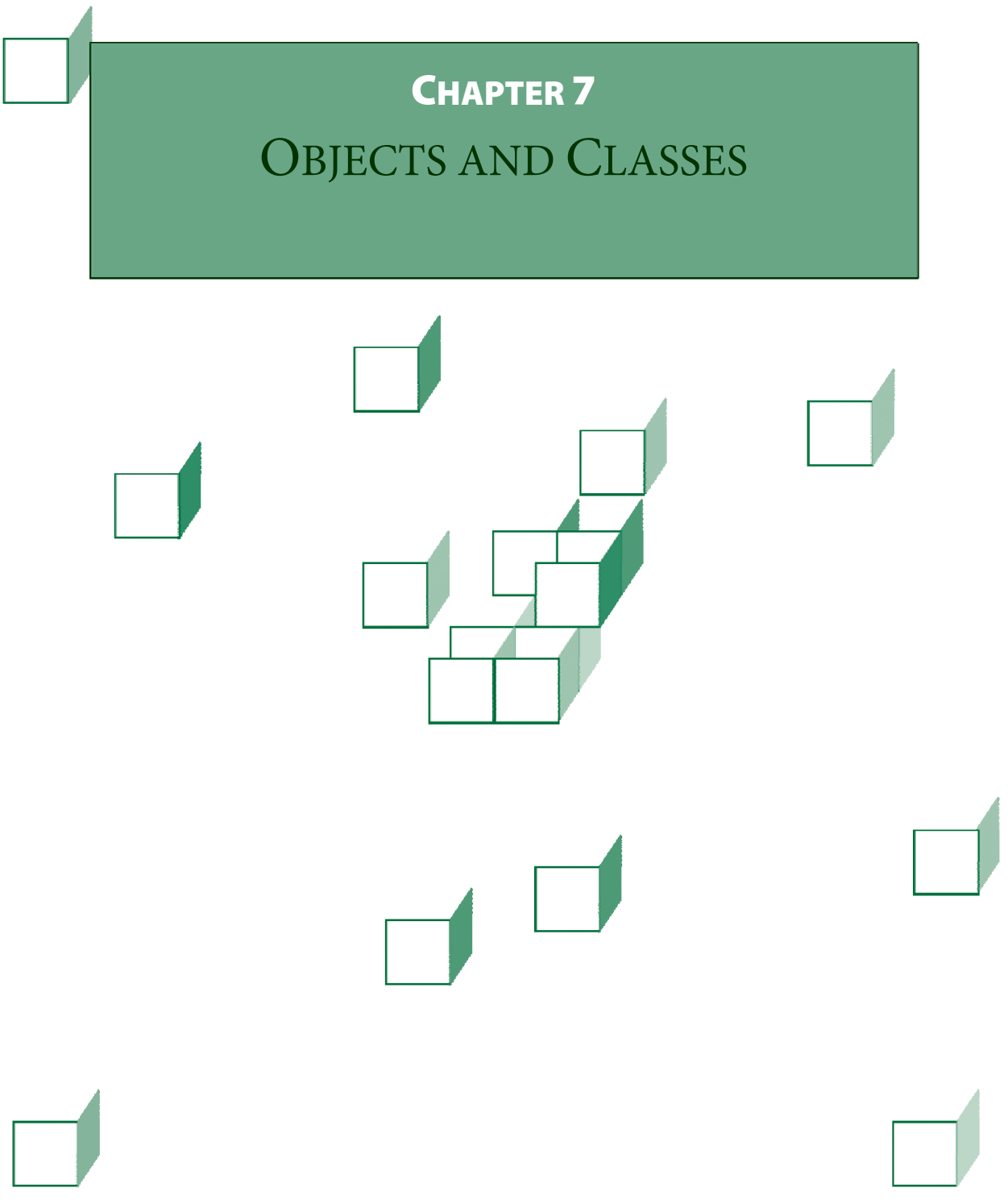
Chapter 7

Objects and Classes



OBJECTIVES

After completing "Objects and Classes," you will be able to:

- Explain the use of classes in Java for representing structured data.
- Distinguish between objects and classes.
- Describe how Java classes support encapsulation through member data and methods.
- Use the public and private access specifiers in Java programs.
- Explain the use of the new operator to instantiate objects from classes.
- Describe the use of references in Java and explain the role of garbage collection.
- Use constructors to initialize your objects.

Structured Data

- **Java defines primitive data types that are built into the language.**
  - Data types such as **int, double,** and **boolean** can be used to represent simple data.
- **Java provides the class mechanism to represent more complex forms of data.**
- **Through a class you can build up structured data out of simpler elements.**

```
public class Product
{
    String description;
    double price;
}
```

  - **Product** is now a new data type.
  - A product has a description (e.g. "Airplane toy") and a price (e.g. 2.55).
- **The Java libraries predefine many classes that you can use in your programs.**
  - **String**, although not a primitive data type, is predefined for you in the standard Java library.

Classes and Objects

- **A class represents a "kind of" or type of data.**
  - It is analogous to the built-in types like **int** and **double.**
- **A class can be thought of as a template from which individual instances can be created.**
  - When you create a **class**, you are only creating a **definition** .
- **An instance of a class is called an object.**
  - Just as you can have several individual integers that are instances of **int**, you can have several products that are instances of **Product.**

References

- **There is a fundamental distinction between the primitive data types and the extended data types that can be created using classes.**
- **When you declare a variable of a primitive data you are allocating memory and creating the instance.**

```
int x;     // 4 bytes of memory have been allocated
```

- **When you declare a variable of a class type (an "object variable"), you are only obtaining memory for a reference to an object of the class type.**
  - No memory is allocated for the object itself, which may be quite large.

```
Product p;
// p is a reference to a Product object
// The object itself does not yet exist
```

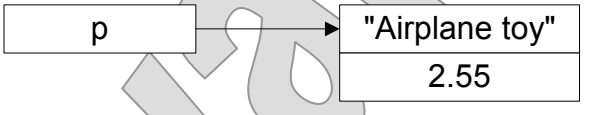
Instantiating and Using an Object

- You instantiate an object with the **new** operator.

```
p = new Product ();  // a Product object now exists
                     // and p is a reference to it
```

- Once an object exists you work with it, including accessing its data members (or "fields") and methods.
  - Our simple **Product** example at this point has no methods, only two data members.
  - You access data members and methods using a dot (normally you will not access data members directly – they are usually "private").

```
p.description = "Airplane toy";
p.price = 2.55;  // values have now been assigned
```



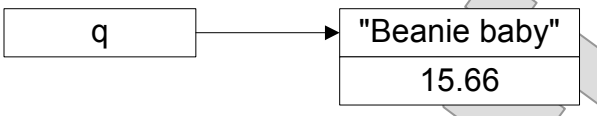
- Before the object exists, its object reference is **null**.
  - You can test for an object reference being **null**.

```
if (p != null)
{
    // OK to assign values
    p.description = "Airplane toy";
    p.price = 2.55;
}
```

Assigning Object Variables

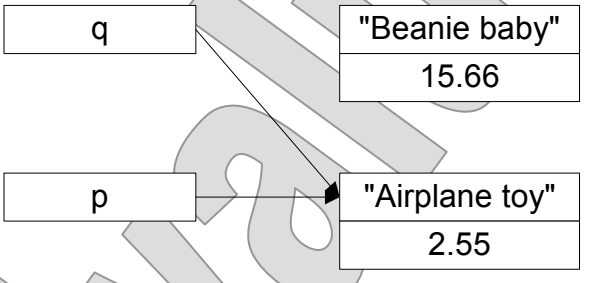
- Consider a second object variable referencing a second object.

```
Product q = new Product();
q.description = “Beanie baby”;
q.price = 15.66
```



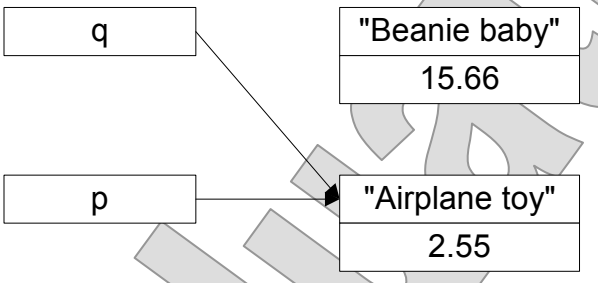
- When you assign an object variable you are only assigning the reference, there is no copying of data.

```
q = p;  // q now refers to same object p does
```



Garbage Collection

- **Through the assignment of a reference, an object may become orphaned.**
  - Such an orphan object (or "garbage") takes up memory in the computer which can now never be referenced.
  - "Beanie baby" is now garbage.



- **The Java Virtual Machine automatically reclaims the memory of unreferenced objects.**
  - This process is known as **garbage collection**.
  - Garbage collection takes up some execution time, but it is a great convenience for programmers, helping to avoid a common program error known as a "memory leak".

Programming a Class

EXAMPLE

- **To program a class in Java do the following:**
  - Enter the class definition in a file.
  - The file should have the same name as the class with the extension **.java.**
  - Make the class **public** in order to be able to use it in other classes.
  - Declare **fields**, which are like variables but they exist for the life of each object of the class, rather than just for the duration of a method invocation.
- **Example program shows definition and use of a simple Product class.**
  - See **Product_Step0**.

```
public class Product
{
    String description;
    double price;
}
```

Default Values

- Notice that you do not need to initialize fields, though you can if you so choose.
- If you don't, the compiler will assign a default value, which is essentially the "zero representation" appropriate to the data type of the field.
- The following table shows the default value for the primitive data types:

Type	Default
boolean	false
char	\u0000
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0

- Object references default to **null**.

Using a Class

EXAMPLE

- **To use a class:**
  - Declare a reference to the class and instantiate an object instance by **new** (can be done on the same line).
  - Access members (and methods) by using the dot notation.
  - See **Product_Step0**.

```
public class TestProduct
{
    public static void main(String[] args)
    {
        Product p1 = new Product();
        Product p2 = new Product();
        Product q;
        p1.description = "Airplane toy";
        p1.price = 2.50;
        p2.description = "Beanie baby";
        p2.price = 15.66;
        q = p1;
        System.out.println(q.description + " " +
                           q.price);
        q = p2;
        System.out.println(q.description + " " +
                           q.price);
    }
}
```

Methods

- **Typically a class will specify behavior as well as data.**
  - A class **encapsulates** data and behavior in a single entity.
- **A method consists of:**
  - An access specifier, typically **public** or **private.**
  - A return type (can be **void** if the method does not return data).
  - A method name, which can be any legal Java identifier.
  - A parameter list, enclosed by parentheses, which specifies data that is passed to the method (can be empty, if no data is passed).
  - A method body, enclosed by curly braces, which contains the Java code that the method will execute.

Methods

- **Example:**
  - The return type is **void** (no data is passed back).
  - The method name is **adjustPrice**.
  - The parameter list consists of a single parameter of type **double**.
  - The body contains one line of code that increments the member variable **price** by the value that is passed in.

```
public void adjustPrice(double p)
{
    price += p;
}
```

- **Example:**
  - The return type is **double** (data of type **double** will be passed back).
  - The method name is **getPrice**.
  - There are no parameters.
  - The body contains one line of code that returns the current value of member variable **price**.

```
public double getPrice()
{
    return price;
}
```

Public and Private

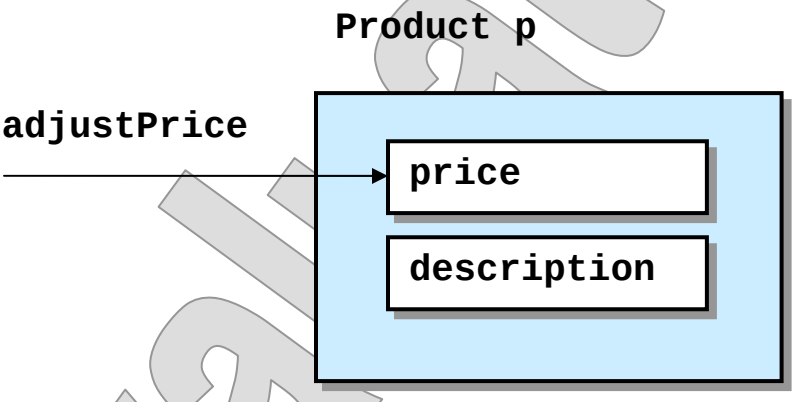
- **Data members and methods of a Java class can be specified as public or private.**
- **Normally you declare data members as private.**
  - A **private** member can only be accessed from within the class, not from outside.
- **Methods may be declared as either public or private.**
  - Public methods are called from outside the class and are used to perform calculations and to manipulate the private data.
  - You may also have private methods, which can be thought of as "helper functions" for use within the class (rather than duplicating code several places, you may create a private method, which will be called wherever it is needed).

Abstraction

- **Our Product class captures the essential features of a product, suppressing unnecessary details.**
  - There are many possible features of a *product*, but for our purposes the only essential things are its **description** and its **price.**
- **All instances of the Product class share these common features.**
- **This helps us deal with complexity.**

Encapsulation

- Encapsulation is an important feature of Object Orientation.
- The implementation of a class should be hidden from the rest of the system, or **encapsulated.**
- Objects have a public and a private side.
- Public side is what the rest of the system knows, while private side implements the public side.



- Data itself should be private and only accessible through methods with a public interface.
- There are two kinds of protection:
  - Internal data is protected from corruption.
  - Users of the object are protected from changes in the representation.

Constructors

- **Another important issue for classes is initialization.**
  - When an object is created, what initial values are assigned to the instance data?
- **Through a constructor you can initialize an object in any way you wish.**
  - Besides initializing instance data you can perform other appropriate initializations (e.g. open a file).
- **A constructor is like a special method which is automatically called when an object is created.**
  - A constructor has no return type.
  - A constructor has the same name as the class.
  - A constructor may take parameters, which are passed when invoking **new.**

```
public Product(String d, double p)
{
    description = d;
    price = p;
}
```

A Product Class

EXAMPLE

- Another version of the **Product** class illustrates the features we have been discussing.
  - See **Product_Step1**.

```
public class Product
{    // constructor
     public Product(String d, double p)
     {
         description = d;
         price = p;
     }
     public void adjustPrice(double p)
     {    // method increments the price by p
         price += p;
     }
     public double getPrice()
     {    // method retrieves the price
         return price;
     }
     public String getDescription()
     {    // method retrieves the description
         return description;
     }
     public String toString()
     {    // method converts to string
         return String.format("%s $%,1.2f",
                              description, price);
     }
     private String description;
     private double price;
}
```

A Product Class

EXAMPLE

```
public class TestProduct
{
    public static void main(String[] args)
    {
        Product p1, p2, p3;
        p1 = new Product("Airplane toy", 2.50);
                                // uses constructor
        p1.adjustPrice(.50);  // uses method
        p2 = new Product("Beanie baby", 15.66);
        p3 = new Product("Cat carrier", 5.50);
                                // uses constructor
        System.out.println(p1.toString());
        System.out.println(p2.toString());
        System.out.println(p3.toString());
                              // conversion to string
        System.out.println(p3);
                             // implicit conversion
    }
}
```

- Try it out:

```
compile
run TestProduct
Airplane toy $3.00
Beanie baby $15.66
Cat carrier $5.50
Cat carrier $5.50
```

The toString Method

- **It is often useful to define a toString method in your class.**
  - This method should return a **String** value that represents the current object instance.
- **toString is known to Java.**
  - If an object reference is used when a **String** reference is expected, your object will be converted to a **String** using the **toString** method.
  - For example, the **println** method expects a **String** parameter, and a **Product** variable will be converted to a **String** variable using the **toString** method.

Formatted Output

- **Java provides a simple means of formatting output to a stream.**
  - This is a feature long known in the C language, and to some it has been conspicuous by its absence in Java.
- **The System.out object offers a method format.**
  - The **format** method takes a formatting string as its first parameter, and then any number of objects after that.

```
System.out.format ("%8d %s%n", 77, "Hello");
```

  - The call shown above produces the following – note the leading spaces to observe an eight-character "width" for the numeric argument:

```
    77 Hello
```

- **The String class offers a format method as well.**
  - It takes the same parameters, but instead of producing the results to the console it returns a formatted string object.

```
String representation =
  String.format ("%8d %s%n", 77, "Hello");
```

  - This provides a means of capturing formatted presentations and passing them around between objects.

```
public String toString ()
{
  return String.format ("(%d, %d)", x, y);
}
```

Formatting Strings

- **The formatting string passed to `format` consists of literal text and `fields`.**
- **Each field in the string must correspond to an argument passed to the method – with a few exceptional field types that produce special characters and don't need a value to be provided.**
- **Common field types are**
  - **`%d`**, which formats an integer
  - **`%s`**, which formats a string
  - **`%f`**, which formats a floating-point number
  - **`%n`**, which produces an end-of-line sequence appropriate for the runtime operating system
- **Most fields can be modified in a few simple ways:**
  - A number preceding the one-letter field code defines the field width, which is a minimum number of characters to be produced by the field.
  - A hyphen preceding the number indicates the field will be **left-justified**, where **right-justified** is the default, if a width has been specified – even for strings.

Formatting Examples

- The following pairs of lines show the Java code to format certain values and the resulting output:

```
System.out.format ("%d %s %s%n",
  77, "Hi", "there");
77 Hi there

System.out.format ("%4d %8s %8s%n",
  77, "Hi", "there");
  77       Hi    there

System.out.format ("%4d %-8s %-8s%n",
  77, "Hi", "there");
  77 Hi       there

System.out.format ("%-16s $%,8.2f",
  "Dog bone", 70);
Dog bone         $      70.00(and no line break!)

System.out.format ("The value is $%,-1.2f.", 70);
The value is $70.00.
```

- Formatted output is simple to use, but for many purposes it is still simpler and easier to simply **print** or **println**.
- Formatted output is especially helpful in establishing regular widths for fields, and in justifying the output text, so that columns of values are neatly aligned for the reader of the output.
- We will use a combination of the two practices in the rest of our code examples and exercises.

Using a Money Class

LAB 7

Suggested time: 60 minutes

In this lab you will create a **Money** class that can be used to hold and display currency values. The **Money** class will encapsulate formatting logic, hiding this detail from the rest of the program.

Detailed instructions are found at the end of the chapter.

SUMMARY

- Classes are used in Java for representing structured data.
- An object is an instance of a class.
- Java classes support encapsulation through private member data and public methods.
- The new operator is used to instantiate objects from classes.
- Java uses references to refer to objects.
- An object without a reference represents unused memory, which will be cleaned up by garbage collection.
- Constructors are used to initialize objects.