Chapter 14

How to work with collections, generics, and lambdas
Objectives

Applied

• Write code that creates and works with an array list that stores one or more elements.

• Use generics to specify the type of element that can be stored in a collection.

• Write methods that accept a functional interface as a parameter.

• Write code that calls a method and passes a lambda expression to that method as an argument.
Objectives (cont.)

Knowledge

• Describe the similarities and differences between arrays and collections.
• Name the two main types of collections defined by the collection framework and explain how they differ.
• Describe how the diamond operator works.
• Describe how autoboxing works.
• Explain when it makes sense to use an array list and when it makes sense to use a linked list.
• Describe two benefits of lambda expressions.
How arrays and collections are similar

- Both can store multiple elements of the same type.

How arrays and collections differ

- Arrays are fixed in size. Collections automatically increase their size if necessary.
- Arrays can store primitive types without using wrapper classes. Collections must use wrapper classes to store primitive types.
- Arrays don’t provide methods for operations. Collections provide methods that perform these operations.
Code that uses an array

String[] codes = new String[3];
codes[0] = "java";
codes[1] = "jsp";
codes[2] = "mysql";
for (String s : codes) {
    System.out.println(s);
}

Code that uses a collection

ArrayList<String> codes = new ArrayList<>();
codes.add("java");
codes.add("jsp");
codes.add("mysql");
for (String s : codes) {
    System.out.println(s);
}
The collection framework

- Collection
  - Set
    - HashSet
  - List
    - ArrayList
    - LinkedList
  - Map
    - HashMap
    - TreeMap
Collection interfaces

Collection
Set
List
Map

Common collection classes

ArrayList
LinkedList
HashSet
HashMap
TreeMap
The syntax for specifying the type of elements in a collection

```
Class<Type> collectionName = new Class<Type>();
```

A statement that creates an array list of String objects

```
ArrayList<String> codes = new ArrayList<String>();
```

A statement that creates an array list of integers

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
```

A statement that creates an array list of Product objects

```
ArrayList<Product> products = new ArrayList<Product>();
```
The syntax for using type inference (Java 7 or later)

```
Class<Type> collectionName = new Class<>();
```

A statement that creates an array list of String objects

```
ArrayList<String> codes = new ArrayList<>();
```
The ArrayList class

`java.util.ArrayList`

Common constructors

`ArrayList<E>()`

`ArrayList<E>(intCapacity)`

Code that creates an array list of String objects

With the default starting capacity of 10 elements

```java
ArrayList<String> codes = new ArrayList<>();
```

With a specified starting capacity of 200 elements

```java
ArrayList<String> codes = new ArrayList<>(200);
```
Common methods of the ArrayList class

- `add(object)`
- `add(index, object)`
- `get(index)`
- `size()`

**Code that adds three elements to an array list**

```java
codes.add("jsp");
codes.add("mysql");
codes.add(0, "java");
```

**Code that gets the last element**

```java
int lastIndex = codes.size() - 1;
String lastCode = codes.get(lastIndex);  // "mysql"
```
Code that gets and displays each element

```java
for (String code : codes) {
    System.out.println(code);
}
```

Resulting output

```
java
jsp
mysql
```
An easy way to display a collection

System.out.println(codes);

Resulting output

[java, jsp, mysql]
More methods of the ArrayList class

clear()
contains(object)
indexOf(object)
isEmpty()
remove(index)
remove(object)
set(index, object)
toArray()
Code that replaces an element

codes.set(2, "android");
System.out.println(codes);

Resulting output

[java, jsp, android]
Code that removes an element

String code = codes.remove(1); // removes "jsp"
System.out.println("'" + code + "' was removed.");
System.out.println(codes);

Resulting output

'jsp' was removed.
[java, android]
Code that searches for an element

```java
String searchCode = "android";
if (codes.contains(searchCode)) {
    System.out.println("This list contains: '" +
        searchCode + "'");
}
```

Resulting output

```
This list contains: 'android'
```
Code that stores primitive types in an array list

```java
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(2);
numbers.add(3);
System.out.println(numbers);
```

Resulting output

```
[1, 2, 3]
```

Code that gets a primitive type from an array list

```java
int firstNumber = numbers.get(0);  // 1
```
Code that gets a primitive type from an array list

```java
for (int number : numbers) {
    System.out.println(number);
}
```

Resulting output

```
1
2
3
```
Welcome to the Invoice application

Enter product code: java
Enter quantity: 2
Another line item? (y/n): y

Enter product code: jsp
Enter quantity: 1
Another line item? (y/n): n

<table>
<thead>
<tr>
<th>Description</th>
<th>Price</th>
<th>Qty</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Murach's Java Programming</td>
<td>$57.50</td>
<td>2</td>
<td>$115.00</td>
</tr>
<tr>
<td>Murach's Java Servlets and JSP</td>
<td>$57.50</td>
<td>1</td>
<td>$57.50</td>
</tr>
</tbody>
</table>

Invoice total: $172.50
The Invoice class

```java
package murach.business;

import java.text.NumberFormat;
import java.util.ArrayList;

public class Invoice {

    // the instance variable
    private ArrayList<LineItem> lineItems;

    // the constructor
    public Invoice() {
        lineItems = new ArrayList<>();
    }

    // a method that adds a line item
    public void addItem(LineItem lineItem) { 
        lineItems.add(lineItem);
    }
}
```
The Invoice class (cont.)

    // the get accessor for the line item collection
    public ArrayList<LineItem> getLineItems() {
        return lineItems;
    }

    // a method that gets the invoice total
    public double getTotal() {
        double invoiceTotal = 0;
        for (LineItem lineItem : lineItems) {
            invoiceTotal += lineItem.getTotal();
        }
        return invoiceTotal;
    }

    // returns the invoice total in currency format
    public String getTotalFormatted() {
        NumberFormat currency =
            NumberFormat.getCurrencyInstance();
        return currency.format(getTotal());
    }
The InvoiceApp class

package murach.ui;

import murach.db.ProductDB;
import murach.business.Invoice;
import murach.business.LineItem;
import murach.business.Product;

public class InvoiceApp {
    public static void main(String args[]) {
        System.out.println("Welcome to the Invoice application\n");
        Invoice invoice = new Invoice();
        getLineItems(invoice);
        displayInvoice(invoice);
    }
}
The InvoiceApp class (cont.)

```java
public static void getLineItems(Invoice invoice) {
    String choice = "y";
    while (choice.equalsIgnoreCase("y")) {
        String productCode = Console.getString(
            "Enter product code: ");
        int quantity = Console.getInt(
            "Enter quantity: ");

        Product product =
            ProductDB.getProduct(productCode);
        invoice.addItem(
            new LineItem(product, quantity));

        choice = Console.getString(
            "Another line item? (y/n): ");
        System.out.println();
    }
}
```
public static void displayInvoice(Invoice invoice) {
    StringBuilder sb = new StringBuilder();
    sb.append(StringUtil.pad("Description", 34));
    sb.append(StringUtil.pad("Price", 10));
    sb.append(StringUtil.pad("Qty", 5));
    sb.append(StringUtil.pad("Total", 10));
    sb.append("\n");

    for (LineItem lineItem : invoice.getLineItems()) {
        Product product = lineItem.getProduct();
        sb.append(StringUtil.pad(product.getDescription(), 34));
        sb.append(StringUtil.pad(product.getPriceFormatted(), 10));
        sb.append(StringUtil.pad(lineItem.getQuantityFormatted(), 5));
        sb.append(StringUtil.pad(lineItem.getTotalFormatted(), 10));
        sb.append("\n");
    }
}
sb.append("\nInvoice total: ");
sb.append(invoice.getTotalFormatted());
sb.append("\n");
System.out.println(sb);
Some benefits of lambda expressions

• They can reduce code duplication.
• They can allow you to write methods that are more flexible and easier to maintain.

Some drawbacks of lambda expressions

• They don’t work well with the integrated debugger.
• They can be inefficient.
• They can result in stack traces that are very difficult to understand.
• They can result in code that’s difficult to understand.
• They can result in code duplication for commonly used functionality.
The Contact class

    public class Contact {
        private String name;
        private String email;
        private String phone;

        // constructor and get / set methods here ...
    }

Code that creates a list of contacts

    List<Contact> contacts = new ArrayList<>();
    contacts.add(new Contact(
        "John Doe", "john_doe@foo.com", "555-1212"));
    contacts.add(new Contact(
        "Jane Doe", null, null));
    contacts.add(new Contact(
        "George Doe", "george_doe@foo.com", null));
A method that returns contacts that don’t have phone numbers

```java
public List<Contact> filterContactsWithoutPhone(
    List<Contact> contacts) {

    List<Contact> contactsWithoutPhone =
        new ArrayList<>();
    for (Contact c : contacts) {
        if (c.getPhone() == null) {
            contactsWithoutPhone.add(c);
        }
    }

    return contactsWithoutPhone;
}
```
Code that gets contacts that don’t have phone numbers

```java
List<Contact> contactsWithoutPhone = filterContactsWithoutPhone(contacts);
```

Code that prints contacts to the console

```java
for (Contact p : contactsWithoutPhone) {
    System.out.println(p.getName());
}
```

The output

```
Jane Doe
George Doe
```
A functional interface

```java
public interface TestContact {
    boolean test(Contact c);
}
```

A method that uses a functional interface to specify the filter condition

```java
public List<Contact> filterContacts(
    List<Contact> contacts, TestContact condition) {

    List<Contact> filteredContacts = new ArrayList<>();
    for (Contact c : contacts) {
        if (condition.test(c)) {
            filteredContacts.add(c);
        }
    }
    return filteredContacts;
}
```
Code that gets contacts that don’t have phone numbers

```java
List<Contact> contactsWithoutPhone = filterContacts(contacts, c -> c.getPhone() == null);
```

The list after it’s printed to the console

Jane Doe  
George Doe

Code that gets contacts that don’t have email addresses

```java
List<Contact> contactsWithoutEmail = filterContacts(contacts, c -> c.getEmail() == null);
```

The list after it’s printed to the console

Jane Doe
The Predicate interface that’s available from the java.util.function package

```java
public interface Predicate<T> {
    boolean test(T t);
}
```

A method that uses the Predicate interface to specify the condition

```java
public static List<Contact> filterContacts(
    List<Contact> contacts,
    Predicate<Contact> condition)
{
    List<Contact> filteredContacts = new ArrayList<>();
    for (Contact c : contacts) {
        if (condition.test(c)) {
            filteredContacts.add(c);
        }
    }
    return filteredContacts;
}
```