UNIVERSITY OF LONDON | INTERNATIONAL PROGRAMMES

Graphical object-oriented and internet programming in Java Volume 2

T. Blackwell

CO2220

2009

Undergraduate study in
Computing and related programmes

This is an extract from a subject guide for an undergraduate course offered as part of the University of London International Programmes in Computing. Materials for these programmes are developed by academics at Goldsmiths.
For more information, see: www.londoninternational.ac.uk

Goldsmiths
UNIVERSITY OF LONDON

This guide was prepared for the University of London International Programmes by:

Tim Blackwell

This guide was produced by:

Sarah Rauchas, Department of Computing, Goldsmiths, University of London.

This is one of a series of subject guides published by the University. We regret that due to pressure of work the author is unable to enter into any correspondence relating to, or arising from, the guide. If you have any comments on this subject guide, favourable or unfavourable, please use the form at the back of this guide.

University of London International Programmes
Publications Office
32 Russell Square
London WC1B 5DN
United Kingdom

www.londoninternational.ac.uk

Published by: University of London

Contents

Preface

Introduction

The course is split into two parts, with a separate volume for each part. This volume constitutes the second part of the course. Part I (Volume 1) covers Object-Oriented programming in Java, graphical user interfaces and event-driven systems. Part II, in this volume, is concerned with the principles of client-server computing, techniques of interconnectivity in Java and interactive web-based computing systems.

Aims

The course as a whole aims to give students an insight into the object-oriented approach to the design and implementation of software systems. The course also considers specific features of the programming language Java, in particular, graphical interfaces and event driven applications.

The second part of the course, which is covered in this volume, is intended to give students the necessary background to understand the technical software aspects of how computers communicate across the internet. Students will be introduced to the underlying principles of client-server computing systems and will gain the required conceptual understanding, knowledge and skills to enable them to produce simple web-based computing systems in Java.

Objectives

1. To re-enforce students' knowledge of object-oriented programming in Java. (Part I)
2. To introduce students to the notion of graphical user interfaces. (Part I)
3. To introduce students to the notion of event-driven systems. (Part I)
4. To teach students the principles of client-server computing. (Part II)
5. To introduce the main techniques for interconnectivity in Java. (Part II)
6. To produce students able to develop rudimentary interactive web-based computing systems. (Part II)

Learning outcomes

On completion of this course students should be able to:

1. Analyse and represent problems in the object-oriented programming paradigm. (Part I)
2. Design and implement object-oriented software systems. (Part I)
3. Build an event-driven graphical user interface. (Part I)
4. Explain the main principles for client-server programming. (Part II)
5. Design and implement rudimentary client side system. (Part II)
6. Design and implement a rudimentary server-side system. (Part II)
7. Integrate his or her knowledge and skills to produce a rudimentary web-based application. (Part II)

Assessment

Coursework contributes 20 per cent of the final mark on the complete unit. An unseen examination paper will contribute 80 per cent of the final mark.

Important note. The information given above is based on the examination structure used at the time this guide was written. Please note that subject guides may be used for several years. Because of this we strongly advise you to always check the current Regulations for relevant information about the examination. You should also carefully check the rubric/instructions on the paper you actually sit and follow those instructions.

How to use this subject guide

This subject guide is not a self-contained account, but is a companion to the course text *Head First Java* 2nd edition (*HFJ*) by Kathy Sierra and Bert Bates. **It is essential that you obtain this book**.

It will also be helpful to have access to *Java Network Programming* (*JNP*) by Elliotte Rusty Harold.

There are a number of other books listed in the section below called **Reading**, which expand on a number of topics and you are advised to deepen your understanding by referring to these additional texts where directed.

There are eleven chapters in this guide.Most chapters are in two parts. The first part is based on a number of readings from *HFJ*. The second part is devoted to programming. Here you will find programming examples and activities based on the material covered in the first part of the chapter.

In short, the chapters are comprised of

- readings from *HFJ* (not Chapter 3), followed by a summary of the key points from each reading

- a bulleted chapter summary
- programming
- learning outcomes.

It is important that you read *HFJ* when directed, and then read the commentary to check that you have understood the main points. The commentaries are not sufficient in themselves. You must refer to *HFJ* and you are recommended to engage with the many interesting activities that the authors suggest. The chapter summaries collect together the main points. All chapter summaries are reproduced in the revision chapter. These summaries can be used to ensure that you are on top of the material, and as a revision guide.

The programming sections are an integral part of the course. Aside from the program examples, you will find programming activities. You **must** attempt these activities before reading on. The activities are followed by a programming solution. Please realise that there is rarely a unique solution to a programming exercise, so do not feel disheartened if your solution differs from mine. However you should read my program, and the accompanying commentary, to understand my solution. Moreover, new material, especially concerning Java graphics, will be found in the commentaries.

The CD-ROM The accompanying CD-ROM provides all the source code and compiled programs. Many activities are centred on making a drawing or an animation. You should run these programs before attempting the activity so that you can see what to aim for (but do not peek at the code!).

The CD-ROM contains the following:

- an Index which serves to navigate through the folders
- demonstration programs
- source and compiled code for all programming examples and exercises
- source and compiled code for Goo, a special animation package developed for this course
- the Goo API, which is the reference document for the Goo code
- the Java API, which is the reference document for the Java code.

You may wish to develop your programs with an integrated program development environment (IDE) such as Eclipse. It is beyond the scope of this course to show you how to use Eclipse but it is well worth investing some time in learning to use this valuable programming tool yourself. Eclipse can be downloaded for free from `http://www.eclipse.org/`. This guide tells you how to write code in a simple editor and compile and run from the command line. However an IDE such as Eclipse simplifies many programming tasks and greatly helps with debugging.

At the end of each volume, there is a revision guide that summarises the main concepts you should have acquired from each chapter, and also gives you some sample examination papers that can guide some of your study.

Reading

The following is a list of essential and supplementary reading.

Essential reading

Head First Java (second edition), Kathy Sierra and Bert Bates (Sebastopol, Calif.: O'Reilly 2005) [ISBN 0596009208 (pbk)].

Recommended reading

Java in a Nutshell, David Flanagan (Sebastopol, Calif.: O'Reilly, 2005) [ISBN 0596007736].
Learning Java, Patrick Niemeyer and Jonathon Knudsen (O'Reilly, 2005).
Effective Java (second edition), Joshua Bloch (Upper Saddle River, NJ; Harlow: Addison Wesley, 2008) [ISBN 0321356683 (pbk)].
Java Network Programming, Elliotte Rusty Harold (Sebastopol, CA; Farnham: O'Reilly, 2005) [ISBN 0596007213 (pbk)].
Java Cookbook, Ian F. Darwin (O'Reilly Media Inc., 2004) [ISBN 0596007019; 978-05960007010].

Notation

`Java keywords, source code, variable names, method names and other source code` are printed in typewriter font. **Filenames, directories and the command line** in bold type. Three dots, . . . , denotes omitted source code in a code excerpt. Concepts and things, where they are distinguishable from their representative Java names (classes , interfaces, method names . . .), are printed in normal type.

Before you do anything else

Insert the CD-ROM into your computer, open the **demo** folder, and run ManyWorldsApp.

You are watching the flight of clouds of blobs (left hand world) and flapping polygons (right hand world). The two worlds are connected by two 'worm holes'. The exit and entrances to the worm holes are shown by the dark grey discs. Objects from either world, when flying over the exit disc in their world will 'fall' into the other world, appearing at the exit disc.

You will observe that the objects are mini-goo animations, similar to those you coded in Volume 1, except that these mini animations move across the screen to form a larger animation. You can think of each type of mini-animation as a creature.

In this demo, the two worlds are launched from a single application on a single computer.

Imagine instead that the worlds are running on separate machines, connected by worm holes that extend across the Earth.

Imagine that there are many worlds connected by a tangle of wormholes, and many different species of goo creature populating the worlds. You are watching your world, awaiting the arrival of an exotic species. Perhaps you are designing your own species, introducing creatures of this species into your world, and letting them disperse throughout the goo'niverse.

Chapter 1

Advantages

Essential reading

JNP Chapter 1.

1.1 Introduction

Java is the first language designed with networks in mind. Java provides solutions to many network problems such as platform independence, security and international character sets. And thanks to the extensive API, little code is needed even for full-scale applications. In brief, some of the peculiar advantages of Java are:

1. Java applications are safer than off-the-shelf software.
2. Network programs in Java are (relatively) easy to write.
3. Java uses threads rather than processes; this is important for scalable web servers.
4. Java has an exception-handling mechanism; this is important for web applications that need to run continuously.
5. Java is object oriented, and all the software engineering principles that you saw in Volume I can be employed.
6. Java is a full language, rather than a scripting language, and is therefore very versatile. Java has only a small execution-speed disadvantage compared to C++, which is another object language.
7. Java has an extensive Java API, especially for networking. The important packages in this regard are:
   - java.io
   - java.nio
   - java.net
   - java.applet
   - java.rmi
   - javax.servlet
   - javax.servelt.http
   - java.lang.Thread.

1.2 Important networking features of the Java Programming Language

There are four important parts of the JPL which we need to find out about: Threads, Java IO, Serialization and Exception Handling. These features play a vital role in Java network programming. Additionally, some further work on Swing will enhance your graphical interfaces.

This volume begins therefore with these Java techniques (which aren't restricted to networks, but are applicable to all Java programming). The course then continues with some fundamental networking: how to connect to a host, writing an internet browser, a spider and peer-to-peer messaging. The course ends with an introduction to distributed computing.

But before we can do any of that, there are some general programming aspects that we need to look at.

1.3 Learning outcomes

By the end of this chapter, the relevant reading and activities, you should be able to:

- describe the advantages of Java as a programming language
- describe the main networking features of Java: threads, Java IO, serialisation and exception handling.

Chapter 2

Statics

Essential reading

HFJ Chapter 10 (not number formatting or static imports).

2.1 Introduction

This chapter covers some aspects of 'general programming' in Java. By this we mean useful, non-object Java techniques. Many useful programs can be written as a sequence of procedures (method calls) and do not need sophisticated data structures. Top-down programming is sufficient in many situations.

2.2 Statics

Reading: pp. 273–281 of *HFJ*.

A mathematical function such as $ROUND$ does the same thing every time it is invoked. There is no associated state. These functions are pure behaviour. Such functions are represented by static methods in Java.

A *static* method can be invoked without any instances of the method's class on the heap. Static methods are used for utility methods that do not depend on a particular instance variable value. This means that static methods are not associated with any particular instance of that class.

A static method (such as `main`) cannot access a non-static (i.e. instance) method. If you write a class with only static methods then it usually makes no sense to instantiate objects of that class. You can prevent instantiating by marking the constructor as `private`.

There is only one copy of a static variable and it is shared by all members of the class. A static method can access a static variable.

2.3 Final

Reading: pp. 282–284 of *HFJ*.

Some quantities such as the speed of light, or the value of pi just do not change. Constants such as these are marked `static final`.

A final static variable is either initialised when it is declared, or in a static initializer block,

```
static{
    SPEED_OF_LIGHT = 299792458;
}
```

Static finals are conventionally named in uppercase with underscores separating words. Note that this is a convention, rather than contributing to the semantics of the language.

A final variable cannot be changed after it has been initialised. An instance variable can also be marked final. It can only be initialised in the constructor or where it is declared.

Methods and classes may also be final. A final method cannot be overridden and a final class cannot be extended.

2.4 Maths

Reading: pg. 286 of *HFJ*.

`java.lang.Maths` has two static final variables (constant variables, that is), and various static methods. The methods correspond to mathematical functions such as $SQUAREROOT$, COS and $ROUND$.

2.5 Wrapping a Primitive

Reading: pg. 287 of *HFJ*.

All primitive values can be wrapped into objects, and wrapper reference classes can be unwrapped:

```
Integer intObj = new Integer(i);
...
int j = Integer.intValue(intObj);
```

2.6 Autoboxing

Reading: pp. 288–291 of *HFJ*.

Java 5 and beyond supports autoboxing: the automatic wrapping and unwrapping of values in method arguments, return values, Boolean expressions, operations on numbers, assignments, list insertion and removal . . . in fact almost anywhere a primitive or a wrapper type is expected.

For example,

```
Integer i = 42;
i++;
```

```
...
Integer j = 5;
...
Integer k = i + j;
```

Note that `+, ++` operators are NOT defined to act on objects. In fact the compiler will convert wrapper objects to their primitive types before the operations are applied.

2.7 Wrapper methods

Reading: pp. 292–293 of *HFJ*.

Wrappers have static utility methods e.g.

```
String s = "2997792458";
int c = Integer.parseInt(s);
...
double kmPerSec = c / 1000.0;
s = Double.toString(kmPerSec);
```

2.8 Number formatting

Reading: pp. 294–301 of *HFJ*.

In a gesture of conciliation towards C programmers, Java 5 has introduced number formatting in print statements. The syntax is similar to C and C++'s `printf`.

2.9 Dates

Reading: pp. 302–306 of *HFJ*.

`Date()` is fine for getting today's date, as in `Date today = new Date();` but use `Calendar` for date manipulation. In fact `Calendar` is abstract. You can obtain an instance of a concrete subclass by calling a static method:

```
Calendar c = Calendar.getInstance();
```

2.10 Static imports

Reading: pp. 307–309 of *HFJ*.

A static class or a static variable can be imported in an effort to save typing. This can make code easier to read, but may create name conflicts. Only use a static import if the static member is called often from within your class, and you are sure there are no naming collisions.

2.11 Summary

- A static method can be called using the class name rather than an object reference variable.
- A static method can be invoked without any instances of the method's class on the heap.
- Static methods are used for utility methods that do not depend on a particular instance variable value.
- Static methods are not associated with any particular instance of that class.
- A static method (such as `main`) cannot access a non-static (i.e. instance) method.
- Mark the constructor as `private` if you wish to prevent clients instantiating the class.
- There is only one copy of a static variable and it is shared by all members of the class.
- A static method can access a static variable.
- Java constants are marked `static final`.
- A final static variable is either initialised when it is declared, or in a static initializer block.
- Static finals are conventionally named in uppercase with underscores separating words.
- A final variable cannot be changed after it has been initialised.
- An instance variable can also be marked final. It can only be initialised in the constructor or where it is declared.
- A final method cannot be overridden.
- A final class cannot be extended.
- `java.lang.Math` only has static methods. Some very useful methods in this class are: `random(), abs(), round(), min(), max()`. These, and others are listed in the API.
- All primitive values can be wrapped into objects (`Integer intObj = new Integer(i);` and unwrapped: `(int i = Integer.intValue(intObj);`.
- Java 5 and beyond supports autoboxing: the automatic wrapping and unwrapping of values.
- Wrappers have static utility methods e.g. `Integer.parseInt(s)` and `Double.toString(kmPerSec);`
- A format string uses its own special little language; the format string enables precise control of number printing.
- Date is fine for getting today's date, but use `Calendar` for date manipulation.
- Calendar is abstract: get an instance of a concrete subclass like this: `Calendar c = Calendar.getInstance();`
- Static imports save typing, but can lead to name collisions.
- Static methods encourage the procedural style of programming.

2.12 Programming

The Math class has many useful operations on numbers. Suppose we wish to define a similar utility class for *vectors*. Here are some common vector functions:

An n-dimensional vector **a** is a list of real (components), $\mathbf{a} = [a_1, a_2 \ldots a_n]$.

A vector can be multiplied by a number, s, $s\mathbf{a} = [sa_1, sa_2 \ldots sa_n]$.

The vector *dot product* is $\mathbf{a} \cdot \mathbf{b} = [a_1b_1 + a_2b_2 + \ldots + a_nb_n]$;

and the *cross product* in three dimensions is defined as
$\mathbf{c} = \mathbf{a} \times \mathbf{b} = [a_2b_3 - a_3b_2,\ a_3b_1 - a_1b_3,\ a_1b_2 - a_2b_1]$.

Vectors can be *added* and *subtracted* (component by component).

The `length` of a vector is calculated from Pythagoras' formula: $|\mathbf{a}| = \sqrt{\mathbf{a}.\mathbf{b}}$

and vectors can be *normalised* to unit length. If $\hat{\mathbf{a}}$ is the normalised form of **a** then $|\hat{\mathbf{a}}| = 1$.

Given point $A = (a_1, a_2 \ldots a_n)$ then $\mathbf{a} = [a_1, a_2 \ldots a_n]$ connects O to A i.e. $\mathbf{a} = \texttt{OA}$. The *distance* between A and B is therefore $|\mathbf{b} - \mathbf{a}|$.

The *angle* θ between **a** and **b** can calculated from the relation: $\mathbf{a} \times \mathbf{b} = |\mathbf{a}||\mathbf{b}|cos\theta$

Learning activity

Write a vector utility class, `VecMath`. The class should implement the vector functions listed above.

2.13 Vector Maths

```
package numbersandstatics;

import static java.lang.Math.*;

public class VecMath {

  private VecMath() {
  }

  public static double length(double[] a) {

    return sqrt(dot(a, a));
  }

  public static void normalise(double[] a) {

    double mag = length(a);
    for (int i = 0; i < a.length; i++) {
      a[i] = a[i] / mag;
    }
```

```
}

public static double distance(double[] a, double[] b) {

  return length(subtract(a, b));
}

public static double[] add(double[] a, double[] b) {

  int numCpts = a.length;
  double[] c = new double[numCpts];
  for (int i = 0; i < numCpts; i++) {
    c[i] = a[i] + b[i];
  }
  return c;
}

public static double[] subtract(double[] a, double[] b) {

  int numCpts = a.length;
  double[] c = new double[numCpts];
  for (int i = 0; i < numCpts; i++) {
    c[i] = a[i] - b[i];
  }
  return c;
}

public static double[] mult(double scalar, double[] a) {

  double[] b = new double[a.length];
  for (int i = 0; i < a.length; i++) {
    b[i] = scalar * a[i];
  }
  return b;
}

public static double[] cross(double[] a, double[] b) {

  if (a.length == b.length && b.length == 3) {
    double[] c = new double[3];
    c[0] = a[1] * b[2] - a[2] * b[1];
    c[1] = a[2] * b[0] - a[0] * b[2];
    c[2] = a[0] * b[1] - a[1] * b[0];
    return c;
  } else
    return new double[0];
}

public static double dot(double[] a, double[] b) {

  int numCpts = a.length;
  double result = 0.0;
  for (int i = 0; i < numCpts; i++) {
    result += a[i] * b[i];
  }
  return result;
}

public static double angle(double[] a, double[] b) {
```

```
    return acos(dot(a, b) / (length(a) * length(b)));
  }
}
```

The methods are a straightforward implementation of the common vector operations. The constructor has been marked private to prevent instantiations of `VecMaths`, and `java.lang.Math` has been statically imported. (This static import is only of marginal benefit since Math is only called twice.)

Learning activity

Write a test class for your VecMath, and test each method. Numerical output should contain four decimal places. Use static imports wherever appropriate.

2.14 Vector maths test program

```
package numbersandstatics;

import static java.lang.Math.*;
import static java.lang.System.out;
import static java.lang.String.format;
import static numbersandstatics.VecMath.*;

public class VecMathTest {

  public static String vecFormat(String formatStr, double[] a) {

    String s = "(";
    for (int i = 0; i < a.length - 1; i++) {
      s += format(formatStr, a[i]) + ", ";
    }
    s += format(formatStr, a[a.length - 1]) + ")";
    return s;
  }

  public static double toDegrees(double radians) {

    return (180 / PI) * radians;
  }

  public static void main(String[] args) {

    // initialise two vectors
    double[] a = { 5, 0, 0 };
    double[] b = { 5 * cos(PI / 6), 5 * sin(PI / 6), 0 };

    // format String
    String formatStr = "%.4f";

    // print a = OA and b = OB
    out.println("a = " + vecFormat(formatStr, a) + ", " + "b = "
        + vecFormat(formatStr, b));

    // print lengths of a and b
```

```
    out.println("|a| = " + format(formatStr, length(a)) + ", |b|
        = "
         + format(formatStr, length(b)));

    // find unit vector pointing along b
    double[] c = { 3, 4 };
    normalise(c);
    out.println("c hat = " + vecFormat(formatStr, c));

    // is it really a unit vector?
    out.println("length of c hat = " + format(formatStr, length(c
        )));

    // vector addition and subtraction
    c = add(a, b);
    out.println("a + b = " + vecFormat(formatStr, c));
    c = subtract(a, b);
    out.println("a - b = " + vecFormat(formatStr, c));

    // scalar multiplication
    c = mult(0.5, b);
    out.println("0.5b = " + vecFormat(formatStr, c));

    // dot product
    out.println("a.b = " + format(formatStr, dot(a, b)));

    // distance between A and B
    out.println("dist AB = " + format(formatStr, distance(a, b)))
        ;

    // cross product
    c = cross(a, b);
    out.println("c = a x b = " + vecFormat(formatStr, c));

    // is c at 90 degrees to a and b?
    out.println("a.c = " + dot(a, c) + ", b.c = " + dot(b, c));

    // angle AOB
    double angle = toDegrees(angle(a, b));
    out.println("angle AOB  = " + format(formatStr, angle));
  }
}
```

`main` initialises two test vectors and systematically calls each method in VecMath. Some static imports shorten the code; probably justifiable in a small program such as this. However one problem did become apparent when coding: how to format the elements of a numerical array? The solution used here is to write a utility method that applies String.format to each element in turn. In order to prevent naming ambiguity, this method was not named `format`.

Learning activity

The above tests were too conservative. There are two problems with the implementations within VecMath which could easily cause a client program to crash. Can you spot them? Write more tests to demonstrate which methods are unsafe and what happens when they are called by a careless client.

2.15 Better test program

```
package numbersandstatics;

import static java.lang.Math.*;
import static java.lang.System.out;
import static java.lang.String.format;
import static numbersandstatics.VecMath.*;

public class VecMathUnsafeTest {

  final static int TEST_0 = 0;
  final static int TEST_1 = 1;
  final static int TEST_2 = 2;

  public static String vecFormat(String formatStr, double[] a) {

    String s = "(";
    for (int i = 0; i < a.length - 1; i++) {
      s += format(formatStr, a[i]) + ", ";
    }
    s += format(formatStr, a[a.length - 1]) + ")";
    return s;
  }

  public static double toDegrees(double radians) {

    return (180 / PI) * radians;
  }

  public static void main(String[] args) {

    int test = TEST_0;
    if (args.length > 0)
      test = Integer.parseInt(args[0]);

    double[] a = new double[] { 5, 0, 0 };
    double[] b = new double[] { 5 * cos(PI / 6), 5 * sin(PI / 6),
        0 };

    switch (test) {
    case TEST_0:
      b = new double[] { 5 * cos(PI / 6), 5 * sin(PI / 6), 0 };
      break;
    case TEST_1:
      a = new double[] { 5, 0, 0, 1 };
      break;
    case TEST_2:
      b = new double[] { 0, 0, 0 };
      break;
    default:
    }

    // format String
    String formatStr = "%.4f";

    // print a = OA and b = OB
```

```
    out.println("a = " + vecFormat(formatStr, a) + ", " + "b = "
        + vecFormat(formatStr, b));

    // print lengths of a and b
    out.println("|a| = " + format(formatStr, length(a)) + ", |b|
        = "
        + format(formatStr, length(b)));

    // find unit vector pointing along b
    double[] c = {3, 4};
    normalise(c);
    out.println("c hat = " + vecFormat(formatStr, c));

    // is it really a unit vector?
    out.println("length of c hat = " + format(formatStr, length(c
        )));

    // vector addition and subtraction
    c = add(a, b);
    out.println("a + b = " + vecFormat(formatStr, c));
    c = subtract(a, b);
    out.println("a - b = " + vecFormat(formatStr, c));

    // scalar multiplication
    c = mult(0.5, b);
    out.println("0.5b = " + vecFormat(formatStr, c));

    // dot product
    out.println("a.b = " + format(formatStr, dot(a, b)));

    // distance between A and B
    out.println("dist AB = " + format(formatStr, distance(a, b)))
        ;

    // cross product
    c = cross(a, b);
    out.println("c = a x b = " + vecFormat(formatStr, c));

    // is c at 90 degrees to a and b?
    out.println("a.c = " + dot(a, c) + ", b.c = " + dot(b, c));

    // angle AOB
    double angle = toDegrees(angle(a, b));
    out.println("angle AOB  = " + format(formatStr, angle));
  }
}
```

2.15.1 Test report

Three tests were performed by supplying *0, 1, 2* as an argument to the Java interpreter (i.e. by typing **(java numbersandstatics/VecMathUnsafeTest 0** at the command line).

1. TEST 0

```
a = (5.0000, 0.0000, 0.0000), b = (4.3301, 2.5000, 0.0000)
|a| = 5.0000, |b| = 5.0000
b hat = (0.8660, 0.5000, 0.0000)
```

```
length of b hat = 1.0000
a + b = (9.3301, 2.5000, 0.0000)
a - b = (0.6699, -2.5000, 0.0000)
0.5b = (2.1651, 1.2500, 0.0000)
a.b = 21.6506
dist AB = 2.5882
c = a x b = (0.0000, 0.0000, 12.5000)
a.c = 0.0, b.c = 0.0
angle AOB  = 30.0000
```

Everything is working fine.

2. TEST 1

```
a = (5.0000, 0.0000, 0.0000, 1.0000), b = (4.3301, 2.5000, 0.0000)
|a| = 5.0990, |b| = 5.0000
b hat = (0.8660, 0.5000, 0.0000)
length of b hat = 1.0000
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
at numbersandstatics.VecMath.add(VecMath.java:35)
at numbersandstatics.VecMathUnsafeTest.main(VecMathUnsafeTest.java:70)
```

This test crashes the program. The bug is traced back to VecMaths line 35 where the code assumes that the two vectors have the same number of components.

3. TEST 2

```
a = (5.0000, 0.0000, 0.0000), b = (0.0000, 0.0000, 0.0000)
|a| = 5.0000, |b| = 0.0000
b hat = (NaN, NaN, NaN)
length of b hat = NaN
a + b = (5.0000, 0.0000, 0.0000)
a - b = (5.0000, 0.0000, 0.0000)
0.5b = (0.0000, 0.0000, 0.0000)
a.b = 0.0000
dist AB = 5.0000
c = a x b = (0.0000, 0.0000, 0.0000)
a.c = 0.0, b.c = 0.0
angle AOB  = NaN
```

Some numerical values have been set at `NaN` (not-a-number). This 'value' is the result of dividing by zero. Ensuing calculations will reveal surprising results. The divide by zero error happens whenever a `normalise` and `angle` is called with a vector of zero length.

We shall see one way of fixing these runtime errors in the next chapter.

2.16 Learning outcomes

By the end of this chapter, the relevant reading and activities, you should be able to:

- describe the advantages of Java as a programming language
- describe the main networking features of Java: threads, Java IO, serialisation and exception handling
- explain how a static method can be called or invoked
- describe what static methods are used for

- explain the term **static final**
- describe how static final variables are initialised and how they are usually named
- understand that a final method cannot be overridden and that a final class cannot be extended
- describe how primitive values can be wrapped into objects and how wrapper reference type classes can be unwrapped
- explain what autoboxing is
- understand that + and ++ operators are **not** defined to act on objects
- understand that wrappers have static utility methods
- describe how number formatting works
- understand how a static class or a static variable can be imported, the advantages of this, as well as the limitations
- describe how a static method can be called using the class name, or invoked without any instances of the method's class on the heap
- understand that static methods are used for utility methods that do not depend on a particular instance variable value
- understand that static imports can lead to name collisions
- understand that static methods encourage the procedural style of programming.

Chapter 3

Exceptions

Essential reading

HFJ Chapter 11.

3.1 Introduction

Most of the time you should aim to write safe code so that clients of your classes are not surprised by the messages they get back, and so that your classes do not crash their program.

However, certain programming activities such as asking the operating system to do something (e.g. pause execution for 100 ms) or interfacing with a device (print a file), or connecting to a network are inherently dangerous because your program cannot know exactly what may happen.

Flaws in your own code are known as bugs and we hope that OO techniques can minimise these. But some runtime errors are outside your control. Luckily Java includes a mechanism so that you can prepare for the unexpected and unusual at runtime with special *exception handling* code.

3.2 Catch!

Reading: pp. 319–328 of *HFJ*.

The compiler makes sure that checked-exceptions are caught by your code. For example, the API tells us that the method prototype for `read(byte[] b)` of `java.io.InputStream` is:

```
public int read(byte[] b) throws IOException
```

This means that read throws an exception object which the calling method must 'catch':

```
try{
   int numRead = inputStream.read(byte[] b);
}
catch (IOExeption e){
   System.out.println(e);
}
```

The risky call is placed in a try block. The accompanying catch block contains emergency code in case something goes wrong. The JVM will execute this code in that eventuality. Here, the catch just prints the exception to the terminal, but normally you would want to do something (e.g. wait a while and then try and read from the input stream again). Occasionally you may wish to write a `finally` block, which will run regardless of an exception.

3.3 Multiple exceptions

Reading: pp. 329–324 of *HFJ*.

A method can throw more than one exception and they must all be caught, preferably by one catch block after another. Exceptions are polymorphic and it is possible to catch all the exceptions with a single supertype catch. Certainly a `catch (Exception e)` block will catch everything because all exceptions subclass `java.lang.Exception`, but this is not advisable because your handler will not have precise information about what went wrong. Since the JVM works its way down the list of catch blocks, it is better to place more specific catch calls (i.e. lowest subclasses in the inheritance tree) higher up the list.

An important subclass of Exception is `java.lang.RuntimeException`. Subclasses of this, such as `java.lang.ArrayIndexOutOfBoundsException` (which is thrown to indicate that an array has been accessed with an illegal index), are ignored by the compiler. Runtime exceptions are usually due to faulty code logic, rather than unpredictable or unpreventable conditions that arise when the code is running.

Exceptions should only be used for very unusual circumstances caused by interactions with the outside world, and not by the internal logic of your code.

3.4 Duck!

Reading: pp. 335–357 of *HFJ*.

The thrown exception can be ducked by your code by declaring it, as in

```
public int riskyMethod() throw IOException{

      return inputStream.read(byteArray);
  }
```

The stack diagrams on pg. 336 of *HFJ* show how the duck mechanism works, and if main ducks as well, the uncaught exception will shut the JVM down.

3.5 Relevance to network programming

Network programs must function for many hours without supervision, and are in contact with the outside world. Therefore the exception handling mechanism of Java is invaluable.

3.6 Summary

- A method can throw an exception object if something goes wrong at runtime.
- All exceptions subclass java.lang.Exception.
- The compiler does not check that possible runtime exceptions are handled in your code.
- However the compiler does care about **checked exceptions**, and insists that they are declared or wrapped in a try/catch block.
- A method throws an exception with the keyword throw followed by a new exceptions object.
- If your code calls a checked exception throwing method, you must either duck the exception or enclose the call in a try/catch block.

3.7 Programming

We noticed in the last chapter that our VecMath class could be dangerous if a client called a method with two double[]'s of different size; and if normalise() or angle() is called with a vector of zero length.

We saw from the test report that if an array is accessed outside its defined range a `java.lang.ArrayIndexOutOfBounds exception` is thrown. Division by zero (double) does not raise any exceptions, but sets the result to NaN.

Learning activity

Copy and paste length(), normalise() and dot() methods from VecMaths into a new class, VecMathException. Alter normalise() so the method throws an ArithmeticException if a division by zero is attempted. (Consult the API to find out about ArithmeticException and its superclass, RuntimeException.)

Write a test class which catches any arithmetic and array index out of bounds exceptions thrown by VecMathException.

3.8 Vector maths with exception throwing

```
package exceptionhandling;

import static java.lang.Math.*;

public class VecMathException {

  private VecMathException() {
  }

  public static double length(double[] a) {

    return sqrt(dot(a, a));
```

```
  }

  public static void normalize(double[] a) throws
      ArithmeticException {

    double mag = length(a);
    if (mag == 0)
      throw new ArithmeticException(
          "SafeVecMath.java 18: Attempt to divide by zero");
    for (int i = 0; i < a.length; i++) {
      a[i] = a[i] / mag;
    }
  }

  public static double dot(double[] a, double[] b) {

    int numCpts = a.length;
    double result = 0.0;
    for (int i = 0; i < numCpts; i++) {
      result += a[i] * b[i];
    }
    return result;
  }
}
```

3.9 Catching VecMath exceptions

```
package exceptionhandling;

import static java.lang.System.out;
import static java.lang.String.format;

import static exceptionhandling.VecMathException.*;

public class VecMathExceptionTest {

  public static String vecFormat(String formatStr, double[] a) {

    String s = "(";
    for (int i = 0; i < a.length - 1; i++) {
      s += format(formatStr, a[i]) + ", ";
    }
    s += format(formatStr, a[a.length - 1]) + ")";
    return s;
  }

  public static void main(String[] args) {

    // format String
    String formatStr = "%.4f";

    // dot product with different size vectors
    try {
      double[] a = new double[] { 1, 1, 0 };
      double[] b = new double[] { 1, 1 };
      out.println("a.b = " + format(formatStr, dot(a, b)));
    } catch (ArrayIndexOutOfBoundsException e) {
      out.println(e);
```

```
    }

    // normalise a zero vector?
    try {
      double[] c = { 0, 0, 0 };
      normalize(c);
      out.println("c hat = " + vecFormat(formatStr, c));
    } catch (ArithmeticException e) {
      out.println(e);
    }

  }
}
```

In fact the exception handling mechanism is rather clumsy for this example. VecMath throws only runtime exceptions, which are unchecked by the compiler; hence the programmer must remember to include try/catch blocks without helpful compiler reminders. Furthermore, runtime exceptions are generally used to escape unknowable problems. It is far better to handle internal errors with code (i.e. not exception throwing) which corrects program logic.

For example we might decide that an attempt to normalise a zero vector should not alter the vector in any way and attempts to dot product unequal length arrays result in a returned NaN.

Learning activity

Write a new class, SafeVecMath, with length(), dot() and normalise() methods which check for unequally sized arrays and zero-length vectors. Write a test class to show what happens if a client calls dot and normalise carelessly. Also demonstrate how to write safe client code for calls to these methods.

3.10 Safe vector maths

```
package exceptionhandling;

import static java.lang.Math.*;

public class SafeVecMath {

  private SafeVecMath() {
  }

  public static double length(double[] a) {

    return sqrt(dot(a, a));
  }

  public static void normalize(double[] a) {

    double mag = length(a);
    if (mag == 0)
      return;

    for (int i = 0; i < a.length; i++) {
      a[i] = a[i] / mag;
```

```
    }

  }

  public static double dot(double[] a, double[] b) {

    int numCpts = a.length;
    if (b.length != numCpts)
      return Double.NaN;

    double result = 0.0;
    for (int i = 0; i < numCpts; i++) {
      result += a[i] * b[i];
    }
    return result;
  }
}
```

3.11 Safe test

```
package exceptionhandling;

import static java.lang.System.out;
import static java.lang.String.format;

import static exceptionhandling.SafeVecMath.*;

public class SafeVecMathTest {

  public static String vecFormat(String formatStr, double[] a) {

    String s = "(";
    for (int i = 0; i < a.length - 1; i++) {
      s += format(formatStr, a[i]) + ", ";
    }
    s += format(formatStr, a[a.length - 1]) + ")";
    return s;
  }

  public static void main(String[] args) {

    // format String
    String formatStr = "%.4f";

    // dot product with different size vectors
    double[] a = new double[] { 1, 1, 0 };
    double[] b = new double[] { 1, 1 };
    out.println("a.b = " + format(formatStr, dot(a, b)));

    // normalise a zero vector?
    double[] c = { 0, 0, 0 };
    normalize(c);
    out.println("c hat = " + vecFormat(formatStr, c));

    // how to write safe client code
    if (a.length != b.length) {
      out.println("unequal sized vectors");
    } else {
```

```
      if (dot(a, b) == 0) {
        // carry on...
      }
    }
    if (length(c) == 0) {
      out.println("zero length vector");
    } else {
      normalize(c);
      // carry on...
    }

  }
}
```

SafeVecMathTest shows how the good programmer should prepare for runtime exceptions due to coding errors by what might be termed 'defensive programming'.

In Volume I we saw a situation where we just had to catch an exception; this was the checked InterruptedException, thrown by the JVM if the operating system could not service the request for program execution pause.

Checked exceptions are an important and valuable aspect of internet Java and we shall be using them often in the remainder of this volume.

3.12 Learning outcomes

By the end of this chapter, the relevant reading and activities, you should be able to:

- explain why it is important to write safe code
- understand when it might not be possible to write safe code, and the implications of this
- describe how a method can throw an exception object if something goes wrong at runtime
- understand how all exceptions subclass `java.lang.Exception`
- understand that the compiler does not check that possible runtime exceptions are handled in code
- understand that the compiler does care about **checked exceptions**, and insists that they are declared or wrapped in a `try/catch` block
- describe how a method throws an exception with the keyword `throw` followed by a new exceptions object
- understand that if your code calls a checked exception throwing method, you must either `duck` the exception or enclose the call in a `try/catch` block.