Fundamentals of OOP and Data Structures in Java

RICHARD WIENER
University of Colorado, Colorado Springs

LEWIS J. PINSON
University of Colorado, Colorado Springs

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK http://www.cup.cam.ac.uk
40 West 20th Street, New York, NY 10011-4211, USA http://www.cup.org
10 Stamford Road, Oakleigh, Melbourne 3166, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain

First published 2000

Printed in the United States of America

Typeface Century Schoolbook 10/12.5 pt. and ITC Franklin Gothic *System* LaTeX 2_ε [TB]

A catalog record for this book is available from the British Library.

Library of Congress Cataloging in Publication Data

Wiener, Richard, 1941–

Fundamentals of OOP and data structures in Java / Richard Wiener, Lewis Pinson.

p. cm.

ISBN 0-521-66220-6 (hb)

1. Java (Computer program language) 2. Object-oriented programming (Computer science) 3. Data structures (Computer science) I. Pinson, Lewis J. II. Title.

QA76.73.J38 W53 2000
005.1′17 – dc21

99-087328

ISBN 0 521 66220 6 hardback

Contents

1

Cornerstones of OOP

The principles and practices of object-oriented software construction have evolved since the 1960s. Object-oriented programming (OOP) is preoccupied with the manipulation of software objects. OOP is a way of thinking about problem solving and a method of software organization and construction.

The concepts and ideas associated with object-oriented programming originated in Norway in the 1960s. A programming language called Simula developed by Christian Nygaard and his associates at the University of Oslo is considered the first object-oriented language. This language inspired significant thinking and development work at Xerox PARC (Palo Alto Research Center) in the 1970s that eventually led to the simple, rich, and powerful Smalltalk-80 programming language and environment (released in 1980). Smalltalk, perhaps more than any programming language before or after it, laid the foundation for object-oriented thinking and software construction. Smalltalk is considered a "pure" object-oriented language. Actions can be invoked only through objects or classes (a class can be considered an object in Smalltalk). The simple idea of sending messages to objects and using this as the basis for software organization is directly attributable to Smalltalk.

Seminal work on object-oriented programming was done in the mid-1980s in connection with the Eiffel language. Bertrand Meyer in his classic book *Object-Oriented Software Construction* (Prentice-Hall, 1988; Second Edition, 1997) set forth subtle principles associated with OOP that are still viable and alive today. The Eiffel programming language embodies many of these important principles and, like Smalltalk, is considered a pure object-oriented language. Eiffel, with its strong type checking (every object must have a type), is closer in structure to the languages that we use today than to Smalltalk.

OOP was popularized by a hybrid language developed at AT&T Bell Labs in the early 1980s, namely C++. This language evolved from the popular C language. C++ evolved rapidly during the late 1980s. Because of this rapid evolution and the need to retain a C-like syntax and backward compatibility with C, the syntax of C++ has become arcane and complex. The language continued to grow in complexity during the early 1990s before finally becoming standardized and is today considered one of the most complex programming languages ever devised. It is a hybrid language because one can invoke functions without classes or objects. In fact most C programs (C is not an object-oriented language) will compile and run as is using a C++ compiler. The hybrid nature of C++ makes it even more challenging

to use since it allows a mixture of styles of software thinking and organization. In order to use C++ effectively as an object-oriented language, the programmer must impose rigorous constraints and style guidelines. Even with such discipline, the C-like nature of C++ allows programmers to work around basic OOP rules and principles such as encapsulation by using casts and pointers. The preoccupation with pointers in C++ makes the language potentially dangerous for large software projects because of the ever present specter of memory leakage (failure to de-allocate storage for objects that are no longer needed).

The Java programming language invented in the mid 1990s at Sun Microsystems and popularized in the late 1990s may be considered to be a third almost pure object-oriented language. Like Smalltalk and Eiffel, actions may be invoked only on objects and classes (except for a limited number of predefined operators used with primitive types). Also like Smalltalk and Eiffel and unlike C++, Java objects that are no longer needed are disposed of automatically using "garbage collection." The programmer is unburdened from having to devote time and effort to this important concern. It might be argued that the presence of primitive types in Java makes the language impure from an OOP perspective. Although this is strictly true, the basic nature and character of Java is that of a pure object-oriented language and we consider it such.

OOP got its popular start in Portland, Oregon in 1986 at the first Association for Computing Machinery (ACM)-sponsored OOPSLA (object-oriented programming, systems, languages, and applications) conference. At that time the first versions of C++ and Eiffel had recently been released. The three most highly developed languages that were showcased at this first OOPSLA conference were Object Pascal, Objective-C, and Smalltalk. The first release of the Java programming language was ten years away.

During the early days of object-oriented programming, attention was focused on the construction and development of OOP languages. Associated with these newly emerging languages were problem-solving methodologies and notations to support the software analysis and design processes. It was not until the late 1990s that standardization of the object-oriented analysis and design notation occurred with the Unified Modeling Language (UML).

The early application areas of OOP were the construction of libraries to support graphical user interfaces (GUIs), databases, and simulation. These application areas continue to provide fertile soil to support OOP development.

As we enter the twenty-first century, OOP has become widely accepted as a mainstream paradigm for problem solving and software construction. Its use may be found in a large number of application areas including compiler construction, operating system development, numerical software, data structures, communication and network software, as well as many other application areas.

In the following sections we introduce some fundamental concepts of OOP. Many of these concepts are elaborated on in later chapters of Part One.

1.1 Data Abstraction

The oldest cornerstone of OOP is the concept of data abstraction. This concept pre-dates OOP.

Data abstraction associates an underlying data type with a set of operations that may be performed on the data type. It is not necessary for a user of the data type to know how the type is represented (i.e., how the information in the type is stored) but only how the information can be manipulated. As an example, consider the notion of integer in a programming language. An integer is defined by the operations that may be performed on it. These include the binary operations of addition, subtraction, multiplication, and division as well as other well-known operations. A programmer can use an integer type without any knowledge of how it is internally stored or represented inside of the computer system. The internal representation is not accessible to the user.

Data abstraction derives its strength from the separation between the operations that may be performed on the underlying data and the internal representation of these data. If the internal representation of the data should be changed, as long as the operations on these data remain invariant, the software that uses the data remains unaffected.

1.2 Encapsulation

The fusion of underlying data with a set of operations that define the data type is called encapsulation. The internal representation of the data is encapsulated (hidden) but can be manipulated by the specified operations.

1.3 Object

OOP is based on the notion of object. A software object represents an abstraction of some reality. This reality may be a physical object but is more often an idea or concept that may be represented by an internal state. As an example consider a bouncing ball. If we were simulating the motion of the bouncing ball with software we would model the ball as an object and its dynamic state as its height above the surface on which it was bouncing. Here the software object represents a physical object. As a more abstract example consider a cashier line at a supermarket. If we were to represent the line as a software object, its internal state might be the number of customers waiting to check out. Associated with the line would be a set of behavioral rules. For example, the first customer to arrive would be the first customer to be served. The last customer to arrive would be the last to be served.

OOP is also based on the notion of sending messages to objects. Messages can **modify** or **return** information about the internal state of an object. We can send a line object the message *addCustomer*. This causes the internal state of the line to change. We can send a ball object the message *currentHeight*. This returns the ball's height above the surface.

The behavior of an object is codified in a class description. The object is said to be an instance of the **class** that describes its behavior. The class description specifies the internal state of the object and defines the types of messages that may be sent to all its instances. A class *Queue* might be defined to describe the behavior of line objects.

In a program an object is a program variable associated with a class type. The object encapsulates data. An object's "value" or information content is given by its

internal state. This internal state is defined in terms of one or more fields. Each field holds a portion of the information content of the object. As indicated above, an object can receive messages that either change the internal state (i.e., change the value of one or more fields) or return information about the internal state of the object. These messages represent the operations that may be performed on the object.

1.4 Message

Messages are sent to or invoked on objects. In most object-oriented languages the syntax used to accomplish this is given as follows:

```
someObject.someMessage
```

The object precedes the message since it is the recipient of the message. A "dot" operator separates the object from the message. Reading from left to right places the emphasis on the first entity, the object. A message may sometimes have one or more parameters. For example,

```
line.addCustomer(joe)
```

Here the object *line*, an instance of class *Queue*, receives the message *addCustomer* with *joe* as a parameter. The object *joe* is presumed to be an instance of class *Customer*. Since a *Queue* object needs to hold other objects, in this case *Customer* objects, the method *addCustomer* must take a *Customer* object as a parameter.

Messages can be cascaded. Suppose we wish to determine the last name of the first customer in a line. The following expression might be appropriate:

```
line.first.lastName
```

Here *line* is assumed to be an instance of class *Queue*. The message *first* returns a *Customer* object (the lead customer in the *Queue*). The message *lastName* returns the last-name field of this lead customer. We are assuming that class *Queue* has a method *first* that returns the lead customer. We are assuming that class *Customer* has a method *lastName* that returns the last-name field.

1.5 Method

A method is a function or procedure that defines the action associated with a message. It is given as part of a class description. When a message is invoked on an object the details of the operation performed on the object are specified by the corresponding method.

1.6 Class

A class describes the behavior of objects, its instances. The external or "public" view of a class describes the messages that may be sent to instances. Each possible message is defined by a method. These include messages that affect the internal state of the object and messages that return information about this internal state. The internal or "private" view of a class describes the fields that hold the information content of instances. In addition to fields, the private view of a class may define private methods that are used to support public methods but cannot be invoked outside of the class.

The user of a class is concerned only with the public or external view of the class. The producer of a class is concerned with the public and private view. Chapter 3 describes the construction of Java classes in detail.

Let us consider a simple example to illustrate some of the ideas presented above. Consider class *Point*. The "actions" that one may take on a point object include:

1. setX(xValue)
2. setY(yValue)
3. x( )
4. y( )
5. distanceFromOrigin( )

Note: We prefer to use a noun phrase rather than a verb phrase for a message that returns internal information about an object. This is justified in Chapter 3.

The five external actions that have been defined for class *Point* are called accessor methods. They allow us to set and get the values of the *x* and *y* coordinates of a point object and get the distance of the point to the origin. The first two accessors, *setX* and *setY*, require a parameter.

Listing 1.1 presents a full Java class definition for *Point*.

Listing 1.1 Class *Point*

```
/** Details of class Point
*/
public class Point {

  // Fields
  private double x;          // x coordinate
  private double y;          // y coordinate
  private double distance;   // length of point

  // Methods
  public void setX (double x) {
    this.x = x;
    updateDistance();
  }
```

```
  public void setY (double y) {
    this.y = y;
    updateDistance();
  }

  public double x () {
    return x;
  }

  public double y () {
    return y;
  }

  public double distanceFromOrigin () {
    return distance;
  }

  // Internal methods
  private void updateDistance () {
    distance = Math.sqrt(x*x + y*y);
  }
}
```

As will be our practice throughout this book, class names and public features shall be presented in boldface type. This highlights the external view of the class.

The three fields are designated with the *private* access specifier. This encapsulates the information content. This content can be modified using only the methods *setX* and *setY*. When either of these methods are invoked, the *distance* field is automatically updated and is available to the user with the method *distanceFromOrigin*.

If the fields information were not encapsulated, a user of class *Point* could directly modify the *x* coordinate or *y* coordinate and forget to update the *distance* field. Of course this quantity could be computed each time it is needed instead of updated each time the *x* or *y* coordinate of the point object is modified. In general, information about an object can be obtained either through storage (as in Listing 1.1) or through computation.

1.7 Inheritance

Another cornerstone of OOP is inheritance. Inspired from biological modeling, inheritance allows new classes to be constructed that inherit characteristics (fields and methods) from ancestor classes while typically introducing more specialized characteristics, new fields, or methods. A subclass is logically considered to be a specialized version or extension of its parent and by inference its ancestor classes.

In Java, every object before its creation must be declared to be of a given type, typically the class that the object is an instance of. This sometimes changes in the presence of inheritance because of an important principal, the principal of **polymorphic substitution**. This principal states that wherever an object of

a given type is needed in an expression, it may be substituted for by an object that is a descendent of the given type. Although it may be difficult upon first contemplation to fully appreciate the power and implications of this principal, it is one of the most important and fundamental concepts in object-oriented software construction.

Since polymorphic substitution states that a descendent class object may be used in place of its ancestor object, the descendent class object must be considered to be of the ancestor type. This makes sense. Consider a high-level class *Vehicle* that encapsulates the properties of all vehicle objects. Now consider a more specific class *Bicycle* with its unique behavior that represents a specialization of class *Vehicle*. At the least, the methods of class *Vehicle* can be interpreted by class *Bicycle*. Thus it makes sense that a *Bicycle* object can be used in place of a *Vehicle* object (it will know how to respond to *Vehicle* messages). Clearly the opposite is not true. A *Vehicle* object cannot be used in place of a *Bicycle* object since it will not necessarily be able to respond to the specialized methods of class *Bicycle*. A bicycle is a vehicle.

In general a subclass should logically satisfy the constraint that it can also be considered to be of the parent class type. This is most fundamental. Regardless of what other purpose one may wish to achieve in using inheritance, this logical constraint should be satisfied. A *TeddyBear* class should not be construed to be a subclass of *Refrigerator*. This constraint is often referred to as the "is a" or "is kind of" relationship between subclass and parent. The subclass should satisfy the logical condition that it "is kind of" an instance of its parent. This logical constraint is sometimes referred to as **behavioral inheritance**. The subclass enjoys the same behavioral characteristics as its parent in addition to the more specialized behavior that distinguishes the subclass from the parent.

Another use of inheritance (some might argue "misuse") is **implementation inheritance**. Here the only purpose of creating a parent class is to factor code that is needed by other subclasses. Since ancestor methods are generally inherited by descendent classes (unless they are redefined in one or more descendent classes), the descendent class can consider the ancestor method to be one of its own. Although implementation inheritance makes it possible to reuse code, if the logical constraints of behavioral inheritance (the "is kind of" relationship) are not satisfied, the software architecture may become muddled and difficult to maintain. Often implementation inheritance flows as a natural and useful byproduct from behavioral inheritance.

It is not the goal of this introductory section on inheritance to present all the details of inheritance in Java. This is the goal of Chapter 4.

To clarify the above ideas, an example that illustrates the use of inheritance is presented in this section without extensive detail. Consider a *SpecializedPoint* class that extends the *Point* class presented in Listing 1.1.

SpecializedPoint Class

Suppose we wish to create a point class in which the x and y coordinates are constrained to be positive. That is, we wish our *SpecializedPoint* objects to be located in the first quadrant of the complex plane.

First we need to make small modifications to class *Point*, given in Listing 1.1. The Modified Point class is given in Listing 1.2.

Listing 1.2 Modified *Point* Class

```
/** Modified Point class
*/
public class Point {

  // Fields
  protected double x;
  protected double y;
  protected double distance;

  // Constructor
  Point () {
    setX(0);
    setY(0);
  }

  Point (double x, double y) {
    setX(x);
    setY(y);
  }

  // Methods
  public void setX (double x) {
    this.x = x;
    updateDistance();
  }

  public void setY (double y) {
    this.y = y;
    updateDistance();
  }

  public double x () {
    return x;
  }

  public double y () {
    return y;
  }

  public double distanceFromOrigin () {
    return distance;
  }

  public String toString() {
    return "<" + x + "," + y + ">" ;
  }

  // Internal methods
  protected void updateDistance () {
    distance = Math.sqrt(x*x + y*y);
  }
}
```

Brief Explanation of Listing 1.2

The access modifiers for the three fields are changed from *private* to *protected*. This allows all subclasses to inherit these fields without changing the accessibility of the fields in outside classes – encapsulation of internal information is preserved while providing access to all descendent classes. If the fields were kept as *private* as in Listing 1.1, the subclass *SpecializedPoint* would effectively have no fields directly accessible. This violates the concept of behavioral inheritance in which a subclass is a kind of its parent. In order for a *SpecializedPoint* object to be of type *Point*, it must retain the three internal fields (i.e., have an *x* value, a *y* value, and a *distance* value).

Two constructors are added to the class definition. As shall be explained further in Chapter 3, a constructor is a function that always bears the name of its class and is used to produce new instances of the given class. In Listing 1.1 no constructor was provided. In this case Java provides a default constructor that initializes all fields to zero (if they are scalar fields as in Listing 1.1) and null if the fields are objects (reference types). This shall be explained in Chapter 2. Notice that the field *distance* is automatically updated based on the values used in the two constructors for fields *x* and *y* by invoking the *setX* and *setY* commands. This is an example of a good object-oriented design principle in action. A consistent set of steps is followed for setting the value of *distance*.

The method *toString()* is useful because it is automatically invoked whenever a string representation of a *Point* is desired. This is useful when doing input/output (I/O) as in the expression *System.out.println("pt = " + pt)*, where *pt* is a *Point* object. Here the "+" or concatenation operator causes the *toString()* method to be automatically invoked, converting the *pt* object to a string object. Class *String* and its important properties are discussed in Chapter 2.

Listing 1.3 Class *SpecializedPoint*

```
/** Details of a specialized Point class that extends class Point
*/
public class SpecializedPoint extends Point {

  // Constructor
  SpecializedPoint () {
    super(); // Invokes the parent class constructor
  }

  SpecializedPoint (double x, double y) {
    super(x, y);
  }

  // Methods
  public void setX (double x) { // Redefined method
    if (x < 0)
      throw new UnsupportedOperationException(
        "x must be greater than 0" );
```

```
    else {
      this.x = x;
      updateDistance();
    }
  }

  public void setY (double y) { // Redefined method
    if (y < 0)
      throw new UnsupportedOperationException(
        "y must be greater than 0" );
    else {
      this.y = y;
      updateDistance();
    }
  }
}
```

Brief Explanation of Listing 1.3

The key word **extends** establishes that class *SpecializedPoint* is a subclass of class *Point*. The methods *setX* and *setY* are redefined. Code is written to ensure that the values of the parameters *x* and *y* are non-negative. If this is violated an *UnsupportedOperationException* is generated. It is the responsibility of the caller (the block of code that invokes the constructor or *setX* or *setY*) to ensure that *x* and *y* are non-negative. This shall be explained in more detail in Chapter 4. All other methods from class *Point* are inherited in class *SpecializedPoint* and may be used as is.

Listing 1.4 presents a small test class that exercises some of the methods of classes *Point* and *SpecializedPoint*.

Listing 1.4 Class *PointTest*

```
/** A test program that exercises classes Point and SpecializedPoint
*/
public class PointTest {

  public static void main(String[] args) {
    Point p = new Point (-3, -4);
    SpecializedPoint sp1 = new SpecializedPoint ();
    SpecializedPoint sp2 = new SpecializedPoint ();

    sp1.setX(3);
    sp1.setY(4);
    System.out.println("sp1 = " + sp1);
```

```
    sp2.setX(-3); // Should cause an exception to be generated
    sp2.setY(4);
    System.out.println("sp1 = " + sp1);
  }
}
```

Brief Explanation of Listing 1.4

The code works fine and predictably until the method *setX* with parameter −3 is invoked on the *SpecializedPoint* object *sp2*. This causes the *UnsupportedOperationException* to be generated. Exceptions are discussed in Chapter 7. The program output is:

```
sp1 = <3.0,4.0>
Exception in thread "main" java.lang.
  UnsupportedOperationException: x
and y must be greater than 0
at SpecializedPoint.setX(SpecializedPoint.java:24)
at PointTest.main(PointTest.java:15)
```

1.8 Late Binding Polymorphism

Late binding is closely related to inheritance. Since methods may be redefined in descendent classes (like methods *setX* and *setY* in Listing 1.3), it is common for several specialized versions of a given method to exist in a class hierarchy, each with the same method signature (same function name, same return type, and same set of parameters). The runtime system is able to bind the correct version of a method to an object based on the specific type of the object. This late binding is an important characteristic of object-oriented systems. The word *polymorphism* derives from "many forms." In the case of OOP, many forms refer to the different versions of a specific method defined in different subclasses. An example that illustrates late binding is presented in the next section.

1.9 Abstract Classes

A class in which one or more methods are not implemented is defined as an **abstract class**. A class in which all methods are implemented is a **concrete class**. Abstract classes are often defined near the top of a hierarchical structure of classes. Undefined or abstract methods are used in an abstract class to establish required behavior in any descendent concrete class. **An instance of an abstract class cannot be created**.

Since some methods in an abstract class may be fully implemented, the benefit of implementation inheritance can be realized along with behavior inheritance.

We illustrate the concepts of abstract class and late binding by considering skeletal portions of a small hierarchy of *Vehicle* classes. We employ UML notation (see Appendix A) to represent the *Vehicle* hierarchy, shown in Figure 1.1.

Class *Vehicle* is shown as the root class in the hierarchy. Class *Vehicle* is abstract. This implies that no instances of *Vehicle* can be constructed. The fields

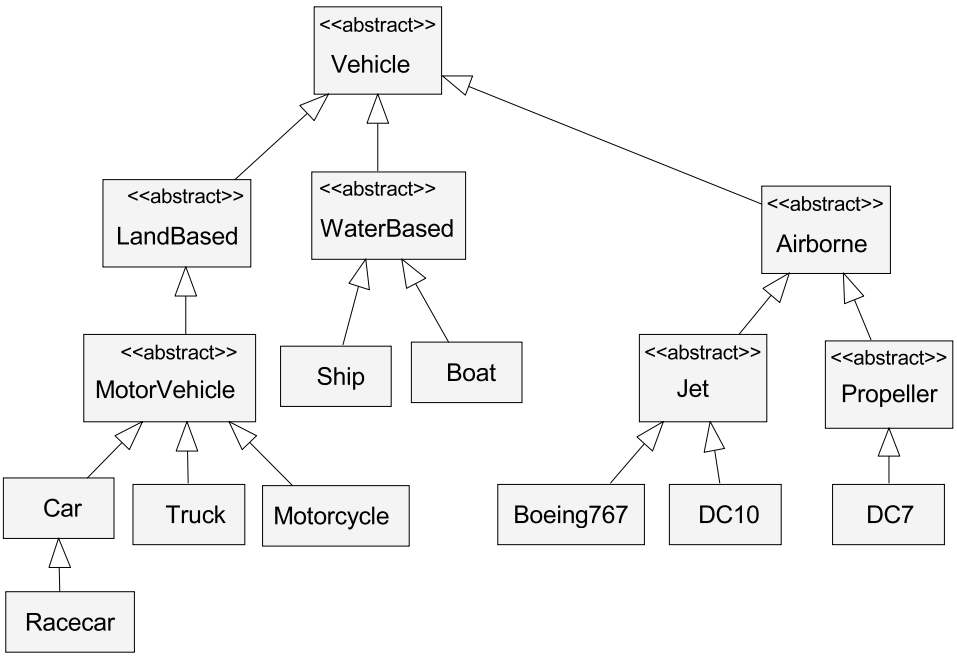

Figure 1.1. UML diagram of *Vehicle* class hierarchy.

of *Vehicle*, if any, and all its methods, are inherited by every class in Figure 1.1. The fields and methods of *Vehicle* describe behavior and state common to all subclasses of *Vehicle* (all vehicle types). The three immediate subclasses of *Vehicle* – *LandBased*, *WaterBased*, and *Airborne* – are also abstract classes (no instances can be constructed). More specialized characteristics (fields and methods) for each of these vehicle types are defined. Under *LandBased* is the abstract class *MotorVehicle*. There are three concrete subclasses of *MotorVehicle*: *Car*, *Truck*, and *Motorcycle*. Each of these inherits the fields and methods defined in the abstract classes *MotorVehicle*, *Landbased*, and *Vehicle* as well as introducing more specialized behavior. Class *Racecar* is shown as a subclass of *Car*. It inherits the fields and methods of *Car* as well as introducing its own specialized behavior (additional fields or methods).

What is the type associated with an instance of class *Racecar*? The answer: *Racecar*, *Car*, *MotorVehicle*, *LandBased*, *Vehicle*, and *Object* (all classes inherit from *Object*). Yes, a *Racecar* instance is of six distinct types. What does this mean in practice?

Consider the following variable declaration:

```
Vehicle rc = new Racecar();
```

Here an object *rc* of formal type *Vehicle* is constructed of actual type *Racecar*. The principle of polymorphic substitution discussed in Section 1.7 is utilized. This allows an object of some descendent type to be substituted for the ancestor type.