Contents

Introduction

0.1 The Subject of this Course

This is, first and foremost, a course about object-orientation: the course is structured around the principal concepts of object-orientation. Our main hope for this course is that, at the end of it, you will have a good understanding of the main principles and techniques of object-orientation and that you will understand the reasons for, and uses of, these principles and techniques.

Why are we so concerned that you learn object-oriented principles? To understand this you need to have an idea of the crisis that has faced, and still continues to face, the software industry. Real commercial software systems increase in size and complexity year on year. It is impossible to design a modern software system all in one go; it would be impossible to understand an unstructured system well enough to maintain it by, for example, making changes when an error is found or when supporting technologies change. Object-orientation has grown as a response to this problem.

Object-orientation is a portfolio of related techniques and concepts that help you to handle the complexity of software systems. The techniques and concepts encourage well-structured systems with reusable and reused components. The idea of reusability is central to modern program development: most development involves the reuse of code you have written before or code that has been supplied for you by such vendors as Sun Microsystems. To be able to reuse this code effectively entails an understanding of object-oriented principles.

One of the themes stressed throughout the course is the use of abstraction to help control the design of complicated systems. Object-orientated languages allow for greater possibilities for abstraction than procedural languages do. These techniques can be seen as generalisations of the notion of abstract data type. The abstractions studied in this course take the form of abstract views of procedures, abstract data types (which are abstractions of packages of data and procedures), design patterns (which are abstractions of techniques for solving problems), and frameworks (which are abstractions of solutions to large problems).

This is, secondly, a course about Java. We have chosen Java as the language on which to base this course, and much of the whole degree programme for several reasons:

a. Java is a purer and cleaner implementation of object-oriented concepts than C++

b. Java has more powerful facilities in the form of libraries of reusing code than any other language

c. Java is platform independent

d. Java is more commercially relevant than any other modern computer language, with the possible exception of C++.

The application domain for most of this course is graphical programming. There are several reasons we believe graphical programming is appropriate to this course and useful for you to study at this point in the programme. Some of these are:

a. Most important object-oriented principles come up naturally in graphical programming

b. Almost all modern computing systems of any complexity include a graphical interface

c. Java's graphical library is particular rich

d. Graphical programming is fun and rewarding.

Modern computing systems generally are meant to be accessed by a multitude of users, some of whom will not be as comfortable with Java code as you. Most users, especially uninitiated ones, find it easier and more pleasant to interact with a system graphically, using buttons etc, than textually by typing into the system. Java has been designed with this in mind and has provided developers with a great deal of pre-defined classes to re-deploy in their applications. And the way that the Java graphics classes are put together follows object-oriented structuring principles. Moreover, graphical applications are very fruitfully seen as semi-independent objects on a screen. And, lastly, it is a particularly satisfying feeling to write a program and see a picture animated as a consequence.

0.2 How To Study This Course

This is an intermediate programming course. It is intended for students with some previous programming experience; ideally you will have Java experience up to the level of the first

year course in this programme. While only a very small amount of specific Java knowledge will be drawn upon without explanation, you are likely to find this course very difficult if this is your first exposure to Java and, particularly so, if this is your first experience of an object-oriented language of any sort.

Unlike other programming courses in this programme, this course follows a textbook very closely. The subject guide will repeatedly refer you to the textbook to read particular sections and to do exercises. It is very important that you do as many of the exercises as you can. Some exercises are suggested throughout the subject guide; you should do at least those, and preferably more. It is as impossible to learn to program without doing programming exercises as it would be to learn to ride a bicycle purely by reading a book about it. And, just as in learning to ride a bicycle, you should expect to fall sometimes. Nobody gets either bicycling or programming right straight off. The difference is that while most people eventually learn to ride a bicycle well enough to stay upright, learning to program is a continuous, never completed process: there will always be new things to make you fall. That can be frustrating, but it can also make learning to program a source of continuous small satisfactions. We very much hope that, at the end of the session, you will find this a fitting description of your experience of the course.

The purpose of the subject guide is to guide you through the material, adding explanations of topics in the text, adding examples, and occasional hints and solutions to exercises.

You will notice a substantial shift in the use of the subject guide in the middle of the course: there is quite a lot written in the guide to accompany the first three chapters of the book and comparatively little afterwards. There are two reasons for this. First, the first three chapters are introductory and you may feel the need for extra support at that stage. By the time you get to Chapter Four, you should be comfortable enough in the subject and in Laszlo to go through the book with only a small amount of guidance. And, secondly, the book is more detailed and slower in its treatment of material from Chapter Four on.

As well as the subject guide and the text, you should have a CD of additional materials. The CD contains the code that is referred to in the texts, and the programming environment—*BlueJ*— that you will be using throughout the course. Instructions for setting up *BlueJ* and your source code for the course are given in the Appendix.

We suggest you devote your time to the course in roughly these proportions:

Chapter One	10%
Chapter Two	15%
Chapter Three	10%
Chapter Four	15%
Chapter Five	15%
Chapter Six	20%
Chapter Seven	15%

0.3 Topics

The major topics of the course are given below:

- Classes and Objects
- Methods
- Data Abstraction
- The Graphics2D API (Application Program Interface)
- Composition
- Inheritance
- Design Patterns
- Building Graphical User Interfaces

0.4 Assessment

Examination

Important: the information and advice given in the following section is based on the examination structure used at the time this guide was written. However, the University can alter the format, style or requirements of an examination paper without notice. Because of this, we strongly advise you to check the instructions on the paper you actually sit.

There will be one three-hour unseen written examination paper for 2910212. This will contain two main sections, the first section corresponding to this guide. You will be asked to answer three questions from each of these sections.

The first section of the examination will not differ radically from the sample section A of the examination paper included in Appendix 1 of this guide. A sample of section B of the paper is included in volume 2 of the subject guide.

Coursework

Two coursework assignments (issued separately each year) will be set for students to practice their program design and development skills using theoretical knowledge gained from the course.

The approach to solving a given problem by implementing an algorithm differs from person to person and from problem to problem, however it generally includes the following stages of work:

1. Analysis of the given problem
2. Decomposition into sub-problems
3. Derivation of a general plan to solve the sub-problems
4. Development of the program
5. Tests and possibly formal proofs of the correctness of the program
6. Comments or review of the limitations of the program.

At the end, a full document should be written which includes a section for each of the above stages of work.

0.5 Books

Course text

The main text for this course is:

> Laszlo, Michael *Object-Oriented Programming featuring Graphical Applications in Java*. (Boston; London: Addison Wesley, 2003) [ISBN 0201726270].

It is imperative that you have access to this book. Throughout the course, you will be referred to it for reading and exercises. It will be referred to variously as "Laszlo" and "the course text" in this guide. Note that any undirected reference to a book chapter or exercise is, in fact, a reference to Laszlo.

For most of the course, the structure and content of the book determines the structure and content of the guide. Indeed, in later chapters of the subject guide, care has been taken to try to synchronise the section numbers of the guide with that of the course text. For example, where possible, the section numbered 3.2 in the guide refers to the section numbered 3.2 in Laszlo.

Other books

There are three other books that we have drawn upon while writing these notes. You may well find these books useful as well:

> Budd, Timothy *Understanding Object-Oriented Programming With Java.* (Reading, Mass: Addison Wesley, 2000) [ISBN 0201612739]. Updated Edition (New Java 2 Coverage).
>
> Barnes, David J. and Michael Kölling*, Objects First with Java: A Practical Introduction using BlueJ*. (Prentice Hall / Pearson Education, 2003) [ISBN 0-13-044929-6].
>
> Knudsen, Jonathan *Java 2D Graphics*. (Sebastopol, Calif.: O'Reilly and Associates) [ISBN 1-56592-484-3].

Another book that we recommend is:

> Eckel, Bruce *Thinking in Java*. (Upper Saddle River, NJ: Prentice Hall, 2000) second edition [ISBN 0130273635]. http://www.bruceeckel.com/

This book is freely downloadable from the Internet. To find a website near you, consult:

www.mindview.net

Almost any Java, object-orientation, or design pattern book would be useful. Some of these are listed in a booklist in section 0.7 below.

0.6 Compiling and Running Java Programs

The subject guide is written under the assumption that you will be using *BlueJ* to compile and run your programs. If you do not do so, you will find it difficult to follow some of the writing. The *BlueJ* provided on the CD runs under several operating systems, including, Windows, Linux and MacOS. If you have any problems with *BlueJ*, you should consult the tutorial that is included on the course CD.

0.7 What you will have achieved when you have completed this subject guide

Having completed this subject guide you will understand how object-orientation is used as a way of developing computer applications that are robust and reusable. You should also know how to deploy object-oriented development techniques in Java, and how to write Java applications that have substantial graphical facets. The learning outcomes at the end of each chapter summarise the examinable topics.

0.8 Some suggested Books

Deitel and Deitel *Java: How to Program*. (Prentice Hall International, 2000) third edition [ISBN 0-13-012507-5].

Flanagan, David *Java in a Nutshell*. (O'Reilly, 1999) third edition [ISBN 1-56592-487-8].

Hubbard, John R. *Schaums: Outlines Programming with Java*. (McGrawHill, 1998).

Jia, Xiapong *Object-Oriented Software Development Using Java: Principles, Patterns, and Frameworks*. (Addison Wesley, 2000).

Lambert, Kenneth A. and Martin Osborne *Java: A Framework for Programming and Problem Solving*. (Brookes Cole, 2002).

Liang, Y. Daniel *Introduction to Java Programming with JBuilder 4.* (Prentice Hall, 2001) second edition [ISBN 0-13-033364-6].

Liang, Y. Daniel *Rapid Prototyping*. (Prentice Hall, 2001) [ISBN 0-13-033364-6].

Richter, Charles *Designing Flexible Object-Oriented Systems with UML*. (Indianapolis, IN: Macmillan Technical Publishing, c.1999) [ISBN 1578700981].

Vlissides, John *Pattern hatching: design patterns applied*. (Harlow: Addison Wesley, 1998) [ISBN 0201432935].

1 Objects and Classes

1.1 Introduction to the Chapter

This chapter serves as a quick introduction to object orientation. It is meant to remind you of things you have probably learned in previous programming courses and to introduce you to concepts that will be expanded throughout this course. The relation between the subject guide and the course text is different in this chapter than it is for the rest of the course. Chapter 1 of the subject guide develops independently of the book.

1.2 Learning Objectives

After working through this chapter, you should be able to:

- state the rudiments of building object-oriented systems
- explain the relationship between objects and classes
- use *BlueJ* to write, edit, compile and run simple Java programs.

You should also have the beginnings of an understanding of:

- how graphical programs fit together in Java
- the uses of frameworks in Java.

1.3 Objects and Classes

An object-oriented program, when it is running, consists of several objects, performing computations and sending messages to each other. That said, the first perplexing thing to note is that when you write an object-oriented program your main concern is not in defining objects.

To understand this conundrum, and to begin to get a feeling for object-orientation, imagine that you are a software developer who has been asked to develop a computer game called

Cops and Robbers. When the game is running there will be a few cops chasing a few robbers around a virtual city on a computer screen.

Each Cop and each Robber will be an object in the system. When we are writing the software for this system, we do not define the behaviour of each Cop and each Robber separately. There are many reasons for this, among these are:

i. Different Cops will have identical behaviour and it would be wasteful to keep redefining it.

ii. We may not want to say in advance how many Cops will be in the game.

iii. Defining objects directly will not lead to reuse and to a well structured system. This is important and will become increasing clear throughout the course.

So, instead of directly defining the several Cops as objects, we define an abstraction of them all. This abstraction is called a **Class**. The class in this example would be called *Cop*,[1] and it does not represent any particular Cop but, instead, represents all Cops, and it contains definitions of all the behaviour that is common to all Cops. Here the behaviour includes two types of things:

i. *Attributes*, *datafields*, or simply *fields*: this is the data associated with the class. In this case, it may include information about position, size, etc. Sometimes these aren't given values in the class definition but are left to be given values in each object.

ii. *Methods*: these are the actions or procedures that all members of the class (in this case, all Cops) can perform. In this case, that might include *running*, *drawing itself on a screen*, and *hitting a Robber over the head with a stick*.

A very important kind of method that we will need to define in the class is one that will allow us to make new objects of the class. Methods like this are called **Constructors**. Constructors are recipes for making objects: they give

---

[1] Notice that the name is a singular noun: this is the usual convention for naming classes unless the class represents a collection of objects.

the objects the attributes they need. If we didn't have constructors there would be no way of making objects of that class.

The other thing about the system when running is that the objects will send each other messages. This idea of messages is broader than it sounds, and in this example, might include a cop sending a message to a robber by hitting him over the head with a stick. The Robber Class must then be defined in such a way that it knows how to respond to this message: by perhaps losing consciousness for some time.

One more thing to note about this brief example: there is likely to be quite a bit of behaviour that Cops and Robbers both have in common. They both have positions; they both run. We can avoid defining these behaviours twice by defining a new class, which we may call *Actor*, in which all of these common behaviours are defined. Then in the definitions of both *Cop* and *Robber* we say that these classes are **subclasses** or **child classes** of the Actor class. (This is done using the keyword **extends**). In this situation, we also say that *Cop* and *Robber* both **inherit** from *Actor*. There are great advantages of doing things this way beyond the saving of duplication of typing. It means that both the development of the system and its later maintenance will be made easier and safer. If we want *Cop* and *Robber* to always run in the same manner, for example, but we later decide to change the way that is, we would be much less likely to make a mistake if the change only needed to be made in one place. We may otherwise, for example, change *Cop* but forget to change *Robber*.

1.4 The Rest of this Chapter

The discussion of classes and objects given above was very quick and fairly abstract. The discussion in Chapter 1 of the course text is more leisurely but is, if anything, slightly more abstract. In the rest of this chapter, we will try to make all of this more concrete by devising a simple (though not trivial) example step-by-step.

Activity

Before continuing with the subject guide, you should read Chapter 1 of Laszlo.

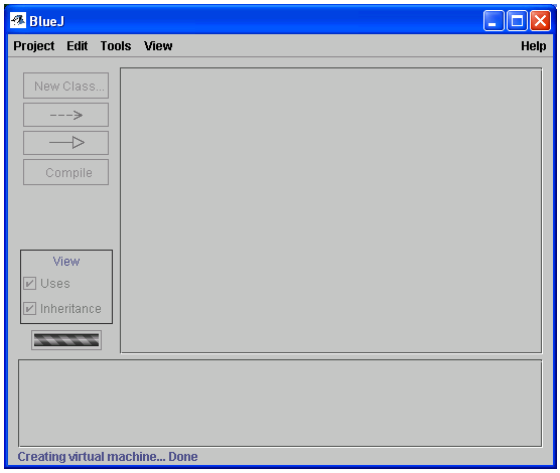
1.5 First Class: Ball

In this section we will start to use *BlueJ* and see an example that will help elucidate the distinction between objects and classes. We are assuming that you have set up *BlueJ* and the course source code as in Appendix A; if you have not already done so, do it now.

In the next few sections, we will be moving towards a program that defines and displays a ball bouncing around a screen. This idea is adapted from Chapter 5 in Budd's book, and if you look there you can see a substantially different implementation and further development of the ideas through Chapters 6 and 7.

The first thing we will do is to implement a very simple ball. The ball will have a position a size and a colour. There will be methods for changing its colour, changing its size, and moving it.

7. Start *BlueJ*. One way to do this is to double-click on the *BlueJ* icon on your desktop. You should get a window that looks like this:



8. In the upper left you will see a pull down menu marked ***Project***. Go to that menu and choose ***NewProject…,*** which is one down in the menu.

9. A new window will open in front of the first one. In that window, go to the Look in box at the top and navigate until you get to C:*BlueJ*\212 Source Code\Chapter 1. Type into the filename box: chapter1Project, and then click on the create button on the right of the window.

10. Go to the Edit pull down menu, which is next to the Project one. From the menu choose Add Class from File...

11. Navigate to C:*BlueJ*\212 Source Code\Chapter 1. Click on Ball.java and click on the Add button on the bottom right of the window.

12. That window should disappear, leaving you only with the original *BlueJ* window. The window should now contain a rectangle that represents the class Ball. What is happening in that window is that a class diagram[2] for our application (or project) is being developed. Notice that the rectangle that represents the Ball class is striped. This indicates that it is not compiled. You should now compile it by pressing the compile button on the left side of the window. The stripes should disappear.

13. Now let's briefly look at the code for this class.

```
1.  public class Ball
    {
2.  protected int diameter;
3.  protected int xPosition;
4.  protected int yPosition;
5.  protected String colour;

6.  public Ball()
        {
        a.  diameter = 30;
        b.  xPosition = 100;
        c.  yPosition = 100;
        d.  colour = "red";
        }

7.  public void blowUp ()
        {
        a.  diameter = diameter + 10;
        }

8.  public void move (int dx, int dy)
        {
        a.  xPosition = xPosition + dx;
        b.  yPosition = yPosition + dy;
        }

9.  public int getX ()
        {
        a.  return xPosition;
        }
 }
```

[2] This is a UML diagram used for describing object-oriented systems. We will learn more about UML throughout this course.

It is very important to note that the numbers (and letters) at the front of the lines are not part of the program, but are put in only as tags to allow us easily to refer to parts of the program. Therefore you will not find them in the code on your computer. Moreover, there is indentation, as in Structured Programming, to indicate which lines are part of which blocks of code. For example, line b under line 8—which we will refer to as 8b—is part of the method whose heading is line 8. Moreover, if line 8b had been a For statement, that is the beginning of a For loop, the lines within the body of the loop would be intended one level deeper and would be labelled with such numbers as 8.a.ii.

Line1 is the heading of the Class definition: it says that we are now about to define a class called Ball and that it will be publicly accessible. The fact that it is public is important if it is to be used by objects of other classes.

Lines 2-5 define what attributes every ball (that is an object of class Ball, or an **instance** of the Ball) will have to have. We could have given them default values there, but chose not to.

Line 6 heads the definition of a constructor. Remember that a constructor is a recipe for making instances (or objects) of the class. What you need to define a ball is to give values of each of the attributes. Lines 6a-d do just that. Notice that these values are all default values in the sense that all balls made with this constructor will have the same values. We will see later that this is not the only way to define these things. You can always spot a constructor in the source code for a class: they have the same name as the class (here, Ball) and they have no place for an output type (that is you go straight from the access modifier—in this case public—to the name of the method.

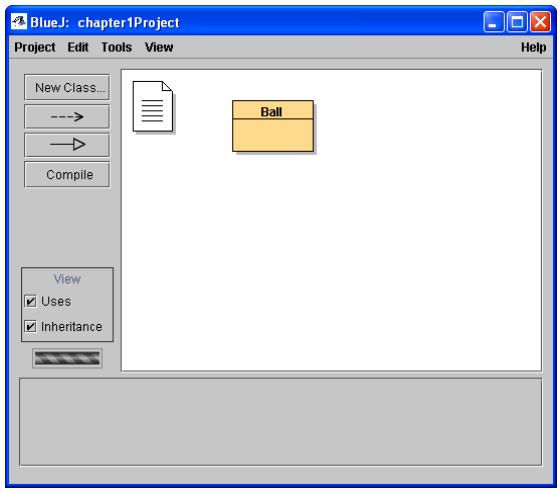
Line 7 begins the definition of a simple method for changing the value of one of the attributes of an object: in this case its size. You should notice the keyword void between public and blowUp. This means that the method has no output, and unlike the case with the constructor, we need to say that explicitly. It has no output: its only effect is to change the value of an attribute of ball. This is known as an **effect**. The distinctions between outputs and effects will be treated in Chapter 2. This method also has no input, which can be seen by the fact that there is nothing written between the brackets after the name of the method.

Line 8 begins a slightly more complicated method. This method moves the Ball a set amount. It is very similar to blowUp, and has a parallel effect, the differences are that, first, two attributes change instead of one and, secondly, that there are inputs to the method that affect how far the ball will move.

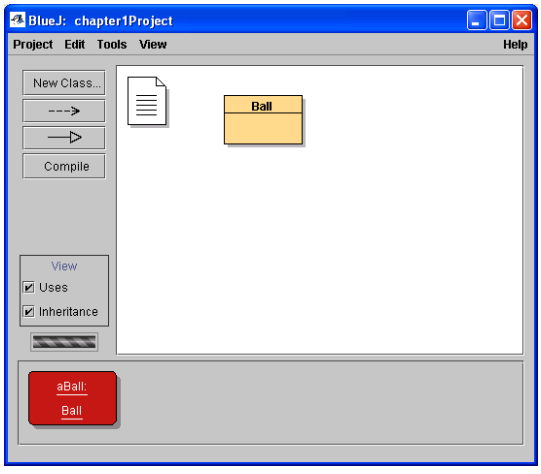
Line 9 begins a new method. This one has an output of type int. This has no effect: the object is in no way changed by outputting this value. This method is an example of a very common kind of method in Java, called a getter. Getters allow safe access to values of an object's fields. Because xPosition is protected, another object cannot refer to it directly, but needs to use the getter.

1.6 Back to BlueJ

Having compiled Ball, your *BlueJ* window should look like the following picture:



1. **Make a Ball Object**: Now, right click on the Ball class. You will be presented with several options including new Ball(). This is an invitation to apply the constructor that is part of the class definition. Notice that you are not invited to apply any of the other methods from the class here. Choose new Ball (), and in the box that is presented to you change the name in the *name of instance* box to aBall. You have now defined an object of class Ball. It should have appeared as a red rectangle on the bottom of the window of your screen. The screen should look something like the following picture:



2. **Inspect the State of your object**: Now right click on the red rectangle that represents **aBall**. You will be presented with several options, from which you should choose Inspect. This will give you a window that displays the values of all the attributes of aBall.

3. **Alter the State**: Leaving that window open, right click on the red rectangle again. This time choose blowUp and notice what happens to the attributes in the Inspection window. Unlike the constructor, blowUp belongs to the object or instance. Such methods are called **instance methods** or **object methods**, as opposed to **class methods**, which are methods that belong to the class. Note that the same distinction is made between instance fields and class fields.

4. Click the red rectangle again and click on move. This time you will be asked to supply values for the parameters. Note the results in the Inspect Window.

Things learned in the exercise:

- The constructor always belongs to the class. It is used to make objects.
- Other methods are used to change the value of attributes and to output values.
- By default, methods belong to individual objects, rather than classes.

1.7 Static Methods and Data Fields

We saw in the last section, that while constructors belong to classes, other methods and fields tend to belong to objects. However, sometimes it is more appropriate for a particular method or field to belong to the Class. These methods and fields are said to be **class methods** and **class fields** or **static methods** and **static fields**.

What differences would it make to a method or field if it were static? There are two principal differences. If a method or field is static then:

- The values will be the same in every instance of the class.
- It can be referred to without creating any objects.

The second point explains the need to have constructors as static. If they were not, they could not be referred to without creating an object, and since they are the way of making objects, there would be no way of ever creating an object at all.

There is a second very important special case of a static method: the main method. We will learn more about the main method in Chapter 2. The point of it is, as you probably already know, that when a program is run automatically, the main method is immediately called. Now, since this is the first thing that happens, it is called before any objects are created, so it must be called from the class. That is why we have the keyword static in the heading of the main method in every application.

To explore the effects of the keyword static, we have provided a new version of the Ball class with the keyword **static** inserted in some places. The important part of the code follows:

```
1.  protected static int diameter;
2.  protected int xPosition;
3.  protected static int yPosition;
……..
4.  public static void blowUp ()
        a.      {
        b.      diameter = diameter + 10;
        c.      }
5.  public void move (int dx, int dy)
        a.      {
        b.      xPosition = xPosition + dx;
        c.      yPosition = yPosition + dy;
        d.      }
}
```

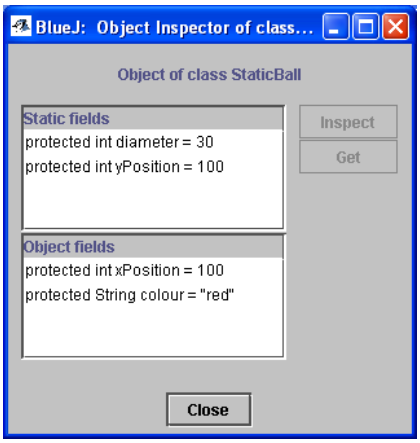
Notice that ***diameter*** and ***yPosition*** are now static, while ***xPosition*** is not. Notice, too, that blowUp is static but move is not.

Now to explore the differences between instance methods and fields and class methods and fields, open *BlueJ* again if it is not still open and add the class StaticBall.java to your Project, following the instructions in Section 1.5 above.

1. Compile the new class and make **two** objects following the instructions in Section 1.6.

2. Using the Inspect method in each of your two objects (by right clicking on each red rectangle), open an Inspect window for each. Lay everything out on the screen so that you can see both inspect windows and the *BlueJ* window.

Your inspect windows should both look like this:



3. Notice that the window clearly separates the Static from the Object Fields.

4. ***Applying a static method***: Right click on the object icon in the *BlueJ* window. Besides using the constructor, you are enabled to apply blowUp. This is because blowUp was declared to be static in the class definition. Now, choose blowUp and notice the effect on both of the inspector windows. Did you notice that the diameter changed in both? A static method affects all instances of the class.

5. ***Applying a non-static method***: Now right click on one of your objects. You will be able to apply move. Choose move, and type non-zero numbers in for both parameter values in the dialogue box that appears. Look closely at the effect in both of the Inspect windows. You should find that only one of the xPositions is affected by the method, but both yPositions are affected. This is because the yPositions are declared to be static, and therefore, every change that happens has to keep the yPositions of all objects the same.

6. ***Applying a Constructor***: Without closing any windows, apply the constructor to make a third object. Notice the effect on the values of the fields in the objects you have already created. The instance or object fields remain unchanged, but the static fields go back to the default values. This is done to ensure that all of the instances have the same value for all of the static fields.

7. ***Remove the class from the project***: We will not have any further use for this class, so you can remove it by right clicking on the class icon and choosing Remove.

Why would you have static fields and methods? Well, we have already given some examples in which it is necessary that methods be static because they get called before any objects are created. Another reason to have static fields is to take advantage of the fact that all of the instances get updated at the same time. For example, consider the Cops

and Robbers example from early in this chapter. One datafield may represent the health of an individual. The health may be affected by how many times you have been hit over the head with a stick for example. If you hit one Cop you should not expect all cops to feel the blow, so health must not be static: an individual's health must belong to him. On the other hand, suppose we have another field that represents visibility, and is a measure of the external lighting. That should change in all instances at the same time, and therefore should be defined as static.

Summary of what you have learned:

- The keyword **static** makes methods and fields belong to the class, instead of the object.
- Static methods have the same effect on all instances at once.
- Static variables always have the same value in all instances, and therefore all change together, even when the change is effected by a non-static method.

1.8 Exercise 1: Adding behaviour to Ball

Note: you must do these exercises because as we continue, we will be working under the assumption that you have adapted the classes appropriately.

Still in *BlueJ*, double click on the rectangle that represents the Class Ball. That will open up an editor window that contains the code for Ball.

In that window, alter the code in the following ways:

1. **Add a Getter**: Add a getter for the yPosition: call it `getY`.
2. **Add a Setter:** Add a method that changes the xPosition of a ball. Call it `setX`. Notice that it will have an int input. Also add a method `setY`. The name setX should remind you of names like getX. These methods are generally called **setters**. Setters and getters are jointly known as **accessor methods**.

3. **Add a Constructor:** Add a new constructor that uses default values for the position and color,[3] but that has an int parameter that will be the diameter of the new ball. Notice that the constructor still needs to have the same name as the class. Therefore, all constructors for a particular class will have the same name. However, no two can have the same signature. That is, different constructors must have different parameter numbers or types. So, for example, you cannot have two constructors for Ball, one of which has just the diameter as a parameter and the other of which has just the x coordinate as a parameter: the compiler would have no way of distinguishing them. Having more than one method in a class with the same name is called **overloading**. Overloading constructors is a particularly important case of overloading.

4. **The solution to 3:** try not to look at this before you have tried the exercise:

```
1. public Ball2 (int diam)
        {
         a.      diameter = diam;
         b.      xPosition = 100;
         c.      yPosition = 100;
         d.      colour = "red";
           }
```

There are several other constructors we could define: they differ in what we use as default values and what we supply as parameters.

1.9 Inheritance for Extension: Drawing Balls

As we have declared graphics to be the application domain we are using for studying object orientation, you may well be disappointed that we have yet to draw anything. We will now rectify that by defining a class of balls that know how to draw themselves on screens. Notice that this is a typical, and important, way of thinking about object-oriented systems: objects are responsible for themselves—it is up to the object to draw itself, it is not up to some superpower to know everything about the World.

[3] We sometimes write “color”, instead of “colour”. This is because, while the real-world property is spelled with a u in British English, the Java datatype is spelled without one. If we are thinking of the datatype we will use the Java spelling; if we are thinking of the real property we will put in the u.

A complication of drawing things in Java is that we need to define the drawing surface as well as what is being drawn. Luckily, this is largely provided by the Java system, and you will learn about that later. For now, we will just provide a class of a particular drawing surface called a **Canvas**. We will leave the code of Canvas unexplained for now. The other class we will introduce in this section is called SelfDrawingBall. A SelfDrawingBall is a Ball with the added ability to be able to draw itself on the screen.

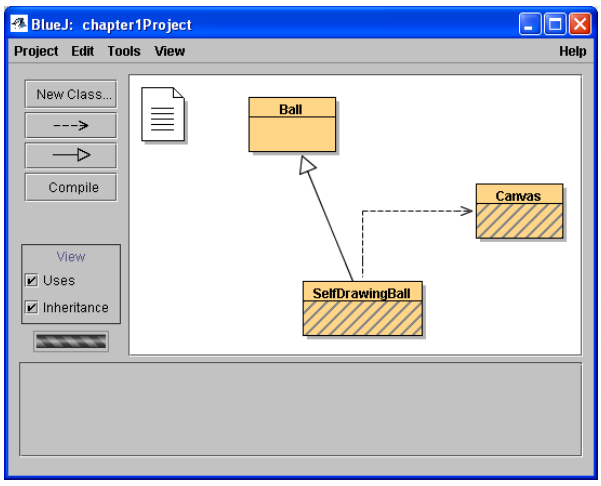
Since SelfDrawingBalls are kinds of Balls, we will define the class as a child class of Ball. Because of this, SelfDrawingBalls will inherit all of Ball's abilities, and we will extend these with the one new drawing ability. This is one of the principal uses of inheritance—adding new behaviours while retaining old ones. We refer to this as **Inheritance for Extension**. You will learn more about this, and the other reasons for inheritance, in Chapter 5.

For now, we will see how the application works and then discuss the main points of the code.

1. Add Canvas.java to your project following the instructions in Section 1.5 above.

2. Similarly, add SelfDrawingBall.java.

3. Now, spread the classes out on the screen by choosing each in turn and, holding down the left mouse button, move the mouse around. You should end up with a window that looks something like the following picture:



4. Notice the two arrows in the class diagram. The arrow made from a solid line and an unfilled triangle that goes from SelfDrawingBall to Ball indicates that the former **inherits** from the latter. The dashed arrow from SelfDrawingBall to Canvas indicates that the former **uses** the latter. The use of these kinds of arrows for these

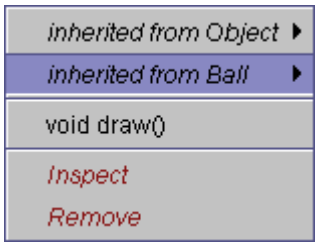
relationships is part of the UML standard. Both of these will be clarified when we look at the code later.

5. Now push the compile button. The stripes should disappear.

6. Make a new SelfDrawingBall object by right clicking on the class and choosing the constructor: new SelfDrawingBall ().

7. Right click on the object; the red rectangle at the bottom of the window. You should get a new window that is the canvas with the ball drawn in it.

8. Now right click on the red rectangle again. Notice that among your options is a new menu called Inherited from Ball,



9. This will give you access to all of the methods that have been inherited from Ball.[4] Choose that menu item and then the move method from Ball. Here you will see it as **void Ball (int, int)**. Choose numbers between 50 and 200. What has happened to the canvas?

10. Nothing has happened to the canvas, even though the ball thinks of its position as having changed. You can test this out by opening up the Inspection window (by choosing Inspect after right clicking on the red rectangle). To make the change on the screen you have to draw the ball again. You can do this by right clicking the red rectangle and choosing void draw (). You should repeatedly choose methods that change the state of the object, and then draw them.

We will now turn our attention to the key parts of the code that make this work.

1. import java.awt.*;
2. import java.awt.geom.*;

---

[4] Notice that you also have the chance to choose methods inherited from Object. This is because all classes are descendent classes of the class Object, which is the root of the inheritance hierarchy. We will learn more about this later.

```
3.  public class SelfDrawingBall extends Ball
    {
4.  public SelfDrawingBall  ()
        a.  {
        b.  super ();
        c.  }
5.  protected void draw()
        {
        a.  Canvas myCanvas = Canvas.getCanvas();
        b.  myCanvas.draw(this, colour, new
            Ellipse2D.Double(xPosition, yPosition,  diameter,
            diameter));
        }
    }
```

Lines 1and 2 are import statements. They tell the complier to get some classes from the Java System. In this case, we need the geometry classes to be able to draw the ball on the screen.

Line 3 is the declaration of what class this is. The new thing here is the keyword **extends**. It signifies that the class we are defining—SelfDrawingBall—is a child class of Ball. That keyword tells the compiler to give us access to all the Ball Methods. It is important to note that this is dynamically worked out so that if Ball changes later SelfDrawingBalls will use the new behaviour of Balls, not the behaviour that Ball had at the time of SelfDrawingBall's definition. The keyword is also the reason that *BlueJ* put the solid arrow in the class diagram.

Lines 4-4c is the definition of the constructor for the class. The key word **super** always refers to the parent class, but this is a very particular use of the word. Within a constructor definition super refers to the constructor of the parent class. So line 4b says that the constructor for this class calls on the constructor for the parent class—Ball. Since that is all that the constructor does, it is essentially equivalent to the parent constructor.

The block 5-5b is the one really new thing here: it is the drawing method. 5b says that we will need to get a drawing surface, which will be a canvas. To be able to do this depends on the class Canvas. This is why *BlueJ* put in the dashed arrow in the class diagram: it represents this dependency. 5c says that to draw the ball, you rely on the draw method that belongs to canvas.

Recall the dot notation: myCanvas.draw refers to the draw method in the object myCanvas. We think of this as sending a draw **message** to the myCanvas object. The **sender** of the message is the ball object; the **receiver** of the message is the canvas object. It is important to understand that when we speak of sending a message to an object we mean invoking one of

the methods that is defined in that Object. Hence, in the Cops and Robber example, we pointed out that if you mean to send a hitOverTheHead message to a robber, then the Robber class needs to have a method hitOverTheHead method. In this case, the draw method in canvas will be defined in the Class Canvas or in one of its ancestors (its parent, its parent's parent, etc.). The draw method in Canvas knows how to draw ellipses. To do so, you give the dimensions of the upper-left hand corner and then the length and width of the bounding box of the ellipse (the bounding box is the smallest rectangle that contains the ellipse). If the bounding box is a square, then the ellipse will be a circle. The reason the compiler at this point knows about ellipses is that we have told it to import the geometry classes from the Java system—had we not done so we would have received an error at this point.

Summary of the section:

1. One reason for using inheritance is to supplement the abilities of a parent class.
2. When you define a class as a child class of another then all of the methods and datafields of the parent class are available to objects of the child class.
3. We use the word *super* to refer to the parent class.
4. Objects can send messages to each other. A message for an object myObject will look like myObject.method.

1.10 Inheritance for Specialisation: Balls that move and draw themselves

It is clearly not always sensible to need to call a separate draw method to see the effects of each change we make. It would be much better to have the drawing as part of the other methods. So we will now define a child class of SelfDrawingBall that exhibits this new behaviour.

So, for example, in the new class we can define a new blowUp method as follows:

```
1.      public void blowUp ()
                {
  a.      super.blowUp();
  b.      draw();
                }
```

Recall that super refers to the parent class. So this definition says that the new version of blowUp will do the same thing as the class's parent would have done if it received a blowUp message, and then draw the screen again. Now, it should be noted that blowUp is not defined in the parent class. What actually happens here is that when you call the super it looks for blowUp in SelfDrawingBall, fails to find it and then looks in Ball. This is the general pattern of things: when you send a message to an object, if it can't respond to it, the message is passed on to the parent, and if the parent can't answer it, the message is passed to the grandparent and so on, all the way up to Object, which is the class from which all classes inherit.

This is a different use of inheritance. Here we are not using inheritance as a way of extending the abilities of the parent class, but rather as a way of altering, or refining, existing behaviour. This is normally done to make the child class more specialised than the parent class, which is what we are doing here. This technique is called **Inheritance for Specialisation**.

An often-quoted real-life example of Inheritance for Specialisation is the case of platypuses. Platypuses are mammals and, therefore, in an object-oriented implementation, the class Platypus could be a child of the class Mammal. One of the defining properties of being a mammal is that you give birth to live offspring, as opposed to laying eggs. So, we could define the action giveBirth in the class Mammal. However, platypuses lay eggs. Therefore, in the implementation of platypuses we will have to override the giveBirth method to this more specialised form of the method, in this case the method is only applicable to one kind of mammal.

You will learn a lot more about Inheritance for Specialisation in Chapter 5. For now, you are going to implement this class.

Summary of the section:

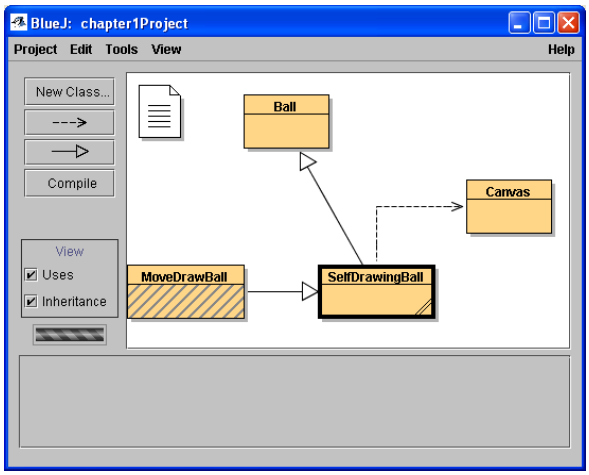
1. A second reason for using inheritance is to revise the abilities of a parent class. This revision usually leads to a more specialised inheritance and so it is usually referred to as *Inheritance for Specification.*

1.11 Exercise 2: Implementing MoveDrawBalls

In this exercise, you are going to implement the class of Balls that will draw themselves after changing their states. You will also learn how to develop new classes from within *BlueJ.*

1. **Define a new class:** In the Edit pull down menu choose New Class… Type MoveDrawBall in the text field and press OK.

2. **Add an inheritance relation**: Move the new class to a convenient part of the window, click on the inheritance arrow button on the left of the screen. Then on the MoveDrawBall rectangle and then on the SelfDrawBall rectangle. Note that the inheritance arrow is the one with a solid line and an unfilled triangle. Your *BlueJ* window should now look something like this:



3. **Open the Class**: Now either right click on the MoveDrawBall class and then choose Edit, or just double click on the MoveDrawBall class.

4. An editor window will pop up, with a template version of the class already in place.

5. Notice that everything that is in blue on your screen is a comment that is ignored by the compiler, so you can ignore it as well for now. The real point of those comments is to provide documentation for you or anybody else looking at your code later. You will learn more about documentation in Chapter 2; for now we will ignore it.

We will go through the template from top to bottom:

1. public class MoveDrawBall extends SelfDrawingBall

Because you put in the arrow, BlueJ took care of the inheritance by putting in the keyword extends.

2. Instance variables: we need no new instance variables; all of the instance variables we need are inherited from Ball. We can therefore delete that section.

3. Constructor: the constructor need not differ from its parent's constructor. So use super to make that happen.

4. Methods: there is one method there to use as a template. Now alter it to be the blowUp method defined in the last section. Redefine move similarly.

5. Try out your new class by making a new object of that class, and exploring the methods.

1.12 A non-trivial method: Bouncing Ball

We are finally in a position to finish the first version of our bouncing ball application. It will be implemented as a child of the SelfDrawingBall Class. It would have been slightly easier—there would have been one fewer line—had it been implemented as a subclass of MoveDrawBall, but that will be left as an exercise.

Bouncing ball has one new method: bouncing. There are two hurdles to be overcome to make the ball bounce for a while around the screen: one is to get the ball to move continually, and the other is to get the ball to bounce off the wall.

Continuous Movement: This can be achieved using any looping structure. We will use a for loop. For the purpose of this example we will just have the ball move on the screen for 500 cycles of the loop, though there are better ways to stop it.

So the way to get this to happen is:

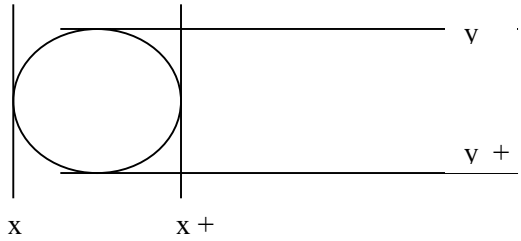
```
1.  for (int i=0;i<500;i++)
                        {
    a.      super.move(dx,dy);
                    }
```

That would just get it to move but it would soon be off the screen, never to be seen again. What we want it to do is bounce off the walls.

Bouncing off walls: Recall that dy is the movement in the y direction (that is vertically: with downwards being positive) and dx is movement in the x direction (that is horizontally: with rightwards being positive).

What happens when you bounce against a vertical wall is that your horizontal motion (your dx) changes. In fact, the change is easy to model: it just becomes the opposite of the motion before the collision—that is if your movement was dx = 6 before hitting the wall afterwards it will be -6; if your movement was dx=-8 before hitting the wall, afterwards it will be 8. Hitting the floor and the ceiling affect your dy similarly.

So the next thing to know is: when do you bounce? Consider the following picture of a ball:



You bounce horizontally if your left-most point is 0 or your right most point is at the right of the canvas (this is at 300); you will bounce vertically if your top point is 0 or your bottom point is at the bottom of the canvas (this is at 300);

Putting all this together we get the following class Definition:

```
1.  public class BouncingBall extends SelfDrawingBall
    {

2.      int dx = 20;
3.      int dy = 17;

4.      public BouncingBall()
                {
        a.        super();
                }

5.      public void bounce()
                {
        a.      for (int i=0;i<500;i++)
        (i)             {
        (ii)    if ((xPosition < 0) || (xPosition + diameter) > 300)
        (iii)           dx = -dx;
        (iv)        if ((yPosition < 0) || (yPosition + diameter) >
                300)
        (v)             dy = -dy;
        (vi)        super.move(dx,dy);
        (vii)       super.draw();
        (viii)      }
                }
    }
```

1.13 Exercise 3

Start a new project. Add the classes: MoveDrawBall, SelfDrawingBall, Ball, and Canvas.

Now define a new version of BouncingBall, call it BouncingBall2, that inherits from MoveDrawBall and has the same behaviour as BouncingBall.

1.14 The future of the bouncing ball example

The way we have implemented the bouncing ball is not typical of the way that we will define graphics applications. We have implemented it this way because it is easier to understand and it brings out the object-orientation principles more clearly. The way we will build graphical applications in the future is by using more powerful drawing surfaces and taking advantage of the inbuilt Java frameworks for graphics. We will re-implement the bouncing ball that way in Chapter 3. The main difference will be that, while in the implementation we have already done, the Ball class refers to the Canvas class; in the later implementation, the drawing surface will own the Ball.

1.15 The chapter so far

In this chapter, so far, we have implemented a simple system as a way of reviewing some object-oriented concepts, reinforcing some key aspects of the object model, and learning about using *BlueJ*.

That is, through the example we have learned about:

- the relationship between objects and classes
- the difference between static and non-static methods and fields
- the general idea and notation for sending messages
- two kinds of use of inheritance: inheritance for extension and inheritance for specialisation.

We also learned how to develop a small application and reviewed how to use loops for repeating actions.

1.16 The rest of the chapter

Now reread Chapter 1 in *Laszlo*. It will take you quickly through some of the basic principles of object-orientation. These principles include some of the ones we have already seen, plus ideas of abstraction, encapsulation and hiding (which will be covered in more detail later), and class association.

2 Methods and Procedures

2.1 What this chapter is about

A **procedure** is the sequences of actions that a computer goes through to perform a task. A **method** is a procedure that is associated with an object. We say that the method belongs to the object and we generally think of the object, rather than the computer, as performing the sequence of actions.

In this chapter we will consider how to talk about procedures and, to some extent, how to implement them. The Level 1 Programming Course is focused on the question of implementing procedures. This course has a different focus. We are more concerned with programming in the large, that is, how to design larger systems that need to do several interrelated things, and about object orientation as a World view that helps to handle the inherent complexities of such systems. This puts a different complexion on what concerns us about procedures. Here the emphasis is on how they fit into a bigger picture.

An important thing to note about fitting procedures into a bigger picture is that you do not need to know everything about a method to work with it. This is useful when you are designing a system: you can do high-level design, leaving the details for later. It is even more useful when you are building a system in which you are not programming all of the procedures yourself, either because you are part of a team, or if you are using code supplied by a standard supplier, like Sun.

We will now look at what you need to know about a procedure, and about how this can be expressed.

2.2 Procedural Abstraction

You should now read Sections 2.1 and 2.2 of Laszlo, which will tell you about abstraction and specification of procedures, and then come back to the subject guide, where some of the main points will be discussed.

2.2.1 Section 2.1 of Laszlo

The important thing to note from Section 2.1 is that what you need to know about a procedure to be able to use it as part of a larger system is its effects, not its implementation details. You may need to delve into those details at some time—indeed you may have written the procedure before or will need to write it yourself later—but when you are planning a system that uses the procedure you need to know two things: its assumptions about its inputs, and its effects. The effects are of two types: output values and effects. This, of course, works both ways and the same information needs to be documented for the procedures you write if you expect them to be used by anyone else or if you may need to use them yourself at a later date. In practice, you should work on the assumption that all of your methods fall into this category and document them using a notation like the one you will see developed in Section 2.2 of Laszlo. Having announced what is best practice, we will not follow it in the subject guide. This breach is not serious because we are describing the programs as we write them here. We, therefore, feel that the extra clutter of the documentation would detract more than it would add. However, Laszlo documents all of his procedures, and you should follow his example, rather than ours.

There are many benefits of thinking of procedures in terms of their abstract behaviour, rather than their implementation details. These include:

- ***Simplicity***: It cuts down on the complication associated with dealing with code.
- ***Robustness***: Implementations of a method can change without affecting the behaviour of the rest of the system. As long as the new implementation conforms to the behavioural specification, you will not have to change any other code when a particular implementation changes.

Another notable point brought out by these specifications is that there are two kinds of likely effects that a procedure can have: it can produce an output, and it can change the state of one or more object. This second is called an effect. Changing the state of an object is not the only kind of effect. A particularly important effect that comes up in this section, and has not been mentioned in the subject guide before, is the effect of writing to a screen. The code for this is:

```
System.out.println(whatever you want to write)
```

As we have seen, this is sending a message to (or calling a method in) an object called System.out, which is in turn an object that belongs to System. Writing to a screen might not look very different from returning a value. The difference is this: when an object returns a

value it is available to the object that sent it the message (this will become clearer soon); when a value is written to a screen it is simply written to a screen.

2.2.2 Translation

The section in Laszlo works through several versions of a translate method, which moves a point a set distance in the x and in the y direction. To get a better feel for the variations, you should now go back to *BlueJ*, where you will run the following methods:

1. In *BlueJ*, define a new project called Chapter2Project
2. Add the following two classes to the project: Point.java and TranslationMethods.java. They can both be found in the Chapter 2 folder in the source code folder on the CD.

Before proceeding further, we will examine the code for both of these. Point is a very simple class that has two datafields (one for each coordinate of its position), several constructors, and some accessor methods (getters and setters).

```
1.  public class Point
    {
2.   protected int x, y;
3.   public Point(int newX, int newY)
       {
     a.     x = newX;
     b.     y = newY;
      }
4.   public Point(Point p)
       {
     a.     this(p.getX(), p.getY());
      }
5.   public Point()
     {
      a.    this(0, 0);
      }
6.   public int getX()
       {
        a.  return x;
       }
7.   public void setX(int newX)
       {
```

```
    a.   x = newX;
    }
```

… and so on.

There are a few interesting things to note here: one is the use of the keyword this. The keyword is exactly parallel to super, which we have already seen. Where super always refers to the parent class, this always refers to the present class. And in the context of a constructor, this by itself refers to another constructor for the present class. So, for example, in line 4a, the this refers to the constructor defined in 3. How does Java know which constructor is being referred to? The answer is in the signature (the type) of the constructor. At 4a, this has two int inputs, so it must be the constructor at 3.

This small method also exemplifies the difference between outputs and printed values. In line 4a, the method takes as one of its inputs p.getX(). This value is the output, the returned value, of the getX method. That is, by sending the getX() message to p, you get a result returned to you, and you can use that however you want.

The TranslationMethods class has all of the translation methods from Sections 2.1–2.3 of Laszlo in it, plus one other:

```
1.  public static int translate6(Point p, int dx, int dy)
        throws NullPointerException

    {
    a.  p.setX(p.getX() + dx);
    b.  p.setY(p.getY() + dy);
    c.  return(p.getX());
    }
```

This is a method with both an effect (changing the values of the coordinate fields by dx and dy) and an output (the new x coordinate). Notice that the method is declared as static. This is because, although it is possible to make objects of this class, and have the objects send the messages, it is just an unnecessary complication. We will not want different objects to send their own translation methods. It is important to note here that if the methods were not static, they would not belong to points, but to translationMethod objects. It would be perfectly reasonable to have non-static translation methods belonging to the Points. Indeed, we did the equivalent of that in our ball example in Chapter 1. The throws clause will be explained in the next section.

Now go back to *BlueJ* and explore the translation methods. Note that you will need to create a Point object, using one of the constructors in the Point class, before you have anything to

translate. You will then have to supply the name of your object into the textfield on the dialog box that pops up when you call any of the translate methods.

2.2.3 Section 2.2 of Laszlo

In Laszlo Section 2.2, the effects are looked at more closely and methods for specifying the procedures are discussed. The important concept is that of Design by Contract. The idea here is that for each procedure, you describe the minimum necessary conditions you expect of an input and exactly what you guarantee of the output.

The section also contains a description of the Assert class of Java and how it can be used as a way of debugging programs. The Assert class automates the process of ensuring that your code keeps to its contracts. If at any time, a procedure is called with input that does not satisfy the precondition, the program will stop running and alert you that the precondition wasn't satisfied; if after running the procedure, the postcondition is not satisfied, then the program will stop running and you will be told that the postcondition has failed. This can be useful information, especially during debugging. You will learn a more general mechanism for this kind of behaviour in the next section.

You are encouraged to try out the effects of using assertions by adding the TryAssertions class to your Chapter 2 project and explore its use.

What you should take from this discussion is that the important things to say about a procedure are:

- the **requires clause**: Input assumptions:
- the **modifies clause:** Classes affected:
- the **effects clause:** Effects and outputs returned

This trio amounts to a notation for documenting your procedures.

2.3 Exceptions

Section 2.3 of Laszlo concerns exceptions, which you will already have met in your Level 1 programming course. Exceptions are used to handle unusual, but not unexpected, cases.

They are usually associated with erroneous behaviour, such as reading past the end of an array, but they can be used in other ways.

Activity

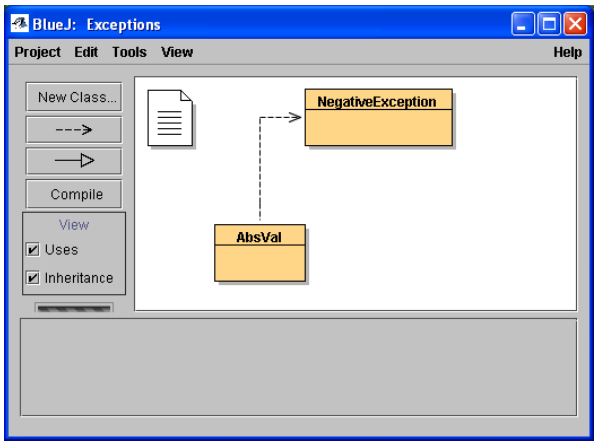
Now read Section 2.3 of Laszlo, where you will learn about the throwing and catching exceptions, and then come back to the subject guide.

The cycle of the use of an exception is: first it is thrown, then it can be caught, and then it may be handled.

To show how this all works we will make a very simple application that includes the full cycle from: defining an exception, throwing the exception, catching the exception, and then handling it. The application is a rather silly implementation of an absolute value function for ints. Recall that the absolute value of a number is never negative, and is defined to be the non-negative number that is as far away from 0 as the input number is. Therefore, if the input is positive the output will be the same as the input; if the input is negative, the output will be the same as the input without the negative sign. For example the absolute value of -4 is 4. Because the product of two negatives is a positive, to get the absolute value of a negative number all you need to do is multiply it by -1: for example -1 * -4 = 4.

In our implementation we will build the absolute value out of the composition of two functions: the first returns the input if it is positive and throws an exception if the input is negative. The exception it throws will be one we define ourselves, called NegativeException. The second function will call the first, and it will catch the exception, and handle it by multiplying the input by -1.

To use these functions in *BlueJ* , open a new project and add the classes: NegativeException and AbsVal. These are both in the Chapter 2 folder supplied with your CD. Press the Compile button. Your window should look something like the following:



Notice the arrow that indicates that AbsVal uses NegativeException, because it can throw it as an exception.

The first thing is to define the exception. The code follows:

```
public class NegativeException extends Throwable
{
}
```

That's the whole class. The fact that it is a child of Throwable is all we really need. Note that we could have defined this exception within the other class. To do so would have been almost the same code embedded within the class definition we are about to see. The one difference necessary would have been the insertion of the keyword static before the word class. We would need the class to be static so that it could be called from static methods.

The second class is given by:

```
1.  public class AbsVal
    {

2.  static int  onlyPositive (int x) throws NegativeException
      a.    {
      b.    if (x<0) throw new NegativeException();
      c.     return x;
      d.    }

3.  static int absValue (int x)
      a.    {
      b.    try
      c.     {
      d.      return onlyPositive (x);
      e.      }
      f.      catch (NegativeException e)
                i.          {
                ii.         return (-x);
                iii.        }
     }
```

onlyPositive: Notice that onlyPositive has to mention that it throws an exception in its heading. When it does throw the exception, the effect is to come out of the computation and to make a new object of class NegativeException. That is why we have the keyword new. If 2b is called, the code never gets to 2c, so nothing is returned.

absValue Notice that we do not have to mention the exception in the heading of this method because it is not thrown, it is handled. Exception catching is always done in a try clause. It essentially says that you should try to do something, though you may be diverted by a thrown exception. In the present case there will be an exception thrown whenever x is negative. In 3f the exception is handled by returning the output, which is -x, or -1* x.

Activity

Before moving on to the next section, you should do exercises 2.3 and 2.4 and 2.5 in Laszlo.

2.4 Procedural Decomposition

2.4.1 Before reading Laszlo Section 2.4

This Laszlo section you are about to read concerns building up procedures from parts, or, equivalently, breaking down specifications to sub-specifications that can be designed separately and assembled into a solution to the original specification.

The section contains this course's first use of the main method. So before proceeding with the Laszlo text, it will be useful to review that. You probably remember that the main method of a class is the method that is called when an application is run automatically. Therefore, every stand-alone application needs to have exactly one main method. The fact that Java knows that it should start by running main, is part of what is called a framework. You will learn more about frameworks later in this chapter and then much more in Chapter Seven. The main method has to be written so that it overrides the main method in the Object class. This implies that the main method must always have the same signature, which is:

```
public static void main (String[] args)
```

The public is needed so it can be called from outside; the static is necessary so it can be called before any objects are created. The **static** is needed so that the method can be called before any objects are created. The **void** means that there are no outputs. The main is the method

name. The input is an array called **args**; the array is an array of Strings. This input type has wide-ranging effects that can be frustrating. Whenever you run a stand-alone application, the inputs are always Strings. This means that if what you really want is integer inputs, for example, you need to do some processing. The inputs must start as appropriate strings (such as "433") and then be turned into ints within the programme. You have already seen that this is not the case with other methods, but since it is true of the main method, the parameters that are called by any stand alone application are Strings.

One way of dealing with this is to define a method that takes as input an array of Strings, and translates them into the kind of inputs you really need to work with. You will see an instance of this in the getNumbers method in Section 2.4.

Activity

You should now read section 2.4 of Laszlo, where decomposition of problems is discussed and the example of a sorting procedure is worked out, and then come back to the subject guide.

2.4.2 After reading Section 2.4

The section followed the development of a sorting procedure. The specification was initially broken into three sub-problems:

1. Read the inputs, turn them into ints, and put them into an array
2. Sort the array of ints
3. Print out the sorted array.

One of the advantages of the decomposition is that we can treat the problems as independent, and, for example, change the sorting procedure later without changing anything else. We will see two choices of sorting methods in this chapter. The first of these is called **Selection Sort**.

The crux of the selection sort method is given in the following loop (which is part of the sort procedure given on page 51 of Laszlo):

```
1.      for (int i = 0; i < n ; i++)
          {
        a.      int indx = min(a, i, n-1);
        b.      swap(a, i, indx);
          }
```

The for statement ensures that whatever is defined within the body of the loop will be performed on the procedure which computes the smallest element not to the left of i on the array. At line b, you take that minimal element put it where i is, and put i where that minimal element was.

Note that this sort procedure is marginally less efficient than it need be. In the last iteration of the loop, min is called with a one-element array and this will never cause any swapping. This call would have been avoided if the loop stopping condition were altered to "i < n-1". This alteration would have no effect on the result. We have left it this way to be in line with the book. However, the slowdown is very small and, indeed, the extra call does not force any extra comparisons (which are the hardest part of the program) because the min procedure is defined to not do any comparisons with a one-element array.

How does this sort the array? When i = 0, the method finds the smallest element on the whole array and puts it in the first place. When i =1, it finds the smallest of the rest and makes it second. The second cannot be smaller than the element that has already been put in the first place, otherwise **it** would have been placed there. At that point, the first element of the array is the smallest and the second element is the second smallest (or they may be tied). And so on.

How efficient is this algorithm? We measure the efficiency of sorting algorithms by the number of comparisons that have to be made in the course of performing them because it is the comparisons that take by far the most time. In this algorithm, if n is the length of the array, the procedure does n-1 comparisons to find the first element; the procedure does n-2 comparisons to find the next; the procedure then does n-3 and so on. Therefore, the number of comparisons is given by:

$$(n-1) + (n-2) + \ldots + 1$$

The evaluation of this sum breaks into two cases, depending on whether n is even or odd.

Case 1: n -1 is even

Rearrange the terms, so that the first and last are added, then the second and second to last are added, and so on. The sum becomes

$$((n-1) + 1) + ((n-2)+2) + ((n-3) + 3) + \ldots$$

or

$n + n + n + \ldots$.

There are (n-1)/2 of these pairs

Therefore, the number of comparisons is n(n-1)/2

Which is $\frac{1}{2} n^2 - \frac{1}{2} n$.

Case 2: n-1 is odd

$1 + \ldots + (n-1) = (1 + \ldots + (n-2)) + (n-1)$

Since n-2 is even, the sum of the first n-2 can be found using the formula proven in Case 1.

So

$1 + \ldots + (n-1) = \frac{1}{2}(n-1)^2 - \frac{1}{2}(n-1) + (n-1)$
$= \frac{1}{2}(n-1)^2 + \frac{1}{2}(n-1) = \frac{1}{2}(n^2-2n+1+n-1)$
$= \frac{1}{2}(n^2-n) = \frac{1}{2} n^2 - \frac{1}{2} n$

We, therefore, say that this is an n^2 algorithm, or that the complexity of the algorithm is $O(n^2)$. Recall that when judging the complexity of an algorithm, we ignore all terms of lower power than the leading term. In this case, we just consider the n^2 and not the n. We also ignore the fact that the n^2 is multiplied by $\frac{1}{2}$. All we care about is that leading power. In this case, what it tells you is that if you double the size of the input array, you multiply the time it takes to sort by approximately 4; if you triple the size of the input, the time it takes to sort goes up by about a factor of 9.

One thing that is unusual about the complexity of Selection Sort is that the same number of comparisons is done on any two arrays with the same number of elements. Because the time taken with most algorithms varies even with the same size inputs, the efficiency of an algorithm is judged on the following two measures: ***average case complexity***, that is how long the algorithm takes on average; or ***worst case complexity***, that is the longest the algorithm may take with arrays of a fixed size. The use of the first is it tells you how long you are likely to have to wait; the use of the second is it tells you when you can be sure you will have your answer. Some algorithms work much more efficiently with some kinds of input than others.

For example, some sorting algorithms work well with data that is nearly sorted before you begin. Therefore, the shape of the data that a procedure is likely to come across can influence the choice of sorting algorithm.

As it turns out, this sorting algorithm is not the most efficient, though it may be the easiest to understand. We will, in the next section, learn about a more efficient one. Before moving on, do the following task and exercise.

Task: Make a project called Sorting, and add the SortIntegerArgs file from Chapter 2 Sourcecode folder that comes on your CD. Experiment with it.

Exercise: Make a copy of the file SortIntegerArgs.java and call it SelectSortComplex.java. Modify that file so that the class has a static int variable compNo. Modify the class further so that this variable will keep track of the number of comparisons that are done in a sort. Your class should print out the value of compNo at the end of sorting. Add the new class to your sorting project and experiment with it. A solution for this exercise has been provided in the Chapter 2 source code folder of your CD, but you should try to do it yourself before looking at the solution.

Summary of the section:

- Decomposition is an important and useful way of handling complex problems.
- The main method always takes an array of strings as input.
- Selection Sort has quadratic, n^2 , complexity.
- Quadratic complexity means that doubling the size of the input, quadruples the time the algorithm takes, etc.

Activity

Before moving on, you should do the following exercises from Laszlo: 2.7, 2.8, and 2.9.

You can explore the example program from 2.8 by adding Triangle.java to your project. If you have trouble doing 2.9, the answer is included in the Chapter 2 source code folder.

2.5 Recursion

The procedures we saw in the last section used looping structures to repeatedly do things. This section is about an alternative way of getting programs to perform repeated tasks: recursion.

A recursive function is one that, in its operation, calls itself. This could easily turn into a process that will never end: if function f calls function f, then when f runs a second time it will call f again, and so on. To avoid this happening we will need a base case where the function no longer needs to call itself, and which is eventually reached from any proper starting point. Usually, the function keeps calling itself on smaller inputs until you get to 0 or an array with one element or some kind of starting place.

To get an idea of how recursion works, consider the follow scenario: you are asked to find out the height of the tallest person in the class. Suppose that you have dutifully worked out the height of the tallest person in the classroom, and just when you are about to hang up your tape measure and go home, John (about whom you had completely forgotten) walks into the room. What do you do now?

Well, you could start over and measure everybody again. This is clearly not necessary. All you really have to do is measure John and compare his height to the answer you have already computed. If John's height is greater than your answer, then the greatest height is John's; if John's height is less than that, then the greatest height is the answer you had before. This is, we hope, fairly obvious. Now, the surprising thing is that that is almost all you would have to tell a computer to get it to do the computation.

Consider the following related program in Java. This program finds the greatest element on n integer array:

```
1.  static int findGreatest(int[] a, int lo, int hi)
    {
     a.     if (lo == hi) return a[lo];
     b.     else {
                (i)     int temp = findGreatest (a, lo+1, hi);
                (ii)    if (a[lo] > temp) {temp = a[lo];}
                (iii)   return temp;
                   }
     }
```

At line 1, we have the header of the method. Notice that there are three inputs: an array and two integers. In practice, the two integers will be the array index of the first and last elements

of the array. They need to be referred to explicitly so that in 1b.i we can refer to a smaller array.

Line 1a gives us our base case: it says when we have an array with one element, the answer will be that element.

Lines b.(i)–(iii) give the recursive call. This is parallel to our height example. The expression (a, lo+1, hi) is equivalent to the classroom without John; while a[0] is equivalent to the height of John. What 1b.i computes is the maximum height in the room before John arrives. 1b.ii says that you alter the maximum height if John is taller than the answer you had before.

And that's really it. This will stop if you start with a non-empty array, because each time it calls itself it is with a smaller input, until eventually the input has one element, at which time line 1a comes into play and you get out of the recursion.

To make this easy to use, we put a non-recursive shell around it, which gives us the usual usage as described above:

```
2.    static int greatest (int[] a)
          {
        a.    return findGreatest(a, 0, a.length-1);
          }
```

What this method does is to call findGreatest giving the first and last array indices as the parameters for the beginning and end. Remember that, since array indices start at 0, the last index is length-1.

Section 2.5 of Laszlo has more explanation and several examples of recursive procedures.

Activity

Now read the section, and then come back to the subject guide.

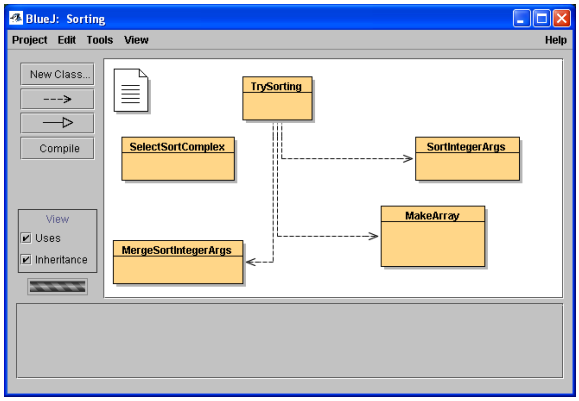
Now do exercise 2.14 (the answer can be found in a file called Power.java the chapter 2 folder in the sourcecode folder on your CD – but try to do it yourself before looking at the answer). Follow this with exercises 2.17 and 2.18. Much of 2.18 is provided on the course CD, but try to do it yourself before opening it.

Mergesort (exercise 2.18) is an example of a divide-and-conquer method for solving a sorting problem: the algorithm works by dividing the original array in two, sorting the two halves separately and then merging the two halves to make a whole which has the same

elements as the original but is now sorted. The halves are sorted in the same way: that is the halves are each broken in two, making quarters that are sorted and merged back to the halves. This process continues until you get down to arrays with one element. This may be harder to understand than the selection sort we learned before; however it is much more efficient. To get a feeling for the difference in timing do the following task and exercise.

Task: Add the following classes to your Sorting project: TrySorting, MakeArray, MergeSortIntegerArgs. They can all be found in the Chapter 2 Sourcecode folder on your CD.

Your *BlueJ* window should look something like this:



1. Compile all and then run the trySorts() method from TrySorting. This will make two arrays of 1000 random numbers from 1 to 1000, sort each of them with one of the two sorting methods and print out the start and finish times for each algorithm.

2. Experiment by changing the value of the variable y in MakeArray, and running trySorts again. This variable controls how large the sample arrays will be. Try it with several values. You should especially do this if your computer is either too slow to do the length 1000 arrays or too fast to notice much difference in the timings with size 1000.

3. Now examine the code of the various classes in this project. The project has been designed to remind you of the differences between class methods and instance methods. Therefore, while the two sorting classes have all their members (fields and methods) defined as static, MakeArray is defined so that an instance of it must be defined before any of its members can be used. Therefore the class TrySorting uses the two sorting classes directly but makes a new object of class MakeArray, in the line:

```
MakeArray newMA = new MakeArray ();
```

Exercise: Make a copy of the file MergeSortIntegerArgs.java and call it MergeSortComplex.java. Modify that file so that the class has a static int variable compNo. Modify the class further so that this variable will keep track of the number of comparisons that are done in a sort. Your class should print out the value of compNo at the end of sorting. Add the new class to your sorting project and experiment with it. A solution for this exercise has been provided in the Chapter 2 source code folder of your CD, but you should try to do it yourself before looking at the solution.

2.6 A brief foray into complexity: why merge sort is faster than select sort

Recall that the time complexity of a sorting algorithm is dominated by the number of comparisons that must be performed in running the algorithm. We have already computed the number of comparisons involved in sorting by a Selection Sort and found that the number is $n(n+1)/2$, which makes Selection Sort an $O(n^2)$ algorithm. We now turn to an analysis of the complexity of mergeSort.

The way that the analysis runs, and this is fairly typical of the genre, is by way of a recurrence relation. It is not necessary that you are able to repeat this argument, but you should at least try to follow the gist of it. Unlike Selection Sort, the number of comparisons depends on the order of the array before it is sorted, so we will analyse the worst case.

Let T(n) represent the number of comparisons needed to sort an array with n elements.

$T(1) = 0$

because you do not need to do any comparisons to sort a 1-element array

$T(n) = 2*T(n/2) + (n - 1)$

because it takes no comparisons to break the array in two, and then you solve the two halves separately (giving the 2 T(n/2)), and then it takes at most n -1 comparisons to put them back together

Since the way the two n/2 arrays are sorted is to halve them as well, we can rewrite T(n) as

$T(n) = 2*T(n/2) + n - 1 = 2*(2*T(n/2) + (n/2)-1) + n-1$
$= 4*T(n/2) + 2n - 3$

Since sorting the n/4 length arrays also involves halving the arrays we can rewrite this further as

$$T(n) = 4*T(n/4) + 2n - 3 = 4*(2*T(n/8) + n/4-1) + 2n$$
$$= 8* T(n/8) + 3n - 7$$

This reasoning leads to the general rule:

$$T(n) = 2^R * T(n/(2^R)) + R\,n - 2^R - 1$$

An important case of this is when you get down to single element arrays. This happens when $R = \log_2 n$ or equivalently, when $n = 2^R$. (Note that we are assuming here that n is a power of 2; for other numbers the analysis should really be done with the smallest power of 2 that is larger than the number.)

In this case, the equation becomes

$$T(n) = n * T(1) + (\log n)\,n - n - 1$$

The first term is zero; the third and fourth are dominated by the second for large numbers, so this algorithm is order n (log n).

Notice that n*(log n) is noticeably smaller than n^2 for large n. That is why this is a much faster algorithm. As it happens, n*(log n) is optimal: there are no sorting algorithms with smaller worst case complexity than n(log n), though there are algorithms that perform better in special cases.

2.7 Summary of the Chapter

This Chapter concerned procedures. The beginning of the chapter stressed the abstract view, in which a procedure is represented by its inputs and effects, rather than in terms of its implementation as Java code. This view of procedures has important benefits, including simplicity and robustness. It also fits well with principal themes of this course, abstraction and programming in the large. These themes will be taken up again in the next chapter.

The chapter also considered implementation issues, primarily that of decomposing a problem into independent chunks and implementing them separately. We saw two sorting algorithms as instances of decomposed systems (one—MergeSort—was given within an exercise). The idea of problem decomposition is to break the problem down to layers of problems, each layer providing solutions for the one above.

As well as seeing procedures built from others, we saw how recursive procedures are built in terms of themselves. MergeSort was implemented as a recursive procedure.

You also saw complexity analyses of these two sorting algorithms.

2.8 Learning outcomes

By the end of this chapter and the relevant reading, you should be able to answer examination questions on:

- Specification of Procedures
- Procedural Decomposition
- Sorting Algorithms
- Recursion
- Complexity Analyses.

3 Data Abstraction

3.1 Abstract Data Types

Laszlo 3.1 contains a discussion of data types in Java: both concrete ones and abstract ones, which are the main subject of this chapter. In the last chapter we talked about methods and how to specify them abstractly. In this chapter, the same idea is brought to a higher level and we learn about specifying abstract data types, which include methods and data. An abstract data type declares a set of values, operations on the values, and axioms that the operations must satisfy. The axioms frequently take the form of preconditions and post-conditions, as we have seen in Chapter 2. The Assert Class in Java can help ensure that the axioms are satisfied.

Abstract data types do not define how the values are represented and how the operations are implemented. Performance — speed, memory requirements — depends on the representation. Therefore, there are frequently several different implementations for the same abstract data type. Which one to use in an application depends on performance issues that have nothing to do with the abstract behaviour.

3.2 Specification and Implementation of abstract data types in Java

There are three ways of packaging behaviour in Java. These are, in increasing order of abstraction: Classes, Abstract Classes, and Interfaces. A class implements all of the methods it mentions; an abstract class may implement no methods, some methods or all methods; an interface cannot implement any. Of the three, only classes can be instantiated, which is to say that only classes can have objects that are instances of them.

The most natural choice for describing an abstract data type is as an interface, but before dealing with that we will briefly discuss abstract classes.

3.2.1 What is an abstract class?

An abstract class looks much like a class, except that it is not necessary to implement all of the methods in it. That a class is abstract is signalled by the keyword abstract that appears before the word class in the header, as in:

```
abstract class Set extends Vector
```

There are two effects on the compiler of calling a class abstract:

1. You will be allowed to leave methods that you supply the headers of unimplemented.

2. You will not be allowed to make objects of the class.

Abstract classes are used in two ways in software development: to defer implementation decisions and to bring out common abstractions used in several classes. In Chapter Five, you will see an extended example of this as different kinds of geometric classes are found to have similarities that are worth formalising. As was mentioned in the Cops and Robber example from Chapter 1, it is more than useful for classes that have some similar behaviour to share a parent. It is more efficient and safer for future changes in the system.

3.2.2 What Is an Interface?

The following definition comes from the Sun website:

Definition: An *interface* is a named collection of method definitions (without implementations). An interface can also declare constants.

This suggests that an interface is nothing more than a special kind of abstract class. However, there are crucial differences, and the existence of the interface mechanism lends great flexibility to Java. One difference in use between the two is that, whereas a class can only inherit from one parent class (and every class except the root class `Object` has exactly one parent), a class can implement any number of interfaces (implementation is the interface equivalent of inheritance for classes).

The benefits of implementing an interface are not so obvious as those of extending a parent class. When you extend a parent class, you tend to inherit implemented behaviour; with an interface there is no implemented behaviour to inherit. All you do inherit is the responsibility of implementing all of the methods specified in the interface. However, there are two very strong repayments to be had from implementing interfaces:

- Other classes know that objects of the implementing class can be relied on to implement certain methods. That is, some messages can

be written to any class that implements a particular interface, and so by implementing the interface, a class enables its objects to become legal recipients of those messages.

- Objects of the implementing class can be collected (in, for example, the same array) with unrelated classes that happen to implement the same interface.

A class can implement many interfaces. As such, the interfaces a class implements tend to say much less about the class's identity than the parent class does. For example, we will see in Chapter 7 that any class that represents a screen that happens to respond to mouse movements and/or clicks must implement the MouseListener interface. This does not mean that the class is primarily a mouse listener, but only that it represents one of the class' aspects.

3.2.3 Class Skeletons

Laszlo does not use abstract classes or interfaces to specify abstract data types. Instead, he uses something he calls a **class skeleton**, which is not a Java construct. It is simply a representation to present the data, operations, and the effects of the operations to people. In section 3.2, two abstract data types are specified as class skeletons and then implemented. The first of these is a class that represents points in the Euclidean plane; the second represents rectangles.

Activity

You should now read Laszlo from the beginning of Chapter 3 to the end of section 3.21. This will cover discussions of concrete and abstract data types and also cover the example of points. When you have finished reading you should return to the subject guide (with the course text to hand) where some of the key points are discussed.

3.2.4 PointGeometry

The first three methods described are Constructors, and we have already seen several of those in previous examples. The next four methods described are accessor methods (getters and setters). Again, we have seen quite a few of these.

The first interesting method specified is distance(PointGeometry p). The interest value of the method is in what it reminds us about the object model. To understand this you must note that the "p" in the signature of the method does not refer to the point we are defining, but to another object that is being passed to our object as a parameter to the distance method. What

is happening here is that we use this to send a message to p1, asking how far it is from another point—the argument of the procedure, p2. In practice, the syntax will look like p1.distance(p2). Here distance is a method that belongs to the object p1, and p2 is a parameter to the method. We say that PointGeomery objects have the **responsibility** of computing the distance to other points.

The line

```
java.awt.Shape shape ()
```

may look a bit perplexing at first. The name of the method is shape. It has no inputs and its output is of type java.awt.Shape. This is actually not quite true: java.awt.Shape isn't a class but is an interface as explained above. Anything that is drawn on a screen has to implement the shape interface. Therefore, if we are to paint an object on the screen, we need to have a method that computes the shape of the object. You will learn more about interfaces in general in Chapter 5, and more about shape later in this chapter.

The next method—translate(dx, dy)— moves the point by dx horizontally and dy vertically (remember the positive direction is down!). This is much like the move method we had in Ball.

The equals method brings up an important and potentially confusing point about equality of objects. Given two objects you can check their equality by using the Java equals sign: == (that is a double equal sign: remember that a single equal sign is an assignment statement!) That equality may have different behaviour than you are expecting. It does not check whether the two objects have the same fields; instead it checks if they are pointing to the same thing. Therefore, if you define two points in the same position, they will not be equal in terms of ==. This initially surprising behaviour is matched by parallel unexpected behaviour of assignment statements for objects.

Consider the following code, using ordinary assignment statements:

```
1. static int x1 = 2;
2. static int x2 = 2;
3. static int x3 = x2;
```

If you were then to ask if any two of them were equal, the answer would come back true. Now if you then add the line

```
4. x2 =x2+1;
```

And then ask about equality, you should find that x3 and x1 are still equal, and both unequal to x2.

The behaviour with objects is markedly different. Consider the following sequence of statements:

```
1.  static PointGeometry p1 = new PointGeometry(1,2);
2.  static PointGeometry p2 = new PointGeometry(1,2);
3.  static PointGeometry p3 = p2;
```

We have two ways of checking their equality: using p1.equals and using the equals sign. These two ways do not agree. Using p.equals, they are all the same. Using the equals sign, p2 and p3 are the same, but p1 is different. That is because == is not checking for the points being in the same place, but rather whether the two points are pointing to the same part of the machine's memory to get the values of its attributes. p1 and p2 happen to be in the same place now, but the information about their x and y components are stored in different parts of the computer's memory. Line 3 is interpreted as making the point p3 look for its attribute values where p2's attributes are stored. So p3 and p2 are equal, even using ==.

Now, translate p2, using its translate method. What do you think the two equals will give as answers now? One surprising thing is that p2 and p3 are still equal in both ways of reckoning equality. The reason for this is that statement p3 is still looking at the same memory location as p2, so it has moved too! Whenever one changes, they both change. This may remind you of the case of static methods that we discussed in Chapter 1: when a static field changes, its value changes in every instance of the class.

Try out your intuitions by compiling and running tryEquals.java from Chapter 3 source code on your CD. Remember that to run a class from *BlueJ*, you right click the class and choose the static method main. Make a new Project called Points, and add TryEquals to the project. Notice that you do not have to add PointGeometry. Indeed you do not have a class for that. PointGeometry is an interface and is part of banana.jar, so is made available through the import statement.

3.2.5 TryPoint

Now add the TryPoint class to your project. The class is almost self-explanatory. The only things worth mentioning at this juncture are:

a) the method main is the method that would run automatically if you ran this from outside *BlueJ* . Within *BlueJ* , you treat main just like any other method and run it by right clicking it.

b) the method main always takes Strings as inputs. To turn the Strings to ints is a two step process:

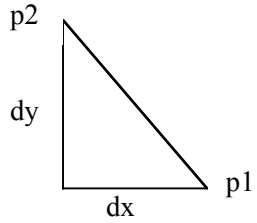
   i. Turn each string into an object that is an instance of the class Integer. (We are assuming that the strings are amenable to that. That is, we are looking for strings like "63". If we get the wrong kind of string we will raise an exception.

   ii. Turn the Integer to an int by sending the Integer a parseInt method.

3.2.6 Implementation of PointGeometry

The two methods of interest here are distance and shape.

Distance is defined using the Pythagorean theorem with the following picture:



The distance between the two points is the length of the hypotenuse, which is the square root of $dx^2 + dy^2$

The shape of a point (that is the way we will draw the point on the screen) is a circle, which is an ellipse with the same width and height. You will learn more about ellipses later in the chapter. Recall from the discussion of balls in Chapter 1, the data needed for an ellipse are the coordinates of the upper-left hand corner of the bounding box, the width of the bounding box and its height. In this case, we see that a point is a circle whose x and y values represent the centre of the point, and whose radius is 2.

Activity

Now return to Laszlo and do exercises 3.1–3.4. It is especially important to do 3.2 because the class you are asked to implement in is used in the RectangleGeometry class still to come.

You should the read the rest of section 3.2 before returning to the subject guide. You will learn about an approach to the specification and implementation of rectangles, and see that an implementation isn't necessarily the obvious one that implements directly the main properties of the class.

3.2.7 RectangleGeometry

One of the interesting things about the RectangleGeometry class is that, unlike the PointGeometry class, the implementation is different from what seems the most obvious implementation of the Specification. The getters and setters for RectangleGeometry are for Positions, Heights and Widths. We say that these are the **properties** of the class. This would lead you to suspect that the data in the implementation was a Point for the Position, plus ints for the Width and Height. (This is, indeed, the way that Graphcs2D implements rectangles). However the implementation given here is in terms of Range Objects, which you implemented in exercise 3.2.

This goes back to the comment in the beginning of this chapter in the subject guide that implementation decisions rely on things other than the logical abstract behaviour. This implementation in terms of Range objects is a good choice if you frequently are going to have to decide whether points are inside or outside of the rectangle.

In exercise 3.5 you are asked to finish the implementation of the RectangleGeometry class. Notice that within the specification of the question you are asked to avoid dependence on the implementation details of the class, even within the definition of the class itself. What that comes down to is using getters to get values rather than accessing the values directly. There is a very good reason to do this as a matter of course; if you use getters then if the representation for the class changes, only those methods that directly access the storage structure must be redefined. The fewer methods that depend directly on the representation, the easier and safer it will be to change that representation.

Activity

Now do exercise 3.5 and then read Laszlo section 3.3. The section is a general discussion of encapsulation and hiding. The specific discussion about using getters is a case in point: the implementation details should be encapsulated in a protected shell and hidden from other users.

3.3 Encapsulation

Section 3.3 contains a general discussion of the concepts of encapsulation and information hiding. **Encapsulation** is the ability of an object to place a boundary around its *properties* and *methods*. Older programs suffered from *side effects* where variables had their contents changed or reused in uncontrolled ways. Object variables can be hidden completely from external access. These *private* variables can only be modified by use of object methods. Access to other object variables can be allowed but with tight control on how it is done. Methods can also be completely hidden from external use. Those that are made visible externally can only be called by using the object's interface. The section in Laszlo stresses that only allowing interaction with an object through its public interface, using getters and setters for example, is a way of protecting both a server and its clients.

Laszlo displays the way that information hiding can protect the client through the use of his second implementation of an up counter.

```
1.  public class UpCounter2 {
2.    private int value;
3.    public UpCounter2()
      {
          a.      value = Integer.MIN_VALUE;
      }
4.    public void inc()
       {
          a.      value++;
        }
5.    public int value()
       {
          a.      return value - Integer.MIN_VALUE;
      }
}
```

On line 2 a datafield called value is defined; on line 5 a method called value (it would perhaps be better named getValue) is defined. The former is declared to be private, while the latter is public. Hence, it is only the method that is available. The method returns exactly what the specification says: it would initially output 0, and then 1 more after each successive increment. On the other hand, the field called value does not begin at 0 but at the minimum value of Int. Therefore, although it is a correct implementation, direct references to the fields will give you unexpected results.

3.4 Graphics2D: Graphical Programming in Java

In section 3.4 of Laszlo, you will learn the rudiments of drawing pictures in Java's Graphics2D API (Application Program Interface). You will then see how to use these to draw rectangles. From the subject guide, you will learn a new implementation of the bouncing ball program.

The usual way of doing graphics in Java differs fundamentally from the bouncing ball programs we developed in Chapter 1. The way we dealt with the drawing surface, Canvas, in Chapter 1 was atypical. We developed the application that way in Chapter 1 because it introduced object-oriented structure in a simple way. However, once you have learned about the Java graphics models, we will be able to develop much the same behaviour in a more flexible and powerful system. The new implementation will be, in some ways, more complicated but it will serve more truly as a model for the graphical programs we will study later in the course.

There are some complications here that you may not understand the first time you read through the program. Do not panic. These ideas will all be explained later in the course, and we shall refer you back to this example when they are.

Activity

You should now read section 3.4 of Laszlo, which will teach you the fundamentals of the Graphics2D Application Program Interface, and then come back to the subject guide.

3.4.1 After Reading Laszlo, Section 3.4

Before we proceed with the examples, we will reiterate a couple of points from the text. Rendering, the process of drawing graphics to an output device, has four steps:

i. Acquiring a graphics2D object that represents the drawing surface

ii. Constructing the content of that which will be rendered. This includes shapes, text and images

iii. Setting attributes of the graphics. These include fonts, pixel paint, etc.

iv. Invoking a rendering method.

We will explain all of these processes by examining a rectangle-painting program from Laszlo.

3.5 Painting Rectangles

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4. import java.awt.geom.*;
5. import banana.*;
```

The awt and swing packages are there because they contain Java's prebuilt drawing object classes. The last package is one that was written by Laszlo to accompany the book. We have already drawn on it and will, later, draw on it extensively. Recall that we had to set *BlueJ* in the beginning to include the banana package among the places it looks for code.

```
6. public class PaintOneRectangle extends ApplicationPanel {
```

Notice that the class is a child class of ApplicationPanel. ApplicationPanel is Laszlo's own class and it represents a top-level window with a border and title bar, with some functionality. This class forms the basis of most of the applications that will be developed throughout the rest of the course. In this and later applications (including the second version of the bouncing ball program), an application panel will be the principal class and contain whatever is to be drawn on it.

```
7.  public static void main(String[] args) {
      a.  parseArgs(args);
      b.  ApplicationPanel panel = new PaintOneRectangle();
      c.  ApplicationFrame frame = new ApplicationFrame("A
          Rectangle");
      d.  frame.setPanel(panel);
      e.  panel.makeContent();
      f.  frame.show();
      }
```

This is the method that gets called when the application is called automatically: it, therefore, lays out the sequence of events of the application running. When the program runs: in (a) the inputs—which are by necessity Strings—are read and turned into the kind of input that the program really needs; in (b), a new ApplicationPanel object is made (a Panel is a container object that belongs to a Frame—in this case it will be the container in which all of the drawing is done) ; in (c), a new ApplicationFrame object is made—this may seem a bit confusing since

we are in a child class of ApplicationFrame, however what is happening here is that the static method main operates in the class to create a new object which will be the principal object in the application; in (d), the panel is set in the frame; in (e) we will make the content that will go in the panel (which is our drawing surface); in (f) we show the frame which will call the PaintComponent method of each of the objects that belong to containers that belong to the frame. This sequence of events is quite general purpose for drawing things: we will see it many times. For now, we need to implement the procedures that are specific to this application.

```
8.  protected static int x, y, width, height;
```

These are the fields associated with the rectangle. They represent position and size.

```
9.  public static void parseArgs(String[] args) {
    a.      if (args.length != 4) {
        i.          System.out.println("USAGE: java
                            PaintOneRectangle x y width height");
        ii.         System.exit(1);
            }
    b.      x = Integer.parseInt(args[0]);
    c.      y = Integer.parseInt(args[1]);
    d.      width = Integer.parseInt(args[2]);
    e.      height = Integer.parseInt(args[3]);
      }
```

Here the arguments are turned into the kind of arguments we need. If the wrong number of arguments is input, 9ai–ii ensures that the system will stop and print out an explanation. b–e takes in the four arguments and turns them into ints. We have seen this behaviour before—in TryPoints—and it comes up quite frequently. At line b, for example: the first input—the first element of the args array, args[0]—is read and the class method that belongs to the Integer class, Integer.parseInt, is applied to the string, turning it into an int. The datafield x is then assigned that int value.

```
10. protected RectangleGeometry rectangle;

11. public void makeContent()
      {
    a.      rectangle = new RectangleGeometry(x, y, width, height);
      }
```

This is where the content is made. Line 10 is declaring a data field: an object of class RectangleGeometry called *rectangle*. Line 11 is overriding an ApplicationFrame method. If we use the makeContent() method, the system will know that that is the content of the panel.

```
12. protected void paintComponent(Graphics g)
      {
    a.      super.paintComponent(g);
```

```
b.      Graphics2D g2 = (Graphics2D)g;
c.      g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                    RenderingHints.VALUE_ANTIALIAS_ON);
d.      g2.setPaint(Color.green);
e.      g2.fill(rectangle.shape());
 }
```

This is the method for painting on the screen. It has a graphics object as its argument. A graphics object represents anything that can be drawn to: a screen, a printer, etc. This is a rather subtle matter in Java. You will never really have to supply this graphics object directly. By using ApplicationFrames, which are children of JFrames, the system will take care of getting the graphics object for you. You will find that if you try running paintComponent directly in *BlueJ*, that you will be asked to supply a graphics object, but if you run main there will be no problem.

In (a), the paintComponent method from ApplicationFrame is called; that will paint everything but the content of the particular application: in this case, the rectangle. In (b), the graphics object is **cast** to turn it into a Graphics2D graphics object. This will allow us more control. The reason that paintComponent still works with graphics objects, which should be superseded by Graphics2D objects, is to stop old programs from becoming obsolete. This is why we have to provide a Graphics object and then cast it as a Graphics2D object. This works because Graphics2D is a child of Graphics. In (c), particular configurations of the painting are set. In (d), the paint colour is chosen, and in (e) the rectangle is coloured with the chosen colour: green. Anything that implements the shape Interface can be painted in two ways: it can be stroked (which outlines the shape) or filled.

You should now open *BlueJ* and, either starting a new project, or with an existing one, compile PaintOneRectangle.java and run the class. Remember, to run a class from *BlueJ*, you right click the class and choose the static method main. (If you have a problem compiling the program—specifically if you are told that banana.jar is unknown—you should go back to the Appendix, and follow the last set of instructions.) Now, change the values of some of the datafields, and note the effects. You can also try running PaintOneRectangle from the console.

Activity

You should now do exercise 3.7 in Laszlo.

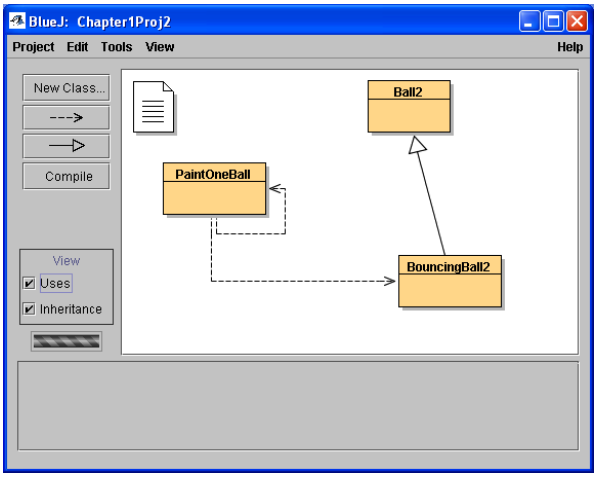
3.6 A different implementation of a bouncing ball

We are now ready to develop the second version of our Bouncing Ball program, following the implementation of PaintOneRectangle.

The main difference between this implementation and the one in Chapter 1 is in the relationship between the balls and the drawing surfaces: in the first version the balls used the drawing surfaces; here the drawing surfaces will use the ball.

Java has several types of drawing surface, all of which have a great deal of inbuilt behaviour. When we build applications, we tend to start with a child class of one of these drawing surfaces, which allows us to draw on the inbuilt behaviour. All of this will get clearer as we go along.

First of all, turn to *BlueJ* and make a project, call it something like Chapter1Proj2, and add the classes Ball2, BouncingBall2, and PaintOneBall from the ***Chapter 3*** folder in the source code supplied in the course CD. Compile the files. Your *BlueJ* window should look something like this:



Notice that whereas in the first project, Ball used Canvas, here PaintOneBall uses BouncingBall2.

3.6.1 Ball2

We will build up this project by looking at each of the classes in turn. We will start with Ball2:

```
1.  import banana.*;

2.  public class Ball2 extends EllipseGeometry
     {

3.  public Ball2 (int x, int y, int width, int height)
      a.    {
      b.      super(x,y,width,height);
      c.    }
     }
```

Line 1 is an Import statement, but it is not importing a package from the standard Java system. Instead, it is importing a package that has been defined by Laszlo to underlie the geometry applications in his book.

Line 2 is announcing the class that we are defining, and it is also declaring it to be a child class of EllipseGeometry, which is one of the classes in the banana package. Among other things, our new class will inherit from its parent class the ability to paint itself on a screen. You will learn more about EllipseGeometry in Chapter 4.

Lines 4–4c is the Constructor for Ball2. It calls on one of the EllipseGeometry constructors to put in the following data: the positions of the upper, right hand corner and the width and the height.

3.6.2 BouncingBall2

BouncingBall2 adds the movement to Ball2.

```
1.  public class BouncingBall2 extends Ball2
     {

2.     int dx = 20;
3.     int dy = 17;

4.  public BouncingBall2(int x, int y, int w, int h, int xmove, int ymove)
      a.    {
      b.    super(x,y,w,h);
      c.    dx = xmove;
      d.    dy = ymove;
      e.    }

5.    public void bounceOnce()
       {
      a.  int xPosition = this.getPosition().getX();
      b.  int yPosition  = this.getPosition().getY();
      c.  if (( xPosition < 0) ||  xPosition  > (400-this.getWidth()))
      d.  dx = -dx;
      e.  if ((yPosition < 0) || ( yPosition >(400-this.getWidth())))
```

```
    f.   dy = -dy;
    g.   super.translate(dx,dy);
     }
}
```

Lines 2 and 3 give the movement data, which is the new data associated with this class.

The Constructor, lines 4-4e, uses no default values, but instead has all of the data fields as parameters. The first four fields are the same as in Ball2, so we can use super (line 4b) to put them in place. 4c and 4d put in the new fields.

The movement method, bounceOnce, has no inputs or outputs. The logic of bouncing is the same as in the first project, but there are a couple of differences worth pointing out.

Note the assignment statement for xPosition:

```
int xPosition = this.getPosition().getX();
```
[5]

The EllipseGeometry method getPosition() returns a Point, which has a method getX() that returns the int that is the x coordinate of the upper left hand corner.

In 5g, we call the translate method that is inherited from EllipseGeometry. This method, as you may suspect, moves the object dx units to the right and dy units down.

We have only defined a single translation in the movement here, whereas we defined the continuous movement as part of the original BouncingBall class. This is actually forced upon us because we can't draw within this class (we have nothing to draw on), so we can't move and then draw and then move from within this class.

3.6.3 PaintOneBall

We are now ready to talk about the really new class here: the one that represents the screen on which everything happens.

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4. import java.awt.geom.*;
5. import banana.*;
```

[5] Recall that ***this*** refers to the object in which the method is called.

```
6.  public class PaintOneBall extends ApplicationPanel {

7.  int counter = 0;

8.   public PaintOneBall()
      a.  {
      b.     setBackground(Color.black);
      c.   }

9.   public static void main(String[] args)
      a.  {
      b.     parseArgs(args);
      c.     ApplicationPanel panel = new PaintOneBall();
      d.     ApplicationFrame myFrame =
                        new ApplicationFrame("A Ball");
      e.     myFrame.setPanel(panel);
      f.     panel.makeContent();
      g.     myFrame.show();
      h.   }

10.  protected int x = 50, y = 50, dx = 20, dy = 30
11.  protected BouncingBall2 aBall;

12.  public void makeContent()
      a.  {
      b.    aBall = new BouncingBall2(x, y,30,30,dx,dy);
      c.   }

13.  protected void paintComponent(Graphics g)
        {
      a.     super.paintComponent(g);
      b.     Graphics2D g2 = (Graphics2D)g;
      c.   g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
      d.                  RenderingHints.VALUE_ANTIALIAS_ON);
      e.     g2.setPaint(Color.green);
      f.     g2.fill(aBall.shape());
      g.      if (counter < 300)
      h.        {counter ++;
      i.         aBall.bounceOnce();
      j.         repaint();}
        }
     }
```

All of this should look quite familiar either from the bouncing ball examples from Chapter 1 or the PaintOneRectangle example above, until you get to 14h, when something interesting and new happens. The combination of the if statement at g and the increment statement at h have the effect of a loop that loops 300 times and then stops. But that's not the interesting part. The interesting part is in lines 14j. Here the program calls the method repaint() that is defined in ApplicationFrame. Repaint calls the paintComponent method; therefore this method ends up calling itself. Therefore, this is a kind of recursion. It is a species of recursion that comes up

very frequently in Java graphics programming. The thing that stops it eventually is that the counter keeps incrementing and you will only repaint until the counter gets to 300.

3.7 Summary of the Chapter

This Chapter was about Abstract Data Types and their implementation. The Abstract Data Types continues the strand of thinking abstractly about programs. The chapter also contains a discussion of the Java mechanisms for abstracting: abstract classes and interfaces. A notation was introduced for specifying abstract data types: skeletons.

You also learned about encapsulation and hiding in their uses in protecting both the server and the client. You saw the distinction between the properties of a data type and its data fields. This was brought out in the RectangleGeometry where none of the properties—Position, Width, and Length—are stored as a data field.

In this chapter we saw the first use of ApplicationFrames as a way of drawing on a screen. The example, which only drew one rectangle on the screen, was abstracted to form a template that we later used to make more sophisticated drawings.

3.8 Learning outcomes

By the end of this chapter and the relevant reading, you should be able to answer examination questions on:

- The Notion of an Abstract Data Type
- The Distinction between Abstract Classes and Interfaces
- The notions of properties, accessors, and mutators
- Equality of Objects
- The Rudiments of the Graphics2D Rendering Model
- Graphics programming using Frames and Panels to draw pictures.