Chapter 10
The ArrayList<E> Class Introduction to Functional Programming

Reference:

A Programmer's Guide to Java SE 8 Oracle Certified Associate (OCA),
Khalid A. Mughal, Rolf W. Rasmussen:
Addison-Wesley Professional, 2016, ISBN: 0132930218
(http://www.ii.uib.no/~khalid/ocajp8/)

Khalid Azim Mughal
Associate Professor Emeritus
Department of Informatics
University of Bergen, Norway.
khalid.mughal@uib.no
http://www.ii.uib.no/~khalid

Version date: 2018-11-06

Overview

• ArrayList<E> Class	• Functional Interfaces • Lambda Expressions

Collections

- A *collection* is a data structure that can maintain a group of objects so that the objects can be manipulated as a *single entity* or *unit*.
  - Objects can be stored, retrieved, and manipulated as *elements* of a collection.

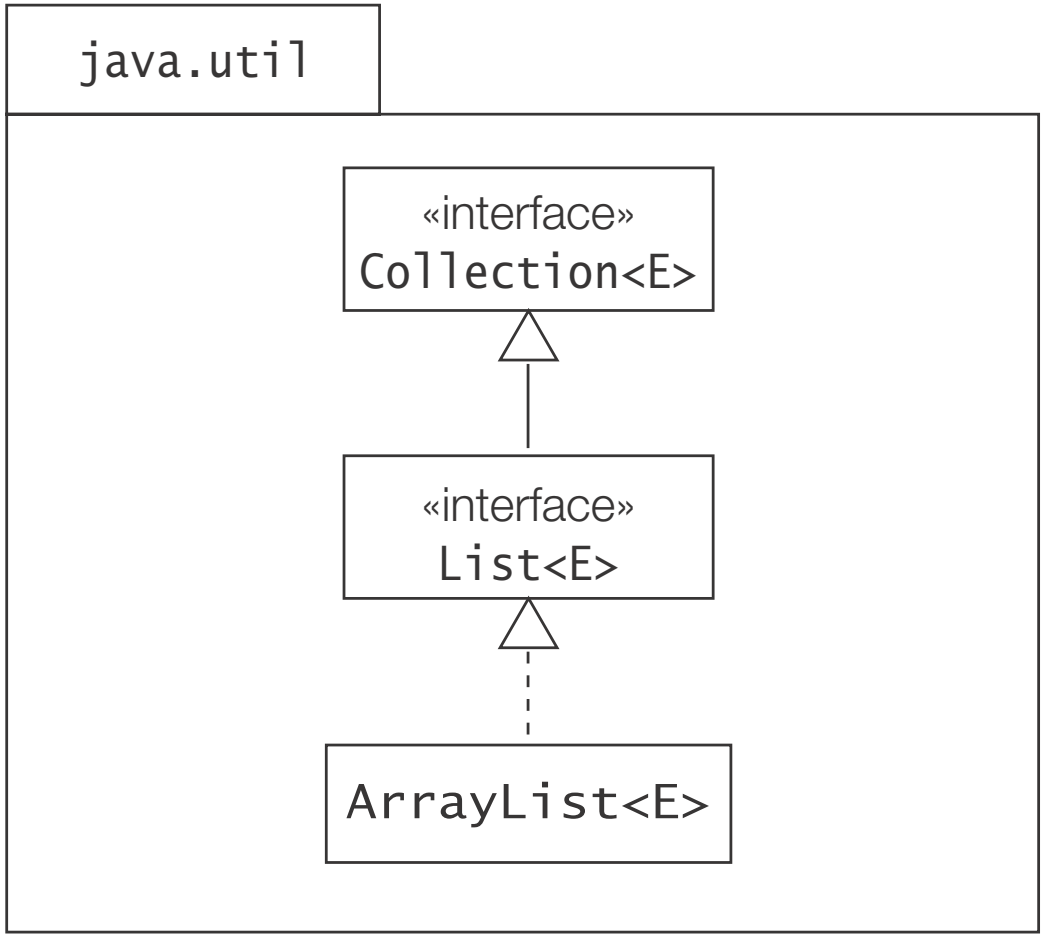
The `ArrayList<E>` Class

- The `java.util.ArrayList<E>` class is a *dynamically resizable* implementation of the `java.util.List<E>` interface using arrays (a.k.a. *dynamic arrays*).
- The order of elements in a list is *positional or insertion order*, given by a zero-based index.

Sorting vs. Ordering

- *Sorting implies ordering* the elements in a collection according to some *ranking criteria*, usually based on the *values* of the elements.
- The elements in such a list are *ordered*, but *not* sorted, as it is not the values of the elements that determine their ranking in the list.
- Ordering does *not* necessarily imply sorting.

Figure 10.1 Partial `ArrayList` *Inheritance Hierarchy in the Java Collections Framework*



Declaring References and Constructing `ArrayLists`

```
ArrayList()
ArrayList(int initialCapacity)
ArrayList(Collection<? extends E> c)
```

The default constructor creates a new, empty `ArrayList` with initial capacity of ten.

The second constructor creates a new, empty `ArrayList` with the specified initial capacity.

The third constructor creates a new `ArrayList` containing the elements in the specified collection.

- Create an empty `ArrayList` of `String`.

  ```
  ArrayList<String> palindromes = new ArrayList<String>(); // (1)
  ```

- The capacity of a list and its size can change dynamically as the list is manipulated.

The Diamond Operator: <>

- The *diamond operator* (<>) can be used in the `ArrayList` creation expression on the right-hand side of the declaration statement.

```
ArrayList<String> palindromes = new ArrayList<>(); // Using the diamond operator
```

- If the diamond operator is omitted, the compiler will issue an *unchecked conversion warning*.

Best practices advocates *programming to an interface***.**

- Great flexibility in substituting other objects for a task when necessary.

```
List<String> palindromes = new LinkedList<>(); // Changing implementation.
```

Appending Elements

- The `add(E)` method appends an element after the last element in list, thereby increasing the list size by 1.

```
palindromes.add("level"); palindromes.add("Ada"); palindromes.add("kayak");
System.out.println(palindromes);

[level, Ada, kayak]                          // Default list format.
```

Constructing from a Collection

- A third constructor allows an `ArrayList` to be constructed from another collection.

```
List<String> wordList = new ArrayList<>(palindromes);
System.out.println(wordList); // [level, Ada, kayak]
wordList.add("Naan");
System.out.println(wordList); // [level, Ada, kayak, Naan]
```

`ArrayList`*s are type-safe.*

```
List<StringBuilder> synonyms   = new ArrayList<>(); // List of StringBuilder
List<Integer> attendance       = new ArrayList<>(); // List of Integer
List<List<String>> listOfLists = new ArrayList<>(); // (3) List of List of String
List<int> frequencies          = new ArrayList<>(); // (4) Compile-time error!
```

Subtyping

- For Arrays:

```
Object[] objArray = new String[10];                 // (5) OK!
objArray[2] = "Green";                              // (6) OK!
objArray[1] = new Integer(2016);                    // ArrayStoreException!
```

  - Array type information available at runtime.
- For the `ArrayList`:

```
ArrayList<Object> objList1 = new ArrayList<String>();// (7) Compile-time error!
List<Object> objList2 = new ArrayList<String>();     // (8) Compile-time error!
```

  - No type information available at runtime because of *type erasure*.

Modifying an `ArrayList`

```
boolean add(E element)
void add(int index, E element)
```

The first method will append the specified element to the *end* of the list. It returns `true` if the collection was modified as a result of the operation.

The second method inserts the specified element at the specified index. If necessary, it shifts the element previously at this index and any subsequent elements one position toward the end of the list.

The method will throw an `IndexOutOfBoundsException` if the index is out of range (`index < 0 || index > size()`).

The type parameter `E` represents the element type of the list.

```
boolean addAll(Collection<? extends E> c)
boolean addAll(int index, Collection<? extends E> c)
```

The first method inserts the elements from the specified collection at the end of the list.

The second method inserts the elements from the specified collection at the specified index, i.e., the method splices the elements of the specified collection into the list at the specified index.

The methods return `true` if any elements were added. Elements are inserted using an iterator of the specified collection.

The second method will throw an `IndexOutOfBoundsException` if the index is out of range (`index < 0 || index > size()`).

The declaration of the parameter `c` essentially means that parameter c can refer to any collection whose element type is `E` or whose element type is a subtype of `E`.

```
E set(int index, E element)
```

Replaces the element at the specified index with the specified element.

It returns the previous element at the specified index.

The method throws an `IndexOutOfBoundsException` if the index is out of range (`index < 0 || index >= size()`).

```
E remove(int index)
boolean remove(Object element)
```

The first method deletes and returns the element at the specified index.

The method throws an `IndexOutOfBoundsException` if the index is out of range (`index < 0 || index >= size()`).

The second method will remove the *first* occurrence of the element from the list, using object value equality. The method returns `true` if the call was successful.

Both methods will contract the list accordingly if any elements are removed.

```
boolean removeAll(Collection<?> c)
boolean removeIf(Predicate<? super E> filter)
```

The `removeAll()` method removes from this list all elements that are contained in the specified collection.

The `removeIf()` method removes from this list all elements that satisfy the filtering criteria defined by a lambda expression that implements the `Predicate<T>` functional interface (page 50).

Both methods return `true` if the call was successful.

The list is contracted accordingly if any elements are removed.

`void trimToSize()`
Trims the capacity of this list to its current size.
`void clear()`
Deletes all elements from the list. The list is empty after the call, i.e., it has size 0.

- The `add(E)` method appends an element to the end of the list.

```
System.out.println("\n(2) Add elements to list:");
for (String str : wordArray) {
  strList.add(str);
  printListWithIndex(strList);
}
// [0:level, 1:Ada, 2:Java, 3:kayak, 4:Bob, 5:Rotator, 6:Bob]
```

- The `add(int, E)` method inserts a new element at a specific index.

```
// [0:level, 1:Ada, 2:kayak, 3:Bob, 4:Rotator, 5:Bob]
strList.add(2, "Java");      // Insert an element at index 2 in the list.
printListWithIndex(strList);
// [0:level, 1:Ada, 2:Java, 3:kayak, 4:Bob, 5:Rotator, 6:Bob]
```

- The index value equal to 0 or the size of the list is always allowed for the method `add(int, E)`.

```
List<String> list1 = new ArrayList<>();   // []
list1.add(0, "First");                    // [First]
list1.add(list1.size(), "Last");          // [First, Last]
```

- Replace an element at a specified index using the `set(int, E)` method.

```
System.out.println("(3) Replace the element at index 1:");
String oldElement = strList.set(1, "Naan");
System.out.println("Element that was replaced: " + oldElement);   // "Ada"
printListWithIndex(strList);
// [0:level, 1:Naan, 2:Java, 3:kayak, 4:Bob, 5:Rotator, 6:Bob]
```

- Remove elements from a list with the `remove()` method, and the list contracts accordingly.

```
System.out.println("(4) Remove the element at index 0:");
System.out.println("Element removed: " + strList.remove(0));      // "level"
printListWithIndex(strList);
// [0:Naan, 1:Java, 2:kayak, 3:Bob, 4:Rotator, 5:Bob]

System.out.println("(5) Remove the first occurrence of \"Java\":");
System.out.println("Element removed: " + strList.remove("Java")); // true
printListWithIndex(strList);
// [0:Naan, 1:kayak, 2:Bob, 3:Rotator, 4:Bob]
```

- The method `remove(Object)` requires that the argument object overrides the `equals()` method from the `Object` class, which only determines reference value equality.

Primitive Values and ArrayList

- Since primitive values cannot be stored in an `ArrayList`, we can use the wrapper classes to box such values first.

```
List<Integer> intList = new ArrayList<>();
intList.add(10); intList.add(20); intList.add(1);
System.out.println(intList);                                    // [10, 20, 1]
System.out.println("Element to be removed: " + 1);              // 1
System.out.println("Element removed: " + intList.remove(1));    // 20
System.out.println(intList);                                    // [10, 1]

System.out.println("Element removed: " + intList.remove(new Integer(1)));// true
System.out.println(intList);                                    // [10, 20]
```

- Overloading chooses the *most specific* method.

Querying an `ArrayList`

`int size()`
Returns the number of elements currently in the list.

In a non-empty list, the first element is at index 0 and the last element is at `size()-1`.

`boolean isEmpty()`
Determines whether the list is empty, i.e., whether its size is 0.

`E get(int index)`
Returns the element at the specified *positional index*.

The method throws an `IndexOutOfBoundsException` if the index is out of range (`index < 0 || index >= size()`).

`boolean equals(Object o)`
Compares the specified object with this list for object value equality.

Returns `true` if and only if the specified object is also a list, both lists have the same size, and all corresponding pairs of elements in the two lists are equal according to object value equality.

`boolean contains(Object element)`
Determines whether the argument object is contained in the collection, using object value equality. This is called the *membership test*.

```
int indexOf(Object o)
int lastIndexOf(Object o)
```
These methods return the index of the first and the last occurrence of the element that is equal (using object value equality) to the specified argument, respectively, if such an element exists in the list; otherwise, the value –1 is returned. These methods provide *element search* in the list.

- The method `get(int)` retrieves the element at the specified index.

```
System.out.println("First element: " + strList.get(0);                 // Naan
System.out.println("Last element: " + strList.get(strList.size()-1));  // Bob
```

- The `equals()` method of the `ArrayList` class can be used to compare two lists for equality with regard to size and corresponding elements being equal in each list.

```
List<String> strList2 = new ArrayList<>(strList);
boolean trueOrFalse = strList.equals(strList2); // true
```

- The membership test is carried out by the `contains(Object)` method. We can find the index of a specified element in the list by using the `indexOf()` and `lastIndexOf()` methods.

```
boolean found = strList.contains("Naan");   // true
int pos = strList.indexOf("Bob");           // 2
pos = strList.indexOf("BOB");               // -1 (Not found.)
pos = strList.lastIndexOf("Bob");           // 4 (the last occurrence)
```

Traversing an `ArrayList`

- Positional access to traverse a list with the `for(;;)` loop.

```
public static <E> void printListWithIndex(List<E> list) {
  List<String> newList = new ArrayList<>();
  for (int i = 0; i < list.size(); i++) {
    newList.add(i + ":" + list.get(i));
  }
  System.out.println(newList);
}

// [0:level, 1:Ada, 2:kayak, 3:Bob, 4:Rotator, 5:Bob]
```

- Since the `ArrayList` class implements the `Iterable` interface (i.e., the class provides an iterator), we can use the `for(:)` loop to traverse a list.

```
for (String str : strList) {
  System.out.print(str + " ");
}
```

- Removing elements from the list while traversing the list.
  - The `for(:)` loop does not allow the list structure to be modified:

```
for (String str : strList) {
  if (str.length() <= 3) {
    strList.remove(str);              // Throws ConcurrentModificationException
```

```
    }
}
```

- Using positional access in a loop to traverse the list and remove elements can be tricky.
- An iterator can also be used explicitly for this purpose.
- Better solution: Using the `ArrayList.removeIf()` method, passing the criteria (as a `Predicate`) for selection as argument (page 50).

- *Streams on ArrayLists (topic for another day).*

Converting an `ArrayList` to an Array

```
Object[] toArray()
<T> T[] toArray(T[] a)
```

The first method returns an array of type `Object` filled with all the elements of a collection.

The second method is a generic method that stores the elements of a collection in an array of type `T`.

If the specified array is big enough, the elements are stored in this array.

If there is room to spare in the array, that is, the length of the array is greater than the number of elements in the collection, the element immediately after storing the elements of the collection is set to the `null` value before the array is returned.

If the array is too small, a new array of type `T` and appropriate size is created.

If `T` is not a supertype of the runtime type of every element in the collection, an `ArrayStoreException` is thrown.

- Using the first `toArray()` method.

```
System.out.println("(14) Convert list to array:");
Object[] objArray = strList.toArray();                  // Object[]
String str = (String) objArray[0];                      // Cast required.

String[] strArray = (String[]) strList.toArray();       // Cast required.
```

- Using the second `toArray()` method.

```
String[] strArray = strList.toArray(new String[0]);        // String[]
```

Sorting an `ArrayList`

- The following static method of the **`java.util.Collections`** class can be used to sort elements of a list.

> `static <T extends Comparable<? super T>> void sort(List<T> list)`
>
> This generic method sorts the specified list into ascending order, according to the *natural ordering* of its elements.
>
> The declaration essentially says that elements of the list have to be *comparable*, i.e., can be compared with the `compareTo()` method of the `Comparable` interface.
>
> `static <T> void sort(List<T> list, Comparator<? super T> c)`
>
> Sorts the specified list according to the order induced by the specified comparator.

- Sorting according to *natural order*:

```
//[Naan, kayak, Bob, Rotator, Bob]
Collections.sort(strList);
//[Bob, Bob, Naan, Rotator, kayak]
```

- Sorting according to a given *total order*:

```
//[Naan, kayak, Bob, Rotator, Bob]
Collections.sort(strList, Comparator.reverseOrder());
//[kayak, Rotator, Naan, Bob, Bob]
```

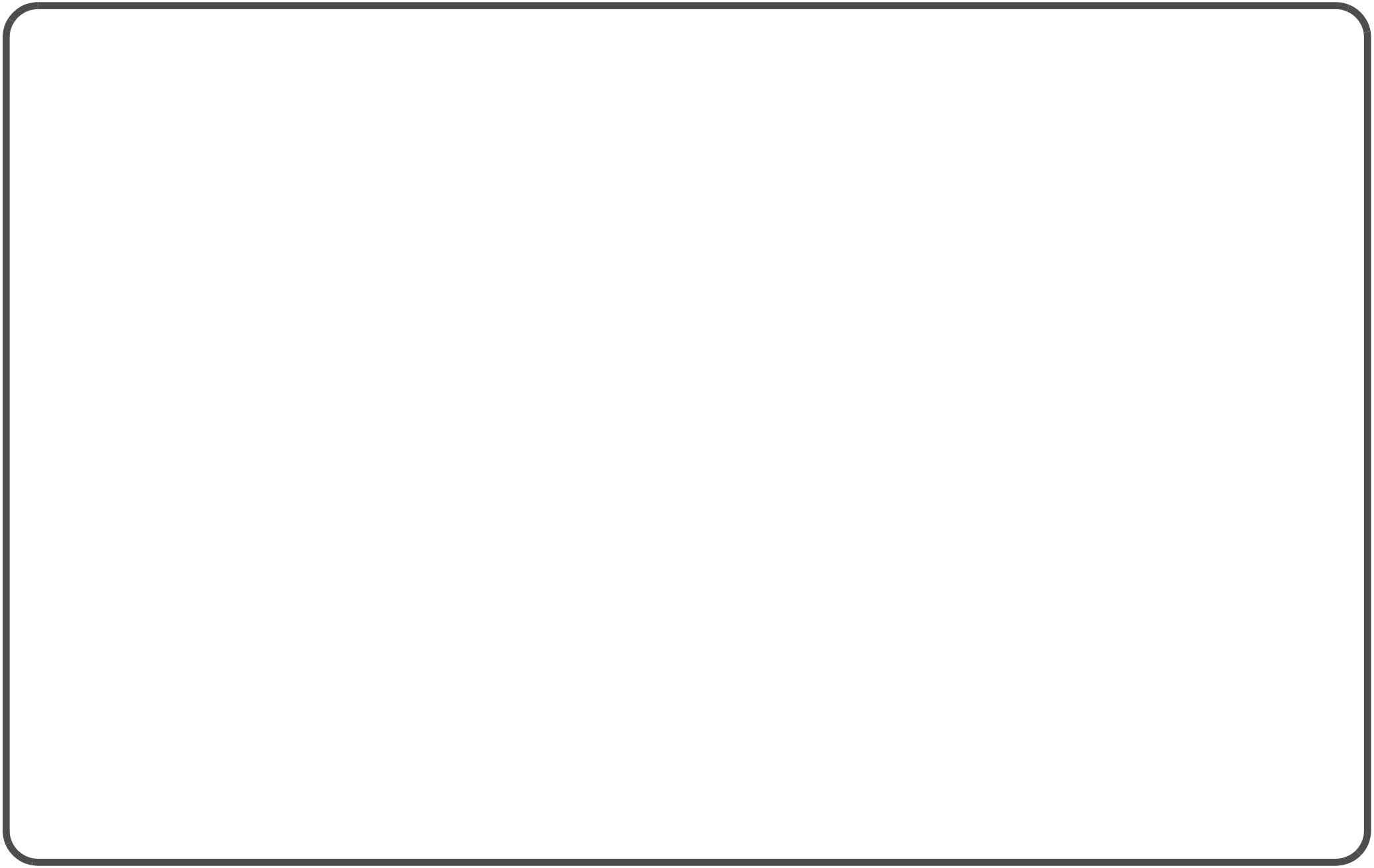


Table 10.1 Summary of Arrays vs. `ArrayList`

	Arrays	`ArrayList`
Construct support	Built into the language	Provided by a generic class
Initial length/ size specification	Length is specified in the array construction expression directly or indirectly by the initialization block.	Cannot specify the size at construction time. However, initial capacity can be specified.
Length/ Size	The length of an array is static, i.e., fixed, once it is created. Each array has a `public final int` field called `length`. (The `String` and the `StringBuilder` class provide the method `length()` for this purpose).	Both size and capacity can change dynamically. `ArrayList` provides the method `size()` to obtain the current size of the list.
Element type	Primitive and reference types	Only reference types
Operations on elements	An element in the array is designated by the array name and an index using the `[]` operator, and can be used as a simple variable.	The `ArrayList` class provides various methods to add, insert, retrieve, replace, and remove elements from a list.

Table 10.1 Summary of Arrays vs. `ArrayList` *(Continued)*

	Arrays	`ArrayList`
Iterator	Arrays do not provide an iterator, apart from using the `for(:)` loop for traversal.	The `ArrayList` class provides customized iterators for lists, in addition to the `for(:)` loop for traversal.
Generics	Cannot create arrays of generic types using the `new` operator. Runtime check required for storage at runtime.	`ArrayList` is a generic type. Can create `ArrayList` of reference types using the `new` operator. No runtime check required for storage at runtime as type-safety is checked at compile time.
Subtype relationship	Subtype relationship between two reference types implies subtype relationship between arrays of the two types.	Subtype relationship between two reference types does not imply subtype relationship between `ArrayList`s of the two types.
Sorting	`java.util.Arrays.sort(array)`	`java.util.Collections.sort(list)`
Textual representation	`java.util.Arrays.toString(array)`	`list.toString()`

Functional-Style Programming

- Before Java 8, the language only supported object-oriented programming (OOP).
  - Packing *state* and *behavior* into objects that communicate in a *procedural* manner.
- Java 8 brings *functional-style programming* into the language.
  - Code representing *functionality* can be passed as values to tailor the *behavior* of methods.
- *Behavior Parameterization*: implementing behavior using lambda expressions and functional interfaces.
- Functional-style programming is also beneficial in developing parallel code.
- *Filtering*: select elements that satisfy a certain criteria.
- Running example: filter a list for one-word *palindromes*, i.e., words that spell the same forwards and backwards. For example, `"anana"` is a palindrome, but `"banana"` is not.

Filtering Using Customized Methods

- Disadvantage: Requires a new method for each new criteria (`FunWithPalindromesV0.java`).

Filtering Using an Interface

- String selection criteria defined by an interface (`FunWithPalindromesV1.java`):

```
public interface StrPredicate {                                    // (1)
  boolean test(String str);
}
```

- The selection method `filterStrings(List<String> strList, StrPredicate predicate)` returns a list with elements that satisfy the selection criteria.
- A *predicate* is a function that takes an argument and returns a `boolean` value.
- Usage:
  - Define *concrete* classes that implement the `StrPredicate` interface.
  - Use *anonymous classes* to instantiate the criteria object.
- Disadvantage: Too much code to encapsulate a single method.

Functional Interfaces and Lambda Expressions

- A functional interface has *exactly one abstract method*.
  - Knowing the name of the functional interface, all information about its sole abstract method can be inferred.
- The `StrPredicate` interface is an example of a *functional interface*.
- Implementation of the sole abstract method of a functional interface can be provided by a *lambda expression*.

```
StrPredicate predicate1 = (String str) ->
    str.equals(new StringBuilder(str).reverse().toString());          // (2)
```

  - A *parameter list* that is analogous to the parameter list of a method.

```
(String str)
```

  - The `->` operator (a.k.a. the *arrow*) that separates the parameter list from the lambda body.
  - A *lambda body* that is either a *single expression* or a *statement block*.

```
str.equals(new StringBuilder(str).reverse().toString()); // Lambda body expr
```

- The lambda expression at (2) defines a *anonymous function*:

```
(String str) -> boolean
```

  - Same signature as the `test()` method of the `StrPredicate` functional interface.

- Lambda expressions can be stored as *values* in references.
- *Type-safety checking*: The compiler can type checks that the lambda expression represents an anonymous function that is compatible with the sole abstract method `of` a functional interface.
- *Lazy Evaluation* of a lambda expression:
  - It is only executed when the sole abstract method is called on the interface with the appropriate argument.
- Lambda expressions provide a precise, concise and readable solution.

Filtering Using the `Predicate<T>` Functional Interface

- The generic `java.util.function.Predicate<T>` functional interface can be to implement predicates.

```
public interface <E> Predicate<E> {
  boolean test(E element);
  // ...
}
```

- Class `FunWithPalindromesV3` uses the Predicate functional interface for filtering palindromes.

Functional Interfaces

- A functional interface can only have one abstract method, called the *functional method* for that interface.
  - Abstract methods declared in an interface are implicitly `abstract` and `public`.

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

- The `@FunctionalInterface` annotation:
  - The compiler will issue an error if the declaration violates the definition of a functional interface.

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

- A functional interface can also provide *explicit* `public` `abstract` method declarations for *non-final* `public` instance methods in the `Object` class, but these are *excluded* from the definition of a functional interface.

```
@FunctionalInterface
```

```
interface StrFormat {                                  // Compile-time error!
  @Override String toString();                         // From Object class
}
```

- Like any other interface, a functional interface can have any number of static and default methods.

```
@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
  @Override boolean equals(Object obj);                  // (4) From Object class
}
```

  - The generic functional interface `java.util.function.Predicate<T>` also has one static method (`isEqual()`) and three default methods (`and()`, `or()`, `negate()`).
  - See also the `java.util.Comparator<E>` functional interface.

- The functional subinterface `IStrPredicate` below is customized to the `String` type by extending the `java.util.function.Predicate<T>` functional interface, where the type parameter `T` is `String`.

```
@FunctionalInterface
interface IStrPredicate extends Predicate<String> { }
```

Built-in Functional Interfaces in Java SE platform API

- In the `java.lang` package there are five functional interfaces: `Runnable`, `Comparable<T>`, `AutoCloseable`, `Iterable<T>`, and `Readable`.
- The `java.util` package contains the `Comparator<E>` functional interface.
- Main support for functional interfaces is found in the `java.util.function` package.

Table 10.2 Selected Functional Interfaces from the **`java.util.function`** *Package*

Functional Interface (T and R are type parameters.)	**Abstract Method Signature**	**Function**
`Predicate<T>`	`test: T -> boolean`	Evaluate a predicate on a `T`.
`Consumer<T>`	`accept: T -> void`	Perform action on a `T`.
`Function<T, R>`	`apply: T -> R`	Transform a `T` to an `R`.
`UnaryOperator<T>`	`apply: T -> T`	Operator on a unary argument.
`BinaryOperator<T>`	`apply: (T, T) -> T`	Operator on binary arguments.
`Supplier<T>`	`get: () -> T`	Provide an instance of a `T`.

Specialized Built-in Functional Interfaces for Primitive Values

- The `java.util.function` package includes functional interfaces that are specialized for primitive values.
- Avoid excessive boxing and unboxing of primitive values when such values are used as objects.
- The functional interfaces `IntPredicate`, `LongPredicate` and `DoublePredicate` provide an abstract `test()` method to evaluate predicates with `int`, `long` and `double` arguments, respectively.

```
Predicate<Integer> integerPred = (Integer i) -> i%2 == 0;// i as operand unboxed.
System.out.println(integerPred.test(2015));              // Argument boxed. false

IntPredicate intPred = (int i) -> i%2 == 0;
System.out.println(intPred.test(2016));                  // true
```

Defining Lambda Expressions

- Lambda expressions implement functional interfaces.
- A lambda expression has the following syntax:

  formal_parameter_list `->` *lambda_body*
- The parameter list and the body are separated by the `->` operator.
- 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 parenthesis, `()`, analogous to the parameter list in a method declaration.
  - Other shorthand forms introduced shortly.
- If the types of the parameters are specified, they are known as *declared-type parameters*.
- If the types of the parameters are not specified, they are known as *inferred-type parameters*.
- Parameters are either all declared-type or all inferred-type.
- Parentheses are mandatory with multiple parameters.
- For a parameter list with a single inferred-type parameter, the parentheses can be omitted.
- Only declared-type parameters can have modifiers.
- Examples of lambda expressions:

```
() -> ..                                  // Empty parameter list
(Integer x, Integer y, Integer z) -> ..   // Multiple declared-type parameters
(x, y, z)    -> ..                        // Multiple inferred-type parameters
(String str) -> ..                        // Single declared-type parameter
(str)        -> ..                        // Single inferred-type parameter
```

```
str          -> ..                          // Single inferred-type parameter
String str   -> ..                          // Illegal: Missing parentheses
Integer x, Integer y, Integer z -> ..       // Illegal: Missing parentheses
i, j, k      -> ..                          // Illegal: Missing parentheses
(String str, j)      -> ..  // Illegal: cannot mix inferred and declared type
(final int i, int j) -> ..  // Ok. Modifier with declared-type parameter
(final i, j)         -> ..  // Illegal: no modifier with inferred-type parameter
```

Lambda Body

- A lambda body is either a *single expression* or a *statement block*.
- Execution of a lambda body results in one of the following actions:
  - a non-`void` return
  - a `void` return
  - throws an exception
- 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, i.e., has a non-`void` return.

```
() -> 2015                                          // 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.

```
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
```

  - However, if the abstract method of the functional interface returns `void`, the non-`void` return of a lambda expression with an expression statement as body, can be interpreted as a `void` return, i.e., the return value is ignored.

- The following examples are not legal lambda expressions:

```
(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 braces (`{}`).
  - The `return` statement is only allowed in a block lambda body.

```
() -> {}                                        // Block body, void return
() -> { return 2015; }                          // Block body, non-void return
() -> { return 2015 }      // 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!");
}
```

Accessing Members in Enclosing Class

- Since a lambda expression is not associated with any class, there is no notion of the `this` reference.
  - If the `this` reference is used in a lambda expression, 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 in the enclosing object.
  - In the case of shadowing member names, the keywords `this` can be explicitly used, and also the keyword `super` to access any members inherited by the enclosing object.

Example: Accessing Members in Enclosing Object

```
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("Otto"); strList.add("ADA"); strList.add("Alyla");
  strList.add("Bob"); strList.add("HannaH"); strList.add("Java");

  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: " + strList);        // (9)  MembersOnly.strList
          banner.append(str);                            // (10) this.banner
          return str.length() > 5;                       // (11) boolean value
        };
      }
    }
```

- Output from the program:

```
List: [Otto, ADA, Alyla, Bob, HannaH, Java]
true love never dies!
```

Accessing Local Variables in Enclosing Context

- A lambda expression does not create its own scope.
  - It is part of the scope of the enclosing context, i.e., it has *lexical or block scope* (§4.4, p. 119).
  - Local variables not accessible outside of the lambda expression.
  - Cannot *redeclare* local variables already declared in the enclosing scope.
- 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, its value does not change in the method.
  - 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.

  Variable Capture
  - A lambda expression might be executed at a later time, after the method has finished execution.

- At that point, the local variables used in the lambda expression are no longer accessible.
- To ensure their availability, *copies of their (reference) values* are maintained with the lambda expression.
- 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 in the reference.

Example: Accessing Local Variables in Enclosing Method

```
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("Otto"); words.add("ADA"); words.add("Alyla");
    words.add("Bob"); words.add("HannaH"); words.add("Java");

//  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)
      banner.append(str);                              // (9)
      return str.length() > 5;
    };
  }
}
```

- Output from the program:

```
List: [Otto, ADA, Alyla, Bob, HannaH, Java]
true love never dies!
```

Type Checking and Execution of Lambda Expressions

- A lambda expression can 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.
- The compiler uses *type inference* to determine the *target type* that is required in the context where the lambda expression is defined.

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

  – The *type* of the lambda expression in (1) is

```
Integer -> boolean
```

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

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

- The *function type* of a functional interface is the method type of its single abstract 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:
-

Target type is context dependent.

```
IntPredicate p2 = i -> i%2 == 0;        // (2) Target type: IntPredicate
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                         // Function type of the target type.
```

- The type of the lambda expression in (2) is

```
int -> boolean
```

- The type of a lambda expression is determined by the context in which it is defined.

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

Lazy Execution

- The compiler does the type checking for using lambda expressions.
- At runtime, the lambda expression is executed when the sole abstract method of the functional interface is invoked.
- A lambda expressions is defined as a *function* and used like a *method*.

```
boolean result1 = p1.test(2015);                    // false
boolean result2 = p2.test(2016);                    // true
```

Filtering Revisited: The `Predicate<T>` Functional Interface

- The filtering examples in this chapter make heavy use of traversing over a list using a loop.
- The functional-style programming frees us from the tyranny of explicit traversing over collection—for example, using *streams*.

Example: Filtering any `ArrayList`

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

public class FunWithPalindromesV4 {

  private static boolean isCaseSensitivePalindrome(String str) {
    return str.equals(new StringBuilder(str).reverse().toString());
  }

  private static boolean isCaseInsensitivePalindrome(String str) {
    return str.equalsIgnoreCase(new StringBuilder(str).reverse().toString());
  }
```

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

  // Create a list of words:                                          // (1)
  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);

  List<String> palindromes1 = filterStrings(words,                    // (2)
      str -> isCaseSensitivePalindrome(str));
  System.out.println("Case sensitive palindromes:     " + palindromes1);

  List<String> palindromes2 = filterStrings(words, str ->             // (3)
      isCaseInsensitivePalindrome(str));
  System.out.println("Case insensitive palindromes:   " + palindromes2);

  Predicate<String> predicate3 = str -> !isCaseSensitivePalindrome(str); // (4)
  List<String> nonPalindromes = filterStrings(words, predicate3);
  System.out.println("Non-palindromes, case sensitive: " + nonPalindromes);

  Predicate<String> predicate4 = str -> str.length() > 3;             // (5)
  List<String> strGT3 = filterStrings(words, predicate4);
  System.out.println("Words with length > 3:          " + strGT3);
```

```
Predicate<String> predicate5 = str ->                                    // (6)
    str.length() > 3 && isCaseSensitivePalindrome(str);
List<String> palindromesGT3 = filterStrings(words, predicate5);
System.out.println("Case sensitive palindromes, length > 3: "
                   + palindromesGT3);

Predicate<String> predicateA = str -> {                                  // (7)
  return str.length() > 3 && isCaseSensitivePalindrome(str);
};
System.out.println("Case sensitive palindromes, length > 3: "
                    + filterStrings(words, predicateA));

Predicate<String> predicateB = str -> {                                  // (8)
  boolean result1 = str.length() > 3;
  boolean result2 = isCaseSensitivePalindrome(str);
  return  result1 && result2;
};
System.out.println("Case sensitive palindromes, length > 3: "
                    + filterStrings(words, predicateB));

Predicate<String> predicateC = str -> {                                  // (9)
  if (str == null || str.equals("") || str.length() <= 3) {
```

```
    return false;
  }
  StringBuilder sb = new StringBuilder(str);
  boolean result = str.equals(sb.reverse().toString());
  return result;
};
System.out.println("Case sensitive palindromes, length > 3: "
                   + filterStrings(words, predicateC));

Predicate<String> predicateD = str ->                                  // (10)
    (str == null || str.equals("") || str.length() <= 3)
    ? false: isCaseSensitivePalindrome(str);
System.out.println("Case sensitive palindromes, length > 3: "
                   + filterStrings(words, predicateD));

Predicate<String> predicateE = str ->                                  // (11)
    str != null &&  !str.equals("") &&  str.length() > 3
    &&  isCaseSensitivePalindrome(str);
System.out.println("Case sensitive palindromes, length > 3: "
                  + filterStrings(words, predicateE));

// Removing elements from a list:
words.removeIf(str -> str.indexOf('a') > 0);                           // (12)
```

```
    System.out.println("List of words, no 'a':      " + words);

    words.removeIf(str -> str.length() > 3);                          // (13)
    System.out.println("List of words, length <= 3: " + words);
  }

  /**
   * 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> filterStrings(List<E> list,               // (14)
                                          Predicate<E> predicate) {
    List<E> result = new ArrayList<>();
    for (E element : list)
      if (predicate.test(element))
        result.add(element);
    return result;
  }
}
```

- Output from the program:

```
List of words:                    [Otto, ADA, Alyla, Bob, HannaH, Java]
Case sensitive palindromes:       [ADA, HannaH]
Case insensitive palindromes:     [Otto, ADA, Alyla, Bob, HannaH]
Non-palindromes, case sensitive: [Otto, Alyla, Bob, Java]
Words with length > 3:            [Otto, Alyla, HannaH, Java]
Case sensitive palindromes, length > 3: [HannaH]
Case sensitive palindromes, length > 3: [HannaH]
Case sensitive palindromes, length > 3: [HannaH]
Case sensitive palindromes, length > 3: [HannaH]
Case sensitive palindromes, length > 3: [HannaH]
Case sensitive palindromes, length > 3: [HannaH]
List of words, no 'a':       [Otto, ADA, Bob]
List of words, length <= 3: [ADA, Bob]
```