# An Introduction to Computer Science with Java

Community College of Philadelphia Computer Science

# Introduction to Computer Science with Java

## Chapter 9 – Classes and Objects in Java

Community College *of* Philadelphia *Computer Science*

*This chapter describes how we can create our own classes in Java. It includes creating classes of objects, passing objects as parameters, use of the* Java *keyword* this, and implementing interfaces.

### Chapter Learning Outcomes

Upon completion of this chapter students should be able to:

- Create a class of objects in java as a collection of properties and methods that manipulate the properties.
- Create constructor, accessor, mutator, utility, and static methods within Java classes.
- Use objects of the students' own creation in java methods in other classes.
- Describe the concept of an inner Class and create java code that uses inner classes.
- describe the use of the Java keyword *this* and properly use it in Java methods.
- create and manipulate an array of objects in java.
- describe the concept of a Java interface and a in Java abstract class, the purpose of the *comparable* interface and compareTo() method, and demonstrate their use in a Java class.

## 9.1 Defining Classes of Objects in Java

The code that defines a class in Java is called a **class declaration**, but it is also known as a **class definition.** Class declarations in Java start with the keyword *class*, then the name of the class, followed by a set of braces enclosing the declarations for the properties and methods in the class. Remember that the Java naming convention is to start a class name with an uppercase letter.

```
class ClassName
{
// declare properties
// declare methods – with constructors first
)
```

If a class is to be a public class, its name should be preceded by the visibility indicator *public.* Only one class in each **.Java* file may be public. If the file has a *main()* method, it must be in the public class.

If no visibility modifier is used, then the class is *package-private*, and is only accessible from within the package.

Usually the properties of an object are declared first in a class declaration, followed by the methods. Putting the set of property declarations together at the top of a class declaration functions like a data dictionary, making it easier for someone to reading a class declaration to know what the properties of the object are.

The constructors are always listed before other methods. As with properties, this makes it easier for someone reading the method to find them.

**Example – Java Class Declaration**

The following code shows a class declaration in Java:

```
public class Book
{
   // declare properties
   private String isbn;
   private String title;
   private String subject;
   private double price;

   // constructor methods ******************************
   public Book()
   {
   } // end Book()

   public Book(String number, String name)
   {
       isbn = number;
       title = name;
   } // end Book(String number, String name)

   // accessor methods ********************************
   public String getTitle()
   {
       return title;
   } // end getTitle()

   public String getISBN()
   {
       return isbn;
   } // end getISBN()

   public String getSubject()
   {
       return subject;
   } // end getsubject()
   public double setPrice()
   {
       return price;
   } // end setPrice()

   // mutator methods ******************************
   public void setTitle(String name)
   {
       title  = name;
   } // end setTtitle()

   public void setISBN(String number)
   {
       isbn = number;
   } // end setISBN()

   public void setSubject(String sub)
   {
       subject = sub;
   } // end setsubject()

   public void setPrice(double value)
   {
       price = value;
   } // end setPrice()

   // method to return information about the book as a String
   public String toString()
   {
       return ("Title: " + title + "ISBN: " + isbn);
   } // end to String()

} // end class Book
```

The Book class is properly encapsulated by making all of its properties private. Only the public methods may be used from outside of the class.

The *Book* class has two constructors, a default constructor and a constructor with two parameters, *number* and *name*. A **default constructor** in Java is a constructor with no parameters. Such a constructor is also called a null constructor or a nullary constructor. (Nullary is a term from mathematics that indicates a function or operation has no operands.) A default constructor sets up a reference variable and memory storage space for an instance of an object.

Each Java class must have a constructor. If a class declaration in Java does not contain any constructors, then the compiler will add a null constructor with an empty method body.

In this example, the standard accessor methods follow the constructor, and then the mutator methods follow the accessors. There are no static methods or properties. Static method and properties are normally listed first in the declarations of properties and methods.

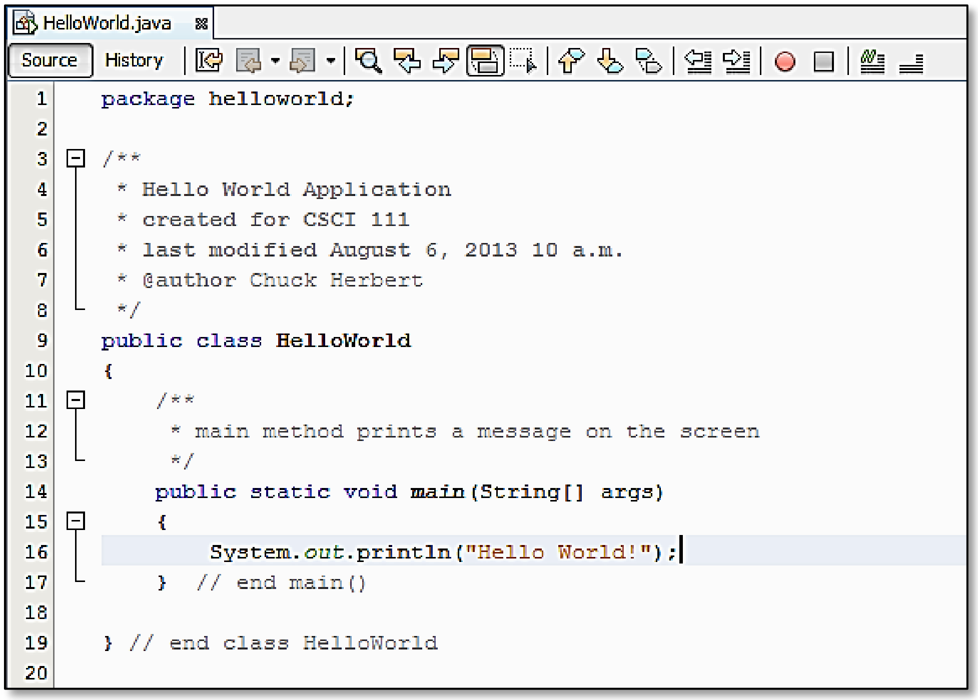
The *toString* method is also considered an accessor method, since it returns information from the properties of an object. It is a good programming practice to include a *toString* method for a class of objects that returns information about the object as a String.

## 9.2 Organizing Class Declarations in Java

Java provides several different ways to organize and access declarations for new classes of objects. In this section we will examine two ways; multiple classes in a single **.java* file, and inner classes.

### Multiple Classes in a Single **.java* file

Java source code should be contained in a text file ending with the extension *".java".* The source code in the file may contain one or more method declarations. If the code is to include a *main()* method that will be executed when the java code runs, then that method must be in a public class, and that class must have a name that matches the main part of the file name. This has happened in all of the programs we have run so far, going all the way back to the our first *HelloWorld* program:



```
package helloworld;

/**
 * Hello World Application
 * created for CSCI 111
 * last modified August 6, 2013 10 a.m.
 * @author Chuck Herbert
 */
public class HelloWorld
{
    /**
     * main method prints a message on the screen
     */
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }  // end main()

} // end class HelloWorld

```

The class is named *HelloWorld*, matching the file name *HelloWorld.java*, which can be seen at the top-left corner of the NetBeans interface. The class contains a static *main()* method; in fact, the only method in the class is a *main()* method, so this class file does not really define a new object so much as it serves as a vehicle for running a simple method that displays a message on the screen. It is a class – the *HelloWorld* class – but the class has no data members and no declared methods other than the static *main()* method.

In contrast, the *Book* class in the example in section 1 of this chapter has a well-defined class – whose declaration includes proper constructors, accessor and mutators – but it does not include any main method, so the class cannot be run as a program by itself.

In most Java programming projects beyond trivial examples like those found at the beginning of a Java textbook or a Java language tutorial, we need both – properly defined classes of objects and a public class with a main method that allows us to execute our code.

The structure of such a *.java* file would look like this:

**SampleClass.java** is the name of the file.

```
public class SampleClass
{

   public static void main()
   {
   // this is the executable main() method
   }

 } // end class SampleClass
/************************************************************

private class firstObject
{
   //the first object to be used -- a car, house, person, ... whatever

}  // end class firstObject
/************************************************************

private class SecondObject
{
   //the second object to be used -- a car, house, person, ... whatever

}  // end class SecondObject
/************************************************************
```

The *main()* method in the example above could then create and use instances of the objects defined in the other classes in the file.

The following is a more specific example, based on *"Hello world."* The file has a class named *HelloWorldDemo*, and a class name *Message*. *HelloWorldDemo* has only an executable *main()* method, which creates and uses an object of class *Message*.

The file includes a package since it will be used as a NetBeans project, which requires packages, along with import statements for some of the things used in the Message class.

**HelloWorldDemo.java** demonstrates two classes in one *.java* file

```
/*
 * HelloWorldDemo.java
 * CSCI 111 Fall 2013
 * last edited November 2, 2013 by C. Herbert
 * This program shows how to use one class from another in the same file.
 *
 * It is only intended to demonstrate the use of classes.
 */

package helloworlddemo;
```

```
import javax.swing.*;
import java.awt.*;

public class HelloWorldDemo
{
    // the main method creates a message object and sends it to the console
    public static void main(String[] args)
   {
       // create a new instance of the Message class named myMessage
       Message myMessage = new Message();

       // set the message text for instance myMessage
       myMessage.setText("Hello, World!");

       // send myMessage to the console
       myMessage.toConsole();

   } // end main()

 } // end class SampleClass
//**************************************************************

class Message
{
   String text;         // the text to be sent as a message
   Font messageFont;    // the Font for the message in GUI components
   Color messageColor; // the Color for the message in GUI components
   // note: Font and Color classes are defined in an imported package

    // constructors methods
    public Message()
   {
   } // end Message()

   public Message(String s, Font f, Color c)
   {
      text = s;
      messageFont = f;
      messageColor = c;
   } // end Message(String s, Font f, Color c)

    // accessor methods ******************************
    public String getText()
    {
      return text;
    } // end getTtext()

    // note: color and font can be set, but not returned

    // mutator methods ******************************

    public void setText(String s)
    {
       text = s;
    } // end setText()


    public void setFont(Font f)
    {
       messageFont = f;
    } // end setFont()

    public void setColor(Color c)
    {
       messageColor = c;
    } // end setTColor()

    // method to display message on the console
    public void toConsole()
    {
       System.out.println(text);
    } // end toConsole()

    // for simplicity, GUI methods using messageColor and messageFont are not shown

} // end class Message
```

## Inner Classes in Java

An **inner class** is class that is declared within another class. Instead of putting the class declarations one after another in a file, a class can be declared inside another class. Inner classes are also known as **nested classes.** Java allows the nesting of classes.

Inner classes are normally just used for a class which will only be invoked from its containing class (the outer class). For example, we might have a class named *bankAccount* to keep track of bank accounts, which works with a class called *bankTransaction*. If *bankTransaction* will only be used within the *bankAccount* class, then it can be declared as an inner class inside the class declaration for *bankAccount.*

Declaring inner classes in this way accomplishes two things:

- it makes *bankTransaction* part of the *bankAccount* class, so wherever the code for *bankAccount* is, it will always have immediate access to objects and methods of type *bankTransaction.*
- it encapsulates the inner class more deeply. Users in the outside world could work with *bankAccount* and not even know *bankTransaction* exists.

Use of inner classes can be good or bad, depending on how the inner class is intended to be used. If objects from a class will only be used by the outer class, then making it an inner class is probably a good idea. If objects from the class might be used by several other classes, then it is probably not a good idea to declare it as an inner class.

The example below shows HelloWordDemo2 , a version of the *HelloWordDemo* from above, written with the *Message* class inside the *HelloWordDemo2* class. This is a good idea only if *Message* will be used by no other classes.

Be aware that static methods, such as *main(),* cannot invoke non-static items, so if an inner class will be invoked from the *main()* method, it must be a static class. Otherwise, it does not need to be static. In the example below, *Message* is a static class within *HelloWordDemo2* so it can be used in *main()*.

**HelloWorldDemo2.java** demonstreates an inner class

```
/*
 * HelloWorldDemo.java
 * CSCI 111 Fall 2013
 * last edited November 2, 2013 by C. Herbert
 * * This program shows how to use one class inside another class
 *
 * It is only intended to demonstrate the use of inner classes
 */

package helloworlddemo2;
import javax.swing.*;
import java.awt.*;

public class HelloWorldDemo2
{
    // the main method creates a message object and sends it to the console
    public static void main(String[] args)
   {
       // create a new instance of the Message class named myMessage
       Message myMessage = new Message();

       // set the message text for instance myMessage
       myMessage.setText("Hello, World!");

       // send myMessage to the console
       myMessage.toConsole();

   } // end main()

   // declare message as an inner class of HelloWorldDemo2

   static class Message
   {
       String text;         // the text to be sent as a message
       Font messageFont;    // the Font for the message in GUI components
       Color messageColor; // the Color for the message in GUI components
       // note: Font and Color classes are defined in an imported package

       // constructors methods
       public Message()
       {
       } // end Message()

       public Message(String s, Font f, Color c)
       {
           text = s;
           messageFont = f;
           messageColor = c;
       } // end Message(String s, Font f, Color c)

        // accessor methods ******************************
        public String getText()
        {
           return text;
        } // end getTtext()
        // note: color and font can be set, but not returned

        // mutator methods ******************************

        public void setText(String s)
        {
           text = s;
        } // end setText()

        public void setFont(Font f)
        {
           messageFont = f;
        } // end setFont()

           public void setColor(Color c)
        {
           messageColor = c;
        } // end setTColor()

        // method to display message on the console
        public void toConsole()
        {
           System.out.println(text);
        } // end toConsole()

        // for simplicity, GUI methods using messageColor and messageFont are not shown

    } // end inner class Message

} // end class SampleClass
//*************************************************************
```

## 9.3 Use of the Java Keyword *this*

The Java keyword **this** may be used in a Java method to specify the instance of the object that has invoked the method. It is most often used in a method where a method variable or a method parameter has the same name as a property of the object invoking the method.

Consider the following example from the *Book* class that appeared near the beginning of this chapter.

The book class has a constructor with two parameters, number and name as follows:

```
   public Book(String number, String name)
   {
       isbn = number;
       title = name;
   } // end Book(String number, String name)
```

However, what if a programmer only sees the method header? The parameter names *number* and *name* don't make it clear what these parameters really are. *Number* might be an ISBN, a Dewey decimal number or, a Library of Congress number. Similarly, *name* might be the name of the book, but it could also be the name of the author, or the name of publisher, or maybe even the name of the subject.

So, we might consider declaring the constructor so that the parameters have same names as the corresponding properties in the class, like this:

```
   public Book(String isbn, String title)
   {
       isbn = isbn;     // note that these two lines are incorrect
       title = title;  // no errors, but they don't work as expected
   } // end Book(String isbn, String title)
```

That makes the parameters more meaningful to a programmer seeing the method header, but now what happens inside the method?

The statement `isbn = isbn;` can cause confusion for a compiler and for a person reading the method. In fact it does not work as we might want it to. The method will assume that *isbn* is the name of the variable or parameter, and not the name of an object's property. It makes sense to the compiler and no error is generated, but it just doesn't work as we wanted it to work. It sets the variable *isbn* equal to itself. That is sensible to a compiler, even though it is a useless instruction that accomplishes nothing. We want to set the property isbn equal to the variable isbn.

The keyword *this* is designed to solve the problem. *this.isbn* will refer to the property *isbn* for the instance of the object that was used to invoke the method.

The method should look like this:

```
public Book(String isbn, String title)
   {
       this.isbn = isbn;      // note: this works as expected
       this.title = title;
   } // end Book(String isbn, String title)
```

The following short programing exercise allows you to work with this example.

## Programming Exercise - Use of the Java Keyword This

The files for chapter 9 include a zipped NetBeans project folder named *ThisBook*.

**STEP 1.**

Download and unzip the NetBeans project folder *ThisBook* included with the Chapter 9 files.

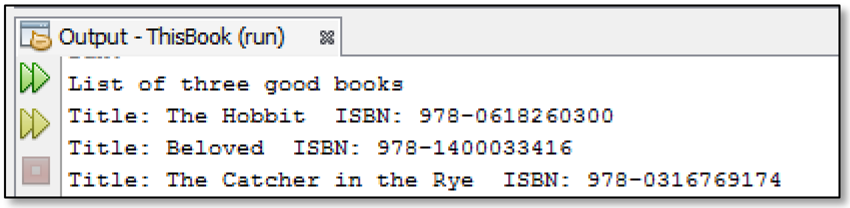
**STEP 2.**

Open the *ThisBook* project in NetBeans and examine the source code. It has a class named *ThisBook* with an executable *main()* method and a class named *Book.*

The main() method in *ThisBook* creates three instances of Book using a constructor that takes two parameters, *number* and *name.* It then uses the *toString()* method in the Book class to print information about each instance of Book. Read through the code and become familiar with how it works.

**STEP 3.**

Run the program. It should print the information for the three books, as follows:


Output - ThisBook (run)

```
List of three good books
Title: The Hobbit  ISBN: 978-0618260300
Title: Beloved  ISBN: 978-1400033416
Title: The Catcher in the Rye  ISBN: 978-0316769174
```

**STEP 4.**

Now, find and edit the code for the second constructor in the Book class:

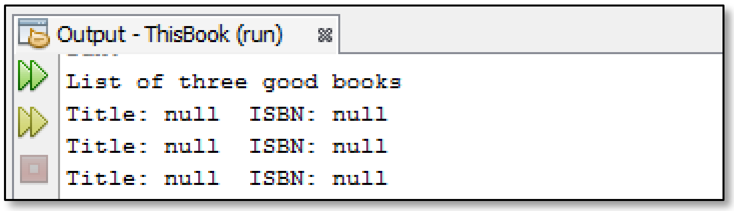
```
    public Book(String number, String name)
    {
        isbn = number;
        title = name;
    } // end Book(String number, String name)
```

Change the parameter ***number*** to ***isbn*** and the parameter ***title*** to ***name*** in the formal parameter list and where they are used in the method:

```
    public Book(String isbn, String title)
    {
        isbn = isbn;
        title = title;
    } // end Book(String number, String name)
```

**STEP 5.**

Now run the program again. Your output should look like this:


Output - ThisBook (run)

```
List of three good books
Title: null  ISBN: null
Title: null  ISBN: null
Title: null  ISBN: null
```

What happened? The compiler is setting the parameters for *isbn* and *title* to themselves, not setting the properties of each instance that calls the method. The properties are not initialized and remain null.

The keyword *this* will correct the problem.

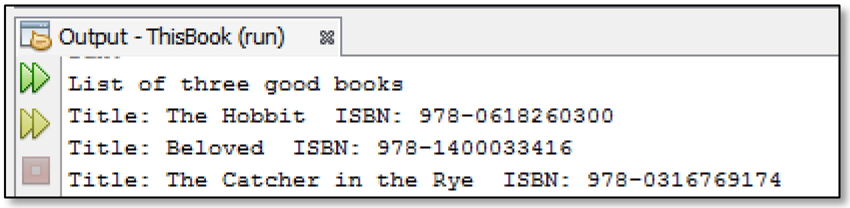
**STEP 6.**

Change the two lines in the constructor to include ***this***, as shown:

```
public Book(String isbn, String title)
    {
        this.isbn = isbn;       // note: this works as expected
        this.title = title;
    } // end Book(String isbn, String title)
```

**STEP 7.**

Now run the program again. Your output should again look like this:



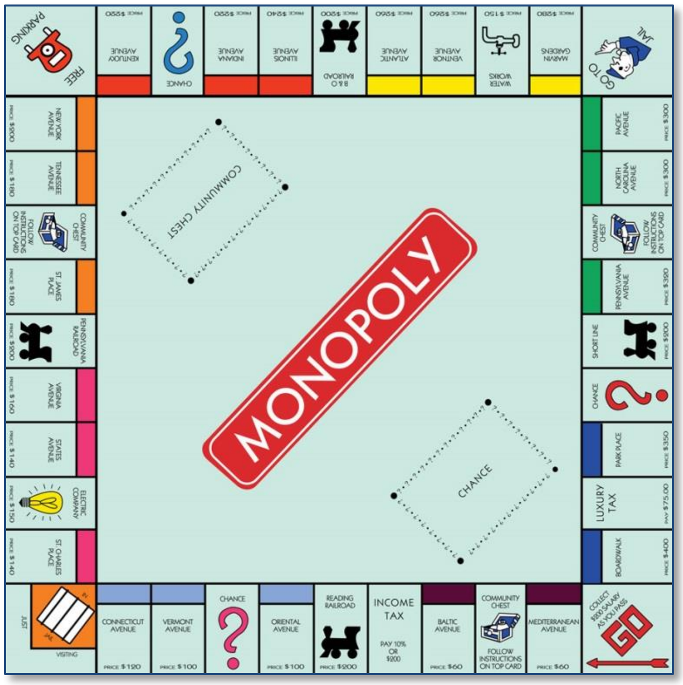
```
List of three good books
Title: The Hobbit  ISBN: 978-0618260300
Title: Beloved  ISBN: 978-1400033416
Title: The Catcher in the Rye  ISBN: 978-0316769174
```

The keyword *this* can be used whenever references to an object's properties or methods are ambiguous. It is useful in the case above, when parameters or variables have the same name as an object's properties, and in certain cases of inheritance and polymorphism.

## 9.4 An Array of Objects in Java

As we saw in Chapter 8, each element in an array of objects is a reference variable holding the address of an object. To store an array of objects in java we first create a class of objects, and then place instances of the object in an array. This is best illustrated with an example.

### Example – Array of Objects: Monopoly Board Squares



In this example we will create an array of squares for a *Monopoly* board, then place the squares in an in an array of objects. First we must define a class for the squares, and then we can place them in an array.

We will start with a description of the squares and a UML diagram of the class.

There are 40 squares on a *Monopoly* board, arranged as shown here.

There are different types of squares:

- **Property** squares can be bought and sold. A player who lands on a property square that player must pay rent. Property squares have rent and a price and belong to a color group.
- **Railroad** squares can be bought and sold, but the rent for a player who lands on a railroad square depends on the number of railroads in possession of the owner. In our example, the rent for each railroad will be set at $25.
- **Utility** squares can be bought and sold. The rent depends on the roll of the dice. In our version of the game will each have a rent of $10.
- **Card** squares cannot be bought and sold and have no rent. Players who land on a card square draw a card.

- **Plain** squares cannot be bought and sold and have no rent. Nothing happens when a player lands on a plain square.
- **Tax** squares require a player to pay a tax.
- There is one **toJail** square. A player who lands on a toJail Square goes to Jail.

Information about each square is summarized in the following table.

| Name | type | Rent | price | color |
|---|---|---|---|---|
| **Go** | plain | 0 | 0 | null |
| **Mediterranean Ave.** | property | 2 | 60 | Dark Purple |
| **Community Chest** | card | 0 | 0 | null |
| **Baltic Ave.** | property | 4 | 60 | Dark Purple |
| **Income Tax** | tax | 200 | 0 | null |
| **Reading Railroad** | railroad | 25 | 200 | null |
| **Oriental Ave** | property | 6 | 100 | Light Blue |
| **Chance** | card | 0 | 0 | null |
| **Vermont Ave.** | property | 6 | 100 | Light Blue |
| **Connecticut Ave.** | property | 8 | 120 | Light Blue |
| **Jail/Just Visiting** | plain | 0 | 0 | null |
| **St. Charles Place** | property | 10 | 140 | Light Purple |
| **Electric Company** | utility | 10 | 150 | null |
| **States Ave.** | property | 10 | 140 | Light Purple |
| **Virginia Ave.** | property | 12 | 160 | Light Purple |
| **Pennsylvania Railroad** | railroad | 25 | 200 | null |
| **St. James Place** | property | 14 | 180 | Orange |
| **Community Chest** | card | 0 | 0 | null |
| **Tennessee Ave.** | property | 14 | 180 | Orange |
| **New York Ave.** | property | 16 | 200 | Orange |
| **Free Parking** | plain | 0 | 0 | null |
| **Kentucky Ave.** | property | 18 | 220 | Red |
| **Chance** | card | 0 | 0 | null |
| **Indiana Ave.** | property | 18 | 220 | Red |
| **Illinois Ave.** | property | 20 | 240 | Red |
| **B & O Railroad** | railroad | 25 | 200 | null |
| **Atlantic Ave.** | property | 22 | 260 | Yellow |
| **Ventnor Ave.** | property | 22 | 260 | Yellow |
| **Water Works** | utility | 10 | 150 | null |
| **Marvin Gardens** | property | 24 | 280 | Yellow |
| **Go To Jail** | toJail | 0 | 0 | null |
| **Pacific Ave.** | property | 26 | 300 | Green |
| **No. Carolina Ave.** | property | 26 | 300 | Green |
| **Community Chest** | card | 0 | 0 | null |
| **Pennsylvania Ave.** | property | 28 | 320 | Green |
| **Short Line Railroad** | railroad | 25 | 200 | null |
| **Chance** | chance | 0 | 0 | null |
| **Park Place** | property | 25 | 350 | Dark Blue |
| **Luxury Tax** | tax | 100 | 0 | null |
| **Boardwalk** | property | 50 | 400 | Dark Blue |

We can see that there are five data properties for each square:

- **name** – the name of the square
- **type** – the type of the square. There are six types: *property*, *railroad*, *utility*, *plain*, *tax*, and *toJail*
- **price** – the cost to buy the square. A price of zero means the square is not for sale.
- **rent** – the rent that a player who lands on the square must pay.
- **color** – the color of the square. only property squares have color, the rest are null.

We need public methods to *get* each property, but not to *set* each property, since users will not be able to change the properties of each square. Their values will be set by a constructor. In this case, the set of methods will be simple – two constructors and the accessor for each square's property, plus a *toString()* method.

The *type* of the square could be stored as an enumerated data type or a String. We will use Strings. The *name* and *color* will also be Strings. The *price* and the *rent* will be integers. Here is an annotated UML diagram for the class, which we will name BoardSquare:

| BoardSquare (board squares for a Monopoly game) | |
|---|---|
| - name: String | name of the square |
| - type: String | property, railroad, utility, plain, tax, or toJail |
| - price: int | |
| - rent: int | |
| - color: String | color group; many are null |
| + Square(): void | |
| + Square(String, String, int, int, String): void | (name, type, price, rent and color) |
| + getName(): String | |
| + getType(): String | |
| + getPrice() int | |
| + getRent() int | |
| + getColor:(() String | |
| + toString:() String | returns data about the square as a String |

Here is the code for the corresponding class of BoardSquare objects:

**from Monopoly.Java** **in the files for Chapter 9 as a NetBeans project**

```java
/* code for a class of Monopoly squares
 *
 * CSCI 111 Fall 2013
 * last edited November 2, 2013 by C. Herbert
 * Each object in this class is a square for the board game Monopoly.
 *
 * This is for teaching purposes only.
 * Monopoly and the names and images in the game are
 * registered trademarks of Parker Brothers, Hasbro, and others.
 */
class Square {

    private String name;    // the name of the square
    private String type;    // property, railroad, utility, plain, tax, or  toJail
    private int price;      // cost to buy the square; zero means not for sale
    private int rent;       // rent paid by a player who lands on the square
    private String color;   // many are null; this is not the Java Color class

 // constructors
    public BoardSquare()
    {
        name = "";
        type = "";
        price = 0;
        rent = 0;
        color = "";
    } // end BoardSquare()

    public BoardSquare(String name, String type, int price, int rent, String color)
    {
        this.name = name;
        this.type = type;
        this.price = price;
        this.rent = rent;
        this.color = color;
    } // end BoardSquare(String name, String type, int price, int rent, String color)

    // accesors for each property
    public String getName()
    {
        return name;
    } //end  getName()

     public String getType()
    {
        return type;
    } //end  getType()

     public int getPrice()
    {
        return price;
    } //end  getPrice()

     public int getRent()
    {
        return rent;
    } //end  getRent()

     public String getColor()
    {
        return color;
    } //end  getColor()

   // a method to return the BoardSquare's data as a String
   public String toString()
    {
    String info = (name +", "+type+", "+price + ", "+ rent+ ", "+color);
    return info;
    } //end  toString()

} // end class BoardSquare
```

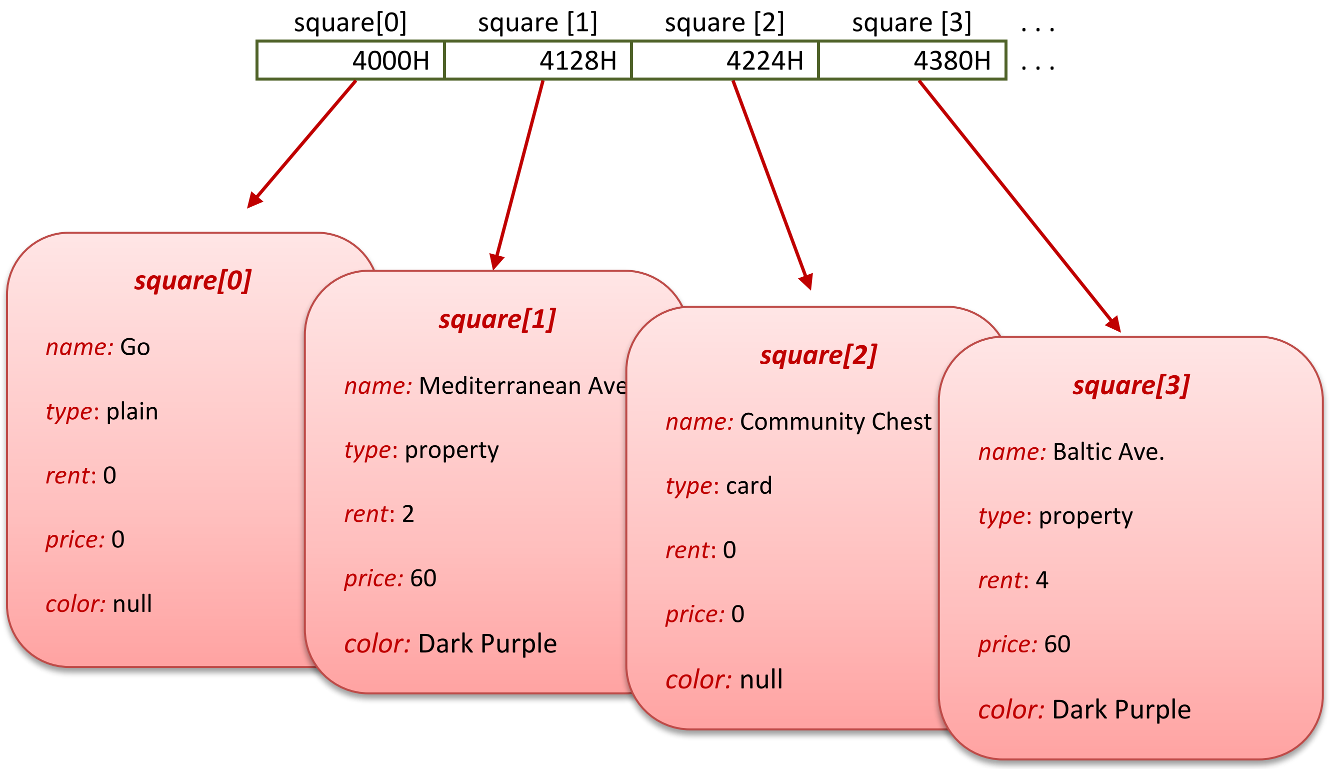
Next, we need code to create the array to hold the *BoardSquare* objects, and then initialize the values of each instance of a *BoardSquare* object using an initializing constructor.

This code creates an array named *square* of type BoardSquare with 40 elements.

**from Monopoly.Java** **in the files for Chapter 9 as a NetBeans project**

```
        /* This code creates an array named square[] of 40 BoardSquare objects
         * for a Monopoly game, then reads data from a file into the properties
         * of each object in the array.
         *
         * The code can be added to an appropriate method.
         */

        BoardSquare[] square = new BoardSquare[40]; // array of 40 monopoly squares

        int i; // a loop counter

        // declare temporary variables to hold BoardSquare properties read from a file
        String inName;
        String inType;
        int inPrice;
        int inRent;
        String inColor;

        // Create a File class object linked to the name of the file to be read
        java.io.File squareFile = new java.io.File("squares.txt");

        // Create a Scanner named infile to read the input stream from the file
        Scanner infile = new Scanner(squareFile);

        /* This loop reads data into the square array.
         * Each item of data is a separate line in the file.
         * There are 40 sets of data for the 40 squares.
         */
        for( i=0; i<40; i++)
        {
            // read data from the file into temporary variables
            // read Strings directly; parse integers
            inName  = infile.nextLine();
            inType  = infile.nextLine();
            inPrice = Integer.parseInt( infile.nextLine() );
            inRent  = Integer.parseInt( infile.nextLine() );;
            inColor = infile.nextLine();

            // intialze each square using the BoardSquare constructor
            square[i] = new BoardSquare(inName, inType, inPrice, inRent, inColor);
        } // end for

        infile.close();

    } // main()
```

Each element in the array *square* will now be a reference variable, holding the memory address where the corresponding BoardSquare object is actually stored in memory.



## 9.5 Abstract Methods, Abstract Classes, and Java Interfaces

An **abstract method** in Java is a method that only has a method header – including its return type, name, and parameter list – but no method body. It's really not a method, just information about the method. Abstract methods are used to create *Java interfaces* and *Abstract classes*, which we learn more about in just a moment.

The header for an abstract method starts with the keyword *abstract* and ends with a semicolon, such as in the following examples:

```
abstract double getArea();

abstract void draw();

abstract void resize();
```

Each of these three might be used in a program that draws shapes on the screen.

A **concrete method** is a method that is not abstract – in other words, the normal methods that we have been using so far.

Obviously, abstract methods can't do anything, but can they form the basis for extensions of Java Abstract classes and implementations of Java interfaces that can do things.

## Abstract Classes

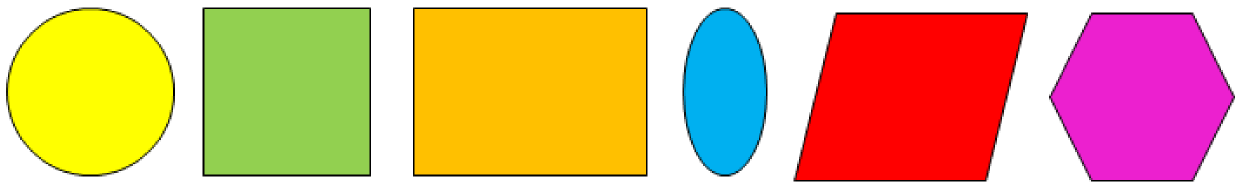
An **abstract class** is a class that cannot be instantiated – in other words, no objects of the class can be created. An **abstract class** serves as the basis for subclasses that can be instantiated.

An abstract class may contain one or more abstract methods. Any class that contains at least one abstract method must be declared as an abstract class. The following example shows how and why abstract classes are used.

## Example of an Abstract Class – the Shape Class

Imagine that we wish to have a graphics program that can draw a variety of shapes on the screen, and can give us information about the shapes it draws – *color*, *perimeter*, *area*, *location*, and so on. The shapes will be in different classes that have different properties.

For example, a *circle* will have a radius, whereas a *square* will have a side, and a *rectangle* will have a long side and a short side. All of the shapes will have an area, a perimeter, a color, and a location.



The methods to draw the shapes on the screen will work differently for different shapes. The software will use the location of the center of a circle and its radius to draw a circle, while it will use the location of the top left corner of a square and the length of its side to draw a square. The classes for rectangles, ellipses, regular hexagons, triangle, parallelograms, and so on, will all use different information to draw the shapes, but they will all have draw methods; The same is true for area methods and perimeter methods; each class will have them , but they will not all work the same way.

This is an obvious case where inheritance can be used. We can set up a parent class, named *Shape*, with subclasses *Circle*, *Square*, *Rectangle*, and so on. All of the objects in the graphics system will be objects of one of the subclasses of Shape – they will all be circles, squares, rectangles, and so on. No objects of the parent class Shape will be created. A few properties – such as *location*, *color*, *area* and *perimeter* – will be common to all shapes but other properties and most methods will be different for each subclass.

This is the perfect case for an abstract class. We will declare *Shape* to be an abstract class, which means there will be no *Shape* objects. The objects will be in subclasses of Shape . We will put the common properties in the Shape class to be inherited by the subclasses.

We will also give the shape class abstract methods for *draw*, *area*, and *perimeter*, which will tell programmers that the subclasses must implement these methods. Concrete classes that are subclasses of abstract classes must implement the abstract classes inherited from the abstract parent class.

Here is what the abstract parent class Shape will look like:

```
abstract class Shape {

   // declare properties
    Point location;     // an object of type Point has an x and y coordinate
    Color color;        // using Java's built in Color class
    double area;
    double perimeter;

   // declare concrete methods -- accessors for location and color
    Point getLocation()
   {
      return location;
    }

    Color getColor()
   {
      return color;
   }

   // declare concrete methods – mutators for location and color
    void setLocation(Point location)
   {
      this.location = location;      }

    void setColor(Color color)
   {
      this.color = color;
   }

   // declare abstract methods
    abstract void draw();
    abstract double calculateArea();
    abstract double calculatePerimeter();

}// end abstract class Shape
```

The abstract class *Shape* can now be used as the basis for subclasses. The subclasses will inherit the four properties – *location, color, area*, and *perimeter* – and the concrete methods, and they must implement the abstract methods *draw(), calculateArea()*, and *calculatePerimeter()*. The code for the Circle class, minus some of the details, is shown below.

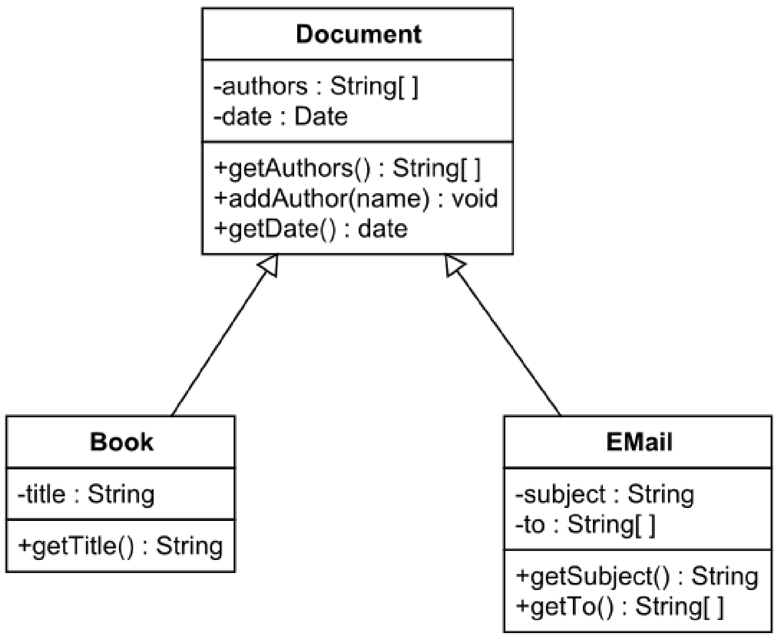
```
class Circle extends Shape {

    // declare Circle specific properties
    private double radius;
    private Point center;   // center and location will be the same point.

    // constructors
    public Circle()
    {
        radius = 0;
        area = 0;
        perimeter = 0;
    }
    public Circle(Point location, double radius, Color color)
    {
        this.location =location;
        this.radius = radius;
        this.color = color;

        // re-calculate area and perimeter whenever radius changes
        area = calculateArea();
        perimeter = calculatePerimeter();
    }

   // add additional accessors
    public double getRadius()
   {
      return radius;
   }

    public double getCenter()
   {
      return center;
   }

   // add additional mutator
    public void setRadius(double radius)
   {
       this.radius = radius;
       // re-calculate area and perimeter whenever radius changes
       area = calculateArea();
       perimeter = calculatePerimeter();
   } // end setRadius

    // implement classes that were abstract in the super class (parent class)
    public void draw()
    {
        // put the code here to draw a circle
    }

    private double calculateArea()
    {
        // area = π * r^2
        return Math.PI * radius * radius;
    }

    private double calculatePerimeter()
    {
        // area = 2 * π * r
        return Math.PI * 2.0 * radius;
    }

    // add additional methods for the Circle class here

} // end class circle
```

To summarize the use of Abstract classes:

- Abstract methods are methods that have no body, just the method header.
- An abstract class is a class that has at least one abstract method.
- An Abstract class cannot be instantiated – no objects of that class can be created.
- Abstract classes are the basis for concrete classes that can be instantiated.
- An Abstract class may contain properties and concrete methods that subclasses have in common, and abstract methods for other methods that the subclasses must have, but which will work differently in different subclasses.

## UML Diagrams for Inheritance and Abstract Classes

We use arrows to denote subclass relationships (inheritance) In UML diagrams, pointing from the subclasses to the superclass. The arrowheads are supposed to be open arrowheads ─▷, but often solid arrowheads ─► are used because they are much easier to draw. The simplified UML diagram below shows this. Book and and EMail both are subclasses of Document.



Technically, the UML standard only calls for abstract class names and abstract method names to be italicized, but often people put the word “abstract” with the names to make it more obvious that they are abstract.

## Java Interfaces

Abstract methods can also be used in Java Interfaces. Java Interfaces are very similar to abstract classes, but they have two differences:

1. Interfaces can only have abstract methods, not concrete methods. Interfaces can have constants, but not properties or concrete methods. (An abstract class can have concrete methods as well as abstract methods.)

2. Interfaces are *implemented* not inherited; they are not used as super classes. A concrete class can implement more than one interface.

Interfaces have no constructors, and are not classes that can be instantiated. They provide a standard set of method calls for other classes.

A class implements an Interface by using the phrase ”implements [interface name]” immediately after the name of the class in the class declaration.

Java interfaces are used to provide a common set of method calls for use within many classes. In essence, they form a contract that says “*This class will include the methods listed in the interface, with the same method names, parameters and return types.*” Effectively, **an interface in Java** is a list of abstract methods that will be implemented in any class that implements the interface.

Imagine that we have several major electronic companies making burglar alarms, cable television boxes, microwave ovens, and so on, all of which use an electronic clock with a display and a programmable clock chip made by one manufacturer. The manufacturer could publish a Java interface showing how to invoke the methods to program the chip. The various other manufacturers could include the interface in the Java code for their devices, so that programmers creating software for the various systems could all use the same method names, parameters, etc, for any clock software, making all of the system more compatible with one another.

The code below shows a clock interface.

```
interface Clock {

        void setTime(int hours, int minutes, int second);

        void setHours(int hours);

        void setTimeZone(int zone);

        void setMinutes(int minutes);

        void setSeconds(int seconds);

        int getHours();

        int getMinutes();

        int getSeconds();

        int getTimeZone();
    }
```

By publishing the interface, the clock chip manufacturer is telling others : “*these methods are guaranteed to work in our system.*” The Java software in the clock would implement this interface, as would any software in any compatible system that also declares that it implements the interface.

To implement an interface, the class header should include the declaration "*implements [name of interface]*" immediately after the class name. For example a class named *Timer* that implements the *Clock* interface might have a class declaration that starts like this:

```
Public class Timer implements Clock
    {
    // body of the Timer class goes here – properties and methods as usual
    }
```

This tells the world that the Timer class has code implementing all of the abstract methods in the Clock interface. Someone who knows the Clock interface would then know how to invoke and use the similar methods in Timer. In a similar manner, someone who knows the Clock interface would also know how to use the related methods in any system that implements Clock – household appliances, entertainment systems, communications equipment, and so on.

In some systems, the interface that specifies the behavior is called the *supplier*, and the class that implements the behavior is called the *client*.

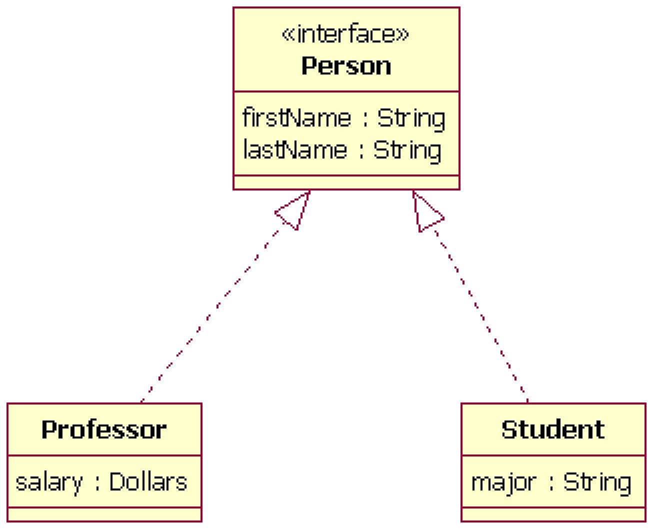
In the modern world, with so many systems interacting with one another all around the Earth via the Web, through mobile devices, via telephone and radio systems, and so on, Java interfaces go a long way toward ensuring some compatibility among software systems. Interfaces are one of the reasons the Java language is so widely in so many large complex computer systems.

To summarize the use of Java interfaces:

- An interface is a collection of methods, similar to those found in a class, but all of the interface's methods are abstract methods.
- An interface could also include properties and constants, but often only has abstract methods.
- A class that implements an interface will have methods that match the abstract methods in the interface, with the same method names and matching parameters.
- An interface is like a contract, with any class that implements the interface guaranteeing that the methods listed in the interface will work in the class.
- classes can implement more than one interface.
- interface implementation is independent of inheritance. Interfaces cannot be inherited; a class cannot extend an interface.
- a programmer declares a class will implement an interface by including "implements [interface name]" immediately after the class name, such as in "*public class Timer implements Clock* { … ".

## Interfaces in UML diagrams

The implementation of interfaces in UML diagrams is shown by using an arrow with a dotted line pointing from a class to any Interfaces it implements. In the following simplified example, both the *Professor* class and the *Student* class implement the *Person* interface:



## Java's Comparable Interface

One commonly used Java interface is the *Comparable* interface, whose only method is the *CompareTo()* method. The *CompareTo()* method compares two objects in a class and returns an integer.

The integer returned by **`a.compareTo(b)`** will be:

- a negative number if *a* comes before *b*;
- a zero if *a* equals *b*;
- a positive number if *a* comes after *b*.

An example of how this is used is the The String class *CompareTo()* method, which uses the lexicographic order (alphabetic order) of the Unicode characters in the String to compare them. It is most often used in an If statement, such as in this code from the BubbleSortDemo example in Chapter 7:

```
if ( a[i+1].compareTo(a[i]) < 0 )      // if the two items are out of order
   {
            // swap the two items and set swapped to true
            c = a[i];
            a[i] = a[i+1];
            a[i+1] = c;
            swapped = true;

   }  // end if
```

The pattern of use for the compareTo() methods in *if* statement basically is:

- `if (a.compare(b) < 0)` // then b comes before a.
- `If (a.compare(b) == 0)` // then a equals b.
- `if (a.compare(b) > 0)` // then a comes before b.

But this only works if we implement the compareTo() method in a way that tells the computer how to compare two items in a newly defined class, as the String class does. How will objects in a new class be compared?

To implement the *comparable interface* in a new class*,* we need to include code for the *compareTo()* method in a class declaration telling the computer how to compare two items in the new class. The following example shows how to do this with the *Book* class from the beginning of the chapter.

## Example – implementing Java's Comparable  Interface

Here is part of the class declaration for the Book class:

```
public class Book
{
   // declare properties
   private String isbn;
   private String title;
   private String subject;
   private double price;

   // constructor methods ******************************
   public Book()
   {
   } // end Book()

    the rest of the class declaration follows  this . . .
```

For the *compareTo()*  method to work ,we need to decide what to use as the basis for the order of our books.  We can use any property that already has a predefined order, such as the ISBN or the title, which are Strings.  If the a new class has a property that is a key field, then that should be used as the basis for the compareTo() method.  A **key field** is a field that has a unique value for each instance of an object. Your Jnumber at CCP is an example of a key field – no two students can have the same Jnumber. Social security numbers, bank account numbers, and Vehicle Identification Numbers (VIN) are all examples of key fields.

*isbn* is a key field. It can be used to compare two objects. The following code shows how to implement the Compareto() method in the book class using the String property *isbn*:

```
   /* compareTo() method using ISBN as the basis for ordering Book objects
    * a.compareTo(b) compares Book a to Book b
  public int compareTo(Book b)
  {
      int x;
      x = this.isbn.compareTo(b.isbn);
      return x;
  } // end Book()
```

In the code shown here, ***this.isbn*** is the ***isbn*** property for the Book object invoking the method.  In code outside of the class, Book object ***a*** can be compared to Book object ***b*** this way - `a.compareTo(b);`

The ***isbn*** property is a String, so one ***isbn*** is compared to another in the method by using the String class *compareTo()* method. ***this.isbn.compareTo(b)*** invokes the String *compareTo()* method for ***this.isbn***. The Book ***compareTo()*** method then returns the result of that String comparison. The result is that Book class objects will now be compared by their ***isbn*** properties whenever a program uses *compareTo()* with Book class objects.

The Book class header can now include the "*implements comparable"* phrase:

```
public class Book implements Comparable
{
   // declare properties
   private String isbn;
   private String title;
   private String subject;
   private double price;

   // constructor methods ******************************
   public Book()
   {
   } // end Book()

    . . . other methods would follow here, including

   public int comareTo(Book b)
   {
       int x;
       x = this.isbn.compareTo(b.isbn);
       return x;
    } // end Book()

the rest of the class declaration follows this . . .
```

We can include the Book *compareTo()* method in the Book class declaration without implementing Java's *comparable* interface, so why should we bother to do so? The answer is to let programmers know that our class of objects can be compared and ordered using this standard method. By putting a *compareTo()* method in the Book class declaration and including the "*implements comparable"* phrase with the Book class header, we are telling other programmers that Book objects can be compared to one another, so they can be sorted, etc.

Think of it like a badge worn by a soldier. A soldier who is a properly trained combat parachutist wears a badge with combat jump wings on his uniform, indicating he is a trained parachute jumper. In a similar manner, the "*implements comparable*" phrase indicates that a class includes a proper *compareTo()* method.

Please note that the *comparable* interface is included in the standard *java.lang* package, so it can be used without any import statement. Other Java interfaces, such as some of those used in GUI programming, may need import statements to work in you code.

## Chapter Review

**Section** 1 of this chapter described class declarations for a new class of objects in Java. Key terms included: **class declaration** and **default constructor.** (Note: Most of the new key terms for this topic were introduced in the previous chapter.)

**Section 2** discussed two ways to organize class declarations in Java code: multiple classes in a single **.java* file, and inner classes. Key terms included **inner class** and **nested classes**.

**Section 3** described appropriate use of the Java Keyword *this.*

**Section 4** described how arrays of objects work in Java, using reference variables.

**Section 5** discussed the Java abstract methods and their use in Abstract Classes and Java interfaces, including the use of Java's Comparable interface. Key terms included: **abstract method, concrete method, abstract class, interface, and key field.**

## Chapter Questions

1. By what other name is a Java class declaration also known? What is a default constructor? What happens if a Java Class declaration does not have a constructor?
2. In a *.java* file, when should a class be public? What class is always public in a *.java* file? How many classes in a *.java* file can be public?
3. How are inner classes declared in Java? When is an inner class a good idea, and when is it a bad idea?
4. What odes the keyword this refer to in a Java method? When should it be used?
5. What is a static property? How are static properties invoked?
6. What is actually stored in an array of objects in java?
7. How is an abstract method declared in Java? How is it different from a concrete method?
8. How is an object of an abstract class instantiated in Java? What are abstract classes used for? What classes must be declared as an abstract class?
9. What do we know about all of the methods in a Java interface? What does implementation of an interface by a class guarantee for a programmer using the class? How many interfaces can a class implement?
10. How is inheritance indicated on a UML diagram? How is implementation of an interface indicated?

## Chapter Group Exercise

**Creating Additional Classes for a Monopoly game**

Section 4 of this chapter described the BoardSquare class, a class of objects for squares on a monopoly board. The NetBeans project *Monopoly* included with the files for this chapter, has that class and a public class named *Monopoly* with an executable main method.

Your task is to create a new class of objects for a player in a monopoly game, and to revise the public method in the Monopoly NetBeans project so that a player can move around the board. We are not implementing a complete monopoly game, but just creating a player class and testing how it works with the BoardSquare class.

The class discussion for this week is about this project. You may work with other students and communicate about the project through the class discussion, through email, or by meeting in person, to discuss the design and programming for this project, but each person should do his or her own work, writing his or her own code. You must participate in the class discussion. Your lab report should indicate who you worked with.

You should:

1. Design and create the class declaration for the player class according to the annotated UML diagram below, and add the class it to the NetBeans Monopoly project.
2. Modify the *main()* method in the Monopoly class so that the player moves around the board, as follows:
    a. the player's location will be the square that the player's token is currently on.
    b. the player's location should start on Square zero, the *Go* square.
    c. the computer should pick a random number beween 1 and 12 to simulate the roll of a pair of dice. It should pick two random numbers between 1 and 6 and add them together to ensure the same probability as rolling real dice.
    d. The main method should then move the player to the new square by adding the random number to the number of the square the player is on, and set the player's location to the new square. The squares are numbered 0 to 39, so whenever the new square number would go above 39 it should be reset (If newsquare > 39, then newsquare = newsquare – 39).
    e. When a player arrives on a square:
        i. the rent for the square should be subtracted from the player's bank balance
        ii. a message should be displayed (console or JoptionPane, it's up to you) with
            1. the name of the player
            2. the name of the player's token

3. the roll of the dice
4. the name of the square the player is now on
5. the rent for the square, if it is not zero
6. the player’s new bank balance

The exact format of the message is up to you.

f. after a player has arrived on a square, the program should say “press enter to continue”, then use an input statement of some kind (keyboard scanner or inputDialog Window) to continue to the next turn.

g. The program should continue (loop) until the user stops the program, or the player’s bank balance is zero.

Remember, this is the first part in building a Monopoly game in Java so we are only testing the player class object moving around the board interacting with the BoardSquare class. Things like picking a card if the player lands on *Chance,* buying and selling properties, collecting $200 for passing *Go*, and so on, are not part of this project. In a real development situation, those things would be added in another phase of the project once the BoardSquare and Player objects work properly. Graphics (and sound) would also be added later. Don’t get carried away with fancy features at this point and make the assignment harder than it needs to be.

| Player (player for a Monopoly game) | |
|---|---|
| - name: String | name of the player |
| - token: String | racecar, wheelbarrow, battleship, top hat, etc. |
| - location: int | the number of the square the player is on |
| | initialized to zero |
| - balance: int | the player’s current bank balance |
| | initialized to 1500 |
| + Player(): void | |
| + Player(String, String, int, int): void | (name, token, location, bank balance) |
| + getName(): String | |
| + getToken(): String | |
| + getLocation() int | |
| + getBalance() int | |
| + setName(String): void | |
| + setToken(String): void | returns data about the player as a String |
| + setLocation(int): void | |
| + setBalance(int): void | |
| | |
| + toString:() String | |

## Contents