier with var-type inferred parameters
(@NonNull int i, int j) -> {};    // Annotation with declared-type parameter
(var i, @Nullable var j)-> {};    // Annotation with var-type inferred parameter

// Parentheses are mandatory with multiple formal parameters:
String str             -> {};     // Illegal: Missing parentheses
var str                -> {};     // Illegal: Missing parentheses
Integer x, Integer y -> {};       // Illegal: Missing parentheses
x, y                   -> {};     // Illegal: Missing parentheses
var x, var y           -> {};     // Illegal: Missing parentheses

// All formal parameters must be either declared-type, inferred-type, or
// var-type inferred parameters.
(String str, j)     -> {};      // Cannot mix declared-type and inferred-type
(String str, var j) -> {};      // Cannot mix declared-type and var-type inferred
(var str, j)        -> {};      // Cannot mix var-type inferred and inferred-type

// Modifiers and annotations cannot be used with inferred-type parameters.
(final str, j)      -> {};      // No modifiers with inferred-type parameters
(str, @NonNull j)   -> {};      // No annotations with inferred-type parameters
```

Lambda Body

A lambda body is either a *single expression* or a *statement block*. Execution of a lambda body either has a non-`void` return (i.e., its evaluation returns a value), or has a `void` return (i.e., its evaluation does not return a value), or its evaluation throws an exception.

A single-expression lambda body is used for short and succinct lambda expressions. A single-expression lambda body with a `void` return type is commonly used to achieve side effects. The `return` keyword is not allowed in a single-expression lambda body.

In the examples below, the body of the lambda expressions is an *expression* whose execution returns a value—that is, has a non-`void` return.

Click here to view code image

```
() -> 2021                                   // Expression body, non-void return
() -> null                                   // Expression body, non-void return
(i, j) -> i + j                              // Expression body, non-void return
(i, j) -> i <= j ? i : j                     // Expression body, non-void return
str -> str.length() > 3                      // Expression body, non-void return
str -> str != null                           // Expression body, non-void return
       && !str.equals("") && str.length() > 3
       &&  str.equals(new StringBuilder(str).reverse().toString())
```

In the following examples, the lambda body is an *expression statement* that can have a `void` or a non-`void` return. However, if the abstract method of the functional interface does not return a value, the non-`void` return of an expression statement body can be interpreted as a `void` return—that is, the return value is ignored.

Click here to view code image

```
val -> System.out.println(val)    // Method invocation statement, void return
sb -> sb.trimToSize()             // Method invocation statement, void return
sb -> sb.append("!")              // Method invocation statement, non-void return
() -> new StringBuilder("?")      // Object creation statement, non-void return
value -> value++                  // Increment statement, non-void return
value -> value *= 2               // Assignment statement, non-void return
```

The following examples are not legal lambda expressions:

Click here to view code image

```
(int i) -> while (i < 10) ++i     // Illegal: not an expression but statement
(x, y) -> return x + y            // Illegal: return not allowed in expression
```

The statement block comprises declarations and statements enclosed in curly brackets (`{}`). The `return` statement is only allowed in a block lambda body, and the rules are the same as those in a method body: A `return` statement with an argument can only be used for a non-`void` return and its use is mandatory, whereas a `return` statement with no argument can only be used for a `void` return, but its use is optional. The `return` statement terminates the execution of the lambda body immediately.

Click here to view code image

```
() -> {}                                     // Block body, void return
() -> { return 2021; }                       // Block body, non-void return
() -> { return 2021 }      // Illegal: statement terminator (;) in block missing
() -> { new StringBuilder("Go nuts."); }            // Block body, void return
() -> { return new StringBuilder("Go nuts!"); }     // Block body, non-void return
(int i) -> { while (i < 10) ++i; }                  // Block body, void return
(i, j) -> { if (i <= j) return i; else return j; } // Block body, non-void return
(done) -> {                      // Multiple statements in block body, void return
  if (done) {

    System.out.println("You deserve a break!");
    return;
  }
  System.out.println("Stay right here!");
}
```

Type Checking and Execution of Lambda Expressions

A lambda expression can only be defined in a context where a functional interface can be used: for example, in an assignment context, a method call context, or a cast context (p. 733). The compiler determines the *target type* that is required in the context where the lambda expression is defined. This target type is always a functional interface type. In the assignment context below, the target type is `Predicate<Integer>`, as it is the target of the assignment statement. Note that the type parameter `T` of the functional interface is `Integer`.

Click here to view code image

```
Predicate<Integer> p1 = i -> i % 2 == 0;  // (1) Target type: Predicate<Integer>
```

The *method type* of a method declaration comprises its type parameters, formal parameter types, return type, and any exceptions the method throws.

The *function type* of a functional interface is the method type of its functional method. The target type `Predicate<Integer>` has the following method, where type parameter `T` is `Integer`:

Click here to view code image

```
public boolean test(Integer t);          // Method type: Integer -> boolean
```

The function type of the target type `Predicate<Integer>` is the method type of the this `test()` method:

```
Integer -> boolean
```

The type of the lambda expression defined in a given context must be compatible with the function type of the target type. If the lambda expression has inferred-type parameters, their type is inferred from the context, and if necessary, from the function type. From the lambda body at (1), it is possible to infer that the type of `i` is `int` (from the expression `i % 2`), and the lambda body evaluates to a `boolean` value (from the evaluation of the `==` operator). Just from the context, it is thus possible to infer that the type of the lambda expression at (1) is:

```
int -> boolean
```

From the function type of the target type `Predicate<Integer>`, the compiler is able to determine that if the inferred `int` type of the parameter `i` in the lambda expression at (1) is promoted to `Integer`, the type of the lambda expression at (1) would then be compatible with the function type of the target type `Predicate<Integer>`, that is:

```
Integer -> boolean
```

The lambda expression defined at (1) above is equivalent to the following implementation using an anonymous class.

Click here to view code image

```
Predicate<Integer> p1 = new Predicate<>() {  // Anonymous class
  public boolean test(Integer i) {
    return i % 2 == 0;
  }
};
```

It is possible to determine that the type of the lambda expression at (1) is compatible with the method type of the only abstract method `test()` defined by the `Predicate<T>` functional interface because it is not ambiguous which method the lambda expression should implement in the functional interface. However, this would not work for an interface that had more than one abstract method, as it would not be clear which abstract method to implement.

In the following assignment, the target type is `java.util.function.IntPredicate`:

Click here to view code image

```
IntPredicate p2 = i -> i % 2 == 0;        // (2) Target type: IntPredicate
```

The `IntPredicate` functional interface has the following abstract method:

Click here to view code image

```
public boolean test(int i);              // Method type: int -> boolean
```

The function type of the target type `IntPredicate` is the method type of its abstract method:

```
int -> boolean
```

The compiler infers that the type of the inferred-type parameter `i` in the lambda expression at (2) should be `int`. As the lambda body returns a `boolean` value, the type of the lambda expression at (2) is

```
int -> boolean
```

The type of the lambda expression is compatible with the function type of the target type `IntPredicate`.

Note that in both examples, the lambda expression is the same, but their types are different in the two contexts. They represent two different values. The type of a lambda expression is determined by the context in which it is defined.

Click here to view code image

```
System.out.println(p1 == p2);            // false
```

The process of type checking a lambda expression in a given context is called *target typing*. The presentation here is simplified, but suffices for our purpose to give an idea of what is involved.

The compiler does the type checking for using lambda expressions. The runtime environment provides the rest of the magic to make it all work. At runtime, the lambda expression is executed when the sole abstract method of the functional interface is invoked.

Click here to view code image

```
boolean result1 = p1.test(2021);          // false
boolean result2 = p2.test(2020);          // true
```

As mentioned earlier, this is an example of deferred execution. Lambda execution is similar to invoking a method on an object. We define a lambda expression as a *function* and use it like a *method*, letting the compiler and the runtime environment put it all together.

Accessing Members in an Enclosing Class

Just like nested blocks, a lambda expression has *lexical or block scope*—that is, names used in a lambda expression are resolved *lexically* in the local context in which the lambda expression is defined.

A lambda expression does *not* inherit names of members declared in the functional interface it implements, which obviously then cannot be accessed in the lambda body.

Since a lambda expression is not a class, there is no notion of the `this` reference. If the `this` reference is used in a lambda expression in a non-static context, it refers to the *enclosing object*, and can be used to access members of this object. The name of a member in the enclosing object has the same meaning when used in a lambda expression. In other words, there are no restrictions to accessing members declared in the enclosing object. In the case of shadowing member names by local variables, the keyword `this` can be explicitly used, and also the keyword `super` to access any members inherited by the enclosing object.

In **Example 13.1**, the `getPredicate()` method at (7) defines a lambda expression at (8). This lambda expression is defined in a non-static context (i.e., an instance method). This lambda expression accesses the `static` field `strList` and the instance field `banner` in the enclosing class at (1) and (2), respectively.

In the `main()` method in **Example 13.1**, an `ArrayList<String>` is assigned to the `static` field `strList` at (3) and is initialized. The `ArrayList<String>` referred to by the `static` field `strList` has the following content:

```
[Tom, Dick, Harriet]
```

A `MembersOnly` object is created at (4). Its `StringBuilder` field `banner` is initialized with the string `"love "`. The local variable `obj` declared at (4) refers to this `MembersOnly` object. At (5), a `Predicate<String>` instance is instantiated by calling the `getPredicate()` method on the `MembersOnly` object referred to by the local variable `obj`. This predicate is first executed when the `test()` method is called at (6) on the `Predicate<String>` instance, with the argument string `"never dies!"`. Calling the `test()` method results in the lambda expression created at (5) by the `getPredicate()` method to be executed in the context of the enclosing `MembersOnly` object referred to by the local variable `obj`.

The parameter `str` of the lambda expression is initialized with the string `"never dies!"`—that is, the argument to the `test()` method. In the body of the lambda expression, the `ArrayList<String>` referred to by the `static` field `strList` in the `MembersOnly` class is first printed at (9):

Click here to view code image

```
List: [Tom, Dick, Harriet]
```

At (10), the parameter `str` (with contents `"never dies!"`) is appended to the `StringBuilder` (with contents `"love "`) referred to by the instance field `banner` in the enclosing object, resulting in the following contents in this `StringBuilder`:

```
"love never dies!"
```

Since the length of the string `"never dies!"`, referred to by the parameter `str`, is greater than 5, the lambda expression returns `true` at (11). This is the value returned by the `test()` method call at (6).

In the call to the `println()` method at (6), the argument

Click here to view code image

```
p.test("never dies!") + " " + obj.banner
```

now evaluates as

Click here to view code image

```
true + " " + "love never dies!"
```

Example 13.1 *Accessing Members in the Enclosing Object*

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class MembersOnly {

  // Instance variable
  private StringBuilder banner;                                  // (1)

  // Static variable
  private static List<String> strList;                           // (2)

  // Constructor
  public MembersOnly(String str) {
    banner = new StringBuilder(str);
  }

  // Static method
  public static void main(String[] args) {
    strList = new ArrayList<>();                                 // (3)
    strList.add("Tom"); strList.add("Dick"); strList.add("Harriet");

    MembersOnly obj = new MembersOnly("love ");                  // (4)
    Predicate<String> p = obj.getPredicate();                    // (5)
    System.out.println(p.test("never dies!") + " " + obj.banner);  // (6)
  }

  // Instance method
  public Predicate<String> getPredicate() {                // (7)
    return str -> {                                        // (8)  Lambda expression
      System.out.println("List: " + MembersOnly.strList);// (9)  Static field
      this.banner.append(str);                             // (10) Instance field
      return str.length() > 5;                             // (11) boolean value
    };
  }
}
```

Output from the program:

Click here to view code image

```
List: [Tom, Dick, Harriet]
true love never dies!
```

Accessing Local Variables in the Enclosing Context

All variable declarations in a lambda expression follow the rules of *block scope*. They are not accessible outside the lambda expression. It also means that we cannot *redeclare* local variables already declared in the enclosing scope. In **Example 13.2**, redeclaring the parameter `banner` and the local variable `words` at (6) and (7), respectively, in the body of the lambda expression results in a compile-time error. Local variables declared in the enclosing method, including its formal parameters, can be accessed in a lambda expression provided they are *effectively final*. This means that once a local variable has been assigned a value to the time when it is used, its value has not changed during that time. Using the `final` modifier in the declaration of a local variable explicitly instructs the compiler to ensure that this is the case. The `final` modifier implies effectively final. If the `final` modifier is omitted and a local variable is used in a lambda expression, the compiler effectively performs the same analysis as if the `final` modifier had been specified.

A lambda expression may be executed at a later time, after the method in which it is defined has finished execution. At that point, the local variables in its enclosing context that are used in the lambda expression are no longer accessible. To ensure their availability, *copies of their values* are maintained with the lambda expression. This is called *variable capture*, although in essence it is the values that are captured. Note that it is not the object that is copied in the case of a local reference variable, but the reference value. Objects reside on the heap and are accessible via a copy of the reference value. Correct execution of the lambda expression is guaranteed, since these effectively final values cannot change. Note that the state of an object referred to by a `final` or an effectively final reference can change, but not the reference value stored in the reference—that is, such a reference will continue to refer to the same object once it is initialized.

In **Example 13.2**, the method `getPredicate()` at (1) has one formal parameter (`banner`), and a local variable (`words`) declared at (2). Although the state of the `Array-List<String>` object, referred to by the reference `words`, is changed in the method (we add elements to it), the reference value in the reference does not—that is, it continues to refer to the same object whose reference value it was assigned at (2). The parameter `banner` is assigned the reference value of the argument object when the method is invoked, and continues to refer to this object throughout the method. Both local variables are effectively final. Their values are captured by the lambda expression, and used when the lambda expression is executed at a later time after the call to the `getPredicate()` method in the `main()` method has completed.

However, if we uncomment (3) and (4) in **Example 13.2**, then both local variables are not effectively final. Their reference values are changed at (3) and (4), respectively. The compiler now flags an error at (8) and (9), respectively because these non-final local variables are used in the lambda expression.

The vigilant reader no doubt will have noticed that no requirement of effectively final was imposed on the field members of the enclosing class or object, when accessed in the lambda body. Reference to the enclosing object or class is captured by the lambda expression, and when the expression is executed at a later time, the reference can readily be used to access values of any fields referenced in the lambda body—no copies of such values need be made, thereby making the effectively final rule unnecessary for accessing fields in the enclosing context.

Example 13.2 *Accessing Local Variables in the Enclosing Method*

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;
```

```
public class LocalsOnly {

  public static void main(String[] args) {
    StringBuilder banner = new StringBuilder("love ");
    LocalsOnly instance = new LocalsOnly();
    Predicate<String> p = instance.getPredicate(banner);
    System.out.println(p.test("never dies!") + " " + banner);
  }

  public Predicate<String> getPredicate(StringBuilder banner) {   // (1)
    List<String> words = new ArrayList<>();                       // (2)
    words.add("Tom"); words.add("Dick"); words.add("Harriet");

//  banner = new StringBuilder();         // (3) Illegal: Not effectively final
//  words = new ArrayList<>();            // (4) Illegal: Not effectively final

    return str -> {                                 // (5) Lambda expression
//    String banner = "Don't redeclare me!";        // (6) Illegal: Redeclared
//    String[] words = new String[6];               // (7) Illegal: Redeclared
      System.out.println("List: " + words);         // (8) Local variable
      banner.append(str);                           // (9) Parameter
      return str.length() > 5;
    };
  }
}
```

Output from the program:

Click here to view code image

```
List: [Tom, Dick, Harriet]
true love never dies!
```

13.3 Lambda Expressions and Anonymous Classes

As we have seen in this chapter so far, both anonymous classes and lambda expressions can be used to provide implementation of functional interfaces. Example 13.3 illustrates using both anonymous classes and lambda expressions for this purpose.

A common operation on elements in a collection is to select those elements that satisfy certain criteria. This operation is called *filtering*. Given a list of words, we would like to filter this list for one-word *palindromes*—that is, words spelled the same forward and backward. For example, `"anana"` is a palindrome, but `"banana"` is not.

Example 13.3 defines the generic method `filterList()` at (7) for filtering a list. Its first parameter is a list of type `T` to filter, and the filtering criteria is given by the second parameter which is a generic functional interface. This functional interface, `Predicate<T>`, specifies the `boolean` abstract method `test(T t)`. The argument passed to the method must implement the `Predicate<T>` interface, supplying the implementation of the `boolean` method `test()` that actually determines if an element satisfies the criteria. The `test()` method is an example of a *predicate*—that is, a function that takes an argument and returns a `boolean` value.

The following `boolean` expressions are used to determine whether a word (given by the reference `str`) is a case-sensitive or case-insensitive palindrome, respectively:

Click here to view code image

```
str.equals(new StringBuilder(str).reverse().toString())
str.equalsIgnoreCase(new StringBuilder(str).reverse().toString())
```

Whether a string is a palindrome is determined by comparing the string for equality with the result of reversing the string using a `StringBuilder`.

Example 13.3 *Filtering an* `ArrayList<E>`

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class FunWithPalindromes {

  public static void main(String[] args) {

    List<String> words = new ArrayList<>();
    words.add("Otto"); words.add("ADA"); words.add("Alyla");
    words.add("Bob"); words.add("HannaH"); words.add("Java");
    System.out.println("List of words:                " + words);

    System.out.println("-----------Using Anonymous Classes---------------------");
    // Use an anonymous class to filter for palindromes (case sensitive).  (1)
    List<String> palindromesA = filterList(words,
        new Predicate<String>() {
          @Override public boolean test(String str) {
            return str.equals(new StringBuilder(str).reverse().toString());
          }
        }
    );
    System.out.println("Case-sensitive palindromes:   " + palindromesA);

    // Use an anonymous class to filter for palindromes (case insensitive). (2)
    List<String> palindromesB = filterList(words,
        new Predicate<String>() {
          @Override public boolean test(String str) {
            return str.equalsIgnoreCase(
                           new StringBuilder(str).reverse().toString());
          }
        }
    );
    System.out.println("Case-insensitive palindromes: " + palindromesB);
    System.out.println("-----------Using Lambda Expressions--------------------");
    Predicate<String> predicate1 = str ->
        str.equals(new StringBuilder(str).reverse().toString());          // (3)
    List<String> palindromes1 = filterList(words, predicate1);            // (4)
    System.out.println("Case-sensitive palindromes:   " + palindromes1);

    Predicate<String> predicate2 = str ->
        str.equalsIgnoreCase(new StringBuilder(str).reverse().toString());// (5)
    List<String> palindromes2 = filterList(words, predicate2);            // (6)
    System.out.println("Case-insensitive palindromes: " + palindromes2);
  }

  /**
   * Filters a list according to the criteria of the predicate.
   * @param list       List to filter
   * @param predicate  Provides the criteria for filtering the list
   * @return           List of elements that match the criteria
   */
  public static <E> List<E> filterList(List<E> list,                      // (7)
                                     Predicate<E> predicate) {
    List<E> result = new ArrayList<>();
    for (E element : list) {
      if (predicate.test(element)) {                                      // (8)
```

```
            result.add(element);
          }
        }
        return result;
      }
    }
```

Output from the program:

Click here to view code image

```
List of words:                [Otto, ADA, Alyla, Bob, HannaH, Java]
-----------Using Anonymous Classes---------------------
Case-sensitive palindromes:   [ADA, HannaH]
Case-insensitive palindromes: [Otto, ADA, Alyla, Bob, HannaH]
-----------Using Lambda Expressions--------------------
Case-sensitive palindromes:   [ADA, HannaH]
Case-insensitive palindromes: [Otto, ADA, Alyla, Bob, HannaH]
```

Filtering Criteria Defined by Anonymous Classes

Example 13.3 uses anonymous classes to instantiate the criteria object, as shown at (1) and (2). The basic idea is that we can both declare and instantiate the class at the same time, where it is needed in the code, and in our case, as an argument in the call to the `filterList()` method. The type parameter `E` in this case is `String`. The anonymous classes at (1) and (2) provide implementations of the `test()` method for strings. The method is called at (8) to determine whether a `String` fulfills the selection criteria.

By using anonymous classes, we have avoided creating separate concrete classes, but the verbosity of declaring anonymous classes to encapsulate a single method is inescapable. And we still have to declare a new anonymous class for each selection criterion, duplicating a lot of boilerplate code.

Filtering Criteria Defined by Lambda Expressions

Ideally we would like to pass the code for the selection criteria as an argument to the `filterList()` method so that the method can apply the criteria to the elements in the list—that is, be able to change the behavior of the `filterList()` method depending on the selection criteria. This is an example of *behavior parameterization*.

Knowing that something is a `Predicate<T>`, all the information about the abstract method it implements can be inferred, as it is the only abstract method in the interface: its name, its parameters, any value it returns, and whether it throws any exceptions.

The assignment at (3) in Example 13.3 uses a lambda expression to provide an implementation for the parameterized `Predicate<String>` functional interface:

Click here to view code image

```
Predicate<String> predicate1 = str ->
        str.equals(new StringBuilder(str).reverse().toString());          // (3)
```

The reference `predicate1` on the left-hand side is of type `Predicate<String>`, and it is assigned the value of the lambda expression on the right-hand side.

The lambda expression at (3) defines an anonymous function that takes a `String` as the only parameter, and returns a `boolean` value. Its type is `String -> boolean`. Recall that the `test()` method of the `Predicate<String>` functional interface type does exactly that. The function type of the `Predicate<String>` functional interface is also `String -> boolean`. The compiler can type check that the lambda expression is assignable to the reference on the left-hand side, since the expression represents an anonymous function that is compatible with the sole abstract method `test()` of the parameterized `Predicate<String>` interface.

The lambda expression at (3) is passed as an argument to the `filterList()` method via the reference `predicate1` at (4). It is only executed when the `test()` method is called with a `String` argument in the `filterList()` method at (8).

The anonymous class declaration at (1):

Click here to view code image

```
new Predicate<String>() {
  @Override public boolean test(String str) {
    return str.equals(new StringBuilder(str).reverse().toString());
  }
}
```

is implemented by the lambda expression at (3):

Click here to view code image

```
str -> str.equals(new StringBuilder(str).reverse().toString())
```

Now we need only pass a new lambda expression to the `filterList()` method to filter a list based on selection criteria. Using lambda expressions is more precise, concise, and readable than using anonymous classes.

Later, when we discuss *streams* (Chapter 16, p. 879), we will also do away with the `filterList()` method for filtering lists.

Lambda Expressions versus Anonymous Classes

Implementation

A lambda expression can only be used to provide implementation of exactly one functional interface. It represents an anonymous function. Unlike an object, it has only behavior and no state.

An anonymous class is restricted to either implementing one interface or extending one class, but it is not restricted to implementing only one abstract method from its supertype.

No separate class file with Java bytecode is created for a lambda expression, in contrast to a separate class file for each anonymous class declared in a source file.

Scope

Lambda expressions do not introduce a new naming scope, and follow the lexical scope rules for nested blocks. Names in a lambda expression are resolved lexically in its enclosing block and enclosing class.

An anonymous class introduces a new naming scope, where names are resolved according to the inheritance hierarchy of the anonymous class, its local enclosing block, and its enclosing class.

Accessing Inherited Members from the Inheritance Hierarchy

Members in the functional interface implemented by a lambda expression are not accessible in the lambda body.

Since an anonymous class can declare and inherit members from its supertype, instances of anonymous classes can have *state*. The `this` and `super` references can be used to access members in the current instance of the anonymous class and its superclass object, respectively.

Accessing Local Declarations in the Enclosing Block

An anonymous class and a lambda expression can only access effectively final local variables in their enclosing local context.

A local variable in a lambda expression cannot shadow a local variable with the same name in the local context because local variables cannot be redeclared. A field name in the anonymous class can shadow a local variable with the same name in the local context.

Accessing Members in the Enclosing Class

A local variable in a lambda expression and a member (either inherited or declared) in an anonymous class can hide a member with the same name in the enclosing class.

A lambda expression and an anonymous class can access any non-hidden members in the enclosing class by their simple names.

In a non-static context, a lambda expression and an anonymous class can access any hidden members in the enclosing class by the `this` reference and the *qualified* `this` reference, respectively.

In a static context, a lambda expression and an anonymous class can access any hidden members in the enclosing class by their qualified names.

Best Practices

Defining an anonymous class can be verbose. Even an implementation of a single method requires a lot of boilerplate code to encapsulate the method in a class definition, with the added risk of making the code hard to read and understand.

The obvious choice for implementing functional interfaces is lambda expressions. Anything beyond that, and there is little choice but to bring in the anonymous classes.

Review Questions

13.1 Which statement is true about functional interfaces and lambda expressions?

Select the one correct answer.

a. A functional interface can only be implemented by lambda expressions.

b. A functional interface declaration can have only one method declaration.

c. In the body of a lambda expression, only `public` members in the enclosing class can be accessed.

d. In the body of a lambda expression, all local variables in the enclosing scope can be accessed.

e. None of the above

13.2 Which of the following statements are true about the following code?

Click here to view code image

```
interface Funky1    { void    absMethod1(String s); }
interface Funky2    { String  absMethod2(String s); }

public class RQ12A99 {
  public static void main(String[] args) {

    Funky1 p1;
    p1 = s -> System.out.println(s);            // (1)
    p1 = s -> s.length();                       // (2)
    p1 = s -> s.toUpperCase();                  // (3)
    p1 = s -> { s.toUpperCase(); };             // (4)
    p1 = s -> { return s.toUpperCase(); };      // (5)

    Funky2 p2;
    p2 = s -> System.out.println(s);            // (6)
    p2 = s -> s.length();                       // (7)
    p2 = s -> s.toUpperCase();                  // (8)
    p2 = s -> { s.toUpperCase(); };             // (9)
    p2 = s -> { return s.toUpperCase(); };      // (10)
  }
}
```

Select the four correct answers.

a. (1) will fail to compile.

b. (2) will fail to compile.

c. (3) will fail to compile.

d. (4) will fail to compile.

e. (5) will fail to compile.

f. (6) will fail to compile.

g. (7) will fail to compile.

h. (8) will fail to compile.

i. (9) will fail to compile.

j. (10) will fail to compile.

13.3 Which statement is true about the following code?

Click here to view code image

```
interface AgreementA { void doIt(); }
interface AgreementB extends AgreementA {}
interface AgreementC extends AgreementB {
  void doIt();
  boolean equals(Object obj);
}
```

```
  class Beta implements AgreementB {
    public void doIt() {
      System.out.print("Jazz|");
    }
  }

  public class RQ12A999 {
    public static void main(String[] args) {
      AgreementA a = () -> System.out.print("Java|");          // (1)
      AgreementB b = () -> System.out.print("Jive|");          // (2)
      AgreementC c = () -> System.out.print("Jingle|");        // (3)
      Object o = a = c;                                        // (4)
      b = new Beta();                                          // (5)
      a.doIt();                                                // (6)
      b.doIt();                                                // (7)
      c.doIt();                                                // (8)
      ((AgreementA) o).doIt();                                 // (9)
    }
  }
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will throw a `ClassCastException`.

c. The program will print `Jingle|Jingle|Jazz|Jingle|`.

d. The program will print `Jingle|Jazz|Jingle|Jingle|`.

e. The program will print `Jingle|Jingle|Jingle|Jazz|`.

13.4 Overview of Built-In Functional Interfaces

Earlier in this chapter, specialized interfaces (including some functional ones) were mentioned that are readily available in the Java SE Platform API (p. 678). To facilitate defining common functions with lambda expressions, the Java SE Platform API also provides a versatile set of functional interfaces for this purpose.

The main support for functional interfaces is found in the `java.util.function` package. The general-purpose generic functional interfaces shown in **Table 13.1** represent the four basic operations that are commonly implemented by functions: to *get* a value (`Supplier<T>`), to *test* a predicate (`Predicate<T>`), to *accept* a value but not return a result (`Consumer<T>`), and to *apply* a function to a value in order to compute a new result (`Function<T, R>`).

The term *arity* refers to the number of arguments that a method requires as its input. A method is called *zero-arity*, *one-arity*, or *two-arity*, depending on whether the method has *zero*, *one*, or *two* arguments, respectively. Depending on whether the functional method is zero-arity, one-arity, or two-arity, its functional interface is likewise referred to as zero-arity, one-arity, or two-arity, respectively. Note that the arity of the functional interface reflects the arity of its functional method. Particularly for a generic functional interface, the arity should *not* be confused with the number of type parameters specified for the generic functional interface.

In **Table 13.1**, except for the `Supplier<T>` functional interface which has a zero-arity functional method, the functional methods for the other three basic functional interfaces are one-arity methods. Accordingly, the `Supplier<T>` functional interface is a zero-arity functional interface, whereas the other functional interfaces are onearity functional interfaces.

Table 13.1 *Basic Functional Interfaces in the* `java.util.function` *Package*

Functional interface (`T` and `R` are type parameters)	Functional method	Function	Arity of function type
`Supplier<T>`	`get: () -> T`	Provide an instance of a `T`.	Zero-arity
`Predicate<T>`	`test: T -> boolean`	Evaluate a predicate on a `T`.	One-arity
`Consumer<T>`	`accept: T -> void`	Perform action on a `T`.	One-arity
`Function<T, R>`	`apply: T -> R`	Transform a `T` to an `R`.	One-arity

It is important to understand the abstract operations that the basic functional interfaces provide before tackling the specialized versions of these functional interfaces in the `java.util.function` package. Since the package provides a wide range of functional interfaces for various purposes, defining new ones should hardly be necessary.

The complete list of all built-in functional interfaces in the `java.util.function` package is given in **Table 13.2**. The table also shows any `default` methods that a built-in functional interface defines. The idea is not to memorize them all, but to understand how they are categorized according to the four basic functional interfaces in **Table 13.1**. The specialized versions of the basic functional interfaces are derived by combining one or more of the following three forms:

- *Two-arity specializations of the basic functional interfaces*
  These functional interfaces (`BiPredicate<T,U>`, `BiConsumer<T,U>`, `BiFunction<T,U,R>`) are two-arity specialized counterparts to the corresponding basic functional interface, except for the `Supplier<T>` interface which does not have a two-arity specialization.
- *Extended versions of the* `Function<T,R>` *and* `BiFunction<T,U,R>` *interfaces*
  The functional interfaces `UnaryOperator<T>` and `BinaryOperator<T>` *extend* the `Function<T,T>` and `BiFunction<T,T,T>` interfaces, respectively. As their names imply, the two specialized functional interfaces `UnaryOperator<T>` and `BinaryOperator<T>` are one-arity and two-arity functional interfaces as their superinterfaces, respectively, *where the parameters and the result in each have the same type*.
- *Primitive type specializations of generic functional interfaces*
  The primitive type specializations avoid excessive boxing and unboxing of primitive values when such values are used as objects.
  The primitive type counterparts are specializations of each generic functional interface where one or more type parameters are replaced by a primitive type. Primitive type specializations primarily involve one or more of the primitive types `int`, `long`, or `double`.
  The naming scheme uses one or more prefixes in front of the name of a primitive type functional interface to indicate its function type—that is, the type of the parameters and that of the result. For example, `IntPredicate` has the function type `int -> boolean`, whereas `IntToDoubleFunction` has the function type `int -> double`, and `LongBinaryOperator` has the function type `(long, long) -> long`.

Table 13.2 *Built-In Functional Interfaces in the* `java.util.function` *Package*

Functional interface (`T, U`, and `R` are type parameters)	Functional method	Default methods unless otherwise

		indicated
`Supplier<T>`	`get:         () -> T`	–
`IntSupplier`	`getAsInt:     () -> int`	–
`LongSupplier`	`getAsLong:    () -> long`	–
`DoubleSupplier`	`getAsDouble:  () -> double`	–
`BooleanSupplier`	`getAsBoolean: () -> boolean`	–
`Predicate<T>`	`test: T -> boolean`	`and(), or(), negate(), static isEqual(), static not()`
`IntPredicate`	`test: int -> boolean`	`and(), or(), negate()`
`LongPredicate`	`test: long -> boolean`	`and(), or(), negate()`
`DoublePredicate`	`test: double -> boolean`	`and(), or(), negate()`
`BiPredicate<T, U>`	`test: (T, U) -> boolean`	`and(), or(), negate()`
`Consumer<T>`	`accept: T -> void`	`andThen()`
`IntConsumer`	`accept: int -> void`	`andThen()`
`LongConsumer`	`accept: long -> void`	`andThen()`
`DoubleConsumer`	`accept: double -> void`	`andThen()`
`BiConsumer<T, U>`	`accept: (T, U) -> void`	`andThen()`
`ObjIntConsumer<T>`	`accept: (T, int) -> void`	–
`ObjLongConsumer<T>`	`accept: (T, long) -> void`	–
`ObjDoubleConsumer<T>`	`accept: (T, double) -> void`	–
`Function<T, R>`	`apply: T -> R`	`compose(), andThen(), static identity()`
`IntFunction<R>`	`apply: int -> R`	–
`LongFunction<R>`	`apply: long -> R`	–
`DoubleFunction<R>`	`apply: double -> R`	–

`ToIntFunction<T>`	`applyAsInt: T -> int`	–
`ToLongFunction<T>`	`applyAsLong: T -> long`	–
`ToDoubleFunction<T>`	`applyAsDouble: T -> double`	–
`IntToLongFunction`	`applyAsLong: int -> long`	–
`IntToDoubleFunction`	`applyAsDouble: int -> double`	–
`LongToIntFunction`	`applyAsInt: long -> int`	–
`LongToDoubleFunction`	`applyAsDouble: long -> double`	–
`DoubleToIntFunction`	`applyAsInt: double -> int`	–
`DoubleToLongFunction`	`applyAsLong: double -> long`	–
`BiFunction<T, U, R>`	`apply: (T, U) -> R`	`andThen()`
`ToIntBiFunction<T, U>`	`applyAsInt: (T, U) -> int`	–
`ToLongBiFunction<T, U>`	`applyAsLong: (T, U) -> long`	–
`ToDoubleBiFunction<T, U>`	`applyAsDouble: (T, U) -> double`	–
`UnaryOperator<T> extends Function<T,T>`	`apply: T -> T`	`compose()`, `andThen()`, `static identity()`
`IntUnaryOperator`	`applyAsInt: int -> int`	`compose()`, `andThen()`
`LongUnaryOperator`	`applyAsLong: long -> long`	`compose()`, `andThen()`
`DoubleUnaryOperator`	`applyAsDouble: double -> double`	`compose()`, `andThen()`
`BinaryOperator<T> extends BiFunction<T,T,T>`	`apply: (T, T) -> T`	`andThen()`, `static maxBy()`, `static minBy()`
`IntBinaryOperator`	`applyAsInt: (int, int) -> int`	–
`LongBinaryOperator`	`applyAsLong: (long, long) -> long`	–

`DoubleBinaryOperator`	`applyAsDouble: (double, double) -> double`	–

The columns in **Table 13.3** list the built-in functional interfaces in the `java.util.function` package according to each category of basic functional interface.

Table 13.3 *Summary of Built-In Functional Interfaces*

`Supplier<T>`	**`Predicate<T>`**	**`Consumer<T>`**	**`Function<T, R>`**	**`UnaryOperat`**
`IntSupplier`	`IntPredicate`	`IntConsumer`	`IntFunction<R>`	`IntUnaryOper`
`LongSupplier`	`LongPredicate`	`LongConsumer`	`LongFunction<R>`	`LongUnaryOpe`
`DoubleSupplier`	`DoublePredicate`	`DoubleConsumer`	`DoubleFunction<R>`	`DoubleUnary`
`BooleanSupplier`			`ToIntFunction<T>`	
			`ToLongFunction<T>`	
			`ToDouble-Function<T>`	
			`IntToLong-Function`	
			`IntToDouble-Function`	
			`LongToInt-Function`	
			`LongToDouble-Function`	
			`DoubleToInt-Function`	
			`DoubleToLong-Function`	
	`BiPredicate<T,U>`	**`BiConsumer<T,U>`**	**`BiFunction<T,U,R>`**	**`BinaryOperat`**
		`ObjIntConsumer<T>`	`ToIntBiFunction<T,U>`	`IntBinaryOpe`
		`ObjLongConsumer<T>`	`ToLongBiFunction<T,U>`	`LongBinaryO`
		`ObjDoubleConsumer<T>`	`ToDoubleBiFunction<T,U>`	`DoubleBinary`

13.5 Suppliers

As the name suggests, the `Supplier<T>` functional interface represents a supplier of values. From **Table 13.4**, we see that its functional method `get()` has the type `() -> T`—that is, it takes no argument and returns a value of type `T`.

Table 13.4 shows all supplier functional interfaces provided in the `java.util.function` package. Apart from the functional method shown in **Table 13.4**, these functional interfaces do not define any additional methods.

Table 13.4 *Suppliers*

Functional interface (`T`, `U`, and `R` are type parameters)	Functional method	Default methods
`Supplier<T>`	`get: () -> T`	–
`IntSupplier`	`getAsInt: () -> int`	–
`LongSupplier`	`getAsLong: () -> long`	–
`DoubleSupplier`	`getAsDouble: () -> double`	–
`BooleanSupplier`	`getAsBoolean: () -> boolean`	–

A supplier typically generates, creates, or produces values. Example 13.4 illustrates defining and using suppliers.

The supplier at (1) in Example 13.4 will always create a `StringBuilder` from the string `"Howdy"`. The `StringBuilder` is not created until the `get()` method of the supplier is called.

Click here to view code image

```
Supplier<StringBuilder> createSB = () -> new StringBuilder("Howdy!");   // (1)
System.out.println(createSB.get());                          // Prints: Howdy!

String str = "uppercase me!";
Supplier<String> makeUppercase = () -> str.toUpperCase();              // (2)
System.out.println(makeUppercase.get());              // Prints: UPPERCASE ME!
```

The supplier at (2) returns a string that is an uppercase version of the string on which the method `toUppercase()` is invoked. Note that the value of the reference `str` is captured at (2) when the lambda expression is defined and the reference `str` is effectively final. Calling the `get()` method of the supplier results in the `toUppercase()` method being invoked on the `String` instance referenced by the reference `str`.

In the examples below, we use a pseudorandom number generator to define a supplier that can return integers between different ranges. The `intSupplier` below generates a number between 0 (inclusive) and 100 (exclusive).

Click here to view code image

```
Random numGen = new Random();

Supplier<Integer> intSupplier = () -> numGen.nextInt(100); // numGen effect. final
System.out.println(intSupplier.get());             // Prints a number in [0, 100).
```

The generic method `listBuilder()` at (12) can be used to build a list of specified length, where the specified `supplier` generates a value every time the `get()` method is called at (13).

The code below builds a list of `Integer` with five values between 0 (inclusive) and 100 (exclusive) by calling the `listBuilder()` method.

Click here to view code image

```
List<Integer> intRefList = listBuilder(5, () -> numGen.nextInt(100));
```

Primitive Type Specializations of `Supplier<T>`

The primitive type versions of the generic supplier interface are appropriately named with a prefix to indicate the type of primitive value returned by their functional methods. For example, the integer supplier is named `IntSupplier`. Their functional methods are also appropriately named with a postfix to indicate the type of value they return. For example, the functional method of the `IntSupplier` interface is named `getAsInt`. These primitive type versions are *not* subinterfaces of the generic `Supplier<T>` interface. `BooleanSupplier` is the only specialization with a primitive type other than `int`, `long`, or `double` in the `java.util.function` package.

Example 13.4 also illustrates defining and using suppliers that return primitive values. Non-generic `int` suppliers are used in the following examples, without the overhead of boxing and unboxing `int` values. Calling the `getAsInt()` method results in the lambda expression to be executed.

Click here to view code image

```
IntSupplier intSupplier2 = () -> numGen.nextInt(100);
System.out.println(intSupplier2.getAsInt());   // Prints a number in [0, 100).
```

The method `roleDice()` at (14) prints statistics of rolling a many-sided dice a specified number of times using an `IntSupplier` as a dice roller. In the call below, a six-sided dice is rolled 100,000 times using the specified `int` supplier that generates numbers from 1 to 6.

Click here to view code image

```
roleDice(6, 100_000, () -> 1 + numGen.nextInt(6));
```

The reader is encouraged to work through Example 13.4, as it provides additional examples of defining and using suppliers.

Example 13.4 *Implementing Suppliers*

Click here to view code image

```
import java.time.LocalTime;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;
import java.util.function.DoubleSupplier;
import java.util.function.IntSupplier;
import java.util.function.Supplier;

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

    Supplier<StringBuilder> createSB = () -> new StringBuilder("Howdy!");   // (1)
    System.out.println(createSB.get());                                  // Prints: Howdy!

    String str = "uppercase me!";
    Supplier<String> makeUppercase = () -> str.toUpperCase();              // (2)
    System.out.println(makeUppercase.get());                 // Prints: UPPERCASE ME!
```

```
  // Pseudorandom number generator captured and used in lambda expressions: (3)
  Random numGen = new Random();

  // Generate a number between 0 (inclusive) and 100 (exclusive):            (4)
  Supplier<Integer> intSupplier = () -> numGen.nextInt(100);
  System.out.println(intSupplier.get());         // Prints a number in [0, 100).

  // Build a list of Integers with values between 0 (incl.) and 100 (excl.): (5)
  List<Integer> intRefList = listBuilder(5, () -> numGen.nextInt(100));
  System.out.println(intRefList);

  // Build a list of StringBuilders:                                         (6)
  List<StringBuilder> stringbuilderList = listBuilder(6,
      () -> new StringBuilder("str" + numGen.nextInt(10)));           // [0, 10)
  System.out.println(stringbuilderList);

  // Build a list that has the same string:                                  (7)
  List<String> stringList2 = listBuilder(4, () -> "Mini me");
  System.out.println(stringList2);

  // Build a list of LocalTime:                                              (8)
  List<LocalTime> dateList1 = listBuilder(3, () -> LocalTime.now());
  System.out.println(dateList1);

  // Generate a number between 0 (inclusive) and 100 (exclusive):            (9)
  IntSupplier intSupplier2 = () -> numGen.nextInt(100);
  System.out.println(intSupplier2.getAsInt());   // Prints a number in [0, 100).

  // Role many-sided dice:                                                  (10)
  roleDice(6, 100_000, () -> 1 + numGen.nextInt(6));
  roleDice(8, 1_000_000, () -> 1 + (int) (Math.random() * 8));

  // Build an array of doubles with values
  // between 0.0 (incl.) and 5.0 (excl.):                                   (11)
  DoubleSupplier ds = () -> Math.random() * 5;                     // [0.0, 5.0)
  double[] dArray = new double[4];
  for (int i = 0; i < dArray.length; i++) {
    dArray[i] = ds.getAsDouble();
  }
  System.out.println(Arrays.toString(dArray));
}

/**
 * Creates a list whose elements are supplied by a Supplier<T>.
 * @param num       Number of elements to put in the list.
 * @param supplier  Supplier that supplies a value to put in the list
 * @return          List created by the method
 */
public static <T> List<T> listBuilder(int num, Supplier<T> supplier) {   // (12)
  List<T> list = new ArrayList<>();
  for (int i = 0; i < num; ++i) {
    list.add(supplier.get());                                            // (13)
  }
  return list;
}

/**
 * Print statistics of rolling a many-sided dice the specified              (14)
 * number of times using an IntSupplier as dice roller.
 */
public static void roleDice(int numOfSides, int numOfTimes,
                            IntSupplier diceRoller) {
  int[] frequency = new int[numOfSides + 1];         // frequency[0] is ignored.
  for (int i = 0; i < numOfTimes; i++) {
```

```
            ++frequency[diceRoller.getAsInt()];                               // (15)
        }
        System.out.println(Arrays.toString(frequency));
    }
}
```

Probable output from the program:

Click here to view code image

```
Howdy!
UPPERCASE ME!
83
[15, 24, 48, 3, 16]
[str8, str0, str6, str8, str6, str7]
[Mini me, Mini me, Mini me, Mini me]
[15:32:05.707, 15:32:05.707, 15:32:05.707]
54
[0, 16747, 16723, 16701, 16607, 16637, 16585]
[0, 124918, 124385, 125038, 125451, 124618, 124600, 125230, 125760]
[0.2129971975975531, 0.6933477140020566, 1.3559818256541756, 1.183773498854187]
```

13.6 Predicates

The `Predicate<T>` interface should be familiar by now, having been used earlier in Example 13.1, Example 13.2, and Example 13.3.

The `Predicate<T>` interface defines a `boolean`-valued function in terms of an instance of its type parameter `T`. From Table 13.5, we see that its functional method `test()` has the type `T -> boolean`—that is, it takes an argument of type `T` and returns a value of type `boolean`.

Table 13.5 shows all the predicate functional interfaces provided in the `java.util.function` package. In addition to the three primitive type predicates recognized by their characteristic prefixes, there is also a generic two-arity specialization (`BiPredicate<T,U>`). Apart from the functional method `test()` shown for each predicate in Table 13.5, these functional interfaces also define `default` methods. Neither the primitive type predicates nor the two-arity predicate are subinterfaces of the `Predicate<T>` interface.

Table 13.5 *Predicates*

Functional interface (`T`, `U`, and `R` are type parameters)	Functional method	Default methods unless otherwise indicated
`Predicate<T>`	`test: T -> boolean`	`and(), or(), negate(), static isEqual(), static not()`
`IntPredicate`	`test: int -> boolean`	`and(), or(), negate()`
`LongPredicate`	`test: long -> boolean`	`and(), or(), negate()`
`DoublePredicate`	`test: double -> boolean`	`and(), or(), negate()`

`BiPredicate<T, U>`	`test: (T, U) -> boolean`	`and(), or(), negate()`

The code below illustrates the `removeIf()` method from the `ArrayList<E>` class that requires a predicate and removes all elements from the list that satisfy the predicate. All palindromes are removed from the list denoted by the reference `words`. The method call `element.test(isPalindrome)` is invoked on each element of the list by the `removeIf()` method.

Click here to view code image

```
Predicate<String> isPalindrome
    = str -> new StringBuilder(str).reverse().toString().equalsIgnoreCase(str);
// Before: [Otto, ADA, Alyah, Bob, HannaH, Java]
words.removeIf(isPalindrome);          // Remove all palindromes.
// After:  [Alyah, Java]
```

We can equally implement the `Predicate<T>` functional interface passed as an argument to the `ArrayList.removeIf()` method using an anonymous class:

Click here to view code image

```
// Before: [Otto, ADA, Alyah, Bob, HannaH, Java]
words.removeIf(new Predicate<String>() {
  public boolean test(String str) {
    return new StringBuilder(str).reverse().toString().equalsIgnoreCase(str);
  }
});
// After:  [Alyah, Java]
```

Similarly, the code below removes all words with even length from the list.

Click here to view code image

```
Predicate<String> isEvenLen = str -> str.length() % 2 == 0;
// Before: [Otto, ADA, Alyah, Bob, HannaH, Java]
words.removeIf(isEvenLen);             // Remove all even length words.
// After:  [ADA, Alyah, Bob]
```

And in this example, all words starting with `"A"` are removed from the list.

Click here to view code image

```
Predicate<String> startsWithA = str -> str.startsWith("A");
// Before: [Otto, ADA, Alyah, Bob, HannaH, Java]
words.removeIf(startsWithA);           // Remove all words that start with "A".
// After:  [Otto, Bob, HannaH, Java]
```

Composing Predicates

The predicate interfaces define `default` methods to compose *compound* predicates—that is, to chain together predicates with logical AND and OR operations.

Click here to view code image

```
default Predicate<T> negate()
```

Returns a predicate that represents the logical negation of this predicate.

Click here to view code image

```
default Predicate<T> and(Predicate<? super T> other)
```

Returns a composed predicate that represents a short-circuiting logical AND of this predicate and the predicate specified by the `other` parameter.

The `other` predicate is only evaluated if this predicate is `true`. Any exceptions thrown during the evaluation of either predicate are conveyed to the caller.

Click here to view code image

```
default Predicate<T> or(Predicate<? super T> other)
```

Returns a composed predicate that represents a short-circuiting logical OR of this predicate and the predicate specified by the `other` parameter.

The `other` predicate is only evaluated if this predicate is `false`. Any exceptions thrown during the evaluation of either predicate are conveyed to the caller.

Click here to view code image

```
static <T> Predicate<T> not(Predicate<? super T> target)
```

This `static` method returns a predicate that is the negation of the specified predicate.

Click here to view code image

```
static <T> Predicate<T> isEqual(Object targetRef)
```

This `static` method returns a predicate that tests if the argument in the call to the `test()` method and the `targetRef` object are equal—for example, `Predicate.isEqual("Aha").test("aha")`. In contrast to the `Object.equals()` method, this method returns `true` if both the argument and the `targetRef` object are `null`.

At (1) below, the `isPalindrome` predicate is negated to define a predicate that tests if a string is *not* a palindrome.

The method calls in a compound predicate are executed from *left to right* with *short-circuit* evaluation of the predicates. The compound predicate `x.or(y).and(z)` at (2) is evaluated as `((x.or(y)).and(z))`, where `x`, `y`, and `z` are constituent predicates `isEvenLen`, `startsWithA`, and `isNotPalindrome`, respectively. The `or()` method is executed first, followed by the `and()` method. However, the `startsWithA` predicate is only evaluated if the `isEvenLen` predicate was `false`—that is, the `or()` method tests the `isEvenLen` predicate first, but not the `startsWithA` predicate unless the first one is `false`. Analogously, the `isNotPalindrome` predicate is only evaluated if the predicate on which the `and()` method is invoked is `true`. Note that the *same* argument is passed to all the constituent predicates that comprise a compound predicate. Schematically, the evaluation of the method call `composedPredicate.test("Adda")` at (3) proceeds as follows:

Click here to view code image

```
  ((x.or(y)).and(z))
= ((true.or(y)).and(z))
= (true.and(z))
= (true.and(false))
= false
// A string that is not a palindrome.
Predicate<String> isNotPalindrome = isPalindrome.negate();       // (1)

// A string with even length or starts with an 'A', and is not a palindrome.
Predicate<String> composedPredicate
    = isEvenLen.or(startsWithA).and(isNotPalindrome);            // (2)
System.out.println("Using composed predicate on \"Adda\": "
        + composedPredicate.test("Adda"));                       // (3) false.

// A string with even length, or it starts with an 'A' and is not a palindrome.
Predicate<String> conditionalOperators
    = str -> str.length() % 2 == 0 || str.startsWith("A")        // (4)
          && !(new StringBuilder(str).reverse().toString().equalsIgnoreCase(str));
System.out.println("Using conditional operators on \"Adda\": "
        + conditionalOperators.test("Adda"));                    // (5) true.
```

The evaluation should be contrasted with the predicate at (4) that is defined with the negation operator `!` and the short-circuit conditional operators `||` and `&&`. Note that the evaluation order is different for the conditional operators because of the precedence rules: `a || b && !c` is evaluated as `(a || (b && (!c)))`, where `a`, `b`, and `c` are `boolean` expressions `str.length() % 2 == 0`, `str.startsWith("A")`, and `(new StringBuilder(str).reverse().toString().equalsIgnoreCase(str))`, respectively. Schematically, the evaluation of the method call `conditionalOperators.test("Adda")` at (5) proceeds as follows:

```
  (a || (b && (!c)))
= (a || (b && (!true)))
= (a || (b && false))
= (a || (false))
= (true || false)
= true
```

Using equality predicates is illustrated by the following examples:

Click here to view code image

```
Predicate<String> isEqualToTarget = Predicate.isEqual("Ada");
System.out.println(isEqualToTarget.test("Adda"));           // false.
System.out.println(Predicate.isEqual("Ada").test("Ada"));   // true.
System.out.println(Predicate.isEqual("null").test("null")); // true.
```

Primitive Type Specializations of `Predicate<T>`

The functional interfaces `IntPredicate`, `LongPredicate`, and `DoublePredicate` evaluate predicates with `int`, `long`, and `double` arguments, respectively, avoiding the overhead of boxing and unboxing of primitive values (see Table 13.5). The primitive type versions are *not* subinterfaces of the `Predicate<T>` interface.

Click here to view code image

```
Predicate<Integer> isEven = i -> i % 2 == 0;          // Operand unboxed.
System.out.println("2021 is an even number: "
                 + isEven.test(2021));                // Argument boxed. false.
```

```
IntPredicate isEvenInt = i -> i % 2 == 0;            // No unboxing.
System.out.println("2021 is an even number: "
                  + isEvenInt.test(2021));           // No boxing. false.
```

Each primitive type version also provides methods for negating a predicate and composing predicates using the methods for short-circuiting logical AND and OR operations.

Click here to view code image

```
IntPredicate isOddInt = isEvenInt.negate();          // Negating a predicate.
System.out.println("2020 is an odd number: "
                  + isOddInt.test(2020));            // false.

IntPredicate isInRange = i -> -100 <= i && i <= 100; // Range: [-100, 100]
System.out.println("21 is in range and odd: "
                 + isInRange.and(isOddInt).test(21));// true.
```

Two-Arity Specialization of `Predicate<T>`: `BiPredicate<T, U>`

The `BiPredicate<T, U>` interface is a two-arity specialization of the `Predicate<T>` interface. From **Table 13.5**, we see that its functional method `test()` has the type `(T, U) -> boolean` —that is, it takes two arguments of type `T` and `U`, and returns a `boolean` value. There are no primitive type specializations of the `BiPredicate<T, U>` interface. **Example 13.5** illustrates defining and using two-arity predicates. The following two-arity predicate tests if an element is a member (or is contained) in a list. The reference `filenames` refers to a list of file names.

Click here to view code image

```
BiPredicate<String, List<String>> isMember
    = (element, list) -> list.contains(element);
System.out.println(isMember.test("X-File4.doc", filenames));  // true.
```

The two-arity predicate below determines if a file name has an extension from a specified set of file extensions.

Click here to view code image

```
BiPredicate<String, Set<String>> selector = (filename, extensions) ->
    extensions.contains(filename.substring(filename.lastIndexOf('.')));
System.out.println(selector.test("Y-File.pdf", extSet));      // true.
```

Determining the file extension is generalized in **Example 13.5** to a list of file names using the generic method `filterList()` which takes three parameters: a list of file names, a set of file extensions, and a two-arity predicate to do the selection. In the method `filterList()`, for each element in the list, the following method call is executed: `selector.test(element, extSet)`.

The `BiPredicate<T, U>` interface also defines `default` methods to compose compound two-arity predicates. As expected, the `or()` and the `and()` methods require a two-arity predicate as an argument. A simple example is given in **Example 13.5** to check if the product or the sum of two numbers is equal to a given number:

Click here to view code image

```
int number = 21;
BiPredicate<Integer, Integer> isProduct = (i, j) -> i * j == number;
```

```
BiPredicate<Integer, Integer> isSum     = (i, j) -> i + j == number;
System.out.println(isProduct.or(isSum).test(7, 3));       // true.
```

Example 13.5 *Implementing the* `BiPredicate<T, U>` *Functional Interface*

Click here to view code image

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.function.BiPredicate;

public class BiPredicateClient {

  public static void main(String[] args) {

    // List with filenames:
    List<String> filenames = new ArrayList<>();
    filenames.add("X-File1.pdf"); filenames.add("X-File2.exe");
    filenames.add("X-File3.fm"); filenames.add("X-File4.doc");
    filenames.add("X-File5.jpg"); filenames.add("X-File6.jpg");
    System.out.println("Filenames: " + filenames);

    // BiPredicate for membership in a list.
    BiPredicate<String, List<String>> isMember =
        (element, list) -> list.contains(element);
    System.out.println(isMember.test("X-File4.doc", filenames));  // true.
    // Set with file extensions:
    Set<String> extSet = new HashSet<>();
    extSet.add(".pdf"); extSet.add(".jpg");
    System.out.println("Required extensions: " + extSet);

    // BiPredicate to determine if a filename has an extension from a specified
    // set of file extensions.
    BiPredicate<String, Set<String>> selector = (filename, extensions) ->
        extensions.contains(filename.substring(filename.lastIndexOf('.')));
    System.out.println(selector.test("Y-File.pdf", extSet));      // true.

    List<String> result = filterList(filenames, extSet, selector);
    System.out.println("Files with required extensions: " + result);

    int number = 21;
    BiPredicate<Integer, Integer> isProduct = (i, j) -> i * j == number;
    BiPredicate<Integer, Integer> isSum     = (i, j) -> i + j == number;
    System.out.println(isProduct.or(isSum).test(7, 3));
  }

  /**
   * Filters a list according to the criteria of the selector.
   * @param list       List to filter
   * @param extSet     Set of file extensions
   * @param selector   BiPredicate that provides the criteria for filtering
   * @return           List of elements that match the criteria
   */
  public static <E, F> List<E> filterList(List<E> list,
                                          Set<F> extSet,
                                          BiPredicate<E, Set<F>> selector) {
    List<E> result = new ArrayList<>();
    for (E element : list)
      if (selector.test(element, extSet))
        result.add(element);
    return result;
```

```
    }
  }
```

Output from the program:

Click here to view code image

```
Filenames: [X-File1.pdf, X-File2.exe, X-File3.fm, X-File4.doc, X-File5.jpg, X-File6.jpg]
true
Required extensions: [.pdf, .jpg]
true
Files with required extensions: [X-File1.pdf, X-File5.jpg, X-File6.jpg]
true
```

13.7 Consumers

The `Consumer<T>` functional interface represents a consumer of values. From **Table 13.6**, we see that its functional method `accept()` has the type `T -> void`—that is, it takes an argument of type `T` and returns no value (`void`). Typically, it performs some operation on its argument object.

Table 13.6 shows all the consumer functional interfaces, together with their functional method `accept()` and any `default` methods that are provided by the interface. There are three primitive type one-arity specializations of the `Consumer<T>` functional interface, recognized by their characteristic prefixes. The generic two-arity specialization (`BiConsumer<T,U>`) also has three two-arity primitive type specializations. Only the one-arity consumers and the two-arity generic consumer define the `default` method `andThen()`.

Table 13.6 *Consumers*

Functional interface (`T`, `U`, and `R` are type parameters)	Functional method	Default methods
`Consumer<T>`	`accept: T -> void`	`andThen()`
`IntConsumer`	`accept: int -> void`	`andThen()`
`LongConsumer`	`accept: long -> void`	`andThen()`
`DoubleConsumer`	`accept: double -> void`	`andThen()`
`BiConsumer<T, U>`	`accept: (T, U) -> void`	`andThen()`
`ObjIntConsumer<T>`	`accept: (T, int) -> void`	–
`ObjLongConsumer<T>`	`accept: (T, long) -> void`	–
`ObjDoubleConsumer<T>`	`accept: (T, double) -> void`	–

Generally, a consumer performs an operation on its argument object, but does not return a value. The formatter below prints a `double` value with two decimal places. The type of the lambda expression is `Double -> void`, and the lambda expression is executed when the method `accept()` is invoked.

Click here to view code image

```
Consumer<Double> formatter = d -> System.out.printf("Value: %.2f%n", d);
formatter.accept(3.145);                         // Value: 3.15
```

In the code below, the `resizeSB` consumer resizes a `StringBuilder` to length 4—a more flexible resizer is presented a little later. The `reverseSb` consumer reverses the contents of a `StringBuilder`. The `printSB` consumer prints a `StringBuilder`. In each case, the type of the lambda expression is `StringBuilder -> void`.

Click here to view code image

```
StringBuilder sb1 = new StringBuilder("Banana");
Consumer<StringBuilder> resizeSB = sb -> sb.setLength(4);
resizeSB.accept(sb1);                            // Bana
Consumer<StringBuilder> reverseSB = sb -> sb.reverse();
reverseSB.accept(sb1);                           // anaB

Consumer<StringBuilder> printSB
    = sb -> System.out.println("StringBuilder: " + sb);
printSB.accept(sb1);                             // StringBuilder: anaB
```

The `ArrayList.forEach()` method requires a consumer that is applied to each element of the list—that is, the method call `consumer.accept(element)` is executed on each element in the list. The consumer below prints an element of the list `words` in lowercase.

Click here to view code image

```
// [Otto, ADA, Alya, Bob, HannaH, Java]
words.forEach(s -> System.out.print(s.toLowerCase() + " "));
// otto ada alya bob hannah java
```

The code below implements the `Consumer<String>` interface using an anonymous class which is passed as an argument to the `ArrayList.forEach()` method:

Click here to view code image

```
// [Otto, ADA, Alya, Bob, HannaH, Java]
words.forEach(new Consumer<String>() {
  public void accept(String s) {

    System.out.print(s.toLowerCase() + " ");
  }
});
// otto ada alya bob hannah java
```

The following consumer prints each element of the list `words` that has an even length:

Click here to view code image

```
// [Otto, ADA, Alya, Bob, HannaH, Java]
words.forEach(s -> {if (s.length() % 2 == 0) System.out.print(s + " ");});
// Otto Alya HannaH Java
```

Composing Consumers

The method `andThen()` can be used to chain together consumers to compose compound consumers. The three consumers used earlier to resize, reverse, and print a `StringBuilder` can be chained together as seen here:

Click here to view code image

```
resizeSB.andThen(reverseSB)
    .andThen(printSB).accept(new StringBuilder("Banana")); // StringBuilder: anaB
```

The constituent consumers are executed one after the other from *left to right*. Note that the reference to the `StringBuilder` instance passed to the `accept()` method is also passed to each constituent consumer.

Click here to view code image

```
default Consumer<T> andThen(Consumer<? super T> after)
```

Returns a composed `Consumer` that evaluates this operation first followed by the specified `after` operation.

Any exception thrown during the evaluation of either operation aborts the evaluation of the composed operation and the exception is conveyed to the caller.

Primitive Type Specializations of `Consumer<T>`

From **Table 13.6**, we see that the non-generic functional interfaces `IntConsumer`, `LongConsumer`, and `DoubleConsumer` define the functional method `accept()` that takes an `int`, a `long`, or a `double` value as an argument, respectively, but does not return a value (`void`). The primitive type versions avoid the overhead of boxing and unboxing of primitive values. They are *not* subinterfaces of the `Consumer<T>` interface, and they all provide the `andThen()` method to compose compound primitive type consumers.

The two `IntConsumer`s below print the square root and the square of their argument, respectively. They are chained by the `andThen()` method that requires an `IntConsumer`.

Click here to view code image

```
IntConsumer sqrt = i -> System.out.printf("%.2f%n", Math.sqrt(i));
IntConsumer sqr = i -> System.out.printf("%d%n", i * i);
sqrt.andThen(sqr).accept(15);    // 3.87
                                 // 225
```

Two-Arity Specialization of `Consumer<T>`: `BiConsumer<T, U>`

The `BiConsumer<T, U>` interface is a two-arity specialization of the `Consumer<T>` interface. From **Table 13.6**, we see that its functional method `accept()` has the type `(T, U) -> void`—that is, it takes two arguments of type `T` and type `U`, and does not return a value (`void`). The `BiConsumer<T, U>` interface provides the `andThen()` method to create compound two-arity consumers.

The following code illustrates defining and using two-arity consumers.

Click here to view code image

```
BiConsumer<String, Double> formatPrinter
    = (format, obj) -> System.out.printf(format, obj);
formatPrinter.accept("Math.PI:|%10.3f|%n", Math.PI); // Math.PI:|     3.142|
```

The `java.util.Map.forEach()` method requires a two-arity consumer that is applied to each entry `(key, value)` in the map—that is, the method call `biconsumer.accept(key, value)` is executed for each entry in the map. The two-arity consumer below formats and prints all entries in the map given by the reference `strLenMap`. The key in this map is of type `String` and the value is of type `Integer` — that is, `HashMap<String, Integer>`. The value is the length of the key string.

Click here to view code image

```
// Map entries (default format): {Java=4, Bob=3, Otto=4, HannaH=6, Alya=4, ADA=3}
strLenMap.forEach((key, value) -> System.out.printf("(%s:%d) ", key, value));
// (Java:4) (Bob:3) (Otto:4) (HannaH:6) (Alya:4) (ADA:3)
```

Primitive Type Specializations of `BiConsumer<T, U>`

Table 13.6 shows the generic functional interfaces `ObjIntConsumer<T>`, `ObjLongConsumer<T>`, and `ObjDoubleConsumer<T>` that are specializations of the `BiConsumer<T, U>` interface. The functional method `accept()` of these primitive type specializations takes two arguments: One is an object of type `T` and the other is a primitive value. These functional interfaces are *not* subinterfaces of the `BiConsumer<T, U>` interface, and they do not provide `default` methods to chain consumers.

The code below shows a new version of resizing a `StringBuilder` written earlier, where the required length was hard-coded in the lambda expression definition and could not be changed. Using an `ObjIntConsumer<StringBuilder>`, the required length can be passed as a parameter in the definition of the lambda expression, as shown below:

Click here to view code image

```
ObjIntConsumer<StringBuilder> resizeSB2 = (sb, len) -> sb.setLength(len);
StringBuilder sb2 = new StringBuilder("bananarama");
resizeSB2.accept(sb2, 6);      // The required length passed as a parameter.
System.out.println("StringBuilder resized: " + sb2);
// StringBuilder resized: banana
```

13.8 Functions

The `Function<T, R>` interface represents a function or an operation that transforms an argument object to a result object, where the object types need not be the same. From Table 13.7, we see that its functional method `apply()` has the type `T -> R`—that is, it takes an argument of type `T` and returns a result of type `R`.

Table 13.7 also shows specialized functions for primitive types, together with their functional methods. They do not define any `default` methods.

The `BiFunction<T, U, R>` interface and its primitive type versions are discussed in §13.9, p. 717. The specialized versions `UnaryOperator<T>` and `BinaryOperator<T>`, which provide functions where the arguments and the result are of the same type, are discussed in §13.10, p. 720, and §13.11, p. 721, respectively.

Table 13.7 *Functions*

Functional interface (`T`, `U`, and `R` are type parameters)	Functional method	Default methods unlesss otherwise
`Function<T, R>`	`apply: T -> R`	`compose()`, `andThen()`, `static identity()`
`IntFunction<R>`	`apply: int -> R`	–
`LongFunction<R>`	`apply: long -> R`	–
`DoubleFunction<R>`	`apply: double -> R`	–
`ToIntFunction<T>`	`applyAsInt: T -> int`	–
`ToLongFunction<T>`	`applyAsLong: T -> long`	–
`ToDoubleFunction<T>`	`applyAsDouble: T -> double`	–
`IntToLongFunction`	`applyAsLong: int -> long`	–
`IntToDoubleFunction`	`applyAsDouble: int -> double`	–
`LongToIntFunction`	`applyAsInt: long -> int`	–
`LongToDoubleFunction`	`applyAsDouble: long -> double`	–
`DoubleToIntFunction`	`applyAsInt: double -> int`	–
`DoubleToLongFunction`	`applyAsLong: double -> long`	–

Example 13.6 illustrates defining and using functions. The first lambda expression tests whether an integer is in a given range. It has the type `Integer -> Boolean`, compatible with the function type of the `Function<Integer, Boolean>` interface. Note that it returns a `Boolean`, as opposed to a lambda expression which implements a `Predicate<T>` that always returns a `boolean` value.

Click here to view code image

```
Function<Integer, Boolean> boolExpr = i -> 50 <= i && i < 100;
System.out.println("Boolean expression is: " + boolExpr.apply(99));
// Boolean expression is: true

Function<Integer, Double> milesToKms = miles -> 1.6 * miles;
```

```
System.out.printf("%dmi = %.2fkm%n", 24, milesToKms.apply(24));
// 24mi = 38.40km
```

The second lambda expression above converts miles to kilometers. It has the type `Integer -> Double`, compatible with the function type of the `Function<Integer, Double>` interface.

The method `listBuilder()` in **Example 13.6** creates a list from an array by applying a `Function<T, R>` to each array element. The `Function<T, R>` is passed as an argument to the method.

Click here to view code image

```
String[] strArray = {"One", "Two", "Three", "Four"};
List<StringBuilder> sbList = listBuilder(strArray, s -> new StringBuilder(s));
System.out.println("Build StringBuilder list: " + sbList);
// Build StringBuilder list: [One, Two, Three, Four]
```

The example above creates a list of `StringBuilder` from an array of `String`. The signature of the method call can be inferred to be the following:

Click here to view code image

```
listBuilder(String[], String -> StringBuilder)
```

with the type parameters `T` and `R` in the generic type `Function<T, R>` inferred as `String` and `StringBuilder`, respectively, resulting in the parameterized type `Function<String, StringBuilder>`.

The second example creates a list of `Integer`s from an array of `String`s, where the functional interface parameter in the method call is inferred to be `Function<String, Integer>`.

Click here to view code image

```
List<Integer> intList = listBuilder(strArray, s -> s.length());
System.out.println("Build Integer list: " + intList);
// Build Integer list: [3, 3, 5, 4]
```

Example 13.6 *Implementing Functions*

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.function.IntFunction;
import java.util.function.IntToDoubleFunction;
import java.util.function.ToIntFunction;

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

    // Examples of Function<T,R>:
    Function<Integer, Boolean> boolExpr = i -> 50 <= i && i < 100;
    System.out.println("Boolean expression is: " + boolExpr.apply(99));
    // Boolean expression is: true

    Function<Integer, Double> milesToKms = miles -> 1.6 * miles;
    System.out.printf("%dmi = %.2fkm%n", 24, milesToKms.apply(24));
    // 24mi = 38.40km
```

```
    // Create a list of StringBuilders from an array of Strings.
    String[] strArray = {"One", "Two", "Three", "Four"};
    List<StringBuilder> sbList = listBuilder(strArray, s -> new StringBuilder(s));
    System.out.println("Build StringBuilder list: " + sbList);
    // Build StringBuilder list: [One, Two, Three, Four]

    // Create a list of Integers from an array of Strings.
    List<Integer> intList = listBuilder(strArray, s -> s.length());
    System.out.println("Build Integer list: " + intList);
    // Build Integer list: [3, 3, 5, 4]

    /* Composing unary functions. */
    Function<String, String> f = s -> s + "-One";    // (1)
    Function<String, String> g = s -> s + "-Two";    // (2)

    // Using compose() and andThen() methods.
    System.out.println(f.compose(g).apply("Three")); // (3) Three-Two-One
    System.out.println(g.andThen(f).apply("Three")); // (4) Three-Two-One
    System.out.println(f.apply(g.apply("Three")));   // (5) Three-Two-One
    System.out.println();

    System.out.println(f.andThen(g).apply("Three")); // (6) Three-One-Two
    System.out.println(g.compose(f).apply("Three")); // (7) Three-One-Two
    System.out.println(g.apply(f.apply("Three")));   // (8) Three-One-Two
    System.out.println();

    // Examples of primitive unary functions.
    IntFunction<String> intToStr = i -> Integer.toString(i);
    System.out.println(intToStr.apply(2021));        // 2021
    ToIntFunction<String> strToInt = str -> Integer.parseInt(str);
    System.out.println(strToInt.applyAsInt("2021")); // 2021

    IntToDoubleFunction celsiusToFahrenheit = celsius -> 1.8 * celsius + 32.0;
    System.out.printf("%d Celsius = %.1f Fahrenheit%n",
                      37, celsiusToFahrenheit.applyAsDouble(37));
    // 37 Celsius = 98.6 Fahrenheit
  }

  /**
   * Create a list from an array by applying a Function to each array element.
   * @param arrayT     Array to use for elements
   * @param func       Function to apply to each array element
   * @return           List that is created
   */
  public static <T, R> List<R> listBuilder(T[] arrayT, Function<T, R> func) {
    List<R> listR = new ArrayList<>();
    for (T t : arrayT) {
      listR.add(func.apply(t));
    }
    return listR;
  }
}
```

Output from the program:

Click here to view code image

```
Boolean expression is: true
24mi = 38.40km
Build StringBuilder list: [One, Two, Three, Four]
Build Integer list: [3, 3, 5, 4]
Three-Two-One
Three-Two-One
```

```
Three-Two-One

Three-One-Two
Three-One-Two
Three-One-Two

2021
2021
37 Celsius = 98.6 Fahrenheit
```

Composing Functions

Both the `default` methods `compose()` and `andThen()` of the `Function<T, R>` interface return an instance of a `Function` that is created from the caller function (i.e., the function on which the method is invoked) and the argument function (i.e., the function that is passed as an argument to the method). The two methods differ in the order in which they apply the caller and the argument functions. Given two functions `f` and `g`, the `compose()` and the `andThen()` methods execute as follows:

Click here to view code image

```
f.compose(g).apply(x) emulates f.apply(g.apply(x)) or mathematically f(g(x)).
f.andThen(g).apply(x) emulates g.apply(f.apply(x)) or mathematically g(f(x)).
f.compose(g).apply(x) and g.andThen(f).apply(x) are equivalent.
```

The `compose()` method executes the argument function `g` first and executes the caller function `f` last. The `andThen()` method does the converse: It executes the caller function `f` first and executes the argument function `g` last. Switching the caller and the argument functions of one method in the other method gives the same result. Since the result of one function is passed as an argument to the other function, the return type of the function executed first must be compatible with the parameter type of the function executed last.

Creating compound functions with the `default` methods `compose()` and `andThen()` is illustrated by the code in **Example 13.6**. Functions `f` and `g` are defined at (1) and (2), respectively. The output from the program shows that the `compose()` method at (3) executes the function `g` first and the function `f` last—the same as the `andThen()` method at (4) with the functions switched and the explicit application at (5). The `andThen()` method at (6) executes the function `f` first and the function `g` last—the same as the `compose()` method at (7) with the functions switched and the explicit application at (8). The return type of the function executed first is also compatible with the parameter type of the function executed last—which is `String` for the functions `f` and `g`.

Click here to view code image

```
Function<String, String> f = s -> s + "-One";    // (1)
Function<String, String> g = s -> s + "-Two";    // (2)

System.out.println(f.compose(g).apply("Three")); // (3) Three-Two-One
System.out.println(g.andThen(f).apply("Three")); // (4) Three-Two-One
System.out.println(f.apply(g.apply("Three")));   // (5) Three-Two-One

System.out.println(f.andThen(g).apply("Three")); // (6) Three-One-Two
System.out.println(g.compose(f).apply("Three")); // (7) Three-One-Two
System.out.println(g.apply(f.apply("Three")));   // (8) Three-One-Two
```

Click here to view code image

```
default <V> Function<V,R> compose(Function<? super V,? extends T> before)
```

This generic method returns a composed function that first applies the `before` function to its input, and then applies this function to the result. (This function refers to the function used to invoke the method.)

Given that the type of this function is `T -> R` and the type of the argument function `before` is `V -> T`, the `compose()` method creates a compound function of type `V -> R`, as the function `before` is executed first and this function last.

Click here to view code image

```
default <V> Function<T,V> andThen(Function<? super R,? extends V> after)
```

This generic method returns a composed function that first applies this function to its input, and then applies the `after` function to the result.

Given that the type of this function is `T -> R` and the type of the argument function `after` is `R -> V`, the `andThen()` method creates a compound function of type `T -> V`, as this function is executed first and the function `after` last.

Any exception thrown during the evaluation of either function aborts the evaluation of the composed function and the exception is conveyed to the caller.

Primitive Type Specializations of `Function<T, R>`

As can be seen in **Table 13.7**, there are three categories of *primitive type one-arity specializations* of the `Function<T, R>` interface, each distinguished by a naming scheme. Also, these primitive type one-arity specializations do not define any `default` methods.

- *Prim*`Function<R>`, where *Prim* is either `Int`, `Long`, or `Double`
  These one-arity generic functions have the functional method `apply`: *primitive* `-> R`, where *primitive* is an `int`, `long`, or `double`—the function takes an argument of primitive type and returns a result of type `R`.
  Click here to view code image

```
IntFunction<String> intToStr = i -> Integer.toString(i);
System.out.println(intToStr.apply(2021));        // "2021"
```

- `To`*Prim*`Function<T>`, where *Prim* is either `Int`, `Long`, or `Double`
  These one-arity generic functions have the functional method `applyAs`*Prim*: `T->` *primitive*, where *primitive* is an `int`, `long`, or `double`—the function takes an argument of type `T` and returns a result of primitive type.
  Click here to view code image

```
ToIntFunction<String> strToInt = str -> Integer.parseInt(str);
System.out.println(strToInt.applyAsInt("2021")); // 2021
```

- *Prim*$_1$ `To` *Prim*$_2$ `Function`, where *Prim*$_1$ and *Prim*$_2$ are `Int`, `Long`, or `Double`, and *Prim*$_1$ `!=` *Prim*$_2$
  These one-arity non-generic functions have the functional method `applyAs` *Prim*$_2$: *primitive*$_1$ `->` *primitive*$_2$, where *primitive*$_1$ and *primitive*$_2$ are `int`, `long`, or `double`, and *primitive*$_1$ `!=` *primitive*$_2$—the function takes an argument of type *primitive*$_1$ and returns a result of type *primitive*$_2$.
  Click here to view code image

```
IntToDoubleFunction celsiusToFahrenheit = celsius -> 1.8 * celsius + 32.0;
System.out.printf("%d Celsius = %.1f Fahrenheit%n",
                  37, celsiusToFahrenheit.applyAsDouble(37));
// 37 Celsius = 98.6 Fahrenheit
```

13.9 Two-Arity Specialization of `Function<T, R>`: `BiFunction<T, U, R>`

The `BiFunction<T, U, R>` interface is the two-arity specialization of the `Function<T, R>` interface. In **Table 13.8**, we see that its functional method `apply()` has the type `(T, U) -> R` — that is, it takes two arguments of type `T` and `U`, and returns a result of type `R`.

The following code illustrates defining and using two-arity functions. The twoarity function `areaOfRectangle` calculates the area of a rectangle.

Click here to view code image

```
BiFunction<Double, Double, Double> areaOfRectangle
    = (length, width) -> length * width;            // (Double, Double) -> Double
System.out.printf("%.2f x %.2f = %.2f%n",
                  25.0, 4.0, areaOfRectangle.apply(25.0, 4.0));
// 25.00 x 4.00 = 100.00
```

Table 13.8 *Two-arity Functions*

Functional interface (`T`, `U`, and `R` are type parameters)	Functional method	Default methods
`BiFunction<T, U, R>`	`apply: (T, U) -> R`	`andThen()`
`ToIntBiFunction<T, U>`	`applyAsInt: (T, U) -> int`	–
`ToLongBiFunction<T, U>`	`applyAsLong: (T, U) -> long`	–
`ToDoubleBiFunction<T, U>`	`applyAsDouble: (T, U) -> double`	–

The two-arity function `concatKeyVal` below concatenates two strings and returns a new string as the result. The reference `map` refers to a `HashMap<String, String>`. The `replaceAll()` method called on this map takes the two-arity function `concatKeyVal` as a parameter, and it replaces the *value* of each entry `(key, value)` in the map with the result of the two-arity function applied to the entry—that is, the method call `concatKeyVal.apply(key, value)` is executed for each entry in the map. The `apply()` method is implemented by the lambda expression below, resulting in the method call `key.concat(value)` being executed for each entry.

Click here to view code image

```
BiFunction<String, String, String> concatKeyVal = (key, val) -> key.concat(val);
// {Dick=silver, Harriet=platinum, Tom=gold}
map.replaceAll(concatKeyVal);
// {Dick=Dicksilver, Harriet=Harrietplatinum, Tom=Tomgold}
```

It is instructive to implement and compare the two-arity function above using an anonymous class:

Click here to view code image

```
// {Dick=silver, Harriet=platinum, Tom=gold}
map.replaceAll(new BiFunction<String, String, String>() {
  public String apply(String key, String val) {
    return key.concat(val);
  }
});
// {Dick=Dicksilver, Harriet=Harrietplatinum, Tom=Tomgold}
```

Composing Two-Arity Functions

The interface `BiFunction<T, U, R>` provides the `andThen()` method for creating compound two-arity functions. Given a *two-arity function* `f` and a *one-arity function* `g`, the method evaluates the functions as follows:

`f.andThen(g).apply(x, y)` emulates `g.apply(f.apply(x, y))`—that is, the two-arity function `f` is executed first and the one-arity function `g` last.

In the example below, the functions chained by the `andThen()` method calls are applied from left to right, first the caller function and then the parameter function.

Click here to view code image

```
BiFunction<String, String, String> concatStr = (s1, s2) -> s1 + s2;
Function<String, String> postfix1 = s -> s + "nana";
```

Click here to view code image

```
Function<String, String> postfix2 = s -> s + "s!";
System.out.println(concatStr.andThen(postfix1).andThen(postfix2)
                   .apply("I am going", " ba"));         // I am going bananas!
```

In the code below, the `concatStr` two-arity function is both the caller and the parameter function in the call to the `andThen()` method. However, the code does not compile. The reason is easy to understand: The two-arity parameter function requires *two* arguments, but the two-arity caller function can only return a *single* value. A *one-arity* function as the parameter function avoids this problem, as it can accept the single-value result of the caller function. Chaining instances of `BiFunction<T, U, R>` is not as straightforward as chaining instances of `Function<T, R>`.

Click here to view code image

```
BiFunction<String, String, String> concatTwice
               = concatStr.andThen(concatStr);           // Compile-time error!
```

Click here to view code image

```
default <V> BiFunction<T,U,V> andThen(Function<? super R,? extends V> after)
```

This generic method returns a composed two-arity function that first applies this function to its input, and then applies the specified `after` one-arity function to the result.

Note the type of the parameter: It is `Function` and not `BiFunction`. Given that the type of this two-arity function is `(T, U) -> R` and the type of the one-arity parameter function `after` is `R -> V`, the `andThen()` method creates a compound two-arity function of type `(T, U) -> V`, as this function is executed first and the one-arity function `after` last.

Any exception thrown during the evaluation of either function aborts the evaluation of the composed function and the exception is conveyed to the caller.

Primitive Type Specializations of `BiFunction<T, U, R>`

Table 13.8 shows that the `BiFunction<T, U, R>` interface has three *primitive type twoarity generic specializations* to `int`, `long`, and `double`, but they do not define any `default` methods for creating compound functions. The specializations are named `To`*`Prim`*`BiFunction<T, U>`, where *Prim* is either `Int`, `Long`, or `Double`. These two-arity generic functions have the functional method `applyAs`*`Prim`*`: (T, U) ->` *primitive*, where *primitive* is an `int`, `long`, or `double`—the function takes two arguments of type `T` and `U`, and returns a result of a primitive type.

In the example below, the `addIntStrs` two-arity function parses two strings as `int` values and returns the sum of the values.

Click here to view code image

```
ToIntBiFunction<String, String> addIntStrs
    = (s1, s2) -> Integer.parseInt(s1) + Integer.parseInt(s2);
System.out.println("10 + 20 = " + addIntStrs.applyAsInt("10", "20"));
// 10 + 20 = 30
```

13.10 Extending `Function<T,T>`: `UnaryOperator<T>`

Table 13.9 shows that the `UnaryOperator<T>` interface *extends* the `Function<T, T>` interface for the special case where the types of the argument and the result are the same. It inherits the functional method `apply()` from the `Function<T, T>` interface. It also inherits the `default` methods `compose()` and `andThen()` from its superinterface `Function<T, T>`, but note that these methods return a `Function<T,T>`, and not a `UnaryOperator<T>`.

Functions where the argument and the result type are the same can easily be refactored to use the `UnaryOperator<T>` interface.

Click here to view code image

```
UnaryOperator<Double> area = r -> Math.PI * r * r;
System.out.printf("Area of circle, radius %.2f: %.2f%n", 10.0, area.apply(10.0));
// Area of circle, radius 10.00: 314.16

UnaryOperator<Double> milesToKms = miles -> 1.6 * miles;
System.out.printf("%.2fmi = %.2fkm%n", 24.0, milesToKms.apply(24.0));
// 24.00mi = 38.40km
```

The `List.replaceAll(UnaryOperator<E>)` method can be used to replace each elements in the list with the result of applying the specified unary operator to that element. The method replaces all strings in the list `team` with their uppercase versions.

Click here to view code image

```
List<String> team = Arrays.asList("Tom", "Dick", "Harriet");
UnaryOperator<String> toUpper = str -> str.toUpperCase();
team.replaceAll(toUpper);     // [TOM, DICK, HARRIET]
```

Since the `UnaryOperator<T>` interface is a subinterface of the `Function<T, T>` interface and inherits the `default` methods `compose()` and `andThen()`, creating compound unary operators is no different from creating compound functions.

Click here to view code image

```
UnaryOperator<String> f = s -> s + "-One";
UnaryOperator<String> g = s -> s + "-Two";
System.out.println(f.compose(g).apply("Three")); // Three-Two-One
System.out.println(f.andThen(g).apply("Three")); // Three-One-Two
```

Table 13.9 *Unary Operators*

Functional interface (`T`, `U`, and `R` are type parameters)	Functional method	Default methods unless otherwise indicated
`UnaryOperator<T> extends Function<T,T>`	`apply: T -> T`	`compose(), andThen(), static identity()`
`IntUnaryOperator`	`applyAsInt: int -> int`	`compose(), andThen()`
`LongUnaryOperator`	`applyAsLong: long -> long`	`compose(), andThen()`
`DoubleUnaryOperator`	`applyAsDouble: double -> double`	`compose(), andThen()`

Primitive Type Specializations of `UnaryOperator<T>`

The `UnaryOperator<T>` interface has three *primitive type specializations* to `int`, `long`, and `double`. The specializations are named *`Prim`*`UnaryOperator`, where *`Prim`* is either `Int`, `Long`, or `Double` (Table 13.9). These non-generic unary operators have the functional method `applyAs`*`Prim`*`:` *`primitive`* `->` *`primitive`*, where *`primitive`* is an `int`, `long`, or `double`—the operator takes an argument of a primitive type and returns a result of the *same* primitive type.

Click here to view code image

```
DoubleUnaryOperator celsiusToFahrenheit = celsius -> 1.8 * celsius + 32.0;
System.out.printf("%.1f Celsius = %.1f Fahrenheit%n",
                  25.0, celsiusToFahrenheit.applyAsDouble(25.0));
// 25.0 Celsius = 77.0 Fahrenheit

DoubleUnaryOperator kms = miles -> 1.6 * miles;
System.out.printf("%.2fmi = %.2fkm%n", 25.0, kms.applyAsDouble(25.0));
// 25.00mi = 40.00km
```

The primitive type unary operators define the `default` methods `compose()` and `andThen()` for creating compound primitive type unary operators. The semantics of these default methods are the same as what we saw earlier (p. 715).

Click here to view code image

```
IntUnaryOperator incrBy1 = i -> i + 1;
IntUnaryOperator multBy2 = i -> i * 2;
System.out.println(incrBy1.compose(multBy2).applyAsInt(4)); // 9
System.out.println(incrBy1.andThen(multBy2).applyAsInt(4)); // 10
```

13.11 Extending `BiFunction<T,T,T>`: `BinaryOperator<T>`

In **Table 13.10**, we see that the `BinaryOperator<T>` interface *extends* the `BiFunction<T, T, T>` interface for the special case where the types of the two arguments and the result are the same. It inherits the functional method `apply()` from the `BiFunction<T, T, T>` interface, as well as its `andThen()` method.

Click here to view code image

```
BinaryOperator<Double> areaOfRectangle = (length, width) -> length * width;
System.out.printf("%.2f x %.2f = %.2f%n",
                  25.0, 4.0, areaOfRectangle.apply(25.0, 4.0));
// 25.00 x 4.00 = 100.00
```

Creating compound binary operators is no different from creating compound twoarity functions using the `andThen()` method, where the parameter function of the method must be a unary operator or a one-arity function.

Click here to view code image

```
BinaryOperator<String> concatTwo = (s1, s2) -> s1 + s2;
UnaryOperator<String> postfix1 = s -> s + "nana";
UnaryOperator<String> postfix2 = s -> s + "s!";
System.out.println(concatTwo.andThen(postfix1).andThen(postfix2)
                  .apply("I am going", " ba"));     // I am going bananas!
```

The two utility methods `maxBy()` and `minBy()` can be used to compare two elements according to a given comparator:

Click here to view code image

```
String maxStr = BinaryOperator.maxBy(String.CASE_INSENSITIVE_ORDER)
                              .apply("aha", "Madonna");            // Madonna
String minStr = BinaryOperator.minBy(String.CASE_INSENSITIVE_ORDER)
                              .apply("aha", "Madonna");            // aha
```

Table 13.10 *Binary Operators*

Functional interface (`T`, `U`, and `R` are type parameters)	Functional method	Default methods unless otherwise indicated
`BinaryOperator<T> extends BiFunction<T,T,T>`	`apply:          (T, T) -> T`	`andThen()`, `static maxBy()`, `static minBy()`
`IntBinaryOperator`	`applyAsInt:     (int, int) -> int`	–
`LongBinaryOperator`	`applyAsLong:    (long, long) -> long`	–

`DoubleBinaryOperator`	`applyAsDouble: (double, double) -> double`	–

The `BinaryOperator<T>` interface also provides two utility methods to create binary operators for comparing two elements according to a given comparator:

Click here to view code image

```
static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)
```

Returns a `BinaryOperator` which returns the greater of two elements according to the specified `comparator`.

Click here to view code image

```
static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)
```

Returns a `BinaryOperator` which returns the lesser of two elements according to the specified `comparator`.

Primitive Type Specializations of `BinaryOperator<T>`

Table 13.10 shows that the `BinaryOperator<T>` interface has three *primitive type specializations* to `int`, `long`, and `double`. The specializations are named `PrimBinaryOperator`, where *Prim* is either an `Int`, `Long`, or `Double` (Table 13.10). These primitive type binary operators have the functional method `applyAsPrim: (primitive, primitive) -> primitive`, where `primitive` is an `int`, `long`, or `double`—the operator takes two arguments of a primitive type and returns a result of the same primitive type.

Click here to view code image

```
DoubleBinaryOperator areaOfRectangle2 = (length, width) -> length * width;
System.out.printf("%.2f x %.2f = %.2f%n",
                  25.0, 4.0, areaOfRectangle2.applyAsDouble(25.0, 4.0));
// 25.00 x 4.00 = 100.00
```

13.12 Currying Functions

The functional interfaces in the `java.util.function` package define functional methods that are either one-arity or two-arity methods. For higher arity functional methods, one recourse is to define functional interfaces whose functional method has the desired arity. The functional interface below defines a three-arity functional method—that is, it takes three arguments.

Click here to view code image

```
@FunctionalInterface
interface TriFunction<T, U, V, R> {
  R compute(T t, U u, V v);             // (T, U ,V) -> R
}
```

The `TriFunction<T, U, V, R>` interface can be used to define a lambda expression to calculate the volume of a cuboid.

Click here to view code image

```
// (Double, Double, Double) -> Double
TriFunction<Double, Double, Double, Double> cubeVol = (x, y, z) -> x * y * z;
System.out.println(cubeVol.compute(10.0,  20.0,  30.0));  // 6000.0
```

Another recourse is to apply the technique of *currying* to transform a multi-argument function into a chain of lower arity functions.

The process of currying is illustrated by implementing the three-arity lambda expression above for calculating the volume of a cuboid. Step 1 below derives *a chain of three one-arity functions* that will together compute the volume of a cuboid. At (1), parentheses are used explicitly to show the nested lambdas that *define* each of the one-arity functions—grouping is from right to left. The nesting of the onearity functions is compatible with the nesting of the types in the parameterized functional interface type. An outer function returns its immediate inner function. Step 2 supplies the `x` argument. This is called *partial application*, as it returns a function where the remaining arguments `y` and `z` are still unknown. Step 3 is also partial application, only the `y` argument is supplied, returning a function where now only the `z` argument is unknown. Only at Step 4, when the `z` argument is supplied at (2), can the final one-arity function be executed. The application of the individual onearity functions can also be chained as shown at (3), without going through the intermediate steps.

Click here to view code image

```
// Step 1:
// Partial application: double -> DoubleFunction<DoubleUnaryOperator>
DoubleFunction<DoubleFunction<DoubleUnaryOperator>> uniFuncA
    = (x -> (y -> (z -> x * y * z)));              // (1)

// Step 2:
// Partial application: double -> DoubleUnaryOperator
DoubleFunction<DoubleUnaryOperator> uniFuncB
    = uniFuncA.apply(10.0);                        // 10.0 * y * z

// Step 3:
// Partial application: double -> double
DoubleUnaryOperator uniOpC = uniFuncB.apply(20.0); // 10.0 * 20.0 * z

// Step 4:
// Application:
double vol1 = uniOpC.applyAsDouble(30.0);      // (2) 10.0 * 20.0 * 30.0 = 6000.0
double vol2 = uniFuncA.apply(10.0).apply(20.0).applyAsDouble(30.0); // (3) 6000.0
```

The sequence in which the arguments are supplied in the currying process is irrelevant, and more than one argument can be supplied in each step in accordance with the target type. Each step is a partial application, except for the last one which executes the final function. The process ensures that the number of unknown arguments decreases in each step.

Here we have provided just a taste of currying, but it is a topic worth exploring further. The technique bears the name of Haskell Curry, a famous mathematician and logician of the twentieth century.

13.13 Method and Constructor References

So far we have seen that a Java program can use primitive values, objects (including arrays), and lambda expressions. In this section, we introduce the fourth kind of value that a Java program can use, called *method references* (and their specializations to constructors and array construction). As we shall see, there is a very tight relationship between method references and lambda expressions.

Quite often the body of a lambda expression is just a call to an existing method. The lambda expression simply supplies the arguments required to call and execute the method. In such cases, method references can provide a more concise notation than a lambda expression, potentially increasing the readability of the code. The following code illustrates the relationship between the two notations.

Click here to view code image

```
// String -> void
Consumer<String> outLE = obj -> System.out.println(obj); // (1a)
Consumer<String> outMR = System.out::println;            // (1b)
outMR.accept("Save trees!");                             // (2)
// Calls System.out.println("Save trees!") that prints: Save trees!
```

The lambda expression at (1a):

Click here to view code image

```
obj -> System.out.println(obj)
```

can be replaced by the *method reference* at (1b):

```
System.out::println
```

The method reference above is composed of the *target reference* (`System.out`) on which the method is invoked and the *name of the method* (`println`), separated by the double-colon (`::`) delimiter:

```
targetReference::methodName
```

Note that the target reference precedes the double-colon (`::`) delimiter, and no arguments are specified after the method name. Table 13.11 summarizes the variations in the definition of a method reference which we will discuss in this section.

Table 13.11 *Method and Constructor References*

Purpose of method reference	Lambda expression/Method reference syntax	Comment
Designate a static method	`(args) -> RefType.staticMethod(args)` `RefType::staticMethod`	
Designate an instance method of a *bounded* instance	`(args) -> expr.instanceMethod(args)` `expr::instanceMethod`	Target reference provided by the method reference.
Designate an instance method of an *unbounded* instance	`(arg0,rest)->arg0.instanceMethod(rest)`	`arg0` is of `RefType`. Target reference provided later

	`RefType::instanceMethod`	when the method is invoked.
Designate a constructor	`(args) -> new ClassType(args)` `ClassType::new`	Deferred creation of an instance.
Designate array construction	`arg -> new ElementType[arg][]...[]` `ElementType[][]...[]::new`	Deferred creation of an array.

A method reference must have a compatible target type that is a functional interface—a method reference implements an instance of a functional interface, analogous to a lambda expression. The compiler does similar type checking as for lambda expressions to determine whether the method reference is compatible with a given target type.

At (1b) above, the function type of the target type `Consumer<String>` is `String -> void`. The compiler can infer from the target type that the type of the argument passed to the `println()` method must be `String`. The type of the method reference is defined by the type of the method specified in its definition. In this case, the type of the `println()` method is `String -> void`, as it accepts a `String` parameter and does not return a value. Thus the target type `Consumer<String>` is compatible with the method reference `System.out::println`. The reference `outMR` is assigned the value of the method reference at (1b).

Analogous to single-method lambda expressions, method references when executed result in the execution of the method specified in its definition. The instance method `println()` at (1b) accepts a single `String` argument. This argument is passed to the method when the functional method `accept()` of the target type is invoked on the reference `outMR`, as shown at (2). The method `println()` is invoked on the same object (denoted by `System.out`) every time this method reference is executed. Execution of the lambda expression at (1a) will give the same result. Not surprisingly, a method reference and its corresponding single-method lambda expression are semantically equivalent.

Static Method References

Sometimes the lambda body of a lambda expression is just a call to a `static` *method*. The lambda expression at (1a) calls the `static` method `now()` in the class `java.time.LocalDate` to obtain the current date from the system clock.

Click here to view code image

```
Supplier<LocalDate> dateNowLE = () -> LocalDate.now();  // (1a) Lambda expression
```

The lambda expression has the type `() -> LocalDate`, and not surprisingly, it is also the type of the static method `now()` which takes no arguments and returns an instance of the `LocalDate` class. The method type is compatible with the function type of the target type—that is, it is compatible with the type of the functional method `get()` of the parameterized functional interface `Supplier<LocalDate>`. The compiler can infer from the context that the type of the static method is compatible with the target type of the functional interface. In such a case, the lambda expression can be replaced by a *static method reference*:

Click here to view code image

```
Supplier<LocalDate> dateNowMR = LocalDate::now;         // (1b) Method reference
```

The double-colon delimiter (`::`) separates the reference type name (class `LocalDate`) from the static method name (`now`) in the syntax of the static method reference.

Analogous to a lambda expression, a method reference can be used as a value in an assignment, passed as an argument in a method or constructor call, returned in a `return` statement, or cast with the cast operator (p. 733). Its execution is deferred until the functional method of its target type is invoked, as at (2).

Click here to view code image

```
LocalDate today = datenowMR.get();    // (2) Method reference at (1b) executed.
System.out.println(today.format(DateTimeFormatter.ISO_DATE)); // 2021-03-01
```

The following rule can be helpful in converting between a lambda expression and a static method reference:

A lambda expression of the form

Click here to view code image

```
(args) -> RefType.staticMethod(args)
```

is semantically equivalent to the *static method reference*:

Click here to view code image

```
RefType::staticMethod
```

where `RefType` is the *name* of a class, an enum type, or an interface that defines the static method whose *name* `staticMethod` is specified after the double-colon (`::`) delimiter.

Arguments are generally not specified in the method reference, and any parameters required by the method are determined by the target type of the context.

Click here to view code image

```
// String -> Integer
Function<String, Integer> strToIntLE = s -> Integer.parseInt(s); // (3a)
Function<String, Integer> strToIntMR = Integer::parseInt;        // (3b)
System.out.println(strToIntMR.apply("100"));                     // (4)
// Calls Integer.parseInt("100") that returns the int value 100 which is boxed
// into an Integer.
```

The `static` method `Integer.parseInt()` in the lambda body at (3a) takes one argument. Its type is `String -> Integer`. Using the one-arity `Function<String, Integer>` as the target type is appropriate as its function type is also `String -> Integer`. We can convert the lambda expression at (3a) to the static method reference shown at (3b). Note that no arguments are specified at (3b). The argument is passed to the static method at a later time when the functional method `apply()` of the functional interface is invoked, as demonstrated at (4).

Similarly, we can define static method references whose static method takes two arguments. A two-arity function or a binary operator from the `java.util.function` package can readily serve as the target type, as demonstrated by the following examples. Note that the static method `Math.min()` is overloaded, but the target type of the context determines which method will be executed. The type of the method `min()` at (5) is different than the type at (6). Analogous to lambda expressions, the same method reference can have different target types depending on the context.

Click here to view code image

```
// (double, double) -> double
DoubleBinaryOperator minDoubleLE = (x, y) -> Math.min(x, y);
DoubleBinaryOperator minDoubleMR = Math::min;           // (5)
System.out.println(minDoubleMR.applyAsDouble(20.0, 30.0));
// Calls Math.min(20.0, 30.0) that returns the double value 20.0.

// (int, int) -> (int)
IntBinaryOperator minIntLE = (x, y) -> Math.min(y, y);
IntBinaryOperator minIntMR = Math::min;                 // (6)
System.out.println(minIntMR.applyAsInt(20, 30));
// Calls Math.min(20, 30) that returns the int value 20.
```

If the static method requires more that than two arguments, we can either define new functional interfaces with the appropriate arity for their functional method, or use the currying technique (p. 723).

Bounded Instance Method References

When the body of a lambda expression is a call to an *instance method*, the method reference specified depends on whether the object on which the instance method is invoked exists or not at the time the method reference is defined.

In the code below, the reference `sb` is declared and initialized at (1). It is effectively final when accessed in the lambda expression at (2a). The reference value of the reference `sb` is *captured* by the lambda expression. When the lambda expression is executed at a later time, the `reverse()` method is executed on the object denoted by the reference `sb`. In this case, the *bounded instance method reference* is simply the reference and the instance method name, separated by the double-colon delimiter, as shown at (2b). The bounded instance method reference at (2b) can replace the lambda expression at (2a). It is executed when the functional method `get()` of the parameterized functional interface `Supplier<StringBuilder>` is called, as shown at (3).

Click here to view code image

```
StringBuilder sb = new StringBuilder("!em esreveR");         // (1)
// () -> StringBuilder
Supplier<StringBuilder> sbReverserLE = () -> sb.reverse();   // (2a)
Supplier<StringBuilder> sbReverserMR = sb::reverse;          // (2b)
System.out.println(sbReverserMR.get());                      // (3)
// Calls sb.reverse() that returns the StringBuilder with character sequence
// "Reverse me!".
```

The case where the object on which the instance method is executed does not exist when the method reference is defined, but will be supplied when the method reference is executed, requires an *unbounded* instance method reference (p. 729).

The following rule can be used for converting between a lambda expression and a bounded instance method reference:

A lambda expression of the form

Click here to view code image

```
(args) -> expr.instanceMethod(args)
```

is semantically equal to the *bounded instance method reference*:

```
expr::instanceMethod
```

where `expr` is an expression that evaluates to a reference value that is captured by the bounded instance method reference and becomes the *target reference* for the bounded instance method reference.

Any reference involved in the evaluation of `expr` must be effectively final, and is captured by the bounded instance method reference. The target reference represented by `expr` is separated from the instance method name by the double-colon delimiter. The target reference is fixed when the bounded instance method reference is defined. The instance method is invoked on the object denoted by the target reference when the method reference is executed at a later time, and any arguments required by the instance method are passed at the same time.

Given an `ArrayList<String>` denoted by the reference `words`, we can pass the method reference `System::println` to the `forEach()` method of the `ArrayList<E>` class in order to print each element of the list. The method takes a consumer as an argument, and the type of the object to print is inferred from the element type of the list.

Click here to view code image

```
words.forEach(obj -> System.out.println(obj));
words.forEach(System.out::println);
```

The syntax of the bounded instance method reference where the instance method has more than one argument is no different. In the code below, the `replace()` method of the `String` class has two arguments. Its type is `(String, String) -> String`, the same as the function type of the target type `BinaryOperator<String>`. The target reference is the reference `str` that is defined at (4). It is effectively final when accessed in the code, and its reference value is captured at (5b) where the method reference is defined. The method replaces each occurrence of `s1` in `str` with `s2`. The arguments are passed to the method when the functional method `apply()` of the target type `BinaryOperator<String>` is executed, as shown at (6).

Click here to view code image

```
String str = "Java Jive";                                         // (4)
// (String, String) -> String
BinaryOperator<String> replaceOpLE = (s1, s2) -> str.replace(s1, s2); // (5a)
BinaryOperator<String> replaceOpMR = str::replace;                // (5b)
System.out.println(replaceOpMR.apply("Jive", "Jam"));             // (6)
// Calls str.replace("Jive", "Jam") that returns the string "Java Jam".
```

In a *non-static* context, the `final` references `this` and `super` can also be used as the target reference of a bounded instance method reference.

Click here to view code image

```
Predicate<String> p1 = s -> this.equals(s);       // String -> boolean
Predicate<String> p2 = this::equals;              // String -> boolean
Supplier<String> s1 = () -> super.toString();     // () -> String
Supplier<String> s2 = super::toString;            // () -> String
```

Unbounded Instance Method References

In the case of an unbounded instance method reference, the target reference is determined when the method reference is executed, as it is the first argument passed to the method reference. This is embodied in the following rule:

A lambda expression of the form

Click here to view code image

```
(arg0, rest) -> arg0.instanceMethod(rest)
```

is semantically equivalent to the *unbounded instance method reference*:

```
RefType::instanceMethod
```

where `RefType` is the reference type of the target reference `arg0`. The names of the reference type and the instance method are separated by the double-colon (`::`) delimiter.

The instance method is invoked on the object denoted by the target reference `arg0` (i.e., the first argument) when the method reference is executed, and any remaining arguments are passed to the instance method at the same time.

In the code below, the type of the unbounded instance method reference `String::length` at (1) is `String -> int`, the same as the function type of the target type `ToIntFunction<String>`. Invoking the functional method `applyAsInt()` on the reference `lenMR` results in the method `length()` being invoked on the string `"Java"` that was passed as a parameter.

Click here to view code image

```
// String -> int
ToIntFunction<String> lenLE = s -> s.length();
ToIntFunction<String> lenMR = String::length;                  // (1)
System.out.println(lenMR.applyAsInt("Java"));                  // 4
// Calls "Java".length() that returns the int value 4.
```

The `static` method `listBuilder()` in Example 13.6, p. 714, creates a list from an array by applying a `Function<T, R>` to each array element. An instance of the `Function<T, R>` is passed as a parameter to the method. Both lines of code below create a list of `Integer` from an array of `String`, where the functional interface parameter in the method call is inferred to be `Function<String, Integer>`. The method `length()` is executed on the first argument of the unbounded instance method reference— that is, on each `String` element of the list.

Click here to view code image

```
List<Integer> intList1 = listBuilder(strArray, s -> s.length()); // Lambda expr.
List<Integer> intList2 = listBuilder(strArray, String::length);  // Method ref.
```

The code below illustrates the case where the unbounded instance method reference `String::concat` at (2) requires two arguments. Its target type is `BinaryOperator<String>` that has the function type `(String, String) -> String`. The instance method `concat()` is invoked on the first argument, and the second argument is passed to the method as a parameter.

Click here to view code image

```
// (String, String) -> String
BinaryOperator<String> concatOpLE = (s1, s2) -> s1.concat(s2);
BinaryOperator<String> concatOpMR = String::concat;           // (2)
System.out.println(concatOpMR.apply("Java", " Jive"));        // Java Jive
// Calls "Java".concat(" Jive") that returns the string "Java Jive".
```

The code below illustrates using parameterized types in method references. At (3), the type of the argument of the generic interface `List<T>` in the method reference is inferred from the context to be `String`. At (4), the type of the argument is explicitly specified to be `String`. This can be necessary if the compiler cannot infer it from the context. The type of the `List::contains` instance method reference is `(List<String>, String) -> boolean`, which is compatible with the function type of the parameterized functional interface `BiPredicate<List<String>, String>`.

Click here to view code image

```
// (List<String>, String) -> boolean
BiPredicate<List<String>, String> containsLE
    = (list, element) -> list.contains(element);

BiPredicate<List<String>, String> containsMR1 = List::contains;          // (3)
BiPredicate<List<String>, String> containsMR2 = List<String>::contains;  // (4)
System.out.println(containsMR2.test(words, "BOB"));  // words is a List<String>.
// Calls words.contains("BOB") that returns a boolean value.
```

If the method in the unbounded instance method reference requires several arguments, compatible target types can be defined by either defining new functional interfaces with the appropriate arity for their functional method, or applying the currying technique (p. 723).

Constructor References

A constructor reference is similar to a static method reference, but with the keyword `new` instead of the static method name, signifying that a constructor of the specified class should be executed. Its purpose of course is to instantiate the class.

We can convert between a constructor reference and a lambda expression using the following rule:

A lambda expression of the form

```
(args) -> new ClassType(args)
```

is semantically equivalent to the *constructor reference*:

```
ClassType::new
```

where `ClassType` is the name of the class that should be instantiated. The class name and the keyword `new` are separated by the double-colon (`::`) delimiter.

Which constructor of `ClassType` is executed depends on the target type of the context, since it determines the arguments that are passed to the constructor.

Execution of the constructor reference `sbCR` defined at (1) results in the zero-argument constructor of the `StringBuilder` class to be executed, as evident from the type of the constructor reference.

Click here to view code image

```
// () -> StringBuilder
Supplier<StringBuilder> sbLE = () -> new StringBuilder();
Supplier<StringBuilder> sbCR = StringBuilder::new;                // (1)
StringBuilder sbRef = sbCR.get();
// Calls new StringBuilder() to create an empty StringBuilder instance.
```

However, execution of the constructor reference `sb4` defined at (2) results in the constructor with the `String` parameter to be executed, as evident from the type of the constructor reference. The target types at (1) and (2) are different. The target type determines which constructor of the `StringBuilder` class is executed.

Click here to view code image

```
// String -> StringBuilder
Function<String, StringBuilder> sb3 = s -> new StringBuilder(s);
Function<String, StringBuilder> sb4 = StringBuilder::new;        // (2)
System.out.println(sb4.apply("Build me!"));                      // Build me!
// Calls new StringBuilder("Build me!") to create a StringBuilder instance
// based on the string "Build me!".
```

The following code illustrates passing two arguments to a constructor using an appropriate target type—in this case, a two-arity function—that ensures applicable arguments are passed to the constructor.

Click here to view code image

```
// (String, String) -> Locale
BiFunction<String, String, Locale> locConsLE
    = (language, country) -> new Locale(language, country);
BiFunction<String, String, Locale> locConsCR = Locale::new;
System.out.println(locConsCR.apply("en","US"));                  // en_US
// Calls new Locale("en", "US") to create a Locale instance with the specified
// parameter values.
```

Note that the constructor reference is *defined* first without any instance being created, and its execution is deferred until later when the functional method of its target type is invoked.

Array Constructor References

Array constructor reference is specialization of the constructor reference for creating arrays. We can convert between an array constructor reference and a lambda expression using the following rule:

A lambda expression of the form

Click here to view code image

```
arg -> new ElementType[arg][]...[]
```

is semantically equivalent to the *array constructor reference*:

```
ElementType[][]...[]::new
```

The array type and the keyword `new` are separated by the double-colon (`::`) delimiter. The `ElementType` is designated with the necessary pairs of square brackets (`[]`) to indicate that it is an array type of a specific number of dimensions. Only the length of the *first* dimension of the array can be created using an array constructor reference. As one would expect, the elements are initialized to the default value for the element type.

The array constructor reference at (1) will create a simple array of element type `int`. The target type `IntFunction<int[]>` is compatible with the type of the array constructor reference (`int -> int[]`). The array of `int` created at (2) has length 4, where each element has the default value `0`.

Click here to view code image

```
// int -> int[]
IntFunction<int[]> intArrConsLE = n -> new int[n];
IntFunction<int[]> intArrConsCR = int[]::new;                          // (1)
int[] intArr = intArrConsCR.apply(4);                                  // (2)
// Creates an int array of length 4.
```

In the code below, we can define a lambda expression to create a two-dimensional array that takes two arguments. However, this is *not* possible using an array constructor reference, as only the length of the first dimension can be passed to an array constructor reference. The line at (3) will not compile, since the target type (`(Integer, Integer) -> int[][]`) is not compatible with the type of the array constructor reference (`int -> int[][]`).

Click here to view code image

```
// (int, int) -> int[][]
BiFunction<Integer, Integer, int[][]> twoDimArrConsLE1 = (n, m) -> new int[n][m];
// BiFunction<Integer, Integer, int[][]> twoDimArrConsCR1
//      = int[][]:: new;                            // (3) Compile-time error!
```

It is only possible to define an array constructor reference to create the length of the first dimension of an array, regardless of how many dimensions it has. This is illustrated by the array constructor reference at (4), which creates a two-dimensional array where only the first dimension is constructed—keeping in mind that in Java, multidimensional arrays are implemented as arrays of arrays. The code at (5) returns an array with three rows, where each row is initialized to the `null` value. Individual arrays can be constructed and stored in the two-dimensional array, as shown at (6).

Click here to view code image

```
// int -> int[][]
IntFunction<int[][]> twoDimArrConsLE = n -> new int[n][];
IntFunction<int[][]> twoDimArrConsCR = int[][]::new;            // (4)
int[][] twoDimIntArr1 = twoDimArrConsCR.apply(3);               // (5)
// [null, null, null]
for (int i = 0; i < twoDimIntArr1.length; ++i)
  twoDimIntArr1[i] = intArrConsCR.apply(i+1);                   // (6) Calls (1).
// [[0], [0, 0], [0, 0, 0]]
```

The example below illustrates constructing an array of objects. The procedure is no different from constructing arrays of primitive values, as explained above. The array returned by the code at (7) has five elements, where each element is initialized to the `null` value.

Click here to view code image

```
// int -> StringBuilder[]
IntFunction<StringBuilder[]> sbaConsLE = n -> new StringBuilder[n];
IntFunction<StringBuilder[]> sbaConsCR = StringBuilder[]::new;
StringBuilder[] sbArr2 = sbaConsCR.apply(5);                      // (7)
// [null, null, null, null, null]
```

Java does not allow creation of generic arrays, as demonstrated by the declaration statement at (8), where an attempt is made to construct an array of formal parameter type `A` (§11.13, p. 627). This can be overcome by using either a lambda expression or an array constructor reference whose target type is a parameterization of `IntFunction<A[]>`, and which is passed to the generic method `createArray()` below, together with the required array length, to create an array of a specific type.

Click here to view code image

```
public static <A> A[] createArray(int length, IntFunction<A[]> creator) {
//A[] arr = new A[length];       // (8) Cannot create generic array!
  return creator.apply(length);  // Lambda expression or

                                 // array constructor reference executed.
}
```

The code below calls the generic method `createArray()` with a lambda expression and an array constructor reference at (9) and (10), respectively, to create a `String` array of length 5. The target type in both cases is parameterized functional interface `IntFunction<String[]>`.

Click here to view code image

```
// n -> String[]
String[] strArrLE = createArray(5, n -> new String[n]); // (9)
String[] strArrACE = createArray(5, String[]::new);     // (10)
```

13.14 Contexts for Defining Lambda Expressions

In this section, we summarize the main contexts that can provide target types for lambda expressions and method references.

Declaration and Assignment Statements

Ample examples of defining lambda expressions and method references in this context have been presented throughout this chapter. As we have seen earlier, the target type is inferred from the type of the assignment target—that is, the functional interface type being assigned to on the left-hand side of the assignment operator.

Click here to view code image

```
DoubleFunction cToF = x -> 1.8 * x + 32.0;           // double -> double
ToIntFunction<String> lenFunc1 = s -> s.length();    // String -> int
ToIntFunction<String> lenFunc2 = String::length;     // String -> int
lenFunc1 = s -> Integer.parseInt(s);                 // String -> int
lenFunc2 = Integer::parseInt;                        // String -> int
```

Method and Constructor Calls

We have seen several examples where a lambda expression is passed as an argument in a method or constructor call. The target type is the functional interface type of the corresponding formal parameter.

Click here to view code image

```
List<Integer> numbers = Arrays.asList(1, 2, 3);
numbers.forEach(i -> System.out.println(i));   // Target type: Consumer<Integer>
numbers.forEach(System.out::println);
```

Expressions in `return` Statements

The expression in a `return` statement can define a lambda expression (or a method reference) whose target type is the return type of the method. The method below returns a function of type `int -> int`:

Click here to view code image

```
static IntUnaryOperator createLinearFormula(int a, int b) {
  return x -> a * x + b;      // int -> int
}

// Client code:
IntUnaryOperator y = createLinearFormula(10, 5);  // 10 * x + 5
y.applyAsInt(2);                                  // 25
```

Ternary Conditional Expressions

For lambda expressions defined in a ternary conditional expression, the target type is inferred from the context of the ternary conditional expression.

In the first ternary conditional expression below, the target type for the lambda expressions that are operands is inferred from the target of the assignment statement, which happens to be an `IntUnaryOperator` interface.

Click here to view code image

```
int ii = 10;
IntUnaryOperator iFunc1 = ii % 2 == 0 ? i -> i * 2 : j -> j + 1;   // int -> int
iFunc1.applyAsInt(4);                                              // 8
//IntUnaryOperator iFunc2 = ii % 2 == 0 ? i -> i * 2
//                          : s -> Integer.parseInt(s);   // Compile-time error!
```

In the second ternary conditional expression above, the code does not compile because the type `String -> int` of the lambda expression that is the second operand is not compatible with the type `int -> int` of the assignment target.

Cast Expressions

The context of a cast expression can provide the target type for a lambda expression or a method reference.

In the two statements below, the cast provides the target type of the lambda expression. Note that the textual lambda expressions are identical, but their target types are different.

Click here to view code image

```
System.out.println(((IntUnaryOperator) i -> i * 2).applyAsInt(10));        // 20
System.out.println(((DoubleUnaryOperator) i -> i * 2).applyAsDouble(10.0));// 20.0
```

The first statement below does not compile, as the `Object` class is not a functional interface and therefore cannot provide a target type. In the second statement, the cast provides the target type of the constructor expression. The type `Function<String, StringBuilder>` is a subtype of the supertype `Object`, and the assignment is allowed.

Click here to view code image

```
// Object obj1 = StringBuilder::new;                  // Compile-time error!
Object obj2 = (Function<String, StringBuilder>) StringBuilder::new;
```

In the code below, the `instanceof` operator at (1) is used to guarantee that at runtime the cast will succeed at (2) and the lambda expression can be executed at (3). Without the `instanceof` operator, the cast at (2) will be allowed at compile time, but a resounding `ClassCastException` will be thrown at runtime, as `DoubleUnaryOperator` and

`IntUnaryOperator` are not related types. In the code below, the body of the `if` statement is not executed.

Click here to view code image

```
Object uFunc1 = (IntUnaryOperator) i -> i * 2;
if (uFunc1 instanceof DoubleUnaryOperator) {                    // (1) false
  DoubleUnaryOperator uFunc2 = (DoubleUnaryOperator) uFunc1;    // (2)
  uFunc2.applyAsDouble(10.0);                                   // (3)
}
```

Nested Lambda Expressions

When lambda expressions are nested, the context of the outer lambda expressions can provide the target type for the inner lambda expressions. This typically occurs when currying functions.

Click here to view code image

```
Supplier<Supplier<String>> f = () -> () -> "Hi";
```

The target type for the nested lambda expressions is inferred from the context, which is an assignment statement to a reference of type `Supplier<Supplier<String>>`. The inner lambda expression `() -> "Hi"` is inferred to be of target type `Supplier<String>`, as its type `() -> String` is compatible with the function type of this target type. The outer lambda expression is then inferred to have the type `() -> Supplier<String>` which is compatible with the target type `Supplier<Supplier<String>>`.

Review Questions

13.4 Given the following code:

Click here to view code image

```
import java.util.*;
public class Test13RQ5 {
  public static void main(String[] args) {
    List<String> values = new ArrayList<>(List.of("ONE","TWO","THREE","FOUR"));
    values.removeIf(s -> s.length() == 3);
    int sum = 0;
    for (String value: values) {
      sum += value.length();
    }
    System.out.println(sum);
  }
}
```

What is the result?

Select the one correct answer.

a. `3`

b. `6`

c. `9`

d. The program will throw an exception at runtime.

e. The program will fail to compile.

13.5 Given the following code:

Click here to view code image

```
import java.util.*;
public class Test13RQ6 {
  public static void main(String[] args) {
    List<String> values
        = new ArrayList<>(List.of("ANNA","JANE","ALICE","JOHN"));
    values.removeIf(s -> s.toLowerCase().startsWith("a"));
    System.out.println(values);
  }
}
```

What is the result?

Select the one correct answer.

a. `[jane, john]`

b. `[anna, alice]`

c. `[JANE, JOHN]`

d. `[ANNA, ALICE]`

e. `[ANNA, JANE, ALICE, JOHN]`

f. `[anna, jane, alice, john]`

g. The program will compile, but it will not produce any output when run.

h. The program will throw an exception at runtime.

i. The program will fail to compile.

13.6 Given the following code:

Click here to view code image

```
import java.util.*;
public class Test13RQ8 {
  public static void main(String[] args) {
    List<String> values
        = new ArrayList<>(List.of("ANNA","JANE","ALICE","JOHN"));
    String s = values.get(0).substring(0,1);
    values.removeIf(s -> s.toLowerCase().startsWith(s));
    values.forEach(s -> System.out.print(s + " "));
  }
}
```

What is the result?

Select the one correct answer.

a. `jane john`

b. `anna alice`

c. `JANE JOHN`

d. `ANNA ALICE`

e. `ANNA JANE ALICE JOHN`

f. `anna jane alice john`

g. The program will compile, but it will not produce any output when run.

h. The program will throw an exception at runtime.

i. The program will fail to compile.

13.7 Given the following code:

Click here to view code image

```
import java.util.*;
import java.util.function.*;
public class Test13RQ9 {
  public static void main(String[] args) {
    List<String> values
        = new ArrayList<>(List.of("PLOT","FLOP","LOOP","LEAP"));
    Predicate<String> filter1 = s -> s.contains("O");
    Predicate<String> filter2 = s -> s.endsWith("P");
    values.removeIf(filter1.and(filter2).negate());
    System.out.println(values);
  }
}
```

What is the result?

Select the one correct answer.

a. `[LEAP]`

b. `[PLOT]`

c. `[FLOP, LOOP]`

d. `[PLOT, FLOP, LOOP]`

e. `[PLOT, FLOP, LOOP, LEAP]`

f. The program will compile, but it will not produce any output when run.

g. The program will throw an exception at runtime.

h. The program will fail to compile.

13.8 Given the following code:

Click here to view code image

```
import java.util.*;
import java.util.function.*;
public class Test13RQ10 {
```

```
    public static void main(String[] args) {
      List<String> values = Arrays.asList("ALICE","BOB","JOHN","JANE");
      UnaryOperator<String> f1 = v1 -> v1.substring(0,1).toUpperCase();
      UnaryOperator<String> f2 = v2 -> v2.substring(1).toLowerCase();
      UnaryOperator<String> f3 = f1.compose(f2);
      values.replaceAll(f3);
      System.out.println(values);
    }
  }
```

What is the result?

Select the one correct answer.

a. `[Alice, Bob, John, Jane]`

b. `[aLICE, bOB, jOHN, jANE]`

c. The program will throw an exception at runtime.

d. The program will fail to compile.

13.9 Given the following code:

Click here to view code image

```
  import java.util.*;
  import java.util.function.*;
  public class Test13RQ11 {
    public static void main(String[] args) {
      List<String> values = Arrays.asList("ALICE","BOB","JOHN","JANE");
      UnaryOperator<String> f1 = v -> v.toLowerCase();
      values.replaceAll(f1);
      Consumer<String> c1 = s -> s = s.substring(0,1).toUpperCase();
      Consumer<String> c2 = s -> System.out.print(s + " ");
      values.forEach(c1.andThen(c2));
    }
  }
```

What is the result?

Select the one correct answer.

a. `Alice Bob John Jane`

b. `alice bob john jane`

c. `A B J J`

d. The program will throw an exception at runtime.

e. The program will fail to compile.

13.10 Which method references are equivalent to lambda expressions at (1) and (2) in the following code?

Click here to view code image

```
  import java.util.*;
  class Test {
```

```
    private List<Integer> values = new ArrayList<>(List.of(1,2,3,4,5));
    public List<Integer> getValues()       { return values; }
    public static boolean isEven(int value) { return value % 2 != 0; }
    public void printValue(int value)       { System.out.print(value + " "); }
  }

  public class Main {
    public static void main(String[] args) {
      Test test = new Test();
      test.getValues().removeIf(v -> v % 2 != 0);                // (1)
      test.getValues().forEach(v -> System.out.print(v + " ") ); // (2)
    }
  }
```

Select the one correct answer.

a. `test.getValues().removeIf(Test::isEven);`
`test.getValues().forEach(Test::printValue);`

b. `test.getValues().removeIf(Test::isEven);`
`test.getValues().forEach(test::printValue);`

c. `test.getValues().removeIf(test::isEven);`
`test.getValues().forEach(test::printValue);`

d. `test.getValues().removeIf(test::isEven);`
`test.getValues().forEach(Test::printValue);`

e. None of the above

13.11 Which statement is true about method referencing?

a. Unbounded instance method reference determines the target reference as the first argument passed to the method reference.

b. Unbounded instance method reference determines the target reference as the last argument passed to the method reference.

c. Bounded instance method reference determines the target reference as the first argument passed to the method reference.

d. Bounded instance method reference determines the target reference as the last argument passed to the method reference.

13.12 Given the following code:

Click here to view code image

```
  import java.util.function.BiFunction;
  public class Test24RQ6 {
    public static void main(String[] args) {
      BiFunction divide = (x, y) -> x/y;
      System.out.print(divide.apply(0.0,0));
    }
  }
```

What is the result?

Select the one correct answer.

a. `0`

b. `0.0`

c. The program will throw an exception at runtime.

d. The program will fail to compile.

13.13 Given the following code:

Click here to view code image

```
import java.util.function.Function;
public class Test13RQ19 {
  public static void main(String[] args) {
    Function f1 = (x) -> "<" + x;
    Function f2 = (x) -> x + ">";
    System.out.print(f2.compose(f1).apply(42));
  }
}
```

What is the result?

Select the one correct answer.

a. `<42>`

b. `>42<`

c. The program will throw an exception at runtime.

d. The program will fail to compile.