`public abstract class` **`ChapterOne`**

Classes and Interfaces

1.1 Prologue 2
1.2 Inheritance 3
1.3 Polymorphism 10
1.4 Encapsulation and Abstraction 12
1.5 Interfaces vs. Abstract Classes 16
1.6 *Case Study and Lab:* Chomp 22
1.7 Summary 27
Exercises 29

1.1 Prologue

If you go to a computer museum, you will find pictures of the first computers from the 1950s. Each of them was custom-designed and built, with special electronics for the CPU and memory. What is now inside a small laptop took up a whole room: rows of cabinets filled with vacuum tubes, connected by massive cables; as in a sci-fi movie, you could enter and walk inside. By contrast, modern computers are designed around standard components and built in automated factories. That's what makes it possible to put one on your lap for a reasonable price.

You won't find pictures of computer software in a museum. But if you could enter and walk around software systems, you would find all kinds of things, from rather elegant structures to huge monstrosities, often barely standing, with pieces held together by the software equivalent of string and duct tape. A small change in one place upsets the whole structure; more strings and tape and props become necessary to prevent the contraption from collapsing. Perhaps these specimens do belong in a museum after all! For better or worse, software remains mostly invisible, hidden behind thousands of lines of code. It is mysterious, sometimes even full of surprises (Figure 1-1).

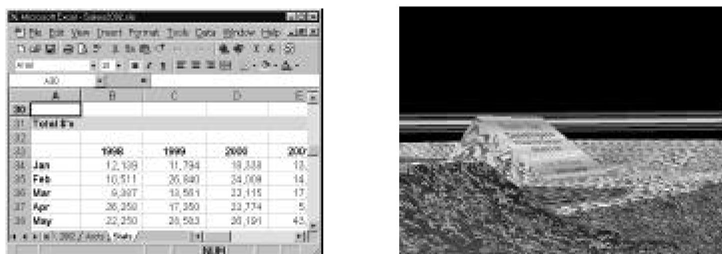


Figure 1-1. Microsoft's *Office 97* version of *Excel*, a popular spreadsheet program used by millions of people for serious work ranging from financial forecasting to scientific modeling, has a well-hidden toy *Flight Simulator* built into it. Did a mischievous developer put it in? In *Excel 97*, press `Ctrl-G`, type `L97:X97`, press `Enter`, press `Tab`, then click on the "Chart Wizard" icon while holding the `Shift` and `Ctrl` keys down. Expert flying eventually reveals a monolith with the names of the product development team scrolling on it. Press `Esc` to return to the spreadsheet.

When we say that a computer program is "written," like a book or a tune, this implies an individual creative effort with few constraints. The notion of programs as being "written" influenced the way we thought about software for a long time. But a while ago, software researchers and professionals started asking themselves: Can we, instead, assemble computer programs from standard pre-tested software components, just as modern computers are assembled from ready-made hardware components? Object-oriented programming has taken up the task of realizing this dream.

The main goals of OOP are team development, software reusability, and easier program maintenance. The main OOP concepts and tools that serve these goals are abstraction, encapsulation, inheritance, and polymorphism. In this chapter we review these key concepts and their implementation in Java. We take a closer look at how Java uses classes and interfaces, inheritance hierarchies, and polymorphism to achieve the goal of better-engineered programs. Our case study and lab for this chapter is an applet that plays the *Chomp* board game with the user. It demonstrates abstraction, encapsulation, inheritance, and polymorphism.

1.2 Inheritance

As we know, in Java all objects belong to classes. A class defines the features and responsibilities of its objects: their fields (data elements) and methods. In strongly-typed computer languages, like Java, the class determines the data type of its objects.

> **Inheritance allows us to take a class and** *extend* **it, "inheriting" all the fields and methods of the** *base class* **in the** *derived class***. The derived class can redefine some of the features or add new features.**

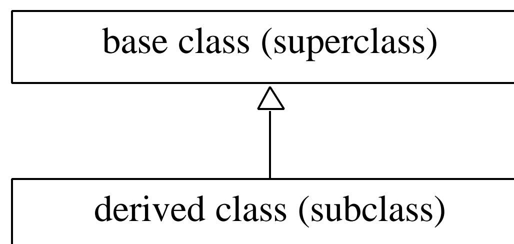


Figure 1-2. Terminology and notation for inheritance: base class is extended to make derived class

The base class is also called the ***superclass***, and the derived class is also called the ***subclass*** (Figure 1-2). In Java, `super` is a reserved word that allows constructors and methods of the subclass to refer to constructors, methods, and fields of the superclass.

Several classes are often derived from the same class. A derived class may in turn become a base class for a new derivation, resulting in a hierarchy of classes. In Java, all classes belong to one big hierarchy derived from the most basic class, called `Object`. This class provides a few features common to all objects; more importantly, it makes sure that any object is an `Object`, an instance of the `Object` class, which is useful for implementing structures that can deal with any type of objects. If we start our own class "from scratch," then our class automatically extends `Object`. For example:

```
public class Player
{
  ...
}
```

is equivalent to:

```
public class Player extends Object
{
  ...
}
```

When we derive new classes from `Player`, we start building a class hierarchy for our application or package. For example:

```
public class HumanPlayer extends Player
{
  ...
}

public class ComputerPlayer extends Player
{
  ...
}
```

This results in the hierarchy shown in Figure 1-3.

The classes that lie closer to the top of the hierarchy are more general and abstract; the classes closer to the bottom are more specialized. Java allows us to formally define an ***abstract*** class. In an abstract class, some or all methods are declared `abstract` and left without code.

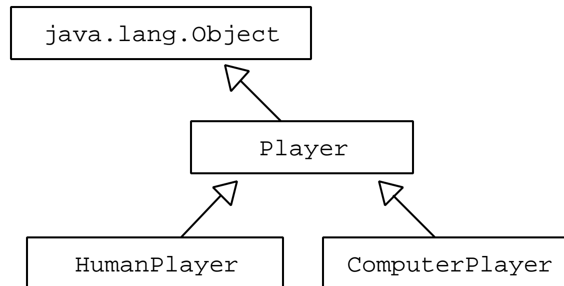


Figure 1-3. `Player` **and two derived classes**

> **An `abstract` method has only a heading: a declaration that gives the method's name, return type, and arguments. An `abstract` method has no code.**

For example, consider the definition of `Player` in Figure 1-4. All its methods are abstract. `Player` tells us what methods a player must have but does not tell us exactly how they work.

> **In an `abstract` class, some methods and constructors may be fully defined and have code supplied for them while other methods are `abstract`.**

A class may be declared abstract for other reasons, too. For example, some of the fields in an abstract class may belong to abstract classes.

More specialized subclasses of an abstract class have more and more methods defined. Eventually, down the inheritance line, the code is supplied for all methods.

> **A class where all the methods are fully defined and which has no abstract fields is called a *concrete* class.**

A program can only create objects of concrete classes. An object is called an ***instance*** of its class. An `abstract` class cannot be ***instantiated***.

```
/**
 *  A player in a board game
 */

public abstract class Player
{
  /**
   *   Returns a prompt to be displayed before
   *   the next move of this player
   */
  public abstract String getPrompt();

  /**
   *   Returns a message to be displayed when
   *   this player has won
   */
  public abstract String getWinMessage();

  /**
   *   Called to initiate this player's next move
   */
  public abstract void makeMove();
}
```

Figure 1-4. The abstract class `Player`

Different concrete classes in the same hierarchy may define the same method in different ways. For example:

```
public class ComputerPlayer extends Player
{
  ...
  /**
   *   Returns a prompt to be displayed before
   *   the next move of this player
   */
  public String getPrompt()
  {
    return " Hmm... Let me think...";
  }
  ...
}
```

```
public class HumanPlayer extends Player
{
  ...
 /**
  *   Returns a prompt to be displayed before
  *   the next move of this player
  */
  public String getPrompt()
  {
    return " Your turn...";
  }
  ...
}
```

Java library packages provide classes for all kinds of things. Classes in your application often extend library classes. This lets you reuse library code and add your own features.

You do not need access to its source code to extend someone else's class or a library class.

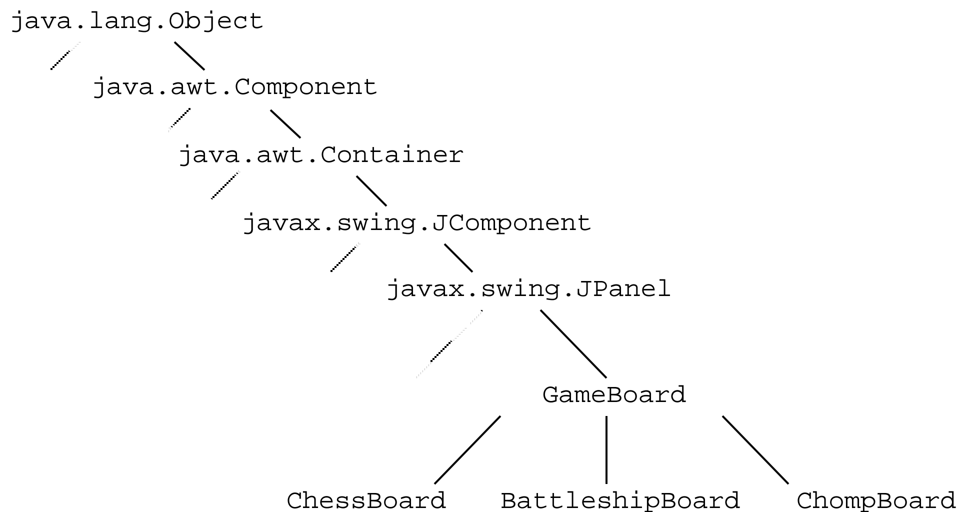


Figure 1-5. A hierarchy of classes (progressing from standard library classes to classes from a hypothetical package for board games)

The class hierarchy shown in Figure 1-5 starts with `Object`. Below it are a few library classes from the standard Java packages AWT and Swing. Under these are classes from a hypothetical package for board games. The `GameBoard` class may be an abstract class, but `ChessBoard`, `BattleshipBoard`, and `ChompBoard` are probably concrete classes.

In Java, constructors are not inherited, so subclasses of the same superclass may have different constructors with different arguments.

But the superclass constructors' code does not have to be duplicated in a subclass.

A subclass's constructor can explicitly call any constructor of its superclass by using the keyword `super`. If `super(...)` is called, then `super(...)` must be the first statement in the constructor.

Suppose, for example, that the `Player` class has a constructor that sets the value of the field `myName` to the string passed to it as an argument:

```
public abstract class Player
{
  private string myName;
  ...
  public Player(String name)  // constructor
  {
    myName = name;
  }
  ...
}
```

`HumanPlayer` and `ComputerPlayer` inherit the field `myName`, but they cannot directly access it because `myName` is `private` in `Player`. However, constructors of these classes can call `Player`'s constructor to set `myName`. This is accomplished using Java's reserved word `super`:

```
public class HumanPlayer extends Player
{
  public HumanPlayer(String name)  // constructor
  {
    super(name);
  }
  ...
}
```

Or:

```
public class ComputerPlayer extends Player
{
  public ComputerPlayer()  // constructor
  {
    super("Computer");
  }
  ...
}
```

Recall that if you do not define any constructors for a class, then Java provides one default constructor that takes no arguments and initializes all the fields to their default values: 0 for numbers, `false` for `boolean`s, `null` for objects. However, if you provide at least one constructor, then the default constructor is not provided automatically.

It is a good idea to define a so-called *no-args* constructor (a constructor with no arguments) for your class if you plan to define other constructors.

Derived classes can not only add fields and methods to the base class's fields, they can also redefine a method from the superclass. Sometimes a method of a subclass needs to expand the functionality of the superclass's method. Again, there is no need to duplicate the code from the superclass, no need even to have access to that code. Instead, the subclass's method can explicitly call any method of the superclass using `super`. For example:

```
public class ChompBoard extends JPanel
{
  ...
  public void paintComponent(Graphics g)
  {
    setBackground(Color.orange);
    super.paintComponent(g);     // fills the background

    // Now add your own drawing:
    ...
  }
}
```

When you derive a class from a library class or someone else's class, you have to be careful not to redefine a base class's method inadvertently.

To recap: inheritance allows us to reuse some code from the base class in a derived class, saving development time. It also helps to avoid duplicate code, thus making program maintenance easier: if you need to change the code, the change needs to be made in only one place.

1.3 Polymorphism

With inheritance and class hierarchies, the notion of an object's data type becomes a little blurred. Suppose we create an instance of the `ComputerPlayer` class. The data type of that object is `ComputerPlayer`. But a `ComputerPlayer` is also a kind of `Player`; if we need to refer to that object in a more generic way, ignoring the specifics of precisely what kind of `Player` it is, Java allows us to do that. For example:

```
Player p;
p = new ComputerPlayer(...);
...
p.makeMove();
```

There may be several situations where we need to refer to an object using its more generic supertype rather than its most specific type. One such situation is when we want to mix different subtypes of objects in the same collection (array, list, etc.). For example:

```
Player players[] = new Player[2];
players[0] = new HumanPlayer(...);
players[1] = new ComputerPlayer(...);
...
player[currentPlayer].makeMove();
```

This is possible because both `HumanPlayer` and `ComputerPlayer` are `Player`s.

Therefore, besides reusing code, inheritance serves a second purpose: it provides a common base data type that lets us refer to objects of specific types through more generic types of references; in particular, we can mix objects of different subtypes in the same collection. This is Java's response to two contradictory demands. On one hand, we want a programming language to strongly enforce the discipline of data types; on the other hand, we want to be able to apply the same algorithms to objects of slightly different data types and to mix different objects in the same collection.

You may be wondering: Once we start referring to objects using variables of more generic types, don't we lose the specific data type information about the object? For example, once we state

```
Player players[] = new Player[2];
players[0] = new HumanPlayer(...);
players[1] = new ComputerPlayer(...);
```

how can we (or the compiler) keep track of exactly what kinds of players `players[0]` and `players[1]` really are? When we write

```
players[currentPlayer].makeMove();
```

how does the compiler know which method to call? Both `HumanPlayer` and `ComputerPlayer` have a `makeMove` method, but these methods do different things.

The answer to this question is ***polymorphism***. When the code is compiled, the decision on which method to call is postponed until the program is executed — a technique called ***late*** or ***dynamic method binding***. When the program is running, each object itself knows exactly what kind of object it is, and it automatically makes sure the correct method is called. Roughly speaking, each object contains a link to a table that holds the entry points of its methods.

Polymorphism ensures that the correct method is called for an object of a specific type, even when the object is disguised as a reference to a more generic type.

Among other things, polymorphism is essential for plugging in your own methods in place of the default calls in library methods. For example, it is common to redefine the `paintComponent` method for the GUI components you have derived from library classes:

```
public class ChompBoard extends JPanel
{
  ...
  public void paintComponent(Graphics g)
  {
    ...
  }
}
```

Java's library code does not know what kind of components you have placed on your application window's content pane — it just repaints all of them in order by calling each one's `paintComponent`. Thanks to polymorphism, your own `paintComponent` methods are called for your components.

Polymorphism is automatic; all you have to do is properly derive your classes from a common base class.

Without polymorphism, the program would have to query objects about their type and then call the appropriate methods. That would look ugly, something like this:

```
String typeOfPlayer = players[currentPlayer].getClass().getName();

if (typeOfPlayer.equals("HumanPlayer"))
  ((HumanPlayer)players[currentPlayer]).makeMove();

else // if (typeOfPlayer.equals("ComputerPlayer"))
  ((ComputerPlayer)players[currentPlayer]).makeMove();
```

Polymorphism streamlines the code by making explicit type checks and conditional branching unnecessary.

1.4 Encapsulation and Abstraction

A well-designed class hides all the details of its implementation from other classes (and from the programmers who work on other classes). In particular, all fields in a class are usually declared `private`, as are "helper methods" used only internally by the class. A class interacts with other classes that use it (its ***clients***) through a well-defined set of public constructors and public methods. This principle is called ***encapsulation***.

It is useful to think of "privacy" as applied to the programmer who defines the class rather than to individual objects of the class. If a program creates several objects of the same class, methods of one such object can call the public and private methods of another and can access its private fields. For example:

```
public class Fraction
{
  private int num, denom;

  // Constructor:
  public Fraction(int a, int b)
  {
    num = a;
    denom = b;
    reduce();
  }

  ...

  public Fraction multiply(Fraction other)
  {
    return new Fraction(num * other.num, denom * other.denom);
          // this object's method can access and even modify
          // other object's private fields, as long as other
          // belongs to the same class.
  }

  ...

  /*
   * Reduces this fraction
   */
  private void reduce()
  {
    ...
  }
}
```

But the private fields and private methods of one class are not accessible in other classes. For example:

```
public class MyMath
{
  public static void main(String[] args)
  {
    Fraction f1(1, 2), f2(1,3);

    Fraction product = f1.multiply(f2);
    product.reduce();                 // Syntax error -- can't call
                                      //   Fraction's private method
    System.out.println(product.num
            + "/" + product.denom); // Syntax error -- can't access
                                      //   Fraction's private fields
  }
}
```

Encapsulation facilitates team development because it minimizes the required interactions between developers. After several programmers agree on the public interfaces for their classes, each can go his own way to develop and test the code for his class.

Encapsulation also makes sure that any changes to the implementation of the class remain hidden from other classes. You can do whatever you want to your class, and other classes in the project are not affected, as long as the class's interface remains the same. A local change in the code does not propagate through the whole program — a principle called ***locality***. This makes software maintenance more manageable.

Finally, encapsulation forces the programmer to focus on the important public features of the class, making the class's definition more abstract. ***Abstraction*** basically means ignoring irrelevant details and emphasizing the relevant ones. Encapsulation forces the programmer to formally define what is relevant. As a result, a class describes well-defined responsibilities of its objects. A more abstract definition makes the class more likely to be reused in other projects.

A class does not have to be declared `abstract` to be abstract to some degree. There are different levels of abstraction; more and more details become clarified as your objects become more specific. Still, it is a good idea to maintain some level of abstraction and not to make your classes very specialized too quickly. At the top levels of the inheritance hierarchy, the classes are often `abstract` in the Java reserved-word sense: the code for some of their methods remains undefined (or some of their fields are variables of abstract class types and remain uninitialized). You cannot create objects of such classes, but they do help you formalize the properties and responsibilities of objects.

More abstract often means more reusable.

For example, the `Player` class in Section 1.2 (Figure 1-4 on page 6) defines three public methods:

```
public abstract String getPrompt();
public abstract String getWinMessage();
public abstract void makeMove();
```

Such a "player" can participate in any game for any number of players.

An encapsulated class usually provides some public methods called ***accessors*** that return the values of private fields and some ***modifiers*** that set the values of private fields. It is common to start an accessor's name with "get" and a modifier's name with "set." For example:

```
public class Player
{
  private string myName;
  ...
  public String getName()
  {
    return myName;
  }

  public void setName(String name)
  {
    myName = name;
  }
  ...
}
```

If necessary, accessors and modifiers can convert the field's value into a different data type or perform additional transformations or adjustments. A modifier can perform additional checks to make sure that the modified field always gets a valid value. For example:

```
  public void setName(String name)
  {
    if (name != null)
      myName = name.trim();
  }
```

The same programmer often works on a whole hierarchy of classes; still, the programmer encapsulates all the classes in the hierarchy, relying on accessors and modifiers for access to private fields. In fact, a programmer may prefer to use accessors and modifiers rather than access the fields directly even within the same class. That increases locality, making the code less dependent on the class's implementation.

Java allows you to declare a field or a method as `protected` rather than `private` or `public`. A protected field is directly accessible in subclasses but acts as private for other classes. You may be tempted to use protected fields in your base class and refer to them directly in your derived class, as opposed to using private fields with accessors and modifiers. However, if you do that and later change the fields in your base class, you might need to change your derived class, too. Bad for locality. Even worse: What if another programmer derives a class from your base class and refers

directly to your class's protected fields? Then you make changes to the implementation of your class in the next release of your package, and the other programmer's class no longer works. That is why it is safer to always make all fields `private`. You may occasionally want to declare a method `protected` rather than `public`. This would indicate that the method is somewhat more technical, like a specialized more intricate tool: perhaps useful in derived classes, but not for general use.

1.5 Interfaces vs. Abstract Classes

Java class hierarchies are great, but the real world is not always quite as neat, and OOP applications may have trouble fitting all the objects into one hierarchy of types. A `HumanPlayer` is a `Player`, but what if we want it to also be a `Person` with such attributes as name, age, e-mail address, and so on? Wouldn't it be nice to be able to combine the features of two or more base classes in one derived class? Something like

```
public class HumanPlayer extends Player, Person // syntax error!
```

Java does not allow such ***multiple inheritance***. The reason is simple: if more than one of the base classes define a method or field with the same name, which one of them is inherited in the derived class? If all of them are inherited, how can we distinguish between them? The compiler (and the programmer) may get confused.

As usual, Java offers a compromise: a class can extend only one base class, but it can ***implement*** several ***interfaces***.

> **An *interface* is akin to an `abstract` class: it gives a formal specification for objects by listing all the required methods, but does not provide the actual code.**

`implements` and `interface` are Java reserved words. The choice of the word "interface" is perhaps unfortunate: this computer term is already overused (or, as a programmer might say, "overloaded"). We have "hardware interface," "user interface," "class's interface..." It is the latter usage that has inspired Java's keyword `interface`.

> **When we state in a class's definition that the class `implements` an `interface`, it means that the class supplies code for all the methods specified in that interface.**

An interface simply gives the prototype for each method (its name, return type, and the argument list) but no code. In that respect an interface is very similar to an `abstract` class. The difference is that an `abstract` class can have some fields, constructors, and public or private methods implemented. An interface does not have any constructors and no code is defined for any of its methods. Thus an interface is even more abstract than an `abstract` class!

Writing interfaces is easy. Let us take, for example, the abstract class `Player` from Section 1.2 (Figure 1-4 on page 6). Figure 1-6 shows `Player` rewritten as an interface. Instead of `public class` we write `public interface`. We can also omit the keyword `public` in all the methods because they are assumed to be public by default. (Likewise, if an interface has fields, they are assumed to be `public static final` by default and must be initialized.)

```
/**
 *  A player in a board game
 */

public interface Player
{
  /**
   *   Returns a prompt to be displayed before
   *   the next move of this player
   */
  String getPrompt();

  /**
   *   Returns a message to be displayed when
   *   this player has won
   */
  String getWinMessage();

  /**
   *   Called to initiate this player's next move
   */
  void makeMove();
}
```

Figure 1-6. `Player`, rewritten as an interface

Now, instead of deriving `HumanPlayer` and `ComputerPlayer` from the abstract class `Player`, we can make them implement the interface `Player`:

```
public class HumanPlayer implements Player
{
  ...
}

public class ComputerPlayer implements Player
{
  ...
}
```

We don't have to change anything else.

The abstract class `Player` did not have any code for its methods anyway: all of them were abstract. So we have not lost anything by rewriting it as an interface. But we have gained something: we have freed the "extends" slot previously occupied by `Player`. Now, if necessary, we can derive `HumanPlayer`, `ComputerPlayer`, and other classes that implement `Player` from something else. For example:

```
public class HumanPlayer extends Person
  implements Player
{
  ...
}
```

If necessary, the same class can implement multiple interfaces.

For example:

```
public class HumanPlayer
  implements Player, MouseListener
{
  ...
}
```

Or:

```
public class ComputerPlayer
  implements Player, ActionListener
{
  ...
}
```

Interfaces are usually pretty short — it doesn't make much sense to declare too many methods in one interface. If necessary, you can split the methods between several interfaces. Then a class can implement all the interfaces it needs. Recall how the library interface `MouseListener` specifies five methods: `mouseClicked`, `mouseEntered`, `mouseExited`, `mousePressed`, and `mouseReleased`. Another interface, `MouseMotionListener`, specifies two methods, `mouseMoved` and `mouseDragged`. A class can implement one or both of these interfaces.

It is also possible to add methods to an interface by deriving a "subinterface" from it. Something like:

```
public interface ChessPlayer extends Player
{
  String getCheckMessage();
}
```

We have already mentioned the dual benefits of inheritance: it helps us reuse code, and it provides a common base data type for polymorphic collections. Java's variety of "multiple inheritance" with interfaces offers nothing for reusability of code because interfaces don't have any code. But interfaces are very helpful for supplying secondary data types to objects and for supporting polymorphism. Just as a real-life object may be used in different ways depending on the situation (e.g., for tossing up in the air, as a container, or to cover one's head), an object in a program may assume different "appearances" for different purposes.

> **If a class implements an interface or several interfaces, an object of that class has the primary data type defined by its class, plus secondary data types defined by all the interfaces that the class implements.**

Consider, for example, a class `HumanPlayer`:

```
public class HumanPlayer
  implements Player, MouseListener
{
  ...
}
```

An object of this class has the data type `HumanPlayer`. For example,

```
    HumanPlayer p = new HumanPlayer("Joe");
```

Once the interface `Player` is introduced, it becomes a legitimate data type. For example, we can create an array of `Player`s:

```
Player players[] = new Player[2];
```

Since `HumanPlayer` implements `Player`, we can put a `HumanPlayer` object into the array `players`, together with objects of other classes that implement the `Player` interface. For example:

```
players[0] = new HumanPlayer(...);
players[1] = new ComputerPlayer(...);
```

`HumanPlayer` also implements the `MouseListener` interface that specifies methods for capturing mouse events (the moves for this player are entered with a click of the mouse). Therefore, a `HumanPlayer` object is a kind of `MouseListener`; `MouseListener` is yet another of its secondary data types. This makes its possible to pass a `HumanPlayer` object as an argument to a method that expects a `MouseListener` and to place a `HumanPlayer` object into a list of `MouseListener`s. For example:

```
ChompBoard board = new ChompBoard(...);
HumanPlayer p = new HumanPlayer(...);
board.addMouseListener(p);
```

Polymorphism ensures that the correct methods are automatically called for an object that implements an interface.

A class that implements an interface must supply all the methods specified by that interface.

For example, even though `HumanPlayer` actually uses only one of `MouseListener`'s methods, `mouseReleased`, all five `MouseListener` methods must be defined, even if the code of the unused methods consists only of empty braces (Figure 1-7).

The reverse is not true: even if you implement all the interface's required methods in a class, that class still doesn't "implement" the interface unless you formally state that it does. If, for example, you supply all five `MouseListener` methods in `HumanPlayer` but forget the words `implements MouseListener` in the class's header and then try

```
HumanPlayer p = new HumanPlayer(...);
board.addMouseListener(p);
```

the compiler will report a syntax error and show a somewhat cryptic error message, something like this:

```
addMouseListener(java.awt.event.MouseListener) in java.awt.Component
cannot be applied to (HumanPlayer)
```

That means the data types do not match: `HumanPlayer` is not a `MouseListener`.

```
public class HumanPlayer
  implements Player, MouseListener
{
  ...

  /**
   *   Called automatically when the mouse button is released
   */
  public void mouseReleased(MouseEvent e)
  {
    if (!myTurn)
      return;
    ...
  }

  // Not used but required by the MouseListener interface spec:
  public void mouseClicked(MouseEvent e) {}
  public void mousePressed(MouseEvent e) {}
  public void mouseEntered(MouseEvent e) {}
  public void mouseExited(MouseEvent e) {}
}
```

Figure 1-7. `HumanPlayer` implements `MouseListener`

Inheritance, encapsulation, accessors and modifiers, abstract and concrete classes, interfaces — all of this may seem overwhelming at first. But this is the essence of OOP. It is not sufficient to say that your program consists of interacting objects; you need to know how objects of different types relate to each other. After all, if you are hoping to assemble programs from reusable components, you need to know the general properties of the components and the rules for putting them together.

1.6 *Case Study and Lab:* Chomp

The game of "Chomp" can be played on a rectangular board of any size. The board is divided into squares (let's say the board represents a chocolate bar). The rules are quite simple: the two players alternate taking rectangular "bites" from the board. On each move, the player must take any one of the remaining squares as well as all the squares that lie below and to the right (Figure 1-8). The square in the upper left corner of the board is "poison": whoever takes it loses the game. Run the *Chomp* applet using the compiled Java files on your student disk in the `Ch01\RunChomp` folder to get a feel for this game.

The number of all possible positions in Chomp is finite and the players make steady progress from the initial position to the end, as the total number of remaining "edible" squares on the board decreases with each move. Games of this type always have a winning strategy either for the first or for the second player. But, despite its simple rules, Chomp turns out to be a tricky game: you can prove mathematically that the first player has a winning strategy, but the proof does not tell you what that strategy is.* You know you can win if you go first, but you don't know how! Frustrating...

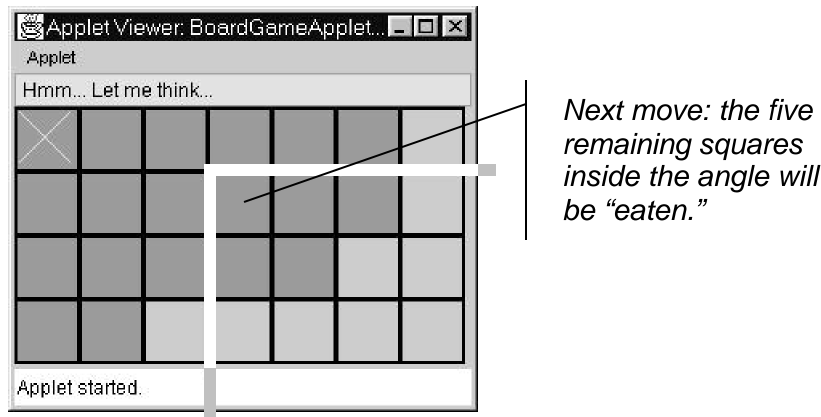


Figure 1-8. The *Chomp* game applet

---

* The proof looks like this. The first player can try to take the lower right corner on the first move. If this is the correct move in a winning strategy, the first player is all set. If it is not, the second player must have a winning move in response. But the first player could "steal" that winning response move and make it his own first move! In the theory of finite games, this argument is called "strategy stealing." Unfortunately, this proof gives no clue as to what the winning strategy might be.

As of now, no one has been able to come up with a formula for the winning Chomp positions (except for the 2 by n and n by n boards). There are computer programs that can backtrack from the final position (where only the "poison" square is left) and generate a list of all the winning positions. Our *Chomp* applet uses such a list, so the computer has an unfair advantage. You could try to "steal" the winning moves from the computer, but the applet's author has foreseen such a possibility and programmed the computer to intentionally make a few random moves before it settles into its winning strategy.

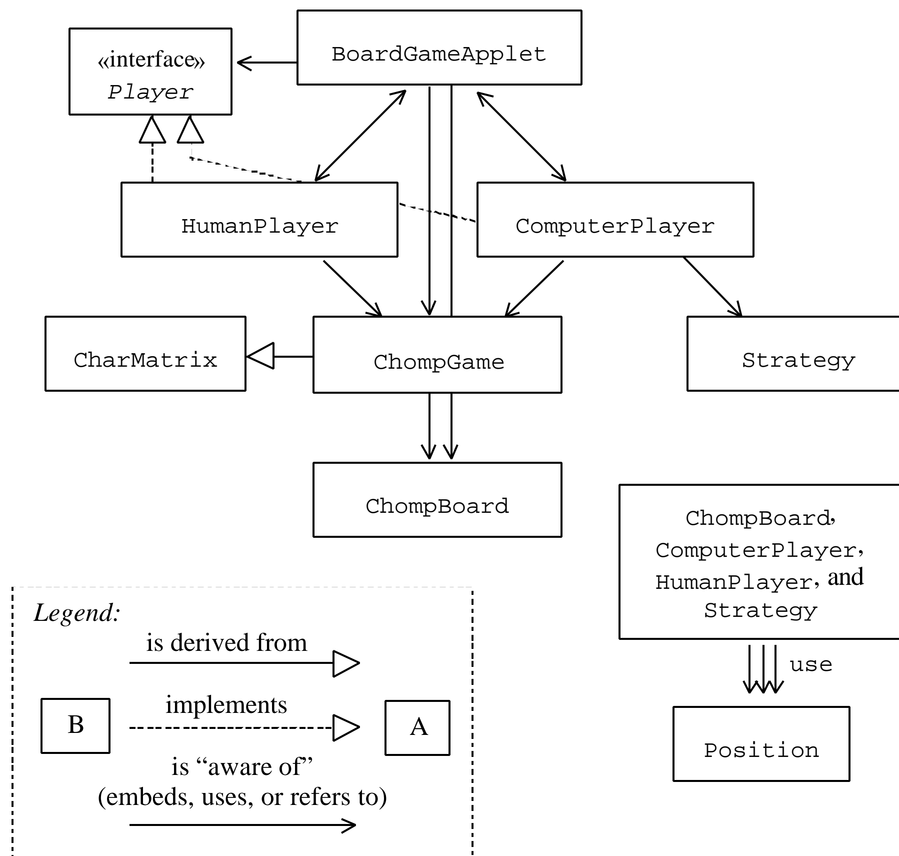


Figure 1-9. The classes in the *Chomp* applet

Luckily, our goal here is not to beat the computer at Chomp, but to learn some principles of object-oriented software design and to practice our Java programming skills. Let us begin by looking at the overall structure of this applet (Figure 1-9). The applet consists of eight classes and one interface, *Player*. An arrow from one

class to another indicates that the first class "knows" something about the second one: it uses the second class's constructors or methods. An arrow with a triangular head represents inheritance: it connects the derived class to its base class. A dotted arrow with a triangle from a class to an interface indicates that the class implements that interface.

In OOP, a large number of classes is not considered a problem, as long as they are reasonably short and manageable and clearly define the responsibilities of their objects. In designing this applet we tried to reduce dependencies between classes, called ***coupling***. `BoardGameApplet`, `HumanPlayer`, and `ComputerPlayer`, for instance, know very little about `ChompBoard`. `ComputerPlayer` is the only class that is aware of `Strategy`'s constructors and methods.

Let us briefly review the responsibilities of each of these classes.

The top class, `BoardGameApplet` (Figure 1-10), derived from `JApplet`, represents the applet as a whole. Its `init` method creates the game object, the board, and the players, and adds a display field and the board to the applet's content pane. Note how general this class is: it doesn't really care what game is played or how the players make their moves, as long as they take turns. The `BoardGameApplet`'s `hasMoved` method is called by each player when that player completes its current move. This method either displays a winning message if the game is over or else prompts the other player to make the next move. (`hasMoved` is needed because a player does not complete the move right away when `makeMove` is called: the "human player" waits for a mouse click and the "computer player" waits until the display stops flashing.)

As discussed in the previous sections, the `Player` interface, implemented by both the `HumanPlayer` and `ComputerPlayer` classes, allows us to place different kinds of "players" into the same array and rely on polymorphism to call their appropriate methods. The `HumanPlayer` object gets its next move from a mouse click on a particular square. The `ComputerPlayer` object creates a `Strategy` object and then consults it for each move.

The `ChompGame` class is designed to keep track of the board's configuration and implement the moves. But `ChompGame` does not display the board — that function is left to a separate "view" class. `ChompGame` only represents the "model" of the game. This class and the `Strategy` class are really the only classes that "know" and use the rules of Chomp.

```
/**
 *  Implements a board game applet with someone
 *  playing against the computer
 */
import java.awt.*;
import javax.swing.*;

public class BoardGameApplet extends JApplet
{
  private ChompGame game;
  private JTextField display;
  private Player players[];
  private int currentPlayer;

  public void init()
  {
    Container c = getContentPane();

    display = new JTextField(20);
    display.setBackground(Color.yellow);
    display.setEditable(false);
    c.add(display, BorderLayout.NORTH);

    ChompBoard board = new ChompBoard();
    c.add(board, BorderLayout.CENTER);

    game = new ChompGame(board);

    players = new Player[2];
    players[0] = new HumanPlayer(this, game, board);
    players[1] = new ComputerPlayer(this, game, board);
    currentPlayer = 0;                    // optional -- default

    display.setText(" You go first...");
    players[currentPlayer].makeMove();
  }

  /**
   *  Called by the player when its move is completed
   */
  public void hasMoved()
  {
    currentPlayer = (currentPlayer + 1) % 2;
    Player p = players[currentPlayer];
    ...
  }
}
```

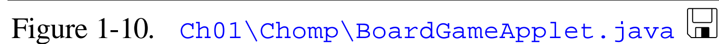


Figure 1-10. Ch01\Chomp\BoardGameApplet.java

The `ChompGame` class extends `CharMatrix`, a general-purpose class that represents a 2-D array of characters. A matrix of characters helps `ChompGame` to represent the current configuration of the board. We could potentially put all the code from `CharMatrix` directly into `ChompGame`. That would make the `ChompGame` class rather large. More importantly, general methods dealing with a character matrix would be mixed together in the same class with much more specific methods that deal with the rules of Chomp. Such a class would become intractable and hardly reusable. By separating the more general functions from the more specific ones, we have created a reusable class `CharMatrix` without any extra effort.

`Strategy` encapsulates the winning Chomp strategy and contains the list of winning positions for a 4 by 7 Chomp board. This class provides two public methods, `findBestMove` and `findRandomMove`. The code for this class is rather cryptic, but we don't really need to know much about it beyond this class's public interface, its constructor, and its two public methods. Whoever wrote and tested this class is responsible for it! This is team development at its best.

`ChompBoard` is the longest class in this project. It implements the "view," the graphics display of the board. The code in this class has to deal with rendering different squares in different colors and to support the "flashing" feedback for computer moves.

Finally, the `Position` class represents a (*row*, *col*) position on the board. It is a simple class: one constructor, `Position(row, col)`, and two accessors, `getRow()` and `getCol()`. We could have used one of the library classes instead but decided to write our own as an example. Such small auxiliary classes often define objects that carry information between other objects. In the *Chomp* applet, `Position` objects are passed from the strategy object to the computer player and from the board to the human player.

Looking ahead a little, we should mention that this applet's design fits the model-view-controller (MVC) design pattern explained in detail in Chapter 10. The main idea of such a design is to clearly separate the "model" (a more abstract object that describes the situation) from the "controller" (the object that changes the state of the model) and from the "view" (the object that displays the model). The "view" is attached to the model and changes automatically (or almost automatically) when the model changes. This way we can easily attach several different views to the same model if we need to. In the *Chomp* applet, the `ChompGame` class implements the "model." The applet's main class and both "players" together work as the "controller." Finally, the `ChompBoard` object is the "view."

We should also point out an intentional flaw in our design. The strategy object is created by the computer player, which makes it hard to change the strategy (e.g., to support different levels of play). A proper design would have a higher level object (e.g., the applet itself) attach a strategy to the computer player. `Strategy` would be an interface, which different strategy classes would implement. This improved design would follow the Strategy design pattern. Fixing the *Chomp* applet along these lines is the subject of Question 7 in the exercises.

As an exercise, restore the missing code in two of *Chomp's* classes. All the *Chomp* source files, as well as the HTML file that runs the applet and an audioclip file, are provided in the `Ch01\Chomp` folder on your student disk. Fill in the blanks in the `hasMoved` method of the `BoardGameApplet` class, calling as necessary the methods specified by the `Player` interface. Supply the missing code for all the methods in the `CharMatrix` class as described in their respective *javadoc* comments. Test your applet.

1.7 Summary

The key OOP concepts — abstraction, encapsulation, inheritance, and polymorphism — aim at better-engineered software, team development, software reusability, and easier program maintenance.

Inheritance allows a programmer to extend a base class (a superclass) into a derived class (a subclass). You can add new fields and methods and redefine some of the base class's methods in the derived class. Constructors are not inherited, but a constructor of a derived class can call any of the superclass's constructors. In Java, all classes with no specific derivation automatically extend the class `Object`.

Inheritance helps to arrange classes into a hierarchy, with more abstract classes near the top and more specialized classes below. Java allows you to declare a class `abstract` and declare some of its methods `abstract` with no code provided. The `abstract` methods eventually get defined further down the inheritance tree. A class in which all the methods are fully defined and implemented is called a ***concrete*** class.

Inheritance serves a dual purpose: it allows us to reuse the base class's code and avoid duplication of code; it also supplies a common generic data type of the base class to objects of different classes derived from that base class. Objects of different but related types can be mixed together in the same array or list or another collection.

Polymorphism ensures that the correct methods are called for an object of a specific type, even when that object is disguised as a more generic type.

A proper class hides the details of its implementation by making all its fields and some methods private. This concept is known as encapsulation. An encapsulated class is accessible to other classes, its clients, through a well-defined and documented interface consisting of public constructors and public methods. An encapsulated class may provide public accessor methods that return the values of the object's fields and public modifier methods that set the values of the fields. Encapsulation enforces locality: a change in the implementation of the class does not affect the rest of the program. This makes program maintenance more manageable. Encapsulation also makes team development easier: to use someone else's class a programmer needs to know only its public interface. Encapsulation forces programmers to pay more attention to the public features of their classes, leading to more abstract, reusable classes.

An interface in Java is akin to an abstract class: it lists a few methods, giving their names, return types, and argument lists, but does not give any code. The difference is that an abstract class may have its constructors and some of its methods implemented, while an interface does not give any code for its methods, leaving their implementation to a class that implements that interface. `interface` and `implements` are Java reserved words. A class that implements an interface must implement all the methods specified by that interface.

The same class can implement multiple interfaces. Each of the implemented interfaces supplies an additional secondary data type to an object of the class. This allows the program to treat the same object in different situations as an object of different data types. For example, an object can be passed to methods that expect an argument of a particular interface type. An interface can also serve as a common data type for mixing objects of different classes in the same polymorphic collection (array, list, etc.), as long as all the classes implement that interface.

Exercises

1. (a) Define an abstract class `WelcomeMessage` with a method

```
String getWelcomeMessage();
```

that returns a welcome message. Define subclasses `WelcomeEnglish`, `WelcomeSpanish`, and `WelcomeFrench` whose `getWelcomeMessage` method returns a welcome message in English, Spanish, and French, respectively. Write a test class that declares an array of three welcome messages (`WelcomeMessage` objects), initializes the elements to three messages in different languages, then displays all three.

(b) Modify the project, replacing the `WelcomeMessage` class with the `WelcomeMessage` interface.

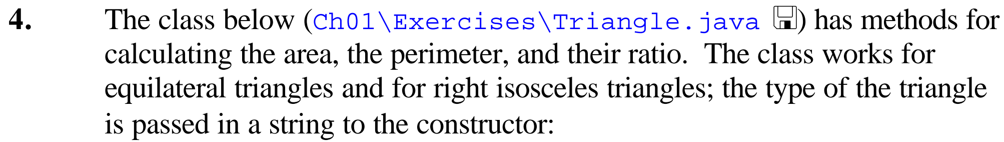
(c) Explain the advantage of using an interface rather than an abstract class in this project.

2. Java documentation indicates that the `Character` class implements the `Comparable` interface. Which method(s) are required for this interface? ✓

3. If class `B` implements interface `I` and class `D` extends `B`, is "implements I" in

```
public class D extends B implements I
```

required, optional, or forbidden? Set up a little test and see what the compiler says.



4. The class below (`Ch01\Exercises\Triangle.java`) has methods for calculating the area, the perimeter, and their ratio. The class works for equilateral triangles and for right isosceles triangles; the type of the triangle is passed in a string to the constructor:

```
public class Triangle
{
  private String type;
  private double side;

  public Triangle(String aType, double aSide)
  {
    type = aType;
    side = aSide;
  }

  public double getPerimeter()
  {
    if ("equilateral".equals(type))
      return 3 * side;
    else if ("right".equals(type))
      return (2 + Math.sqrt(2.0)) * side;
    else
      throw new RuntimeException("Invalid triangle type");
  }

  public double getArea()
  {
    if ("equilateral".equals(type))
      return Math.sqrt(3) / 4 * side * side;
    else if ("right".equals(type))
      return side * side / 2;
    else
      throw new RuntimeException("Invalid triangle type");
  }

  public double getRatio()
  {
    return getArea()/getPerimeter();
  }

  public static void main(String[] args)
  {
    Triangle equilateralTr = new Triangle("equilateral", 12);
    Triangle rightTr = new Triangle("right", 12);
    System.out.println("Equilateral " +
                            equilateralTr.getRatio());
    System.out.println("Right Isosceles " +
                            rightTr.getRatio());
  }
}
```

Continued

(a) Restructure the above program in the OOP style. Make the `Triangle` class abstract. Keep the `side` field but eliminate the `type` field. Make the `getArea` and `getPerimeter` methods abstract. Derive the concrete classes `EquilateralTriangle` and `RightTriangle` from `Triangle`. Provide an appropriate constructor for each of the two derived classes and make them call the superclass's constructor. Redefine the abstract methods appropriately in the derived classes. Put `main` in a separate test class and change it appropriately.

(b) The area of a triangle is equal to one half of its perimeter times the radius of the inscribed circle. If the length of a side of an equilateral triangle is the same as the length of the legs in a right isosceles triangle, which of these triangles can hold a bigger circle inside? ✓

5. (a) Modify the *Chomp* applet to make it played by two human players.

(b) Change the applet further so that it displays different prompts for the two players (e.g., "Your turn, Player 1" and "Your turn, Player 2"). Implement this change by deriving `HumanPlayer1` and `HumanPlayer2` from `HumanPlayer` and redefining the `getPrompt` method in them.

(c) Consider an alternative implementation of different prompts for different human players: instead of using derived classes, accept the name of the player as a parameter in `HumanPlayer`'s constructor; make `getPrompt` return a standard message concatenated with the player's name. Is this implementation more appropriate or less appropriate in an OOP program than the one suggested in Part (b)? Why? ✓

6. Turn the *Chomp* applet into a game for three players: two human players and one computer player.

7. In the *Chomp* applet, a `ComputerPlayer` object creates a `Strategy` for itself. Restructure the applet as follows. Rename the `Strategy` class into `ChompStrategy4by7`. Define a new `Strategy` interface with methods from the former `Strategy` class and state that `ChompStrategy4by7` implements `Strategy`. Rather than creating a `Strategy` object in `ComputerPlayer`, make `BoardGameApplet` pass a `Strategy` object (of the `ChompStrategy4by7` type) to the `ComputerPlayer`'s constructor:

```
public ComputerPlayer(BoardGameApplet applet,
          ChompGame game, ChompBoard board, Strategy s)
```

8. Recall that objects are called ***immutable*** if they cannot be modified after they are created. Explain the relationship between encapsulated classes and classes that define immutable objects. ✓

9. Examine the Java documentation and tell which of the following library classes define immutable objects:

`java.lang.Integer` _______
`java.awt.Point` _______
`java.awt.Color` _______
`java.util.Calendar` _______

10.■ A Java class can be declared `final`, which means that you cannot derive classes from it. For example, `Integer` and `String` are `final` classes. Why? ✓

11. (a) Make the *Chomp* applet scalable. Eliminate the `CELLSIZE` constant in the `ChompBoard` class and obtain the cell's width and height from the current dimensions of the panel when necessary.

(b) Add a constructor to the `ChompBoard` class that sets the row and column dimensions of the board. Make the applet play on a 3 by 6 board. Which properties of the code make this change easy? ✓