Chapter 10

More object-oriented programming skills

Objectives

Applied

1. Add two or more classes to a package and make the classes in that package available to other classes.
2. Create a JAR file that contains a library of one or more packages and make that library available to other applications.
3. Create a module by adding a module-info file to a project.
4. Use the modules you create in other applications.
5. Add javadoc comments to the classes in one or more packages and generate the documentation for those packages.
6. Use your web browser to view the documentation you added to a package.
7. Declare and use an enumeration.

Objectives (cont.)

8. Enhance an enumeration by adding methods that override the methods of the Java and Enum classes. Use methods of the enumeration constants when necessary.
9. Use a static import to import the constants of an enumeration or the static fields and methods of a class.

Knowledge

1. List two reasons that you might store classes in a package.
2. Describe how to create a directory structure for a package.
3. Describe how to make one or more packages available to other applications.
4. Describe the characteristics of a Java module.
5. List the four advantages of the module system.

Objectives (cont.)

6. Describe the three things that a module-info file accomplishes, and name the module that is automatically available to other modules.
7. Explain how to use a module in other applications.
8. Explain why you might add javadoc comments to the packages you create.
9. Explain the purpose of using HTML and javadoc tags within a javadoc comment.
10. Explain what an enumeration is and how you use one.
11. Explain what static imports are and how you use them.

The directories and files for an application that uses packages

```
ch10_LineItem\src
    murach
        app
            LineItemApp.java
        business
            LineItem.java
            Product.java
        database
            ProductDB.java
        presentation
            Console.java
```

The LineItem class

```
package murach.business;

import java.text.NumberFormat;

public class LineItem {...}
```

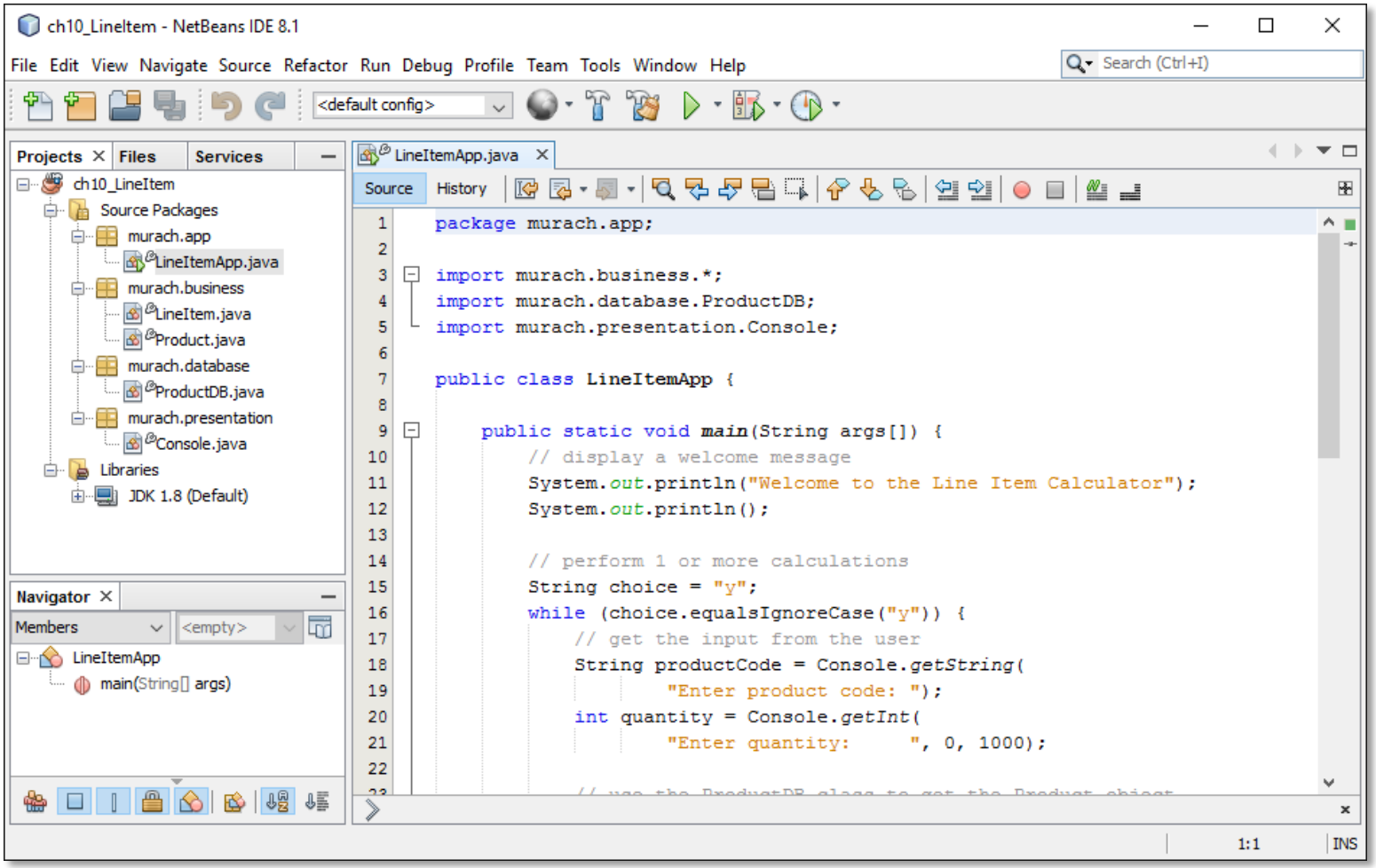
The Product class

```
package murach.business;

import java.text.NumberFormat;

public class Product {...}
```

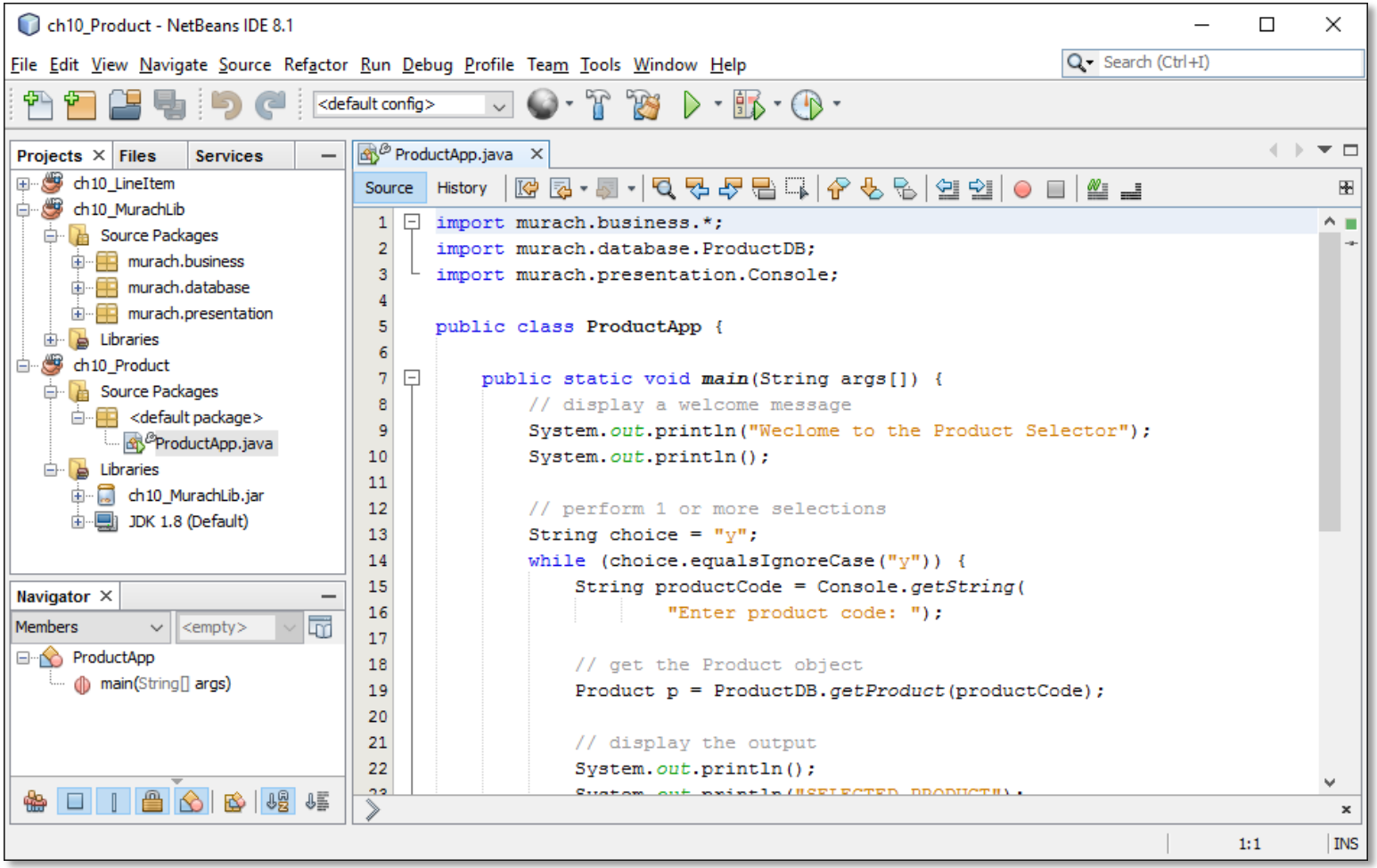
The ProductDB class

```
package murach.database;

import murach.business.Product;

public class ProductDB {...}
```

A NetBeans project with multiple packages



A NetBeans project that uses a library



How to create a library

1. Create a project that contains just the packages and classes that you want to include in the library.
2. Right-click on the project and select the Clean and Build command to compile the project and remove any files that are no longer needed. Then, NetBeans automatically creates a JAR file for the project and stores it in the dist subdirectory for the project.

How to use a library

1. Create or open the project that is going to use the library.
2. Right-click on the Libraries directory and select the “Add JAR/Folder” command. Then, use the resulting dialog box to select the JAR file for the library.
3. Code the import statements for the packages and classes in the library that you want to use. Then, you can use the classes stored in those packages.

A module…

- Has a name that uniquely identifies it.
- Explicitly declares the other modules that it depends on.
- Explicitly declares which of its public types are accessible to other modules.

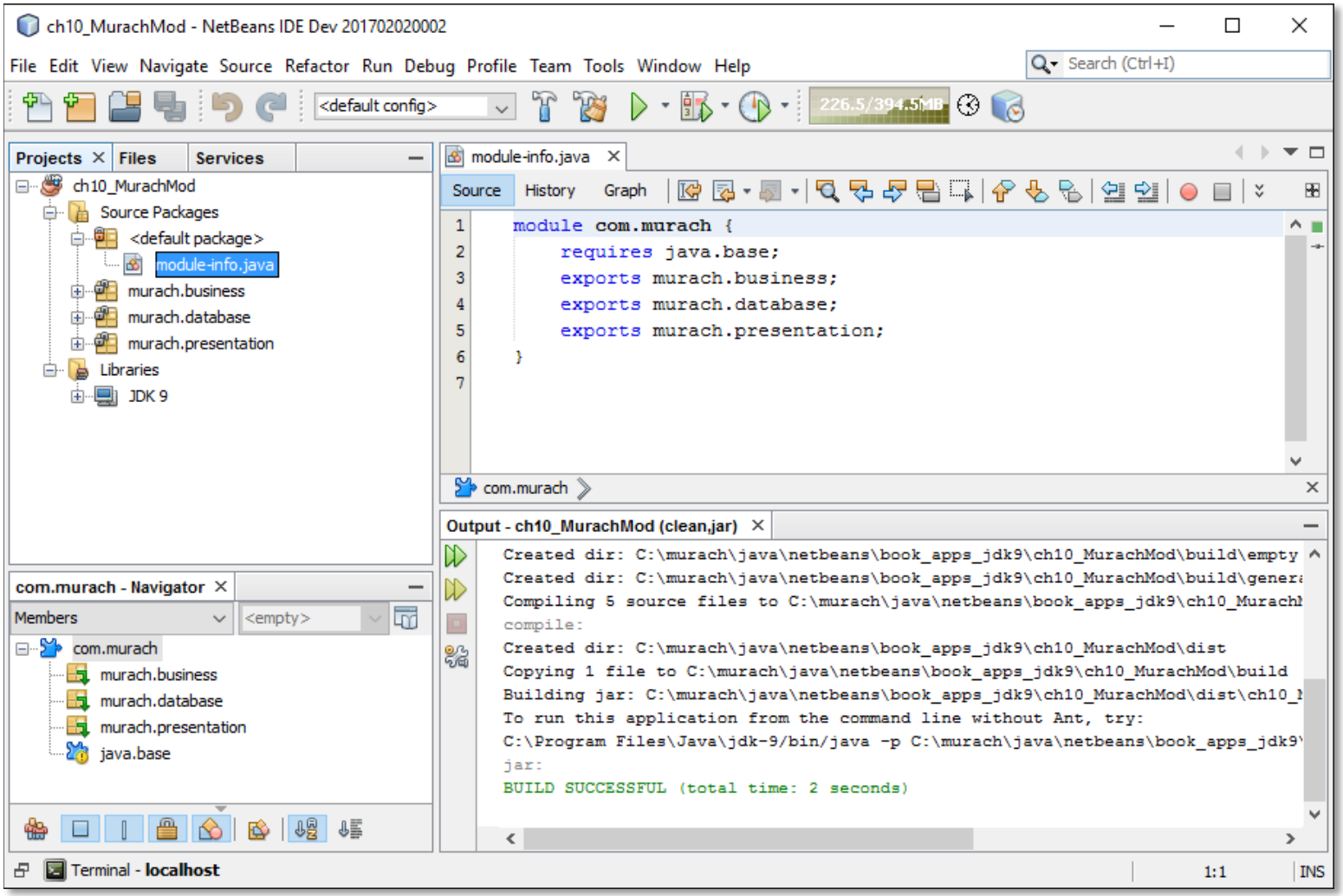
Advantages of the new module system

- Makes the Java platform more easily scalable down to small devices.
- Improves the security and maintainability of the Java platform.
- Improves Java application performance.
- Makes it easier to construct and maintain libraries and large applications.

Code that defines the java.base module

```
module java.base {
    exports java.io;
    exports java.lang;
    exports java.lang.annotation;
    exports java.lang.invoke;
    exports java.lang.module;
    exports java.lang.ref;
    exports java.lang.reflect;
    exports java.math;
    exports java.net;
    ...
}
```

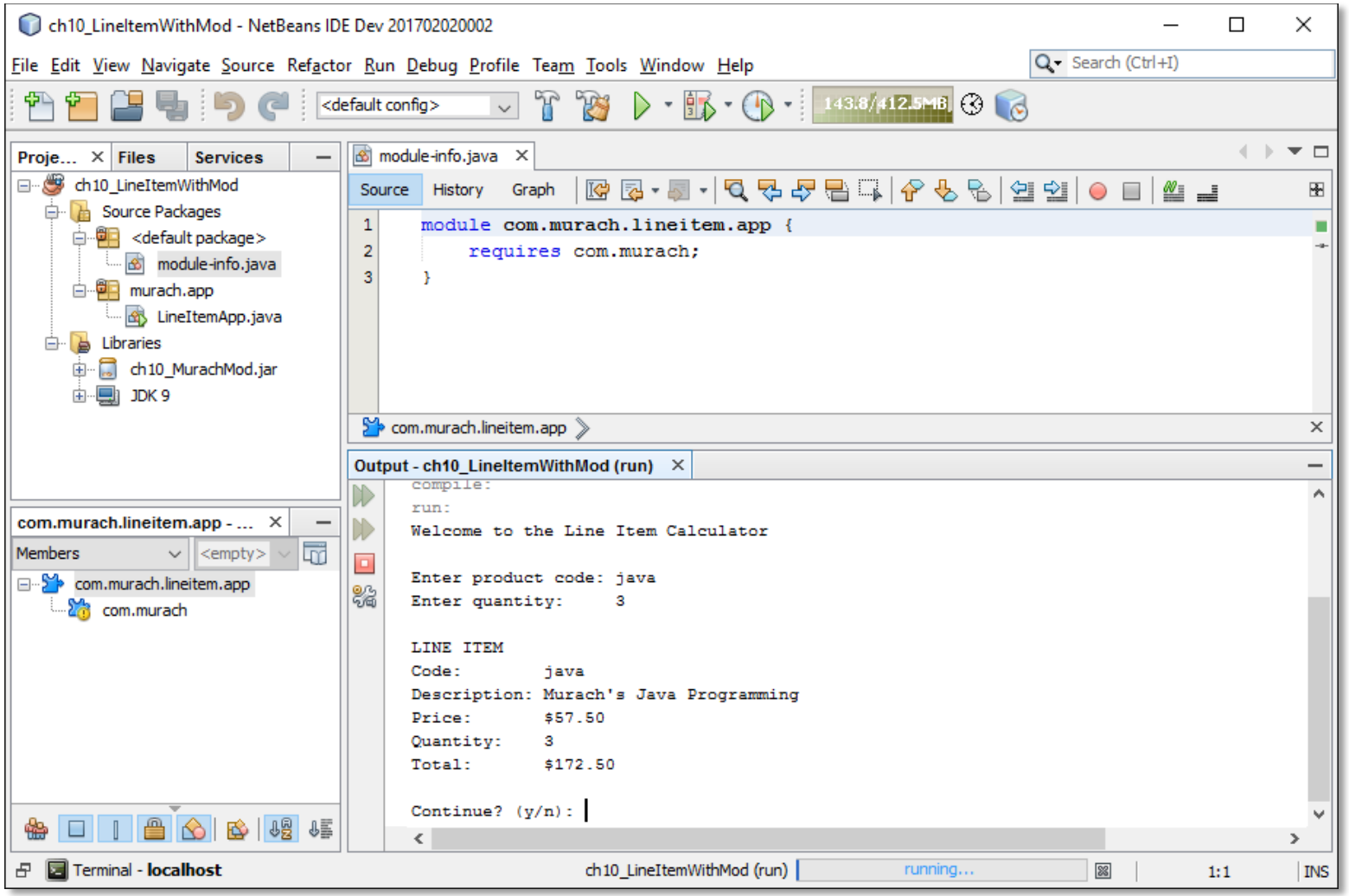
A NetBeans project that defines a module



A module-info file for the com.murach module

```
module com.murach {
    requires java.base;
    exports murach.business;
    exports murach.database;
    exports murach.presentation;
}
```

A NetBeans project that uses a module



A module-info file that requires the com.murach module

```
module com.murach.lineitem.app {
    requires com.murach;
}
```

A module-info file that requires the java.sql and sqlite.jdbc modules

```
module com.murach.product.app {
    requires java.sql;
    requires sqlite.jdbc;
}
```

The Product class with javadoc comments

```
package murach.business;
import java.text.NumberFormat;

/**
 * The Product class represents a product and is used by
 * the LineItem and ProductDB classes.
 */
public class Product  {

    private String code;
    private String description;
    private double price;

    /**
     * Creates a new Product with default values.
     */
    public Product() {
        code = "";
        description = "";
        price = 0;
    }
```

The Product class with javadoc comments (cont.)

```
/**
 * Sets the product code to the specified String.
 */
public void setCode(String code) {
    this.code = code;
}

/**
 * Returns a String that represents the product code.
 */
public String getCode() {
    return code;
}

...
```

Common HTML tag used to format javadoc comments

```
<code></code>
```

Common javadoc tags

```
@author
@version
@param
@return
```

The Product class with comments that use HTML and javadoc tags

```
package murach.business;

import java.text.NumberFormat;

/**
 * The <code>Product</code> class represents a product
 * and is used by the <code>LineItem</code> class.
 * @author Joel Murach
 * @version 1.0.0
 */
public class Product {
    private String code;
    private String description;
    private double price;
```

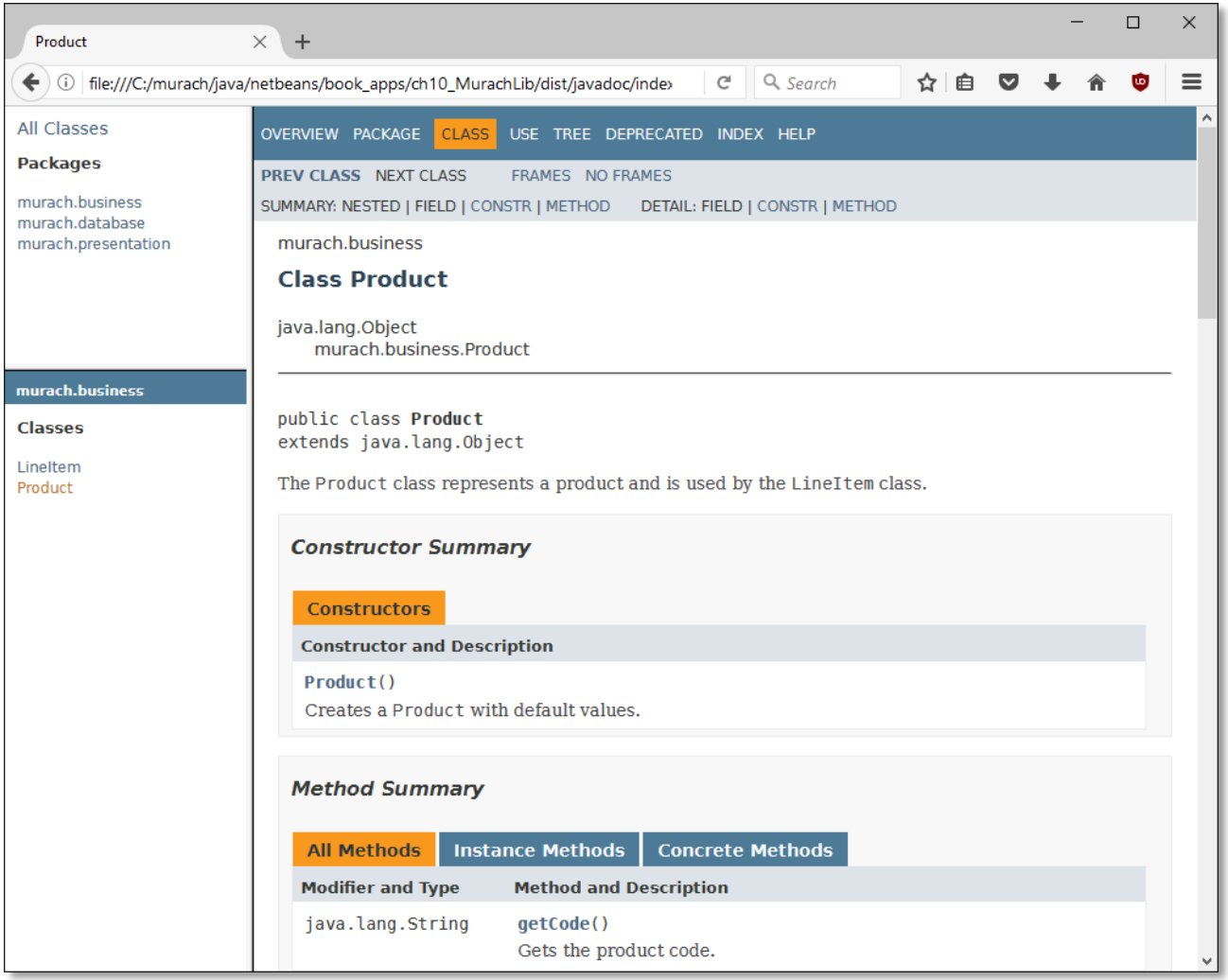
The Product class with comments that use HTML and javadoc tags (cont.)

```
/**
 * Creates a <code>Product</code> with default
 * values.
 */
public Product() {
    code = "";
    description = "";
    price = 0;
}

/**
 * Sets the product code.
 * @param code a <code>String</code> for the product
 * code
 */
public void setCode(String code) {
    this.code = code;
}
```

The Product class with comments that use HTML and javadoc tags (cont.)

```
/**
 * Gets the product code.
 * @return a <code>String</code> for the product code
 */
public String getCode() {
    return code;
}
...
```

The API documentation for the Product class



The syntax for declaring an enumeration

```
public enum EnumerationName {
    CONSTANT_NAME1[,
    CONSTANT_NAME2]...
}
```

An enumeration that defines three shipping types

```
public enum ShippingType {
    UPS_NEXT_DAY,
    UPS_SECOND_DAY,
    UPS_GROUND
}
```

A statement that uses the enumeration and one of its constants

```
ShippingType secondDay = ShippingType.UPS_SECOND_DAY;
```

A method that uses the enumeration as a parameter type

```
public static double getShippingAmount(ShippingType st) {
    double shippingAmount = 2.99;
    if (st == ShippingType.UPS_NEXT_DAY)
        shippingAmount = 10.99;
    else if (st == ShippingType.UPS_SECOND_DAY)
        shippingAmount = 5.99;
    return shippingAmount;
}
```

A statement that calls the method

```
double shippingAmount =
getShippingAmount(ShippingType.UPS_SECOND_DAY);
// double shippingAmount2 = getShippingAmount(1); //
Wrong type, not allowed
```

Two methods of an enumeration constant

```
name()
ordinal()
```

An enumeration that overrides the toString() method

```
public enum ShippingType {
    UPS_NEXT_DAY,
    UPS_SECOND_DAY,
    UPS_GROUND;

    @Override
    public String toString() {
        String s = "";
        if (this.ordinal() == 0)
            s = "UPS Next Day (1 business day)";
        else if (this.ordinal() == 1)
            s = "UPS Second Day (2 business days)";
        else if (this.ordinal() == 2)
            s = "UPS Ground (5 to 7 business days)";
        return s;
    }
}
```

Code that uses the toString() method

```
ShippingType ground = ShippingType.UPS_GROUND;
System.out.println(
    "toString: " + ground.toString() + "\n");
```

Resulting output

```
toString: UPS Ground (5 to 7 business days)
```

How to code a static import statement

```
import static murach.business.ShippingType.*;
```

The code above when a static import is used

```
ShippingType ground = UPS_GROUND;
System.out.println(
    "toString: " + ground.toString() + "\n");
```