Name _____

COMP160
General Programming

Laboratory Workbook
Semester Two - 2014

Department of Computer Science

University of Otago

Table of Contents

F.A.Q.

What do I do if I miss a lab?

Come to any other lab as soon as you can. As long as you can find a seat that is not needed by somebody streamed to that lab, you are welcome.

How many labs can I miss without failing terms?

You may miss two of your streamed labs before the mid-semester break and two after without failing terms. However, we know that every missed lab increases your chances of failing the course. If you need to miss a lab, make sure you catch it up at another time if at all possible.

If I have a lab marked, do I automatically get terms for that lab?

Not necessarily. Only if it was marked on or before its scheduled time. Completing the lab late does not return a missing terms count, though you will get the lab mark. If the scheduled lab was not marked on or before its scheduled lab time the only way of gaining terms is for you to be logged on to a lab machine during that scheduled lab time.

How long do I have to stay in the lab to get terms for that lab session?

It depends how far behind you are. An hour would be permitted occasionally for someone who is generally up-to-date. Someone who is 2 or more labs behind will need to work on the task for pretty well the whole 1 hour and 50 minutes.

How is attendance for terms determined if the current lab is not marked on time?

The server's login records are used to check attendance duration. If you were logged on to your machine for most of the lab stream you will be considered to have attended the lab. (The other way of attending is having your work marked on or ahead of time).

Can I come to other lab sessions *as well as* my own?

Yes, as long as you can find a seat that is not needed by somebody streamed to that lab. If the lab gets too full, demonstrators might ask you to leave or wait until another machine becomes available.

Can I come to other lab sessions *instead of* my own?

Yes, but we ask you to come to your streamed session first where at all possible. You might miss out on terms for the current lab if you come after your streamed session.

Can I work at home?

Yes. You are welcome to bring a completed lab to your streamed lab session. Having your work successfully marked on time will give you terms for that lab. BUT if you are working at home and get stuck, STOP and come to the lab to ask for help. We suggest that spending any more than one hour battling over any particular issue is a waste of your time.

What can I do if I am falling behind?

Come to the lab more often. Attend as many lab sessions as you need to catch up.

What do I do if I can't understand the lab task?

Read it through carefully without touching a computer. Try to identify the end-point. Talk to a demonstrator.

How should I prepare for my exam?

Having a good understanding of the concepts used in the lab tasks should put you in a very good position. There will be practise exams on Blackboard. Talk over any concepts you don't understand with a demonstrator.

Can I go over my mid-semester exam with someone?

Yes, and you can get a mark for it. The Lab 17 mark can be achieved by **either** learning to use a Command Line Interface, **or** reviewing your corrections on your mid-semester exam paper with a demonstrator.

What do I do if I miss a lecture?

The lecture notes are posted on Blackboard for download. Read the lecture notes **before** you attempt the lab work with the same number.

What do I need to know about the final examination?

The final examination is similar to the mid-semester examination in terms of being a combination of multiple choice questions and short answer questions. It counts 60% towards to your final mark. You need to score at least 50% in the final exam AND have a total mark (including the internal component) of 50% or over to pass the course.

Timetable for COMP160

Day	Lec. No.	Lecture	Lab No.	Mark	Lab Name
July 8th	1	Introduction (Chapter 1)			
July 10th	2	Data types and language basics (Chapter 2)	1	1%	Introduction to Java
July 15th	3	Program structure, methods and basics (Chapter 2)	2	1%	Variables
July 17th	4	Expressions. Arithmetic (Chapter 2)	3	1%	Methods
July 22nd	5	Graphics, drawing and GUIs (Appendix F)	4	1%	Expressions
July 24th	6	Objects 1 and special methods (Chapter 3)	5	1%	Graphics
July 29th	7	Objects 2. Strings (Chapter 3)	6	1%	Objects
July 31st	8	Structured programming, more maths (Chapter 3)	7	1%	Constructors
August 5th	9	Boolean expressions, blocks, if else (Chapter 4)	8	1%	Math and Random
August 7th	10	Selection (Chapter 4)	9	1%	Selection 1
August 12th	11	Repetition (loops) 1, iterators, iterable (Chapter 4)	10	1%	Selection 2
August 14th	12	Repetition (loops) 2 (Chapter 4)	11	1%	Strings
August 19th	13	Objects 3 Classes and methods (Chapter 5)	12	1%	Repetition 1
August 20th	**M I D - S E M E S T E R E X A M**			15%	**Covers Lectures and Labs 1 - 12**
August 21st	14	Objects 4 References (Chapter 5)	13	1%	Repetition 2
August 23rd -31st	**M I D - S E M E S T E R B R E A K**				
September 2nd	15	Arrays 1 (Chapter 7)	14	1%	Graphical Objects
September 4th	16	Arrays 2 references to objects (Chapter 7)	15	1%	Arrays
September 9th	17	Graphics 1 components (Chapter 6)	16	1%	Two-Dimensional Arrays
September 11th	18	Graphics 2 events (Chapter 6)	17	1%	Command Line Interface **OR** Mid-semester exam review
September 16th	19	Graphics 3 examples (Chapter 6)	18	1%	Graphical User Interfaces
September 18th	20	Files input output, sorting (Chapter 10 / readings)	19	1%	Calculator
September 23rd	21	Hierarchies, inheritance (Chapter 8)	20	1%	Reading from Files
September 25th	22	Visibility, overriding. (Chapter 8)	21	1%	Shapes 1: Building the Structure
September 30th	23	Hierarchies, abstract classes (Chapter 8)	22	1%	Shapes 2: Animation
October 2nd	24	Collections, ArrayList, Applets	23	1%	Shapes 3: Abstract
October 7th	25	Simulation. Programming	24	1%	Shapes 4: ArrayLists
October 9th	26	Topics in Computer Science	25	1%	Options

Introduction

Welcome

Welcome to the laboratory sessions for COMP160. Programming is a very practical skill. The best way to learn is to sit down and write programs. This means that the laboratory sessions are probably the central part of COMP160 and the place that you will really learn the art and craft of computer programming. We hope you find the laboratory sessions useful and enjoyable.

For general course information please visit the course blackboard page (accessed from http://blackboard.otago.ac.nz) regularly.

Please make sure you read these introductory pages. They contain some general information which we will assume that you know from now on. You are also required to sign page 8 to show that you agree to abide by the rules for use of both the software and the laboratory.

Contact People

Lectures

Professor Anthony Robins

Office : Room 253, Owheo Building, 133 Union St East

Phone : 479- 8314

Email : anthony@cs.otago.ac.nz

Laboratory

Ms Sandy Garner

Office : Room G11A, Owheo Building, 133 Union St East

Phone : 479-5242

Email : sandy@cs.otago.ac.nz

COMP160 Web Page

http://www.cs.otago.ac.nz/comp160

This site holds preliminary information about the course.

http://blackboard.otago.ac.nz

Blackboard is used for notices, study information and clarifications to lab work. Also, files for lab work can be downloaded from the Course Documents link.

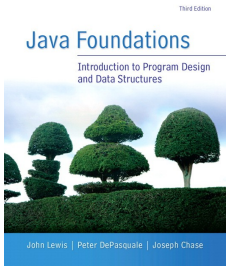
About the Laboratory Sessions and Assessment

The Laboratory Workbook (Lab Book)

This book details the practical work for the 2 laboratory sessions per week required for COMP160. (Each laboratory session is two hours long). A copy of the book is available electronically for free. We prefer you purchase a hardcopy for $20. An alternative is to purchase or print a cut-down version which contains the preparation exercises and planning pages, and use the electronic copy for the lab task instructions. You need to bring one of these books physically with you to each lab session as it makes up part of the work which is marked. We also expect you to have a pen with you.

The Textbook

The text book is:



Java Foundations: Introduction to Program Design and Data Structures

Third edition

John Lewis, Peter DePasquale, Joseph Chase

Addison-Wesley / Pearson Education Inc. 2014

ISBN 10: 0-13-337046-1

ISBN 13: 978-0-13-337046-1

You will need to own a copy, or have ready access to a copy. The labs, including most assessed exercises, are based on the text book (L,D&C). The 2nd edition is also fine to use. Page reference numbers given in this book refer to the 3rd edition first, then the 2nd edition in a smaller font, enclosed in square brackets e.g. page 62 [86].

Labs

Laboratory based work forms 25% of your total assessment for COMP160. There are 25 Labs worth 1% each.

As you can see from the timetable on page 3, both lectures and labs follow the structure of L,D&C (following the chapters more or less in order).

Each lab consists of three sections:

Notes - The notes may highlight points to watch out for while reading the chapter in L,D&C or provide further relevant background notes.

Preparation - These exercises should be completed **before** the scheduled lab session starts. Preparation will usually involve questions taken from L,D&C or the lecture material, and planning for the program which will be written during the lab session. At the very least, you should read through the lab work and understand the task before arriving in the lab.

Lab Work - This is work that should be completed during the lab session. It involves planning and writing a piece of code to perform a specified task.

Assessment

Each lab is marked on an "all or nothing" basis. Preparation, Exercises and Lab Work are assessed. If this work is completed to a satisfactory standard you get the mark for that lab. A "satisfactory standard" for Preparation Exercises means that your answers are mostly correct. For Lab Work it means that the programs you write are working, well designed, and include appropriate comments.

When you have completed the work for a lab session your work will be checked, and your lab book will be signed by a demonstrator. Your code should then be submitted electronically to the COMP160 Drop Box on our server. If your laboratory marks are not entered into our records correctly then your signed lab book is proof, along with the files on our servers, that you did complete the lab.

Labs should be completed in your scheduled lab session. If you cannot complete a lab during that time, you should work on it in your own time. Ideally it should be marked before the **start** of your next lab session. If you are working ahead of schedule – great! – but please don't ask the demonstrators for help with material that has yet to be delivered in lectures.

If you haven't completed a lab and had it marked during the lab session it was assigned, your attendance will be recorded for the purposes of Terms by our machine records.

Terms

To gain terms (to be allowed to sit the final exam) you must attend at least 11 of your first 13 scheduled laboratory sessions (before the break) and at least 10 of your next 12 scheduled labs.

Please do not interpret this as "I only need to complete 21 of the 25 labs". We would expect you to complete all the labs, but 4 of them may be completed to your time schedule rather than ours.

This 4 lab 'freedom' allowance is to allow you to cope with illness, family emergencies and other events that may occur over the semester. Of course, if you use it without good cause, it will not be available if you need it later.

If you have a legitimate need for absence covering more than 2 labs you can apply to the Teaching Fellow for a terms credit. You will need to provide written proof of your circumstances. The workload of other papers you are studying is not sufficient reason for an extension. Managing course workload is a normal part of every student's university experience.

If you have completed your lab work before the end of your lab session, and have had it marked, you do not have to stay for the rest of the session. Submit a copy of your lab code to the COMP160 Drop Box before you leave the lab.

If you have not completed your lab work by the end of your lab session, your attendance in the lab will be recorded by us using our server log records.

Work Load

You will find the laboratory time too short to complete the lab work unless you have attended the lectures and done some preparation beforehand.

COMP160 is an eighteen point paper. This means that during Semester 2 you should expect to spend roughly 12 hours per week working on COMP160. This includes the 6 scheduled hours (2 hours of lectures and 4 hours of laboratories). In other words you have 6 hours per week for your own work on understanding the concepts and preparation for the laboratory sessions. This lab book contains background information and preparation exercises that are designed to be done before each actual laboratory session. It also describes programming exercises to be done during the lab session itself, although you need to think about them **before** you sit down in front of the computer.

Demonstrators may refuse to help you if you haven't completed your preparation exercises, as these are specifically designed to build the knowledge required for your lab work.

The course material builds quickly on itself. Consequently, it is important that you complete and understand the work for one lab session before going on to the next. Some of you may have some programming experience and might feel that the early programming problems are too easy. In this case please try other programming exercises from the text book, or you can work on your own programming projects.

This is a fast moving course – if problems arise it is very important that you ask for help before you fall too far behind. Don't be afraid to ask – we are here to help.

Demonstrators

The demonstrators are there to help you understand the work and help you with any problems. They should give help, especially if you have been stuck for a long time, but they are not there to simply tell you everything or to write the programs for you! They will try to help you work out your own answers to the exercises and develop your own understanding.

Demonstrating is very difficult work and finding the right balance in answering a question is hard. The process is sometimes frustrating for both them and you, so please be patient and understand the reasons for the amount or kind of help that a demonstrator might give.

Illness

If you are ill / contagious, please stay at home until you are well.

Working together

Many students find it helpful to work in small groups, and this cooperation is to be encouraged – you can learn a lot from each other! Feel free to work on the Preparation questions together (this means taking an **active** part in producing the answer).

Shared Work

When you present code for marking, it is very important that all of the code is written by you. It does not help you to learn if you simply copy (plagiarise) a written answer or program. In order to pass COMP160 you must pass the final exam, so gaining your own understanding is essential.

- **Do not copy any portion of another student's code.**
- **Do not lend, send or show your code to any other student.** Copying requires two willing persons, the giver and the taker. If another student copies your code, you will lose your marks as well.
- **Do not assume that changing the variable names and comments makes the copying undetectable.** Several previous students can tell you this is not the case.

If you are falling behind and feeling stressed by the pace of the course, copying someone else's code is not the best solution.

If you are caught plagiarising the matter will be dealt with severely under the University Regulations. We routinely check your code against that of your fellow students, and previous COMP160 students. In order that we may be able to do this, we get you to deliver a copy of your .java files as you get your labs marked. The demonstrators will show you how to do this when it is required.

The University has a clear policy with regard to copying the work of others:

> *Students should make sure that all submitted work is their own. Plagiarism is a form of dishonest practice. Plagiarism is defined as copying or paraphrasing another's work, whether intentionally or otherwise, and presenting it as one's own (approved University Council, December 2004). In practice this means plagiarism includes any attempt in any piece of submitted work (e.g. an assignment or test) to present as one's own work the work of another (whether of another student or a published authority). Any student found responsible for plagiarism in any piece of work submitted for assessment shall be subject to the University's dishonest practice regulations which may result in various penalties, including forfeiture of marks for the piece of work submitted, a zero grade for the paper, or in extreme cases exclusion from the University.*

Access to Java Programming Environments

DrJava is available on both Windows and Mac computers in the Computer Resource Rooms.

DrJava is also available as a free download, for Windows or Macintosh, from `http://www.drjava.org` should you wish to install it on another computer.

In order to run, DrJava needs the Java run-time environment to be installed. This is pre-installed on Macintosh OS operating systems up to 10.6, but if you have a Windows machine or Mac OS 10.7 or above you will need to do this yourself. At the time of going to print, the url

`http://www.java.com/en/download/manual.jsp`

was a good place to begin.

Sorry, we are unable to provide assistance with your home installation of Java. Please use the help facilities available from the download site.

Rules for use of the COMP160 Laboratory

We have to have a section on rules and here it is.

Software Licences

The software and manuals in use in the COMP160 laboratory have been bought under various licences. In general, these licences state that under the Copyright Laws:- No part of this work may be reproduced in any form (in whole or in part) or by any means or used to make a derivative work (such as a translation, transformation or adaptation). Under the law, copying includes translating into another language or format. All rights reserved. Hence, you are NOT permitted to copy the system software or any applications under any circumstances, nor should the systems be used to copy any other licensed or copyright software without the owner's permission. Failure to observe these requirements will be considered a serious breach of University regulations, and will be dealt with under the Discipline Regulations.

Rules

- **Your 24 hour access to the Computer Science Department labs is for you and you alone. You must not allow friends to come in to the building with you after-hours.**
- **You must never allow anyone else to use a departmental computer logged in under your account name. Do not log a friend in and let them work/ play beside you.**
- **Eating is not allowed while sitting at a computer. You may drink out of sipper top bottles, which are to be kept on the floor.**
- **Cell phones must be switched off during streamed lab sessions.**
- Software is provided by the Department for all classes. The use of any other software in the laboratory is NOT allowed without the written permission of the Head of the Department.
- We reserve the right to examine any memory stick or other media brought into the laboratory.
- Students belonging to a particular laboratory session have first priority for the use of the computers and help from the demonstrators in that session. If you want to do extra work and there are no computers available you will have to find another time.
- Large files not related to course work should not be stored in your Home Directory. This includes **mp3**, **movie** and large **graphics** files. Such files may be deleted from your Home Directory without warning.
- Your computer science home directory is to be used responsibly and for coursework only. Its contents can be inspected at any time by departmental staff.

To use the COMP160 Lab you must sign in the box below. By doing so you are accepting notice of these conditions and are agreeing to abide by them.

I have read and agree to abide by the lab rules.

I understand and agree to the terms of usage for my home directory.

Name (please write clearly) **Signature**

_____ _____

Laboratory 1 Introduction to Java

A journey of a thousand miles starts with a single step.

– Lao Tzu, Tao Te Ching

Notes

The reading for this lab is Chapter 1 of Lewis, DePasquale and Chase (the text book), particularly Sections 1.1 and 1.2 (pages 2-18 [*30 – 46 2nd ed.*]). Chapter 1 is very general, and Lab 1 focuses on introducing **DrJava** (the application that we will use to write Java programs). Seldom is a program of any significant length typed in accurately to begin with. This lab explores some of the error messages which DrJava produces to help you correct your code.

On page 15 [*43*], the textbook describes an Integrated Development Environment (IDE). **DrJava** is a simple IDE which can be used on many different kinds of computer, including Macs, Windows, Linux and other forms of Unix, to develop programs that run on many different kinds of computer. Java programs can also run on many other kinds of device including some cell phones, PlayStations, Palm handhelds, and others. We will use **DrJava** to write and run Java programs on Macs running Mac OS 10.9.3.

Preparation

In general, preparation questions are taken from the textbook, and you will need to read the textbook in order to answer them. The preparation questions should be completed before you come to the lab session. Note that the textbook contains answers to its Self Review Questions. Working through these may also help you to prepare for your laboratory session. For this first lab, however, there are no preparation questions.

Printing in the lab:

Unfortunately we are not able to provide a free printing service any more. You may find you have a small printing quota available to you, in which case the printer in front of the office is where your pages will be.

Lab Work

In this lab we introduce **DrJava** (the application that we will use to write Java programs) and use it to write a Hello World program. Most programs will contain some errors when they are first typed. We will explore some of the error messages DrJava produces to help you identify the errors.

Part 1

1. Log On to a computer in the laboratory. If you haven't used Mac OS 10 recently, we suggest you take a look at Mac Help (**Finder > Help**) to familiarise yourself with it. You will find it very useful to know how to customise your Dock.

In COMP160, we expect not to have to spell out every detail of a process, but assume that you will be able to find things in file menus, work out which button/key to press to finish a process and make selections in dialog boxes. To call a demonstrator, we ask that you use the **DemoCall** icon on the desktop. This queues your call and ensures people are seen in the order in which they asked for help.

2. Change your password: select **Apple menu> System Preferences> Users & Groups.** Click on the **Change Password** button. Enter your existing password then your new password. Your new password must be **at least 8 characters long** and contain **at least one digit**, **one uppercase character** and **one lowercase character**. Retype your new password in the **Verify** field then select **Change Password**. You will see a message about your login keychain. Select **OK**.

3. Change your password from time to time and keep it secure!

4. You will see the **coursework** folder on the desktop - open it and explore. This folder is stored on our server, not your local machine. In the **COMP160** directory you will find the **coursefiles160** folder which contains any files needed for your lab work.

5. In the Computer Science Department laboratories, all your files are stored in your **Home** directory. The word *directory* is interchangeable with the word *folder*. Your **Home** directory, like the **coursework** directory, is stored on our server. You can access it from the Finder window under **Go> Home** or under **Places** at the left hand side of any Finder window or using the key combination *command-shift-H*. In your **Home** directory there is a **COMP160** directory where your lab work should be saved during the semester.

6. If you haven't already done so, create a shortcut to the **DrJava** application in your dock by dragging its icon from the **Applications** directory to your dock. Open the **DrJava** application. The icon will start bouncing as the application opens.

DrJava is the application in which you will write and run your Java code. It is an Integrated Development Environment (IDE) providing a graphical user interface (GUI) and tools for developing your computer programs. The IDE allows you to edit, navigate, compile and run your code. You will soon be very familiar with this environment.

7. The physical process required for writing and running a java program involves four steps:

1) Typing in the code. DrJava will display a large window on the right hand side, underneath the **DrJava** tool bar (Figure 1.1). This is the text editing window into which you will type your code.

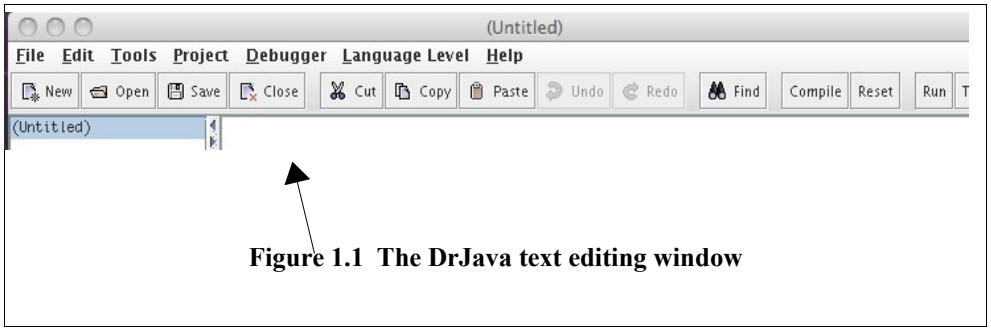


Figure 1.1 The DrJava text editing window

Into the text editing window, type the code for the `HelloApp` class listed below.

```
/** Hello World exercise.
 * Lab 1 COMP160 Part 1
 * (your name) July 2014
 */
public class HelloApp{
   public static void main (String[] args){
      System.out.println("Hello world");
   }
}
```

DrJava will put a space after the star - you will need to remove it

2) Saving your file (class).
Select **File> Save as** and the Save window will appear. Double-click on the COMP160 directory.

Create a new directory for Lab 1 by selecting **New Folder** (Figure 1.2). Name your new folder **Lab1**.

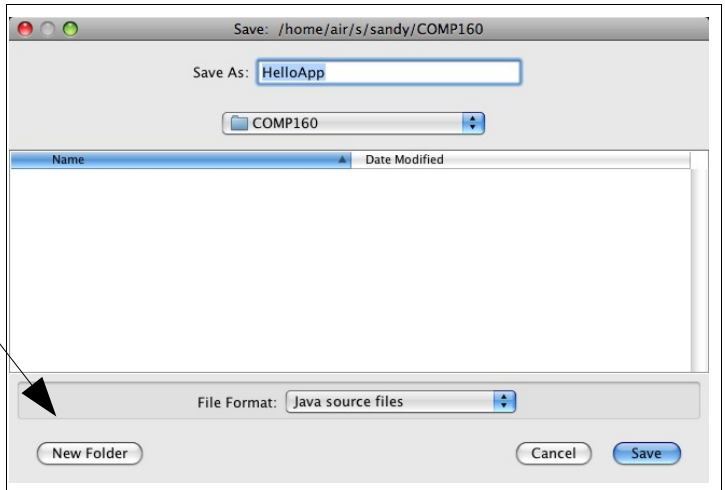
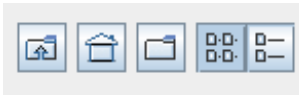


Figure 1.2



It is possible your Save window will look like this instead. **New Folder** is the middle icon.

A java class needs to be saved in a file of the same name, but with a `.java` extension.

Save your file in the **Lab1** folder with the name `HelloApp.java` (Figure 1.3).

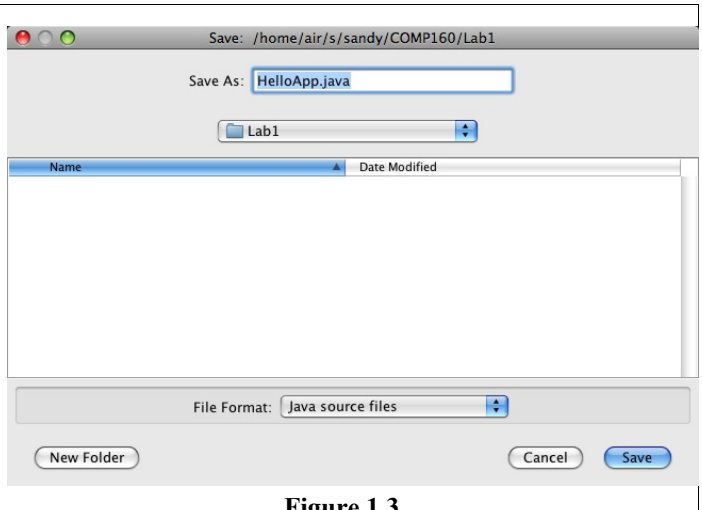


Figure 1.3

3) Compiling your code. Click on the Compile button on the toolbar (or the shortcut `F5` key).

When you see "Compilation completed" in the Compiler Output window, a class file will have been created in your Lab1 directory called `HelloApp.class`. If you see an error message in this window, check your code for typing errors, correct your code, and save again (**File> Save**). Note: Java is case sensitive.

4) Running your program. Run your program using the Run button, or if your version of DrJava doesn't have one, use the **Tools> Run document's main method** (or the (not so short) shortcut `fn` plus `F2` key).

You should see the output of the program in the Console window.

Figure 1.4

Part 2

1. Programming Projects PP 1.1 (L,D&C page 28 [55]). In a new `DrJava` document, enter, compile, and run the following application:

```
public class Test{
   public static void main (String[] args){
      System.out.println("An Emergency Broadcast");
   }
}
```

2. Programming Projects PP 1.2 (L,D&C page 29 [56]). Introduce the following errors, one at a time, to the `Test` class. Record any error messages that the compiler produces. Fix the previous error each time before you introduce a new one. If no error messages are produced, explain why.

You will find it useful to turn line numbering on in **DrJava** : In **`Preferences> Display Options`** check the box for **`Show All Line Numbers`**

a. change Test to test

b. change Emergency to emergency

c. remove the first quotation mark in the string (in programming, a string is a sequence of characters)

d. remove the second quotation mark in the string

e. change `main` to `man` (this will compile but check the Interactions window when you try to run it)

f. change `println` to `bogus`

g. remove the semicolon at the end of the `println` statement

h. remove the last brace in the program

3. Copy the file called `Ex3App.java` that can be found in the **coursework>COMP160>coursefiles160> Lab01** directory to your **`Lab1`** directory. Fix the errors in the code until it produces a correct output.

Make sure you understand:

* A Java program is made up of one or more class definitions.
* One of the classes must contain a method called `main`. By definition, a class with the `main` method in it is called an application class.
* The `main` method is where statement execution begins.
* Each programming statement in the `main` method is executed (performed) in order until the end of the method is reached.
* The Java language is accompanied by a library of extra classes which are called the standard class libraries.

Lab Completed

☐ files in Lab1 folder ☐ Part 1 Hello World ☐ Part 2 Error messages
☐ Part 3 Ex3App fixed ☐ lab rules (page 8) signed ☐ submitted

Date **Demonstrator's Initials**

Laboratory 2 Variables

Notes

Reading: L,D&C Chapter 1, Sections 1.1,1.2 (pages 2-18 [30 - 46]), Chapter 2, Sections 2.1 - 2.3 (pages 34-50 [60 - 75]). You may feel that we have thrown you in at the deep end, but don't panic. Every concept introduced will be revisited in its own time. For now, we ask you to look for patterns in the code examples you are given in lectures, in the text and in this lab book. The lab work can be done by following the patterns. We don't expect you to understand it all at this stage.

Preparation

Preparation questions are (mostly) taken from L,D&C as indicated, and you will need to read the textbook to answer them. The questions below should be completed before you come to the lab session. Write your answers in the space provided.

1. Give examples of two types of Java comments and explain the differences between them.

2. Exercise EX 1.2 (L,D&C page 28 [54]). Which of the following are not valid Java identifiers? Why?

Identifier	Valid?	Reason
a. Factorial	☐	
b. anExtremelyLongIdentifier	☐	
c. 2ndLevel	☐	
d. level2	☐	
e. MAX_SIZE	☐	
f. highest$	☐	
g. hook&ladder	☐	

3. What is the name of the method where statement execution begins in a Java program?

4. Exercise EX 2.4 (L,D&C page 69 [92]). What output is produced by the following statement? Explain.

```
System.out.println("50 plus 25 is " + 50 + 25);
```

5. A Java program contains the following statements:

```
int i = 7;
int x = 3;
x = i;
i = x;
```

What will the value of `i` be after these 4 statements have been executed?

What will the value of `x` be after these 4 statements have been executed?

Lab Work

The three parts to this lab are designed to give you experience working with variables. Note that we will use the javadoc form of block comment `/**` for class descriptions and method headers from this lab on. [See L,D&C Appendix I, 2nd edition only.]

Part 1

1. Open a new **DrJava** file. Type in the following code:

```
/** Lab 2 COMP160 Part 1 TwoNumbersApp.java
 * (your name) July 2014
 */
public class TwoNumbersApp{
   public static void main (String[] args){
      double num1;   //declaration, specifies data type and name
      double num2;
      num1 = 8.5;   //assignment
      num2 = 15.0;
      System.out.println("First number is " + num1);
      System.out.println("Second number is " + num2);
      System.out.println("Sum is " + num1 + num2);
   }
}
```

> **DrJava has a very useful function for automatically indenting your code.** The tab key will indent any selected piece of code. Indent your whole class by choosing **Edit> Select All (⌘A)** then **tab**.

2. Save the file in your **Lab2** directory as `TwoNumbersApp.java`. Compile your code. If there are errors in your code, you will see them now in the Compiler Output window. If you typed carefully, you will again see the message "Compilation completed.". When you alter code, you need to compile the changes to make a new class file before you run it again.

3. Run your program. The program output will appear in the Console window. The Console window accumulates output from all programs run during a session. The Interactions window, however, is reset each time a program is run. It will be displaying the java instruction to run `TwoNumbersApp` and the program output.

The Console window can be refreshed using the instruction **Tools> Clear Console**.

> The sign you have known in mathematics as equals (=) needs in Java to be thought of as "gets the value" or "is assigned the value". It is known as the **assignment** operator.
>
> The statement `num1 = 8.0;` assigns the value `8.0` to the variable `num1`. The order is important! The value on the right of the assignment operator is assigned to the variable on its left.
>
> A variable must be **declared** (given a name and data type) before it can be used.
>
> The + sign can mean **concatenation** or **addition**. If either of the operands is a string (sequence of characters) rather than a number, the `+` operator will be unable to perform addition e.g. there is no sensible answer to `"cat" + 3`. In this case, concatenation is performed – the second operand is treated as a string and is appended to the first – producing for the example given `"cat3"`.

4. Check the output of your code. Does the answer look correct?

5. Fix your `TwoNumbersApp` program so that the numbers are **added** rather than **concatenated**.

6. Save, compile and run your code.

Part 2

1. Use **File> Save As** to save your `TwoNumbersApp` class as a new file called `ThreeNumbersApp.java` then change the class name in the code to match the new filename.

2. Open your **Home> COMP160 > Lab2** directory. Resize the open windows so you can see the `DrJava` window and the **`Lab2`** window at the same time. Watch the `Lab2` window as you compile your `ThreeNumbersApp` class for the first time. What is the name of the file that appears after the compile? _____

The next steps will show you how to alter the class `ThreeNumbersApp` so that it **stores** then displays the sum of **three** numbers.

3. Declare a third variable, of type `int`, called `num3`. Assign a value to `num3`.

4. Insert another `println` statement which describes and displays the value stored in `num3`.

5. Declare a fourth variable called `sum`. It is going to hold the sum of two `doubles` and an `int` so will need to be of the most precise data type involved (`double`).

6. Assign to the variable `sum` the sum of the three variables.

7. Alter the `"Sum is "` statement so that it prints the value stored in the variable `sum`.

Part 3

Copy the file called `FillRestaurant.java` that can be found in the **coursework> COMP160> coursefiles160> Lab02** directory to your **`Lab2`** directory. Open it in `DrJava`.

> You may notice that double-clicking on a `.java` file will not open the file in DrJava. Drag the file to the open DrJava window or use **File > Open**.

The class will not compile, nor produce any useful output, at this point.

```
/** COMP160 Lab2 S2 2014    FillRestaurant.java
 * A restaurant caters for large tour groups and takes bookings by the bus and van load each evening.
 * This class stores the booking data and calculates the number of unallocated seats.
 */
public class FillRestaurant{
    public static void main(String[]args){
        final int MAX_OCCUPANCY = 300;      // number of seats in restaurant
        final int BUS_CAPACITY = 35;        // number of seats in a bus
        final int VAN_CAPACITY = 8;         // number of seats in  a van
        final String DATE = "30th July 2014";  // dining date

        int numBusBooked =  4;              // number of buses expected on DATE
        int numVanBooked = 2;               // number of vans expected on DATE

       // number of diners expected from buses,   * is the multiplication sign
        int busDiners = numBusBooked * BUS_CAPACITY;
       // number of diners  expected from vans
        int vanDiners =
          . . .
```

1. Finish the statement which calculates and stores the number of diners expected to be arriving in vans.

2. Write another statement which calculates and displays the number of seats left in the restaurant for that date, producing the output:

```
Seats left for 30th July 2014 : 144
```

Use the DATE variable to produce the date in the output string rather than typing it.
When you have completed these tasks, call a demonstrator to mark your work and record your mark.

Lab Completed

☐ preparation exercises ☐ Part 1 TwoNumbers ☐ Part 2 ThreeNumbers
☐ Part 3 FillRestaurant ☐ DATE used ☐ comments ☐ submitted

Date **Demonstrator's Initials**

Laboratory 3 Methods

Notes

Readings: L,D&C Chapter 2, Section 2.6 (pages 61-65 [85 - 89]).
Until now all your code has been in the main method. In this lab you will be writing several methods in one class, and sending data from one method to another.

You will need to look carefully at the code examples from your Lecture 3 notes in order to answer some of the preparation questions.

Preparation

1. **a.** The class `Rhyme` below defines three **methods**. What are their names?

b. The class `Rhyme` below has two **variables**. What are their names?

c. The `main` method in the class `Rhyme` below contains two method calls. Number them in order of execution.

```
public class Rhyme{

    public static void main(String [] args){
        displayString2();
        displayString1();
    }// end main

    /** method which displays whatever is stored in a local string variable to the screen*/
    public static void displayString1(){
       String s1 = "Violets are Blue";
       System.out.println(s1);
    }//end displayString1

    /** method which displays whatever is stored in a local string variable to the screen*/
    public static void displayString2(){
       String s2 = "Roses are Red";
       System.out.println(s2);
  }//end displayString2

}//end class
```

d. What will the output of this code be?

2. Since Lab1, you have called the method `println` without having to write a definition for it. How is this possible?

Lab Work

A variable declared **within** a method is sometimes called a local variable, as it is only able to be used/accessed within that method. This is a safety feature for programmers – in different sections of a large program, two programmers in a team can each choose to use the same variable name in their method without affecting each other.

Because a variable declared within a method is local to that method, if its value is needed elsewhere in the program it must be sent (passed) as a parameter.

Part 1

1. Make a copy of the file `Fish.java` from the **coursework>COMP160>coursefiles160> `Lab03`** folder to your **`Lab3`** folder. Open it in **DrJava**. Take a look at the code.

```
/**
 * Lab 3, COMP160, 2014
 */

public class Fish{

    public static void main(String [] args){
        printVerse1();
    }//end main

     /** declares a String variable called verse1 and displays it on the screen*/
    public static void printVerse1(){
       String verse1 = "One fish\nTwo fish\nRed fish\nBlue fish.\n";
       System.out.println(verse1);
    }//end printVerse1

}//end class
```

One fish Two fish Red fish Blue fish.
printVerse1 output

2. Compile and run. It should produce an output such as shown in the box to the right. This output will give you a clue to what the \n is doing. It is called an escape sequence. The character n following the character \ will produce a new line. (Refer to L,D&C page 40 [65])

3. Write another method (call it `printVerse2`) in the class `Fish` which declares a local variable called `verse2` and prints it out. See the output expected for this method in the box to the right.

Black fish Blue fish Old fish New fish.
printVerse2 output

4. Compile your code. If you run your code now, you will see no change because you haven't called this new method yet.

5. Call your new method from the `main` method, in order to produce the output below:

One fish Two fish Red fish Blue fish. Black fish Blue fish Old fish New fish.

You now have a program with 3 methods and 2 local variables.

The phrase for the third verse will be stored as a **local variable** in the `main` method rather than as a local variable in a `printVerse` method. Variables are usually declared at the top of the class or method they belong to.

6. Declare a `String` variable called `verse3` in the **`main` method** and set its initial value to `"This one has a little star."`

7. Write a `printVerse3` method which prints out `verse3` then a blank line. Call this method from `main`. Compile. You will see an error saying

```
cannot find symbol
  variable verse3
  class Fish
```

The local variable `verse3` can only be accessed within the `main` method since this is where it is declared. However, there is a way to get its value to the `printVerse3` method. We can send it as an input to the method via the parentheses (brackets). Here's how:

8. Instead of empty parentheses in the `printVerse3` method header, fill them like this: `(String verse)`. This is called a **formal parameter**. Change the `System.out.println` statement so it prints `verse` rather than `verse3`.

9. Change the method call in `main`. Instead of calling `printVerse3()` call `printVerse3(verse3)`. A copy of the string variable named `verse3` is sent to the method as input. This is called an **actual parameter**, or sometimes an **argument**.

10. Compile and run. So far so good?

On to the next challenge - user input! The fourth verse will be typed in by the user (you) while the program is running.

11. Add the line **`import`** `java.util.Scanner;` to the top of your class, between the comment and the class header. This makes the library class `Scanner` easier to use in your class.

12. Following the `Echo` example (L,D&C page 64 [88]), write code at the start of the `main` method which:

- declares a local `String` variable called `verse4`
- constructs a `Scanner` instance object using the line `Scanner scan = new Scanner(System.in)`
- prompts the user with a meaningful message (the program will sit and wait for input - the user needs to be told what is expected of him/her)
- calls the `nextLine` method on the `Scanner` object to read text input from the user, and assigns it to the `verse4` variable.

13. At the end of the `main` method, call the `printVerse3` method again, this time sending it `verse4` as a parameter. Now that the `printVerse3` method is being used for another verse as well, its name is no longer appropriate. Change its name to `printVerse` and change the method calls to match. Compile the code.

14. Click on the **Interactions** tab which you will find next to the **Console** tab then run your code. The Interactions pane will display an input box in which to type the fourth verse. If you want to be true to Dr Seuss, type `"This one has a little car."` in the input box.

The method `printVerse` receives a copy of the current value of whatever string variable it is sent as input. The new string is considered to be a local variable and is given a name (`verse`) as it is passed into this method. Any name could be used, but meaningful names are best.

Whenever the `printVerse` method is called, it MUST be sent a string variable or the code will not compile.

```
Enter verse 4 text:
This one has a little car.

One fish
Two fish
Red fish
Blue fish.

Black fish
Blue fish
Old fish
New fish.

This one has a little star.

This one has a little car.
```

15. Write comments in your code - there are some examples of commented code in **coursework> COMP160> coursefiles160> `Resources`**. Each method should have a preceding block comment describing its purpose e.g. */** displays the value of the input parameter */* .

Some questions :

1. Does the order in which you **write** the methods make a difference to the output?
2. Does the order in which you **call** the methods make a difference to the output?

Part 2

1. Take a look at the online documentation (link below). **Bookmark this page**, because you will be needing it again and again. The top left window lists Java packages (organised groups of classes for a particular purpose). Scroll to `java.util` and click.

The lower left window will change to give you a list of links relevant to `java.util`. Scroll down until you find `Scanner` and select it.

The main window will now show information relevant to the `Scanner` class. Scroll down until you find Method Summary, then further until you find `nextInt()` and `nextLine()`.

Note the brief description, and the return type of the method in the column on the left. Click on the link for a fuller description of any particular method, but don't worry if it doesn't mean anything to you just yet.

2. Have a look at `java.lang` and from there the class `System`.

This is how you find out about the packages of classes in the libraries, and what they can do.

http://java.sun.com/javase/6/docs/api/

`UML diagrams`

It can be useful to represent the class structure in a diagram. One form of diagram in common use is the Unified Modeling Language. The UML class diagram for the finished `Fish` class is shown below. It has three sections. The top one is for the name of the class. The middle is for the data fields. Data fields are variables declared outside of any method. In this case, there are no data fields. The lower section lists the methods. The methods are listed with their attributes in this order:

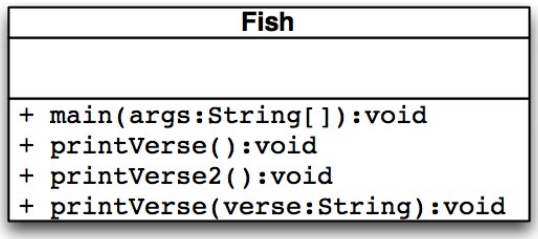
1. visibility **+** represents "public"
(**-** represents "private")

2. the name of the method

3. parentheses
may be empty
or contain the parameter/s in the format
`name:data type`
(the main method's parameter is an array of String values called args – you will cover arrays in the second half of the course)

4. a colon **`:`**

5. the method's return type (these methods are all **`void)`**



Lab Completed

☐ preparation exercises ☐ Part 1 working ☐ relevant names used ☐ comments
☐ parameter (*printVerse*) ☐ Part 2 API bookmarked ☐ submitted

Date **Demonstrator's Initials**

Laboratory 4 Expressions

Notes

Readings: Lecture 4 notes and L,D&C Chapter 2, Section 2.4 (pages 51 – 58 [75 – 82]).

You will be using `return` methods. There are many new concepts incorporated into this lab. Don't panic. Give yourself time and work through them one by one.

Preparation

1. Exercise EX 2.10 (L,D&C page 69 [93]). Given the following declarations, what result is stored in each of the listed assignment statements?

Remember: If both operands are integers the result will be an integer. Any fractional part is lost. If this result is then stored as a `double` or a `float` (as in `b.` or `d.` below), the integer will be automatically widened (displayed with `.0` after it).

```
int iResult, num1 = 25, num2 = 40, num3 = 17, num4 = 5;
double fResult, val1 = 17.0;
```

a. `iResult = num1 / num4;`

b. `fResult = num1 / num4;`

c. `iResult = num3 / num4;`

d. `fResult = num3 / num4;`

e. `fResult = val1 / num4;`

g. `iResult = num1 / num2;`

n. `iResult = num3 % num4;`

o. `iResult = num2 % num3;`

p. `iResult = num3 % num2;`

q. `iResult = num2 % num4;`

2. A `return` method can only ever return one single value.

The type of this value must be specified in the method header in place of the keyword `void`.

What data type is being returned by methods with the following headers (yes, it is as easy as it looks):

a. `public static int aMethod()` returns a value of type _____

b. `public static double bMethod()` returns a value of type _____

c. `public static String cMethod()` returns a value of type _____

d. `public static int dMethod(double d)` returns a value of type _____

e. `public static double eMethod(String s, int i)` returns a value of type _____

f. `public static String fMethod(String s)` returns a value of type _____

3. The result of a `return` method (the value returned) can be used in other parts of the program by simply calling the method. A method is called by its name and a parameter list (which may be empty). Imagine the value which is being returned replacing the method call in the flow of the program.

Legal method calls for the methods in Ex. 2 (above) could look like these (any legal parameter values may be used in the parameter list):

```
double sum = aMethod() + bMethod();   //adds the int returned by aMethod and the double
                                      returned by bMethod
```

```
System.out.println("The result is " + dMethod(25.3));        //displays an int.
```

Note: `dMethod` must be sent a parameter which is (or can be widened to) a `double` or the code will not compile. Widening refers to the automatic upgrade of an integer to a double when necessary e.g. an input of `35` widens to `35.0` if a double is expected.

```
System.out.println(cMethod());        //displays a String

System.out.println("The result is " + eMethod("xE0", 3));   //displays a double.
```

Note: e`Method` must be sent a `String` and an `int` input parameter or the code will not compile.

a. Write a statement which stores in a variable called `difference` the value resulting from subtracting 3.8 from the value returned by `bMethod`. Include the declaration for the variable `difference`.

b. Write a complete method definition for `fMethod`. The method should return the input string twice, separated by an asterisk.

i.e. if the input string is `"dog"` the output would be `"dog*dog"`

A method call for `fMethod`, using "dog" as the input (actual) parameter, would look like this:

```
fMethod("dog");
```

c. Finish the statement which assigns to a variable called `result` the string returned by `fMethod,` using the string `"woof"` as the input (actual) parameter.

```
String result =
```

d. Finish the statement which assigns to a variable called `result` the string returned by `fMethod,` using the string returned by `cMethod` as the input (actual) parameter.

```
String result =
```

Lab Work

You are provided with an incomplete program to finish. When completed it will read, in turn, three Fahrenheit values from the user and convert them to Celsius.

You will find it helpful to refer initially to code listing 2.7 (L,D&C page 55 [80]) which performs a single Celsius to Fahrenheit conversion and was the starting point for the code provided. Where the code-listing in the text book is limited to converting a single Celsius temperature (24), the code for this lab will be more flexible by getting each Fahrenheit value from the user. The Fahrenheit input values will need to be read as `double` rather than `int`, so you will need to use the `Scanner` class's `nextDouble()` method rather than `nextInt()`.

1. Copy the file `FahrenheitToCelsius.java` to your **`Lab4`** directory and open it in **`DrJava`**.

```
/** Lab 4  COMP160 S2 2014
 * Starting code
 */

import java.util.Scanner;
public class FahrenheitToCelsius{
  public static void main(String[]args){
    convertFToC();
    //Step 5
  }

  /** gets input from user representing fahrenheit and displays celsius equivalent*/
  public static void convertFToC(){
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter Fahrenheit temperature");
    double fahrenheit =           ; //Step 2 - assign next double input from Scanner object
    System.out.println( fahrenheit + " degrees Fahrenheit is " ); //Step 4
  }


  /**  calculates and returns the celsius equivalent of a double input parameter called fahrenheit*/
  public static double toCelsius(double fahr){
    final int BASE = 32;
    final double CONVERSION_FACTOR = 9.0/ 5.0;
    //Step 3
    return celsius;
  }
}
```

2. In the `convertFToC` method, assign to the variable `fahrenheit` the next `double` input from the keyboard.

3. In the `toCelsius` method, write an expression which calculates the celsius equivalent of the variable `fahr` and stores it in a variable called `celsius`.

You may like to determine the formula required by working your way from the Celsius to Fahrenheit conversion formula in the space below. Try to isolate celsius on one side of the equation. Maintain equality by performing equal operations on both sides. Call a demonstrator if you need a hand.

```
fahr = celsius * CONVERSION_FACTOR + BASE
```

4. In the `convertFToC` method, change the second `System.out.println` statement so it produces a display such as

```
212 degrees Fahrenheit is 100 degrees Celsius
```

for an input of 212. If you are baffled by this instruction, here is a fuller description:

The first half of the output line (212 degrees Fahrenheit is) is already written. The next item to be displayed is the number of degrees celsius, which is being calculated and returned by the `toCelsius` method.

Remember (see Lecture 3 and preparation exercises) that the result of a **`return`** method can be used elsewhere in a program by calling its name and providing the correct number and type of input parameters.

Call the `toCelsius` method within the `System.out.println` statement, sending it the data that it needs (the variable representing the degrees in fahrenheit).

Finish off the statement by concatenating the string " degrees Celsius" on the end.

This program should now compile. If you run it, it will convert one value from Fahrenheit to Celsius.

5. Add two statements to the `main` method so the program will convert 3 fahrenheit values when it is run.

6. Run your program, and use it to calculate the celsius temperatures for the examples below. (In lecture 8 you will learn how to limit the number of decimal places displayed. Don't worry about it for now.)

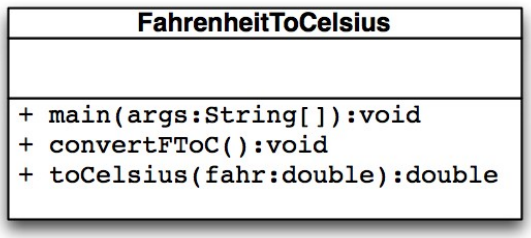
32.0 degrees fahrenheit is degrees Celsius

98.6 degrees fahrenheit is degrees Celsius

212.0 degrees fahrenheit is degrees Celsius

UML class diagram

The UML class diagram for the FahrenheitToCelsius class shows one method with a return type other than `void`. The `toCelsius` method takes one `double` called `fahr` as an input parameter and also returns a `double`.



Lab Completed

- ☐ preparation exercises complete
- ☐ temperature converter working
- ☐ comments
- ☐ submitted

Date **Demonstrator's Initials**

Laboratory 5 Graphics

Notes

Readings: L,D&C Appendix F (pages 965 – 972 [830 - 837]).

In Lab 3 you created an instance of the class `Scanner` using the keyword `new.` In Part 1 of this lab, you will be working with an instance of the class `SnowmanPanel`, which is created by the keyword `new` in the `main` method. This class uses the Graphics class from the package java.awt to "draw" in a new window (frame).

Preparation

1. Exercise EX F.3 (L,D&C page 982 [845]). Assuming you have a `Graphics` object called `page`, write a statement that will draw a line from point `(20, 30)` to point `(50, 60)`.

2. Exercise EX F.4 (L,D&C page 982 [845]). Assuming you have a `Graphics` object called `page`, write a statement that will draw a rectangle outline with height `70` and width `35`, such that its upper-left corner is at point `(10, 15)`.

3. In the diagram of a graphics window below, sketch a circle **centred** on point `(50, 50)` with a **radius** of `20` pixels. Draw the circle's bounding box, and mark where the left and top co-ordinates intersect with the axes.

4. Exercise EX F.5 (L,D&C page 982 [845]). Assuming you have a `Graphics` object called `page`, write a statement that will draw a solid (filled) circle *centred* on point `(50, 50)` with a radius of `20` pixels. This is the circle you sketched in Exercise 3 above.

5. What is the purpose of the `TOP` and `MID` local variables in the `Snowman` class listed on pages 971 – 972 [835 - 836] of L,D&C?

Part 1

Adapted from Programming Projects PP F.1 (L,D&C page 982 [845]). Create a revised version of the `Snowman` program as described below.

You will find a version of the code in the **`coursefiles160`** folder. The `main` method is complicated. You don't need to understand what is happening in the `main` method until much later in the course.

1. Copy the `Snowman.java` file to your **`Lab5`** folder. Open it in **DrJava**.

The `main` method creates an instance of the `Snowman` class using the keyword `new`. The `Snowman` class contains the `paint` method, which is called automatically. It is the code within the `paint` method that we want you to concentrate on at present. Notice how useful the comments are to your understanding.

Compile and run the class. You should see a snowman similar to that pictured on L,D&C page 971 [835].

2. Add two red buttons to the upper torso. Make sure they are centred on the MID line.

3. Make the snowman frown instead of smile. (See "Drawing an arc" below)

4. Move the sun to the upper-right corner of the picture.

5. Display your name in the upper-left corner of the picture.

Note that

- a string is drawn above and to the right of its starting point
- the Snowman window (frame) has a title bar at the top, which takes up the top 22 px of the stated size

6. Shift the snowman to the left of the picture. (**Hint:** This requires just one statement to be altered.)

Drawing an arc (smile / frown)

```
page.drawArc (MID-10, TOP+20, 20, 10, 190, 160);  //arc which draws the Snowman's smile
```

The Snowman's smile is an arc. The first 4 numbers define the bounding box of the oval of which the arc is a portion. In the diagram below (Figure 5.1), the bounding box is outlined in gray, and the complete oval has been drawn in outline.

The fifth number is the start position of the arc. Position `0` is at 3 o'clock, and the measure is degrees (there are 360 degrees in a circle). The direction is anti-clockwise. (No, don't ask why). The Snowman's smile starts at `190` degrees.

The sixth and last number is the **size** of the arc angle. The Snowman's smile is a 160 degree angle, taking the arc from 190° to 190 + 160 = 350° (almost back to `0`). In `SnowmanPanel.java`, because `drawArc` rather than `fillArc` is used, just the edge of the specified arc is drawn. In Figure 5.1, the whole of the specified arc is filled in as though `fillArc` has been used because it is easier to visualise the size and shape of the arc with fillArc.

Diagram of the arc for the Snowman's smile.

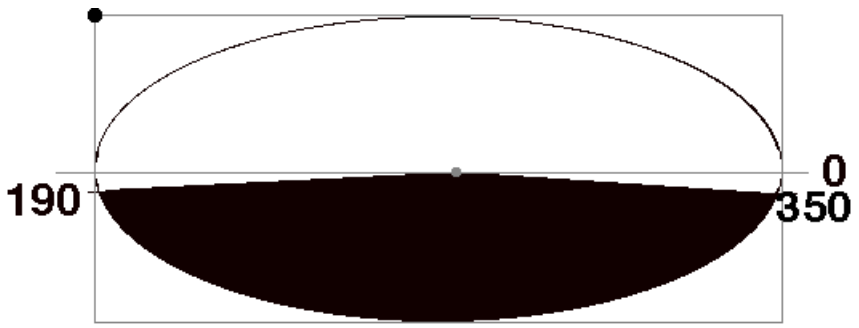


Figure 5.1

Part 2

Adapted from Programming Project F.7 (page 983 [846]). Write an application that shows a pie chart with eight equal slices, all coloured differently. Lastly, move one piece of the pie out a little as shown below in Figure 5.2.

We have provided you with the class structure required. Copy `Pie.java` from **LabFiles** to your **Lab5> Part2** folder. You now have a blank slate on which to draw your pie chart.

L,D&C lists colour names on page 967 [831], and describes how to draw an arc on page 969 [833]. Figure F.5 page 969 [833] will be a useful reference. (Your arcs will need to be filled.)

FIRST, plan your arcs by filling in the table below. There is a table column for each specification required in order to draw an arc. A full circle is 360° (its `arcAngle` would equal 360).

	x location	y location	width	height	startAngle	arcAngle
1st arc						
2nd arc						
3rd arc						
4th arc						
5th arc						
6th arc						
7th arc						
8th arc						

You can see from the table you have filled in that many of the values required are the same for each arc. **These values should be stored in local variables** with sensible names (similar to the way that MID and TOP were stored in the Snowman class). Your code can then be written very quickly, using the copy and paste commands for efficiency.

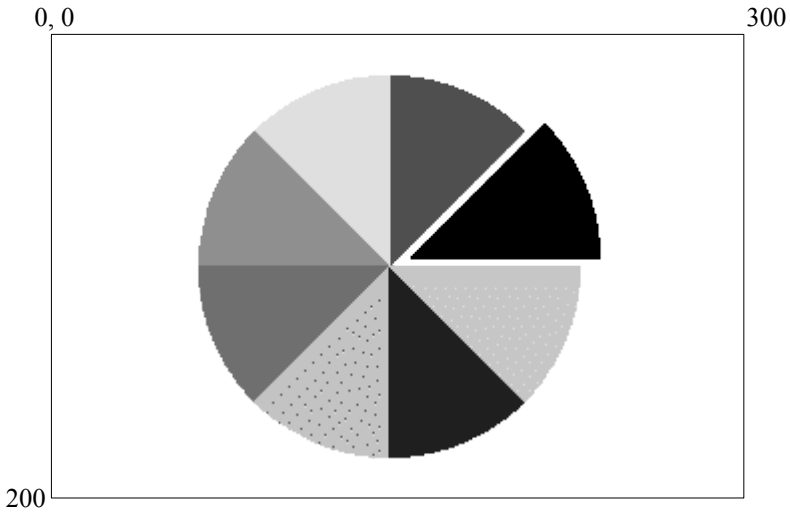


Figure 5.2

Lab Completed

☐ preparation exercises complete ☐ comments
☐ Part1 Snowman ☐ Part2 Pie (using local variables) ☐ submitted

Date **Demonstrator's Initials**

Laboratory 6 Objects

Notes

Readings: L,D&C Chapter 3, Section 3.1 (pages 76 – 80 [100 - 104]).
This lab marks a significant change in the design of our programs. From now on, most of our programs will have a more usual Object Oriented design. As well as the application class we will write one or more support classes, and we will create and use instances of these support classes. Because they will be instance objects their members (methods and data fields) do not need to be static.

You have already created some objects, such as `Scanner` objects in Labs 3 and 4. You have also created `String` objects in Lab 3, using the Java shortcut which doesn't require the keyword `new`. So creating / instantiating an object is really not too scary!

Once an object has been instantiated, its public methods can be accessed from other classes using dot notation. For instance, if the object has been given the name `book1`, its method `displayBook` can be accessed using the expression `book1.displayBook()`. This lab has you creating instances of a class, and manipulating them from an application class.

Preparation

1. Assume there is a class called `MyClass` which contains a `void` method called `display`.

a. What will the following statement in an application class do?

```
MyClass x = new MyClass();
```

b. Write a statement to follow the one above, which will call the `display` method on `x`.

c. Write a statement which will create an instance of `MyClass` called `y`.

2. Look at the listing for `Book.java` on page 31 of this lab book.

What are the names of the three data fields?

Which three of the seven methods are accessor methods?

What is the difference between a data field and a local variable?

3. The class `MyFrame` is listed on page 29. You will be asked to write another method (called `decorate`) in the `MyFrame` class. There are only two places where this method might be written – put asterisks in these two places.

The class `MyFrame` does not contain or need to contain a `main` method. It is a support class. The class will be created (instantiated) by another class, in this case, the application class (which **does** have a main method).

Lab Work

Part 1

DrJava has an Interactions window which allows you to test bits of code straight away. You can type a Java statement directly in to this window and see its effect. You can also type a statement which uses the result of a previous statement. In this lab, you will create some objects, first using DrJava's Interactions window, then using an application class.

1. Copy the file `MyFrame.java` from the **coursefiles160** folder to your **Lab6** folder. Open it in **DrJava**. You can see a listing of the code on the next page. **Compile the code.**

2. Click on the `Interactions` tab. Underneath **Welcome to Java ...** , type

```
MyFrame m1 = new MyFrame();
```

then press <return>. You have just created an object - an instance of the class `MyFrame` - which you can now refer to by the name `m1`. You can't see it yet, because it hasn't been set to be visible.

3. In the interactions pane, type `m1.setVisible(true);`

You should now be able to see a very small frame in the top left hand corner of your screen (it may be behind the DrJava window).

4. Now type `m1.setSize(300,150);` in the interactions window. The frame size is now set so that you can see the contents.

You can set the size of the frame any time you like. Try resizing the frame manually, then call the `setSize` method again. (**Hint:** If you want to repeat an instruction in the interactions pane, use the up arrow ↑ to scroll back through the previous instructions, then press <return> when the one you want is selected.)

The methods `setSize` and `setVisible` are both methods which can be called on a `JFrame` object.

It is usually more convenient to manage your objects using an application class. All the required code can be conveniently gathered into a main method, which can be run with a single command.

5. Open a new **DrJava** document and in it write an application class for `MyFrame` called `MyFrameApp`. (By definition, the application class is the class with the `main` method in it.) The `main` method should contain the three instructions you have typed in the interactions pane, in the following order:

```
MyFrame m1 = new MyFrame();
m1.setSize(300,150);
m1.setVisible(true);
```

6. Save this class in the same folder as your `MyFrame` class. **Compile** and run. You have now used an application class to create an instance of your `MyFrame` class. This instance is called m1, and this name is used to call its methods.

In case you were wondering, the methods `setSize` and `setVisible` are not defined within the `MyFrame` class, but can still be called on it because of the keyword `extends`. They are actually inherited methods of the `JFrame` class, which in turn are inherited from the class `JComponent`. (More later . . .)

7. In the `main` method, create another instance of `MyFrame`. Set its size and make it visible. Compile and run.

You will have to move the top frame aside so that you can see the second frame underneath. Both of these objects look exactly the same, but each instance of a class doesn't have to be the same. In fact, this is where their strength lies. Let's start by locating them in different places. Include the following line in your main method, then compile and run again.

```
m1.setLocation(0,180);
```

Let's add a method to your `MyFrame` class so that your instances can be seen to be different.

8. In the `MyFrame` class (**not** the application class) add the following method:

```
public void decorate(Color shade, String title){
   setBackground(shade);
   setTitle(title);
}// end decorate
```

This method takes two input parameters (a reference to an object of type `Color`, and a reference to an object of type `String`), and calls the `setBackground` and `setTitle` methods of the `JFrame` class (which, like `setSize` and `setVisible`, are inherited from the class `JComponent`).

9. Now, in the `main` method, call the `decorate` method on one of your instances. Your `decorate` method is expecting to be passed a `Color` and a `String`, in that order, separated by a comma. If it doesn't get what it wants, it will not compile, so pass it a `Color` e.g. `Color.pink` (see L,D&C page 967 [831] for a list of Color names, or look up the `java.awt Color` class in the API) and a title (e.g.`"Pink Einstein"`). You will need the statement `import java.awt.Color` at the top of the application class in order for the `MyFrameApp` class to be able to find the `Color` class (or you could use the full name e.g. `java.awt.Color.pink` in the call to the `decorate` method).

10. Compile and run. You should now have two instances of the `MyFrame` class, one showing a background in the default gray, and the other in some other colour. (**Note:** If you are working on some versions of Windows, the background will be transparent. Try it again when you're in the lab.)

11. The `setLocation` statement belongs out in the support class. Move it into the `decorate` method. It will not need the `m1` prefix, since it will now be applied to all instances of `MyFrame`.

The way it stands, all instances of `MyFrame` that have the `decorate` method called on them will be drawn at location `(0, 180)`. Change the `180` to the variable name `yOffset`. (Assume that the `x` coordinate will always be `0`.)

Include the declaration for the `yOffset` in the parameter list in the `decorate` method header. Adjust the method call in the `main` method to suit. Compile and run. It should behave exactly as it did before.

12. Add another statement to the `main` method which sets the background colour, title and y offset for your other instance of `MyFrame`.

```
/**
 * Lab 6, Part 1, COMP160
 * MyFrame.java
 */

// import graphics classes including JFrame
import javax.swing.*;
import java.awt.*;

/** Make our own version of a JFrame class with our own paint method...*/
public class MyFrame extends JFrame {

   public void paint(Graphics g){
      g.drawRect(50, 50, 40, 40);       // square
      g.drawRect(60, 80, 225, 30);      // rectangle
      g.drawOval(75, 65, 20, 20);       // circle
      g.drawLine(35, 60, 100, 120);     // line
      g.drawString("Out of clutter, find simplicity", 110, 70);
      g.drawString("--Albert Einstein", 130, 100);
   }//end paint

}// end class
```

Part 2

1. Copy the file `Book.java` from the **coursefiles160** folder to your Lab6 folder. Open it in **DrJava**. You will find a code listing on the next page.

The major difference between the structure of `Book` and the structure of `MyFrame` is that the `Book` class does not use graphics - its text output will show in the Console window.

`Book` has 3 data fields. They are `private`, so are not accessible from other classes. The only way another class can see what is in them is to call the public methods of `Book` which return their values (the `"get"` methods often called accessor methods) or which display their values (the `displayBook` method).

The only way another class can change the values in the data fields is to use the three public `"set"` methods (mutator methods)

2. Write an application class for `Book` called `BookShopApp`, which creates 3 instances of `Book` using the default constructor. When you have created each `Book` object, use its `set` (mutator) methods to set the values of its data fields (don't bother with user input, just type the values as literals in the code itself).

The first book is called "Life of Pi", has 348 pages and sells for $28.90.

The second book is called "Mister Pip", has 240 pages, and costs $22.70.

The third book is of your own choosing.

3. When the data fields are set, call each book's `displayBook` method.

What would happen if you called the `displayBook` method before **the data fields are set? Try it if you are not sure.**

Your output should look like this:

```
The name of the book is Life of Pi
It has 273 pages.
You can buy this book for $28.90
*********************************
The name of the book is Mister Pip
It has 240 pages.
You can buy this book for $22.70
*********************************
The name of the book is ...
.
.
.
```

The **state** (values stored in the data fields) of these 3 instances is different, but their **behaviour** (the methods that can be called on them) is the same (see Readings at the back of this book, page 110).

Extension Exercise: Write another class for your program called DVD which holds, changes and displays relevant data for a DVD (e.g. name, zone, rating, run time). Get your application class to create some instances of the DVD class, and display the data. You will be duplicating some behaviours from the book class. Later in the course you will learn how related classes in a hierarchy can share behaviours which are common to them.

Lab Completed

☐ preparation exercises complete ☐ comments
☐ Part 1 MyFrameApp ☐ Part 2 BookShopApp ☐ submitted

Date **Demonstrator's Initials**

```
/**
 * Lab 6, Part 2, COMP160 2014
 * Book.java
 * Stores and displays information about an individual Book.
 */

import java.text.NumberFormat; // for formatting price to 2 decimal places. See L,D&C p.94[117] Listing 3.4

public class Book{

   //data field declarations
   private String title;                    // book title
   private int numPages;                    // number of pages in book
   private double price;                    // retail price of book

   /** sets the value of the data field title to the value of the input parameter */
   public void setTitle(String t){
      title = t;
   } //end method

   /** sets the value of the data field numPages to the value of the input parameter */
   public void setPages(int n){
      numPages = n;
   } //end method

   /** sets the value of the data field price to the value of the input parameter */
   public void setPrice(double p){
      price = p;
   } //end method

   /** returns the value of the data field title */
   public String getTitle(){
      return title;
   } //end method

   /** returns the value of the data field numPages */
   public int getNumPages(){
      return numPages;
   } //end method

   /** returns the value of the data field price */
   public double getPrice(){
      return price;
   } //end method

   /** displays formatted Book information to the console window */
   public void displayBook(){
      NumberFormat fmt = NumberFormat.getCurrencyInstance(); //for formatting price
      System.out.println("The name of the book is " + title);
      System.out.println("It has " + numPages + " pages.");
      System.out.println("You can buy this book for " + fmt.format(price));
      System.out.println("**********************************");
   } //end method

} //end class
```

UML class diagram

When describing support classes, the UML class diagrams get more interesting. Often the application class will not contain data fields, as it is more concerned with managing the instantiation of other classes.

The middle section of the UML class diagram is for the list of data fields in the order:

1. visibility **-** represents "private"
(**+** represents "public")

2. the name of the data field

3. a colon **:**

4. the data type of the data field

The lower section lists the methods. As described in lab 3, the order is:

1. visibility **+** represents “public”
(**-** represents “private”)

2. the name of the method

3. parentheses

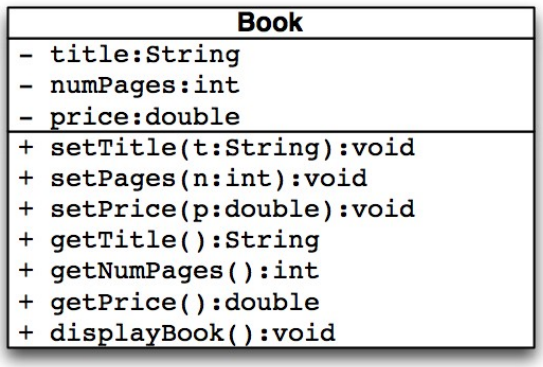
may be empty

or contain the parameter/s in the format **`name:data type`**

if there is more than one parameter, there should be a comma between each e.g. `(n:int, p:double)`

4. a colon **:**

5. the method's return type (may be `void`)



Laboratory 7 Constructors

Notes

Readings: L,D&C Chapter 5, Section 5.1, 5.2, 5.3, 5.4 (pages 170 – 199 [190 - 218]).

Constructors were discussed briefly in section 3.1. In chapter 5 constructors are revisited. These are special methods that have the same name as the class. Like other methods, a constructor can take input parameters. Unlike other methods, a constructor is neither `void` nor does it `return` a result. It is only called on instantiation (creation) of the object.

Constructors explained

There are different ways of setting the values of the data fields of an object. One is to specify values literally when writing the class, e.g.

```
private int age = 21;     // 21 is a "literal" value
```

Another is to use a mutator method

```
public void setAge(int a){
    age = a;      // age is set to the value of the input parameter
}
```

Other ways are to read in values typed by the user or read from a file.

Often the best way to set them is to assign values to the data fields **as the object is created** using a special method called a **constructor**. This

* makes sure that the data fields of the object are set to useful default values
* helps to avoid the problem of methods working with null data fields.

A constructor is a special method which

* has NO return type or void keyword in its header
* has the same name as the class

Every class is assumed to have a **default** constructor, with no parameters, that does nothing but initialise all the data fields of the class to zero values. This constructor doesn't have to be written, but if we wanted to write out an equivalent constructor for a class called `Book`, it would look like this:

```
public Book() {}  // this Book constructor must in a class called Book, saved in a file called Book.java).
```

There are no statements in the body, so nothing gets done (apart from the initialisations).

Objects are created using a constructor method with the same name as the class.

To instantiate a class, the constructor is called in conjunction with the keyword `new`.

```
Book book1 = new Book();   // the constructor method call is underlined
```

Compare the two declaration statements below. The first creates an instance of the class `Book` using the keyword `new` and the default constructor. It is stored in a variable called `book1`.
The second stores the integer value `3` in a variable called `x`.

```
Book book1 = new Book();
     int x = 3;
```

`int` is the data type of the variable called `x`. It is a **primitive** type.

`Book` is the data type of the variable called `book1`. It is a **reference** type.

We could decide to define our own constructor that does something more than just create the object, for example:

```
public Book(){
   title = "Jaws";  //sets the initial value of the data field title to Jaws
}
```

This constructor would set the `title` data field to *Jaws*. The problem is it would do so for every instance of the `Book` class. But every book is not called *Jaws*! We can define a constructor that takes an input parameter and uses it to set the initial value of the data field. The title data can then come from outside the class, allowing each object to be different.

```
public Book(String t){
   title = t;   //sets the initial value of the data field title to the value of the input parameter
}
```

Preparation

1. Write a constructor for the `Book` class from lab 6 which takes 3 parameters and uses them to set the values of all 3 data fields.

2. Why is it useful to be able to pass parameters / arguments to constructors?

3. Both mutators and constructors enable you to store data in an object. How are they different?

4. Which of the following could be legal constructor headers for a class called MyClass?

```
public int MyClass(){}
public MyClass(){}
public MyClass{}
public myClass(int x){}
public MyClass(int x, y){}
public int MyClass(int y){}
public MyClass(y){}
public class MyClass(String s){}
public void MyClass(String s){}
public MyClass(int s, String x){}
```

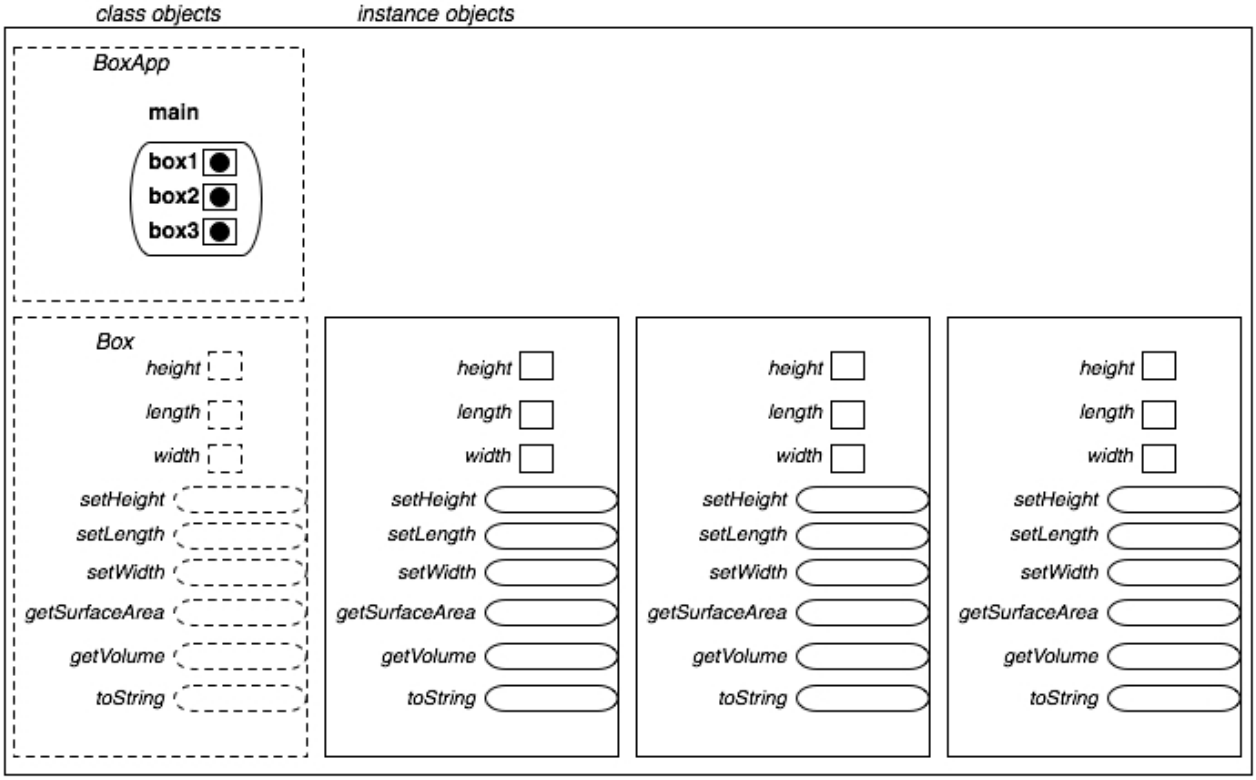
5. A class `Exam` has an `int` data field called `score`. Write a constructor for `Exam` which sets the value of `score` to the value of its input parameter.

```
public class Exam{
  private int score;

}//end class Exam
```

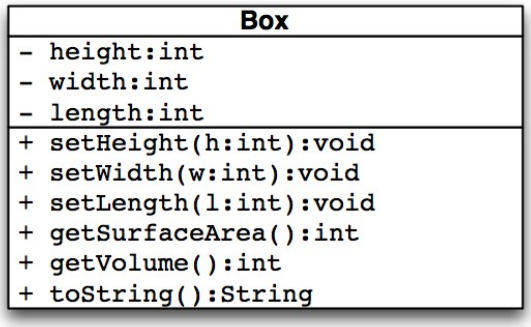
6. Write a single statement in the `main` method of the application class below. It should create an instance of `Exam` called `exam1` by calling the constructor you wrote in the previous exercise. Send the constructor the data required to set its data field `score` to `80`.

```
public class ExamApp{
  public static void main(String [] args){


  }
}
```

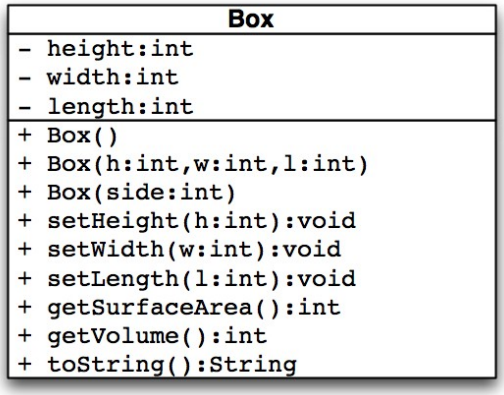
7. Refer to the labwork on the next page, and use it to complete the object diagram below by drawing arrows from the box1, box2 and box3 references to the corresponding instance objects. Write the values requested by the Lab Work in the data fields of each instance object.



Plans for Lab Work



Box UML class diagram after step 4.



The completed Box UML class diagram with its 3 constructor methods.

Lab Work

This lab will guide you through the creation of a support class called `Box` which calculates the volume and surface area of a box given its height, length and width. An application class called `BoxApp` will create instances of `Box` using the constructor methods of `Box`. Refer to the UML class diagrams on the previous page for an overview of the class structure required.

1. Write a support class `Box` which has three data fields as described in the UML class diagrams.

2. Write a mutator method for each of the data fields. Mutators are methods which set the value of the data field to the value of the input parameter. They usually have names beginning with `set` e.g. `setHeight`

3. `Box` will need (accessor) methods which compute and return the volume and surface area. Note that volume and surface area are **not** stored as data fields. The volume is defined by the product of the height, width, and length. The surface area can be calculated by summing the areas of the six sides. (Hint: There are actually only three unique sides).

4. Write a method called `toString()` that will return a string describing the height, length, width, volume and surface area in the Console (see step 11).

The first UML class diagram describes the class to this point.

5. Write a constructor that takes three parameters, and uses them to set the `height`, `length` and `width` data fields.

6. Write a replacement for the default constructor that creates a `Box.` (All dimensions will be set to the default value of `0`.)

7. Create a new class called `BoxApp`. This will be the application class. In the `main` method, create a `Box` object (`box1`) using the replacement for the default constructor.

```
Box box1 = new Box();
```

8. Write a statement to display what is returned by this object's `toString` method.

9. Compile and run your code. You should see something like the line below, because the data fields are all set to 0 (the default value).

```
Height is: 0, Length is: 0, Width is: 0, Volume is: 0, Surface Area: 0
```

10. Call the mutators on the `box1` object to set the data fields to height 4, length 4 and width 6 **before** you call the `toString` method but **after** you have created the object. (You could set the values by user input with Scanner if you wish.)

11. Compile and run. Now the data fields have values, so the output should look something like:

```
Height is: 4, Length is: 4, Width is: 6, Volume is: 96, Surface Area: 128
```

12. There is an easier way to create an instance of `Box` and fill its data fields with values. Have the `main` method create a `Box` object (`box2`) with height set to 3, length set to 4 and width set to 5 by calling the constructor you wrote in Step 5.

```
Box box2 = new Box(3, 4, 5);
```

13. Write a statement to display what is returned by this object's `toString` method.

Note: Which constructor is called depends on the number of input parameters. When you created `box1` you called the constructor with no parameters and the constructor that accepts no parameters is run. When you created `box2` you called the constructor with 3 parameters, so the constructor that accepts 3 parameters is run. This is called method **overloading**.

You now have two objects / instances of `Box` that have different states.

14. When you run your code, you should now generate (for the input values shown) the following output:

```
Height is: 4, Length is: 4, Width is: 6, Volume is: 96, Surface Area: 128
Height is: 3, Length is: 4, Width is: 5, Volume is: 60, Surface Area: 94
```

15. One possible variety of `Box` is a cube, where `height, length` and `width` are all the same. Write a third constructor for the class `Box` which takes just one input parameter and sets all the data fields to this value.

16. In the application class, make an instance of `Box` using this new constructor, setting the sides to `5`.

```
Height is: 4, Length is: 4, Width is: 6, Volume is: 96, Surface Area: 128
Height is: 3, Length is: 4, Width is: 5, Volume is: 60, Surface Area: 94
Height is: 5, Length is: 5, Width is: 5, Volume is:125, Surface Area: 150
```

Lab Completed

☐ preparation exercises complete	☐ comments	☐ three constructors
☐ return methods - volume, surface area	☐ working	☐ submitted

Date **Demonstrator's Initials**

Laboratory 8 Math and Random

Notes

Readings: L,D&C Chapter 3, Sections 3.4 - 3.6 (pages 86 – 97 [110 - 120]).
The Java libraries contain many useful classes. Some of these classes contain methods for performing commonly required specific tasks such as formatting output, performing mathematical calculations and producing a random number. We will explore the use of some of these classes in this lab.

Preparation

1. Write a statement which creates a `Random` object called `generator`.

2. Adapted from Exercise EX 3.6 (L,D&C page 105 [128]). Assuming that a `Random` object called `generator` has been created, what is the range of the result of each of the following expressions? The first two are done for you.

a. `generator.nextInt(20)` 0 -19

b. `generator.nextInt(8) + 3` returns an int between 0 and 7 with an 'offset' of 3, giving 3 - 10

c. `generator.nextInt(50)`

d. `generator.nextInt(5) + 1`

e. `generator.nextInt(45) + 10`

f. `generator.nextInt(100) - 50`

3. Evaluate these expressions and indicate the data type of the result.

	expression	evaluates to	data type of result
a.	`Math.pow(3, 3) =`		
b.	`Math.pow(2, 4) =`		
c.	`(int) –5.8 * Math.sqrt(9) =`  this is called casting – see lecture 4		
d.	`Math.floor(3.2) =`		
e.	`Math.floor(22.1) * Math.sqrt(9) =`		
f.	`Math.ceil(3.2) =`		
g.	`Math.round(3.2) =`		

There are 2 `round` methods in the `Math` class. Their short definitions are:

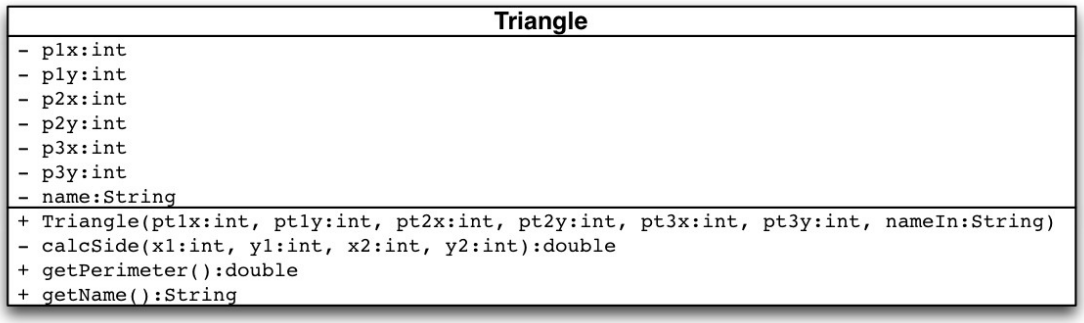
```
static long  round(double a)      Returns the closest long to the argument.
static int   round(float a)       Returns the closest int to the argument.
```

The method that is used depends on the data which is sent as a parameter. A floating point number (e.g. 3.2) is interpreted as a `double` unless it has F or f after it (e.g. 3.2F) in which case it is a `float`.
In **g.** above, the parameter is a `double`, so the first `round` method listed above is called.

4. Exercise EX 3.10 (L,D&C page 106 [128]). Write code statements to create a `DecimalFormat` object that will limit a value to 4 decimal places. Then write a statement that uses that object to print the value of a variable named `result`, properly formatted.

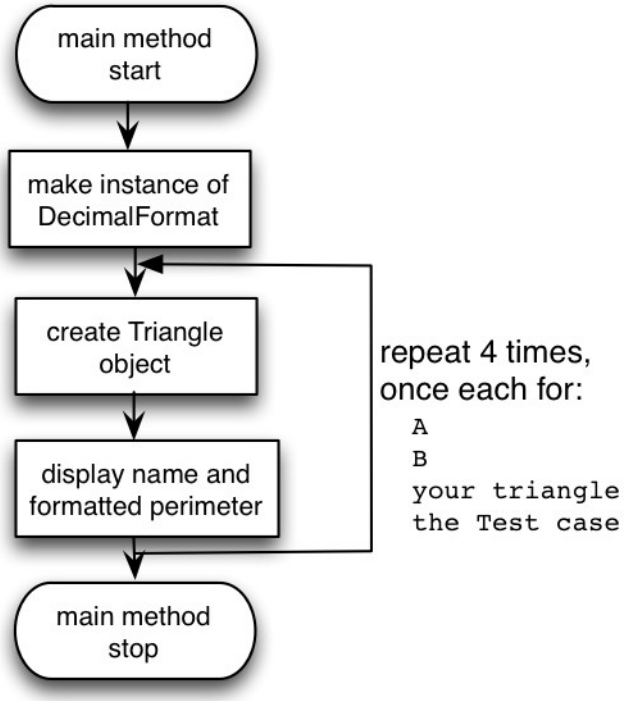
Plans for Lab Work

The UML class diagram for the class `Triangle` as described in the Lab Work



The UML class diagram for the application class (which isn't much use)

A flow diagram showing the sequence of events required in the `main` method.



Lab Work

Part 1

Specification: Write an application that calculates the perimeter of three triangles, given the coordinates for the three corners of each. Two triangles are drawn for you in Figure 8-1. Draw another of your own.

Compute the length of a triangle's side by calculating the distance between two points (corners) using the following formula:

$$distance = \sqrt{(x_2-x_1)^2 + (y_2-y_1)^2}$$

The perimeter of a triangle is the sum of the 3 distance calculations, 1 for each side.

What test case could you use to make sure your calculations are correct?
A good one might be (0,0) (3,0) (3,4) - this gives a right-angled triangle where the legs (sides next to the right angle) are 3 and 4 units long respectively, and the hypotenuse (side opposite the right angle) will be 5 units long according to the Pythagorean formula

$$c^2 = a^2 + b^2$$

The perimeter therefore will be 3 + 4 + 5 = 12 units.

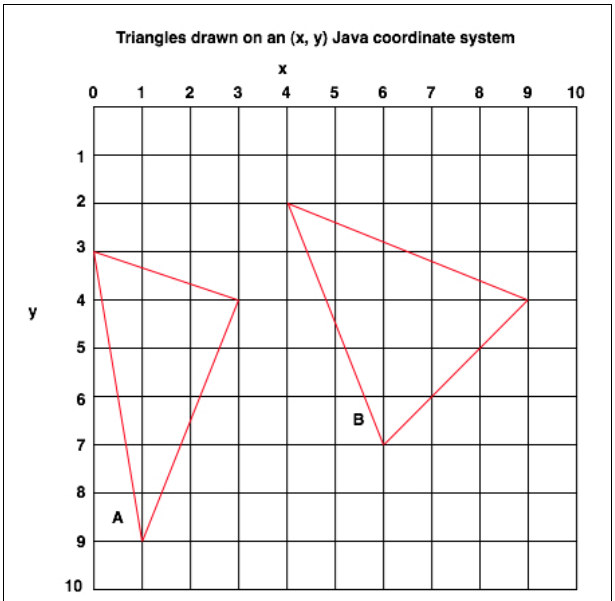


Figure 8-1

The UML class diagram for the support class (Triangle) is shown on the previous page. Each instance of the support class represents one triangle. The following paragraphs describe in a long-winded way what the diagram sums up more succinctly.

- The Triangle class should have data fields to store the coordinate information for the triangle as well as the triangle's name (A, B, . . .)
- The information should be set by a constructor.
- The Triangle class should have a method to calculate and return the length of a side. This method should take 4 `int` values as parameters, representing the x , y values for each end of the side.
- The Triangle class should have a method to add the lengths of its 3 sides together and return the perimeter.
- The Triangle class should have a method which returns the name of the triangle.

The application class should create a test case and three triangle objects, sending co-ordinate data to the constructor e.g.

```
Triangle a = new Triangle(0,3,3,4,1,9,"A");
```

The application class should display the perimeter of each triangle, formatted to 2 decimal places.

```
e.g. Triangle A perimeter is 14.63 units
```

BEFORE YOU START CODING plan your application class and your support class on paper. Look at the UML class diagrams and flowchart provided. Read and re-read the specifications above to make sure you understand the task. What data fields are required? What type of data will they store? Which data needs to travel from one class to the other via the constructor? What methods are required? Will they be void or return? When/how often will they be called? What parameters will they need?

Does it matter which direction you go around the triangle? Does it matter which point you start with?

Make sure your variables and methods have sensible descriptive names.

Extension:

Make the Triangle class calculate the length of the hypotenuse if it is given the length of the 2 legs of a right angle triangle rather than the 3 coordinate pairs.

What changes would you need to make if the corners weren't exactly on the grid intersections i.e. the coordinates aren't necessarily whole numbers?

Part 2

Write a program which displays a random integer within a specified range. The application class should read the high and low integer values required from the user, create an instance of the support class, and display the result of a call to your random range method. The support class should contain no data fields and just one method, which takes two `int` parameters (representing the high and low limits of the random range) and returns a random integer between (and inclusive of) the two parameters.

You will find listed below (and also in a file called `code.txt` in the **`LabFiles> Lab08`** folder) 2 comment blocks, 2 import statements, 2 class definitions and 3 methods. **This is all the code you will need.** Your job is to arrange this code into an application class and a support class as described above.

```
/**
 * RandomApp.java
 * Lab 8, Part 2, COMP160  2014
 * Displays a random integer between high and low limits
 * High and low values are typed in by the user.
 */

/**
 * RandomRange.java
 * Lab 8, Part 2, COMP160  2014
 * Contains a single method which returns random integer between high and low parameters.
 */
import java.util.Scanner;
import java.util.Random;

public class RandomApp{
}
public class RandomRange{
}

public static void main(String[]args){
  int lo = readInt("Enter lowest value");
  int hi = readInt("Enter highest value");
  RandomRange r = new RandomRange();
  System.out.println("Random integer between " + lo + " and " + hi + " : " +
                                                    r.randomRange(lo, hi));
}

/** Returns random integer between high and low parameters.*/
public int randomRange(int low, int high){
  Random generator = new Random();
  return generator.nextInt(high - low + 1) + low;
}

/** Returns an integer entered by the user*/
public static int readInt(String message){
  Scanner sc = new Scanner(System.in);
  System.out.println(message);
  return sc.nextInt();
}
```

Lab Completed

☐ preparation exercises complete	☐ comments	☐ Triangle class
☐ return method for each side	☐ random	☐ submitted

Date **Demonstrator's Initials**

Laboratory 9 Selection 1

Notes

Readings: L,D&C Chapter 4, Sections 4.1, 4.2, 4.3 (pages 112 – 130 [134 - 151]).
Selection is a fundamental part of all programming languages - we need to be able to select what to do next under particular conditions. This lab gives you practice formulating the conditions, which are boolean expressions and introduces the `if else` structure. You will get more practice writing constructors, formatting output and working with objects.

The logic for what appears to be a simple task may get quite complicated. Careful thought and planning are required, but that is not the end of it. After all that work, even if your code is producing answers, it will still need thorough, rigorous testing.

Preparation

1. Calculate the result of each expression if `a` is `5`, `b` is `9`, `c` is `14`, and `flag` is `true`.

a. `a == ( b + a – b)`

b. `( c == ( a + b ) ) || ! flag`

c. `( a != 7 ) && ( c >= 6) || flag`

d. `! ( b <= 12 ) && ( a % 2 == 0 )`

e. `! ( ( a > 5 ) || ( c < ( a + b ) ) )`

2. Given:
```
char ch = 'f', digit = '8';
String aString = "comp";
```
What does each of the following evaluate to?

a. `'a' <= ch && ch <= 'z'`

b. `digit > '0' || digit < '8'`

c. `aString.equals("comp")`

d. `"comp".equals(aString)`

3. Write statements which will swap the values stored in `low` and `high` if `low` is larger than `high`.

```
Scanner scan = new Scanner(System.in);
int low = scan.nextInt();
int high = scan.nextInt();
int tempStore = 0;
```

4. Write a boolean expression for each of the following:

a. `year` is divisible by 4

b. `waterLevel` is greater than 2.5 and less than 3.9 inclusive

c. `ch` is an uppercase letter

5. Write a `boolean` expression to assign a value of `true` to `child` if `age` is in the range `0` to `17`, inclusive; otherwise assign a value of `false`.

6. Write a `boolean` expression to assign a value of `true` to `pass` if `score` is in the range `50` to `100`, inclusive; otherwise assign a value of `false`.

7. What output is produced by the following code fragment?

```
int limit = 100, num1 = 15, num2 = 40;
if (limit <= 200){
   if (num1 == num2)
      System.out.println("lemon");
   System.out.println("lime");
}
System.out.println("grape");
```

8. The two code fragments below perform the same task. Which is easier to read?

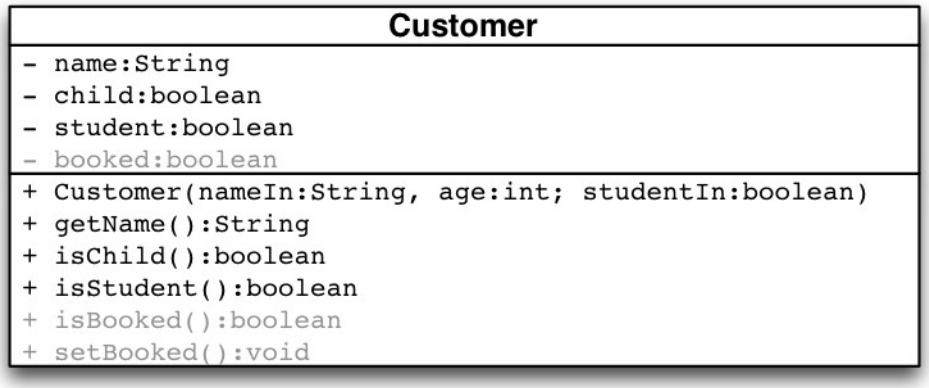
```
if( score > 50){
    if ( score < 100){
       printCertificate();
    }
 }
```

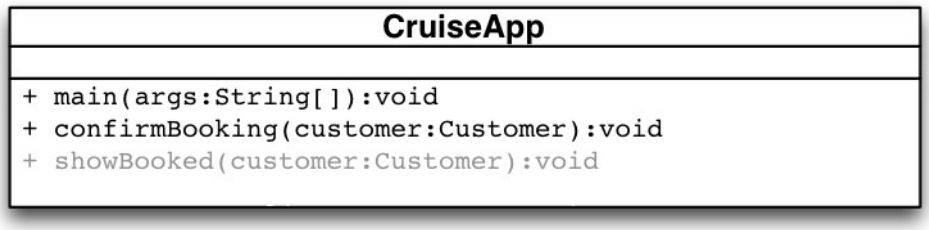
```
if (score > 50 && score < 100){
     printCertificate();
 }
```

Plans for Lab Work

Members in black are described in the initial setup of steps 1 and 2.

Members in gray are described in the later steps, 5 and 6.





Lab Work

If you already know what an array is, and how to use a loop, or are comfortable enough to look ahead to these concepts, please feel free to use them in your lab work.

The process of writing classes as described below uses incremental development, where a piece of program is developed and tested, then another piece of code or more functionality is added. After each step the program should compile, and (as soon as the main method is included) run. This method of breaking tasks down into successively smaller tasks is called a top–down approach.

A class (called `Customer`) will represent a potential customer for a meal and discounted harbour cruise package. Various discounts will be offered for children and students.

The application class will create instances of the `Customer` class, and calculate the costings according to the `child`/`student` status of each `Customer` object.

Since we have not covered reading from files yet, the data for creating each instance of `Customer` is provided for you as constructor calls in `CruiseApp`, the application class starter provided in the **coursefiles160** folder and printed below. The customer's name, age and a boolean value for having presented a student ID card are the information given.

```
public static void main(String[]args){
        // each Customer created with name, age, showed student ID card
       Customer customer1 = new Customer("Aaron Stott",17, true);
       Customer customer2 = new Customer("Betty Adams",17, false);
       Customer customer3 = new Customer("Corin Child",16, true);
       Customer customer4 = new Customer("Doris Stewart",25, true);
       Customer customer5 = new Customer("Edmond Cheyne",12, false);
       Customer customer6 = new Customer("Fiona Chaney",7, false);
       Customer customer7 = new Customer("Ged Still-Child",16, true);
       Customer customer8 = new Customer("Harry Adamson",20, false);
       confirmBooking(customer1);   //and so on
}
```

Fig 9.1 The main method of CruiseApp.java, provided in coursefiles160

1. From this starting point, and with the help of the UML class diagrams, plan and write a `Customer` class which stores the name and two `boolean` data fields, one for whether the customer is a student, the other for whether the customer is a child. A child is defined as between 5 and 16 years (inclusive).

The class should have accessors for these three data fields. Rather than prefix the boolean accessor's name with `get`, use `is` e.g. `isChild()`. The advantage of this will be easier-to-read `if` statements later on.

2. The main method will create `Customer` objects when run. Write another method in the application class called `confirmBooking`. This method will eventually display prices and confirm the booking. It will need to take a `Customer` object as a parameter. Initially, to test that your structure is working so far, just get this method to print out the data returned by the accessors. You will need to call this method 8 times from the main method, once for each `Customer` object (unless you already know about loops and arrays, in which case you could save yourself some typing).

```
Aaron Stott false  true  (the spaces won't be there unless you have explicitly requested them)
Betty Adams false  false
Corin Child true true
Doris Stewart false  true
Edmond Cheyne true false
Fiona Chaney true  false
Ged Still-Child true  true
Harry Adamson false  false
```

3. Now that the structure is working, you can concentrate on what happens inside the `confirmBooking` method.

This method should

- specify that a standard ticket price is 56.0 and a standard meal price is 30.0.
- calculate discounts .
  - ► Students and children receive half price tickets, but anyone else gets a 20% discount.
  - ► Children receive half-price meals. Everyone else gets a 10% discount.
- display the prices and the total for each customer.

```
Aaron Stott   Ticket price:28.0    Meal price:27.0   Total price:55.0
Betty Adams   Ticket price:44.8    Meal price:27.0   Total price:71.8
Corin Child   Ticket price:28.0    Meal price:15.0   Total price:43.0
Doris Stewart  Ticket price:28.0   Meal price:27.0   Total price:55.0
Edmond Cheyne  Ticket price:28.0   Meal price:15.0   Total price:43.0
Fiona Chaney   Ticket price:28.0   Meal price:15.0   Total price:43.0
Ged Still-Child Ticket price:28.0  Meal price:15.0   Total price:43.0
Harry Adamson  Ticket price:44.8   Meal price:27.0   Total price:71.8
```

4. Format the price information as you would expect it to be displayed.

```
Aaron Stott   Ticket price:$28.00  Meal price:$27.00 Total price:$55.00
```

5. So far, so good. The next component of the task is to present the information tidily, one customer at a time, and ask for confirmation of the booking. The user should confirm by pressing **`y`** or **`Y`**.

The booking information will need to be stored. This will require another `boolean` data field in the `Customer` class. Write a mutator and an accessor for it.

Back in the application class, finish your `confirmBooking` method. If the choice is **y** or **Y**, set the customer's `booked` field to `true` and display `"Booked"`.

```
Aaron Stott
Ticket price: $28.00
Meal price:   $27.00
Total price:  $55.00
Confirm booking for Aaron Stott(Y/N)
Y
Booked
```

6. Lastly, the business would like a listing of those who have booked.

In the **application** class, write another method which takes an instance of `Customer` as a parameter. All this method should do is print out the name of the customer if (and only if) they are booked. You will need to call this method 8 times (once again, we are looking forward to having loops and arrays to make our job easier).

The method's print out might look something like this:

```
Aaron Stott is booked
Corin Child is booked
Edmond Cheyne is booked
Ged Still-Child is booked
```

Lab Completed

☐ preparation exercises complete ☐ comments
☐ working ☐ submitted

Date **Demonstrator's Initials**

Laboratory 10 Selection 2

Readings: L,D&C Chapter 4, Sections 4.2 and 4.4 (pages 116 – 127, 130 – 134 [138 – 148, 151 - 155]).

Preparation

1. What will be displayed after each of the following conditional statements have been performed?

(You will need to refer to L,D&C page 124 [146])

```
int a = 7;
int b = 8;
int c = 2;
System.out.println( (a < b) ? a : b);            _____________

System.out.println( (a == b) ? a : b);           _____________

System.out.println( (a < b) ? a + c : a – c);    _____________
```

2. For what value(s) of `a` will the condition `(a > 0 || a < 0)` be `false`?

3. For what value(s) of `a` will the condition `(a < 100 || a > 0)` be `true`?

4. Explain the difference between

```
a = 0;
if (b > 0) a++;
else if (c > 0) a++;
```

and

```
a = 0;
if (b > 0) a++;
if (c > 0) a++;
```

5. What will the output of this code be if `a` has the value 25?

```
if (a > 10){
     System.out.println("a is greater than 10");
} else if (a > 20){
     System.out.println("a is greater than 20");
}
```

6. In the two code fragments below, tick the statements which will be performed if `a` has a value of `12`

1.
```
if(a < 10)
  b = 5;
  c = 6;
  System.out.println(a);
```

2.
```
if(a < 10){
   b = 5;
   c = 6;
}
System.out.println(a);
```

Plans for Lab Work

Lab Work

Programming Projects PP 4.1 (L,D&C page 163 [182]). Design and implement an application that processes a series of integer values representing years. The purpose of the program is to determine if each year is a leap year (and therefore has 29 days in February) in the Gregorian calendar. A year is a leap year if it is divisible by 4, unless it is also divisible by 100 but not by 400.

For example, the year 2010 is not a leap year, but 2012 is. The year 1900 is not a leap year because it is divisible by 100 but not by 400. The year 2000 is a leap year because it is divisible by 100 and also divisible by 400. Produce an error message for any input value less than 1582 (the year the Gregorian calendar was adopted). This information is summarised in Table 10-1.

The application should display the result to the console window, as below

```
2010: is not a leap year
2012: is a leap year
1900: is not a leap year
2000: is a leap year
1565: predates the Gregorian calendar
```

	/4	/100	/400	Leap
2010	no	no	no	**no**
2012	yes	no	no	**yes**
1900	yes	yes	no	**no**
2000	yes	yes	yes	**yes**

Table 10-1

Just one class is sufficient for this task. The code, however, should not be without structure.

The statements which “process” the year and display the output should be in a method which takes each individual year as a parameter.

The `main` method should call this method at least 5 times, once for each of the 4 "test case" years listed in Table 10-1 above, and once more for a year before 1582. e.g.

```
leapYear(2010);
leapYear(2012);
leapYear(1900);
leapYear(2000);
leapYear(1565);
```

Lab Completed

☐ preparation exercises complete ☐ comments
☐ working ☐ submitted

Date **Demonstrator's Initials**

Laboratory 11 Strings

Notes

Readings: L,D&C Chapter 3, Section 3.2 (pages 80 – 83 [104 -107]).

While `int` and `double` are primitive data types, `String` is a reference data type. `String` is a class in the Java libraries (API). When you create a `String` you are creating an instance of this class. The String class contains many methods which may be called on any `String` object. This lab will explore some of them.

Background

Think of a `String` as a series of characters stored in a sequence. Each character is in its own position and has an index number. The numbers start at 0. (See the diagram below.)

A String has a name.
```
String animal = "tree frog";
String colour = "red";
```

> The `String` class is in the `java.lang` package which is imported automatically to all Java programs. When a statement like `String word = "duck"` is run, an object of type `String` is created.
>
> If a variable is an instance of a Java class its data type is a **reference** type. Reference types refer to an object. A reference variable doesn't hold the object itself. It holds the memory address of the object (where it can be found in memory).
>
> The primitive data types are of known size for storing in computer memory. A reference data type does not have a set size. This is clearly demonstrated by the String class. A `String` can be of any length.

Knowing the length of a `String` may be very useful in some circumstances. The `String` class has a method `length` which returns the length of the string. It can be called like this:

```
int stringLength = colour.length(); //assigns the length of a String called colour to an int variable
System.out.println(colour.length()); //displays (doesn't store) the length of the String called colour
```

- The length returned by the `length()` method is the count of characters as humans count, starting at 1. It is an integer.
- Each character has a position or index in the string. The index starts at `0` and finishes at `length() - 1`.
- A space is a character like any other.

The String declaration: `String songLine = "While my guitar gently weeps";` is the basis for the diagram below.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
Character	W	h	i	l	e		m	y		g	u	i	t	a	r		g	e	n	t	l	y		w	e	e	p	s

Trying to access an index out of the valid range will produce a `"StringIndexOutOfBoundsException"` error when the code is run. In DrJava this appears in the Interactions pane in red. (This is a run-time error.)

For the declaration above, `songLine.length()` will return the value _____

For the declaration above, the last index position of `songLine` is _____

The `String` class has many methods which can be called. To access the methods, we use dot notation to identify which particular `String` the method is being called on e.g.

```
animal.length()  colour.indexOf('e')  songLine.charAt(3)  animal.substring(2,4)
```

The text book describes some of these String methods on page 105. A complete list can be found in the APIs/libraries.

Preparation

1. What value is stored in the variable after each of the following statements have been executed? Assume the declaration

```
String drink = "Ginger ale";
```

The API shows good examples of substring being used. Remember that the length of a string is the number of characters it contains (e.g. 10 for `drink` shown above) but the index positions start at 0.

index	0	1	2	3	4	5	6	7	8	9
character	G	i	n	g	e	r		a	l	e

a. `char c = drink.charAt(0)`

b. `char c = drink.charAt(drink.length() - 1)`

c. `String s = drink.substring(1)`

d. `String s = drink.substring(3, drink.length() - 2)`

The String method `indexOf` isn't listed on page 81 [105] (L,D&C), but it is in the Java library documentation. For your convenience, here is the description.

> **`public int indexOf(int ch)`**
> Returns the index within this string of the first occurrence of the specified character.
> **`public int indexOf(int ch, int fromIndex)`**
> Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
> i.e. starts searching at `fromIndex`

Which of these methods is called depends on the parameters used in the method call. A single `char` or `int` e.g. `indexOf('k')` will call the first version. (This is called **method overloading** and is discussed later.)

A `char` or `int` followed by an `int` e.g. `s.indexOf(97,3)` will activate the second version.

These methods can be sent either a character e.g. **`'e'`** or a Unicode value e.g. **`101`**. See Appendix C page 816 for a listing of common Western unicode characters and their numeric values.

```
String s = "cabbage";
System.out.println(s.indexOf(97));  //these 2 lines of code
System.out.println(s.indexOf('a')); //perform the same function
```

e. `int first_e = drink.indexOf('e')`

f. `int second_e =  drink.indexOf('e', 5)`

g. The variable `second_e` is intended to store the position of the second `'e'` found in the string. If the string stored in `drink` is changed to `"green tea"`, would the statement `second_e =  drink.indexOf('e', 5)` still find the second `'e'`? _____

The **5** in part **f.** is not flexible – it is "hardwired" and will not adapt if the String is changed.
Adapt the expression used in **f.** so that it finds the position of the second `'e'` in the String `drink` in terms of the position of the first 'e'.

```
int second_e =  drink.indexOf('e',              )
```

h. `drink.toUpperCase()`

i. `drink.toLowerCase()`

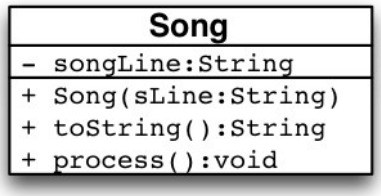
2. Look up the `String` class in the library documentation - it is in the `java.lang` library. Refer back to Lab 3, Part 2 (page 19 for instructions on accessing the library (Java API) documentation. Scroll down until you find its method `length`.

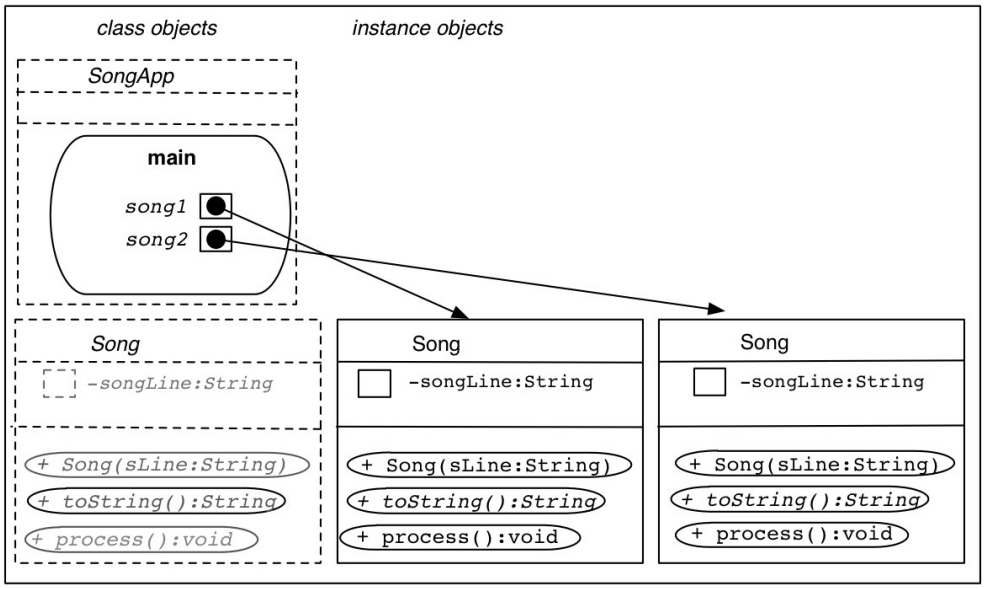
a. What is the short definition given?

b. What is the return type of `length`?

3. Given the statement `String title = "Modern Times";` write a statement which will declare a variable called `titleLength` and assign to it the length of the string called `title`.

Planning for Lab Work





Lab Work

Note: it is not a good idea to call any class you write `String` because this will prevent your code from accessing the `String` class already written in the APIs.

1. Make a new class called `Song`. It should have a (private) **data field** called `songLine` to hold a `String` value, a **constructor** which sets the value of the data field and a `toString` **method** to return the value stored in the data field.

2. Still in the `Song` class, write a `void` method called `process()`. The method can remain empty for now.

3. Make a new application class called `SongApp` (using `App` in the application class's name helps people to locate the `main` method in programs involving more than one class). In the `main` method:

- create an instance of `Song`, initialising the data field to "`While my guitar gently weeps`".
- print out (display) the value stored in the data field of your instance object. You cannot access this data directly from another class because it is private, but you can access it through the public method which returns its value.
- write one more statement which calls the `process` method.

If you compile and run now you should just see the String stored in the data field displayed.

4. Still in the `main` method, make another instance of `Song`, initialising its data field to "`Let it be`". As before, display the value stored in this instance's data field and call its `process` method.

The structure of your program is now in place. The application class creates two instance objects, sends them data, and calls their methods either to perform tasks or retrieve information. You can think of the application class as a manager class, whose job is to create instances of other classes and issue orders which organise the data in and out of them.

All the remaining code is written in the `process` method of the support class.

5. Print out the length of `songLine`. Label the output rather than just showing a number e.g. `Length is: 9`.

6. Print out the last character in `songLine` using `charAt`.

7. Your next task is to print the first two words of the `songLine` data field on one line and all subsequent words on the next line.

(Make sure you have completed and understood Preparation Exercise 1 to gather the knowledge you need for this task. You are welcome to use local variables for storing information. Use descriptive names to aid readability.)

8. Print out the first letter of the third word of `songLine`.

9. Print out `songLine` in uppercase. Use `toUpperCase`.

10. Print out `songLine` with the spaces replaced by the letter '`x`'.

11. Use `indexOf` to find the position of the first occurrence of the letter `'b'` in `songLine`. Display the result in a `println` statement. What is the integer returned if the target character is not found in the string?

_____

12. Has the value stored in `songLine` changed as you have worked through the exercise above? Print out `songLine` to check your answer.

_____

13. Make one more instance of the `SongLine` class, this time sending `"Penny Lane"` to the constructor. Run your code again. You will see a run time error. Use an `if` statement wherever necessary to avoid attempting to access index positions which don't exist.

Lab Completed

☐ preparation exercises complete	☐ comments	☐ embedded questions answered
☐ descriptive names, output labelled	☐ working	☐ submitted

Date **Demonstrator's Initials**

Laboratory 12 Repetition 1

Notes

Readings: L,D&C Chapter 4, Sections 4.5, 4.6, 4.7 (pages 134 – 151 [155 – 172]).
Much of the power of computing lies in the ability to repeat tasks a certain number of times, or until a particular state has been reached, or until the end of the job. Loops are the key to repetition, and there are a number of different loops which may be used. In this lab we will look at the `while` and the `do-while` loops.

Preparation

1. Exercise EX 4.7 (L,D&C page 161 [180]). What output is produced by the following code fragment?

```
int num = 0, max = 20;
while (num < max){
   System.out.println(num);
   num += 4;
}
```

2. Exercise EX 4.8 (L,D&C page 161 [180]). What output is produced by the following code fragment?

```
int num = 1, max = 20;
while (num < max){
   if (num % 2 == 0)
      System.out.println(num);
   num ++;
}
```

3. Write a `while` loop which counts the number of spaces in a String called `mySentence`. Declare any variables required.

4. What is the minimum number of times a `while` loop is executed?

5. What is the minimum number of times a `do-while` loop is executed?

6. Adapted from Exercise EX 4.19 (L,D&C page 163 [181]).

a. Finish the code below so it reads exactly 10 integer values from the user and prints the highest value entered.

```
import java.util.Scanner;
public class HighApp{
    public static void main(String[]args){
        Scanner scan = new Scanner(System.in);
        int count = 1;
        System.out.println("Enter number " + count);
        int highest = scan.nextInt(); //first, assume highest
        count++;
        while(____________){ // loop continues while the condition remains true
            System.out.println("Enter number " + count);
            int input = ____________________
            if (input > highest){
                ________________________
            }
            count++;
        }
        System.out.println("The highest number entered was " + highest);
    }
}
```

b. Why is the initial value of `highest` set to the first input value rather than 0?

Plans for Lab Work

Lab Work

These programs can both be written just in the main method of an application class.

Part 1

Programming Projects PP 4.3 (L,D&C page 163 [182]).

Design and implement an application that reads an integer value and prints the sum of all even integers between 2 and the input value, inclusive. Print an error message if the input value is less than two. Prompt the user when requesting input, and label the output clearly.

A typical output might be:

```
Enter an integer greater than 1
  7
Sum of even numbers between 2 and 7 inclusive is: 12
```

or

```
Enter an integer greater than 1
 -3
Input value must not be less than 2
```

Part 2

Adapted from Programming Projects PP 4.12 (L,D&C page 165 [184]).

Design and implement an application that reads a string from the user, then determines and prints the number of vowels and consonants which appear in the string.

Hint: Use a `switch` statement inside a loop. If you omit the word `"break"` between cases, you can list several cases and do the same thing for all of them e.g. for a variable called `grade`, of type `char`, which has already been declared and assigned a value:

```
switch(grade){
  case 'A':
  case 'B':
  case 'C':
    System.out.println("Pass");
    break;
  case 'D':
  case 'E':
    System.out.println("Fail");
    break;
  default:  //if the character stored in grade is not listed as a case, this will apply
    System.out.println("Uncoded input");
}
```

A typical program output for this exercise might be:

```
Enter a sentence
 My dog has fleas!
Sentence is : My dog has fleas!
VowelCount : 4
ConsonantCount : 9
```

Lab Completed

- ☐ preparation exercises complete
- ☐ comments
- ☐ Part 1 sum
- ☐ Part 2 vowels
- ☐ submitted

Date **Demonstrator's Initials**

Laboratory 13 Repetition 2

Notes

Readings: L,D&C Chapter 4, Section 4.8 (pages 151 – 157 [172 – 177]). Reread L,D&C Chapter 4 Section 4.3 (pages 127 – 130 [149 – 151]). Chapter 5, Section 5.5 (pages 199 – 203 [218 – 220])
The `for` loop is particularly useful for the systematic processing of a series of related data. In this lab you will use a `for` loop to access each character in a `String`.

Preparation

1. Write a `for` loop that displays the following set of numbers:

0, 10, 20, 30, 40, 50 . . . 1000

2. Write a `for` loop to print the odd numbers from 99 down to 1 (inclusive).

3. Finish the `for` loop which prints out all the ASCII characters from `'A'` to `'z'` on one line, with a space between each.

```
for(char c = 'A';                          ){
    System.out.print(             );
}
```

4. What output will the nested for loop below produce?

```
for(int i = 0; i < 3;i++){
    for(int j = 0; j < 3;j++){
        System.out.print("*");
    }

    System.out.println();
 }
```

5. What output will the nested for loop below produce?

```
for(int i = 0; i < 3; i++){
    for(int j = 0; j <= i; j++){
        System.out.print("*");
    }

    System.out.println();
}
```

Planning for Lab Work

Design overview – to produce a sorted string (without using any data structures not yet covered in the course)

make the phrase lower case

make a new empty string which will be built up into the sorted string

find every `'a'` in the lower case phrase and append it to the sorted string

find every `'b'` in the lower case phrase and append it to the sorted string

find every `'c'` in the lower case phrase and append it to the sorted string

etc.

Algorithm:

get first phrase from user
make the phrase lower case
create a new empty string (ready to store each letter in alphabetical order)
for each letter of the alphabet (a to z)
for each index position in the phrase string
if the char at that position matches the letter, append it to the new string
(first it will find all the a's, then the b's etc.)

repeat the process for the second phrase (the process could be in a method which is reused for each phrase)

compare the 2 sorted strings – if they are equal, they are anagrams. (remember Lab 9, Prep 2 c, 2 d)

Lab Work

An anagram is a word or a phrase made by transposing the letters of another word or phrase; for example, "parliament" is an anagram of "partial men," and "software" is an anagram of "swear oft." Write a program that figures out whether one phrase is an anagram of another phrase. The program should ignore white space and punctuation. The phrases should be typed in at the keyboard.

This program can just be written in the main method of an application class.

Typical expected output:

Enter first phrase

replays

Enter second phrase

parsley

aelprsy are the letters of replays in order

aelprsy are the letters of parsley in order

replays is an anagram of parsley

Some more word pairs and phrases to test with:

Resistance Ancestries	Gainly Laying	Admirer Married	Orchestra Carthorse
Creative Reactive	Deductions Discounted	Listen Silent	
Paternal Parental	Angered Enraged		

A highwayman	Away! Hang him!
Internal Revenue Service	I never return even a slice.
Shakespeare	Seek a phrase.
Received payment	Paid me every cent.

Lab Completed

☐ preparation exercises complete ☐ comments
☐ working ☐ submitted

Date **Demonstrator's Initials**

Laboratory 14 Graphical Objects

Notes

Readings: L,D&C Appendix F (pages 966 – 976 [834 – 840]) and Chapter 3, Section 3.8 (pages 100 – 102 [123 – 125]). It is vital for this lab that you read and understand the text book pages covering graphical objects and the `paintComponent` method.

You have already created graphical objects - in Lab 6 you made objects of the type `MyFrame`. In this lab, you will work with three classes: an application class and two support classes. One of the support classes will create and work with several instances of the other support class.

Preparation

1. Given the two classes below, what will be printed out when `TestApp` is run?

```
public class TestApp{
  public static void main(String[]args){
    Test t = new Test();
    System.out.println(t.toString());
  }
}
```

```
public class Test{
  private int x;
  private int y;

  public Test(){
    int x;
    x = 3;
    this.x = 4;
  }
  public String toString(){
    return("The value of x is " + x );
  }
}
```

2. A series of statements in some class looks like this:

```
SecondClass s1 = new SecondClass();
SecondClass s2 = new SecondClass();
System.out.println(s1.toString());   //if no toString method has been written, these will
System.out.println(s2.toString());   //print out the object's memory address (reference)
s1 = s2;
System.out.println(s1.toString());
System.out.println(s2.toString());
```

If the first two `println` statements produced the output

```
SecondClass@b890dc
SecondClass@123c5f
```

what will the output of the other two `println` statements be?

3. The `Graphics` method `drawString` requires three inputs - the string to be 'drawn' and 2 integers representing the coordinates. The code

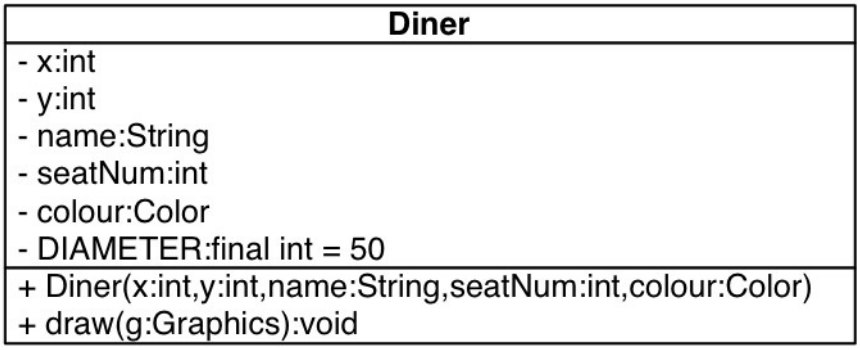
```
int i = 3, x = 100, y = 100;
g.drawString(i, x, y);
```

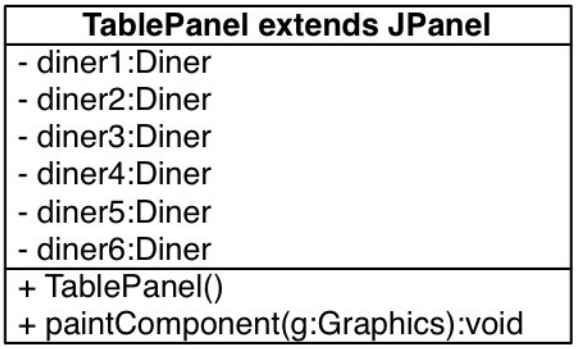
will fail. Why?

4. The `Integer` wrapper class in `java.lang` has a `toString` method which converts the integer to a string. Rewrite the `drawString` statement from Ex. 3 above so that the code will display a string representation of `i`.

```
g.drawString(
```

Plans for Lab Work





Lab Work

Programming Projects PP F.21 (L,D&C page 984 [847]). Write a program that displays a graphical seating chart for a dinner party. Create a class called `Diner` (as in one who dines) that stores the person's name, gender and location at the dinner table. A diner is graphically represented as a circle, colour-coded by gender, with the person's name printed in the circle.

Assume there will be 6 guests around the table. The `Splat` program listing (pp 973 – 976 [837 - 840]) will be a useful example to follow.

Your program should produce a window something like the picture in Figure 14.1. Don't waste too much time on text positioning - that is not the purpose of this exercise.

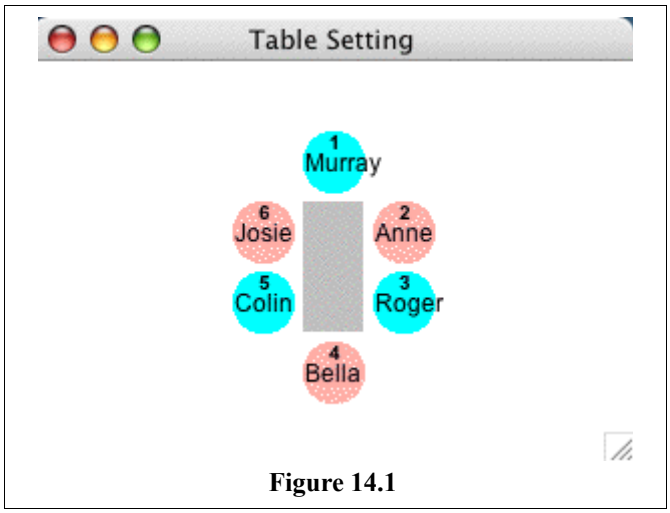


Figure 14.1

In the `Splat` class the `main` method creates the `JFrame` and adds not just a `JPanel` but a user-created extension of `JPanel` called `SplatPanel`.

All the behaviours of the `JPanel` class (such as `add`, `paintComponent`, `setBackground` . . .) are available to `SplatPanel` because of the keyword `extends` in its class header.

Each circle is an instance of the Circle class, stored in `Circle` data fields, created using `Circle` constructors and drawn in the `paintComponent` method using calls to each circle's `draw` method.

Specifications for a Solution for Diner

The first thing to do **before you go any further** is work out the coordinates for your drawing elements - each of the 'diner' circles and the table. You will need to have set a size for your drawing window/panel (we used 300 by 300) and a diameter for your circles (we used 50). Write the coordinates for each element on Figure 14.1 (previous page).

All the graphics drawing in the program will be handled by the `paintComponent` of a `JPanel`, in this case the `TablePanel` class (which extends `JPanel`) described below. The `paintComponent` method of `TablePanel` will call the `draw` method on each `Diner` object, passing it a reference to its `Graphics` object so that `Diner` can draw itself in the `JPanel` graphics context.

The Diner class *(compare with the Circle class LD&C page 840)*

The `Diner` class represents a single diner by means of a filled circle. The circle has an x,y location, a name, a colour representing gender and a number representing the seating position. All this information is sent to it when it is created, then stored in its data fields. Because the diameter of the circle is the same for each diner, it can be set literally in the `Diner` class. It should also be a constant, as it doesn't change. The `Diner` class contains a method to draw itself using the data in its data fields.

1. The `Diner` class will need data fields representing:

- an x location

- a y location

- the name of the diner

- the colour of the circle (the class will need to import `java.awt.*` in order to use `Color`)

- the number of the seating position

- the diameter of the circle, set to a standard size

2. The `Diner` class will need a constructor which sets the data fields representing x and y location, name, colour and seating position to the value of its parameters.

3. The `Diner` class will need a method (call it `draw`) to draw itself which

- takes a graphics object as a parameter

- sets the drawing colour to the colour held in the data field

- draws the circle at the x,y location

- sets the drawing colour to the colour required for text

- sets the font face and size for the name (see the box on the next page for tips on setting a font)

- prints the name at a location relative to the circle. **Note:** `drawString(String str, int x,int y)` locates the **baseline** of the leftmost character at position `(x, y)`. Long names will spill out of the circle.

> The `drawString` text is drawn **above** the `x` value given,
> not below as is the case for `drawOval` and `drawRect`.

- sets the font face and size for the seating position. **Note:** the font size for the place number is smaller than that for the names.

- prints the seating position at a location relative to the circle

Tips for Setting a Font

The code

```
xxx.setFont(new Font("Courier", Font.PLAIN, 8));
```

sets the font for a `Graphics` object called `xxx` to `Courier, plain, 8 point`,

or you can set the font by creating an instance object of type `Font`:

```
Font boldH14 = new Font ("Helvetica", Font.BOLD, 14);
```

and using it later on a particular `Graphics` object e.g.

```
page.setFont(boldH14);
```

4. The `Diner` class will need to import `java.awt.*` in order to use `Color` and `Graphics`.

This class is now complete, so this could be a good opportunity to compile it to check for errors.

The TablePanel class *(compare with the SplatPanel class LD&C page 839)*

The `TablePanel` class needs to be an extension of the `JPanel` class (it wants to inherit all of the methods of `JPanel`, as well define some extra ones that you will write yourself). It creates the 6 `Diner` objects and calls their `draw` method, as well as drawing the `JPanel` and the rectangle representing the table.

1. The `TablePanel` class will need to extend `JPanel`.

2. The `TablePanel` class will need to declare data fields to hold references to 6 `Diner` objects.

3. The `TablePanel` class will need a constructor which

- creates 6 `Diner` objects, sending each appropriate values
- sets the size and background colour of the panel

4. The `TablePanel` class will need a `paintComponent` method which

- takes a `Graphics` object as an input parameter
- draws a panel of a size and background colour by calling the `paintComponent` method of the inherited `JPanel` by calling

  `super.paintComponent(`*whatever your Graphics object is called*`)`

- calls the `draw` method of each `Diner` object, passing the `Graphics` object as a parameter
- draws the rectangle representing the table

5. The TablePanel class will need to import `java.awt.*` in order to use `Color` and `Graphics`. It will also need to import `javax.swing.*` in order to use `JPanel` methods.

This class is complete. You might like to compile it to check for errors.

The Application class *(compare with the Splat class LD&C page 837)*

1. The application class will need a `main` method which

- creates a new `JFrame` object

- adds a new instance of the `TablePanel` class to it

- calls the `setDefaultCloseOperation`, `pack` and `setVisible` methods

- imports `javax.swing.*` in order to use `JFrame` methods.

Lab Completed

☐ preparation exercises complete ☐ comments
☐ working ☐ submitted

Date **Demonstrator's Initials**

Laboratory 15 Arrays

Notes

Readings: L,D&C Chapter 7, Sections 7.1 & 7.2 (pages 340 -351 [354 - 365]).
The power of loops can be realised when data is stored in a systematic way. An array is the first data structure we will use.
This lab uses both `for` and `foreach` loops, loops with sentinel values, application and support classes, `void` and `return` methods, methods with and without parameters, constructors, arrays, random number generators and boolean conditions. If you are able to write and understand the code for this task, you have an excellent grasp of the concepts presented in the first half of the course.

Preparation

1. Declare an array of `int` called `scores` that can hold 10 values.

2. Declare an array of `String` called `words` that can hold 5 words.

3. Declare an array of `double` called `height` which can hold 20 values.

4. Declare an array of `String` called `family` and use an **initialiser list** to fill it with the names of your family members.

5. Write a statement which prints out the `length` of the array called f`amily`.

6. Write a `foreach` loop which prints out the name of each family member in the array called `family` on a new line.

7. Write a `for` loop which prints out the name of each family member in the array called `family` on a new line.

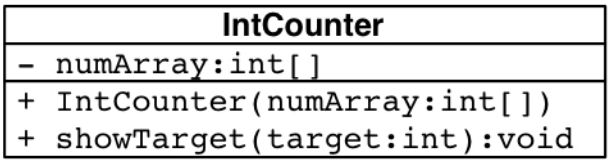
Plans for Lab Work

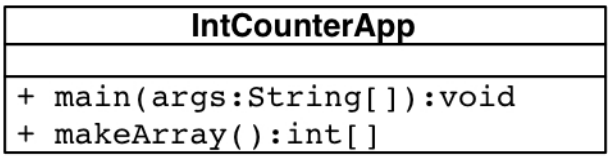
Support class

data field	**array of int**
constructor	**takes an array of int as a parameter** (an array of int is of type `int[]`)
	sets the value of the data field
	displays every element of the array (foreach) and the length of the array on 1 line
showTarget method	**takes an int as a parameter**
	produces a display line each time the target integer is found in the array, stating in which index/position the target is found

Application class

makeArray method	**creates an array of int of random size between 5 and 10**
	fills the array with random values between 0 and 4
	returns the array
main method	**in a loop, asks the user for the target integer (3 times)**
	makes an instance of the support class, sending it an array of int returned by makeArray
	calls the showTarget method of the instance, sending it the target integer





Refer to Lecture 14, topic `this`, for how to successfully use the same name for a data field and constructor parameter.

Lab Work

The purpose of the program you will write for this lab is to display the array position of every occurrence of a 'target' integer that is stored in an array of `int`. We are asking you to do this with a very specific class structure, which is described carefully below, and summarised on the preceding page.

The support class will contain a single **data field** of type `array of int`. The support class will have a **constructor** which is sent a reference to an `array of int` as a parameter, and uses this to set the value of the data field. A foreach loop in the constructor will display all the elements stored in the array and the length of the array on one line (see Figure 16.1). The support class also has a `showTarget` **method** which is sent a target integer as input and uses a `for` loop to access every position of the array in turn, and print out the position number (index) of the positions which hold the target value. (This target integer will be read in from the user, using a `Scanner` object in the `main` method, then passed to the `showTarget` method in the support class.)

The **application** class has a **method** which returns an `array of int`. Let's call it `makeArray`. This method uses the random number generator to decide how big the array is going to be (between 5 and 10 elements), fills it with random `int` values (between 0 and 4), then returns the array.

The **`main` method** contains code in a loop which asks the user to specify the target integer then makes an instance of the support class, sending it a reference to an array returned by the `makeArray` method. It will then call the `showTarget` method (passing it the target value). The loop will finish after three iterations.

Hints:

Use **`"\t"`** to indent the output by 1 tab stop.

An array of `int` is of type **`int[]`**

```
Which number do you wish to find?
[2]
4 4 1 2 0 0 2 4 Array is of length 8
   There is a 2 in position 3
   There is a 2 in position 6

Which number do you wish to find?
[3]
0 2 3 3 1 2 0 Array is of length 7
   There is a 3 in position 2
   There is a 3 in position 3

Which number do you wish to find?
[3]
2 0 4 2 4 2 1 4 0 Array is of length 9


Finished
```

Figure 16.1

If you have the time and the inclination, see if you can add a statement that reports that the target integer isn't found when appropriate e.g.

`"There are no 3's in this array"`

Lab Completed

☐ preparation exercises complete ☐ comments ☐ working
☐ structure as described ☐ submitted

Date **Demonstrator's Initials**

Laboratory 16 Two-Dimensional Arrays

Notes

Arrays of objects and two-dimensional arrays.

Readings: L,D&C Chapter 7, Sections 7.3, 7.4, 7.5, 7.6 (pages 351- 366 [365 – 384]).

1. Draw the data structure which is created by this code, and the values that are stored in it:

```
int cols = 4;
int rows = 5;
int [] [] table = new int[rows][cols];
for (int row = 0; row < rows; row++){
   for (int col = 0; col < cols; col++){
        table[row][col] = row * col;
   }
}
```

2. Using a nested for-each loop, write code to print out the values stored in `table` (Q1 above), with a tab between each column in the row, and each row on a new line.

Lab Work

Part 1

The array for this exercise is an array of `String` objects which is filled by the user via a `Scanner` object.

1. In the `main` method of a new class, declare an array of `String` called `fruits` that can hold 3 `String` objects.

2. Write a foreach loop to print out all the elements in the array. Compile and run. You will see the elements are all null at this point. Unlike arrays of primitive types which are instantiated to 0 or empty, arrays of reference types need to be instantiated.

3. Between the declaration and the foreach loop, use a loop to fill each array position with the name of a fruit. Use a `Scanner` object to get a line of text from the user. Don't forget to prompt the user with a message to explain what is expected of him/her.

4. If you compile and run at this point you should be able to enter three strings then see them displayed in a list.

5. Change the statement in the foreach loop so it prints out `"Guess what fruit I am?"` , displaying the first 2 letters of each fruit and the number of letters in the fruit's name. (Just treat a space as another letter in the name for fruits such as passion fruit.)

e.g. `Guess what fruit I am?  pe  4 letters`

6. Get the guess from the user. If it is correct, print `Correct` and carry on to the next fruit. It it is not correct, print `Try again` and repeat the question.

Part 2

Write a program which displays a multiplication table up to 12 x 12 as in Figure 17.1.

Note `"\t"` will print a tab for spacing the output.

Just use an application class, and put all your code in the `main` method.

1. Declare a two-dimensional array just big enough to store the information, and use a nested **for** loop to fill it with the data. (Make sure you are saving `1*1` in position `0,0`.)

2. Use a nested **foreach** loop to display the data you have saved in the array.

1	2	3	4	5	6	7	8	9	10	11	12
2	4	6	8	10	12	14	16	18	20	22	24
3	6	9	12	15	18	21	24	27	30	33	36
4	8	12	16	20	24	28	32	36	40	44	48
5	10	15	20	25	30	35	40	45	50	55	60
6	12	18	24	30	36	42	48	54	60	66	72
7	14	21	28	35	42	49	56	63	70	77	84
8	16	24	32	40	48	56	64	72	80	88	96
9	18	27	36	45	54	63	72	81	90	99	108
10	20	30	40	50	60	70	80	90	100	110	120
11	22	33	44	55	66	77	88	99	110	121	132
12	24	36	48	60	72	84	96	108	120	132	144

Figure 17.1

Part 3

```
public class Average{

    public static void main(String[]args){

        int [] [] table = {{1,2,3},{4,5,6},{7,8}};



    }

}
```

Given the starting code above, finish the main method so that it calculates and displays the average for each "row" of the 2 dimensional array. Each item in the row, and the average, should be displayed on one line. The output should look like this:

```
Output
1 2 3    Average : 2.0
4 5 6    Average : 5.0
7 8    Average : 7.5
```

Lab Completed

☐ preparation exercises complete	☐ comments	☐ Part 1 Fruit
☐ Part 2 12 x 12	☐ Part 3 Average	☐ submitted

Date **Demonstrator's Initials**

Laboratory 17 Options

You may choose whether to get your mark for this lab from discussing your corrections to your mid-semester exam with a demonstrator OR working through the Command Line Interface work described below.

Notes

Readings: L,D&C Appendix I (pages 867 – 869).
This lab has you creating a Java program using "old fashioned" tools based on a "command line" interface in a simple terminal / console. The point of this is to be aware that there are different ways of writing and running programs. Unix commands are issued by typing appropriate characters and pressing the **<return>** (or **<enter>**) key. In this lab we give you some of the basics to get you started.

Preparation

There are NO preparation questions for this lab. We ask you instead to look at your mid-semester exam paper and improve your answers for questions you did not get full marks for.

Lab Work

Terminal / Console vs GUI
In the "good old days" a computer screen ("terminal" or "console") could only display lines of text, and respond to text commands that were typed in - a command line interface (CLI). Operating systems such as Unix and DOS (on the PC) were all based on CLIs. With the invention of the mouse and good graphics, the use of a graphical user interface (GUI) has become much more popular. The Macintosh was the first popular computer to use a GUI in 1984, and PCs eventually followed with early versions of Windows in the 1990's.

Although GUIs have more or less taken over, many current operating systems still provide a CLI as well, often in a special terminal / console window running within the GUI. See for example modern versions of Unix / Linux, including MacOS X (which has a Terminal application in /Applications/Utilities).

Programs have changed too. Older style programs were text based, reading inputs from and writing outputs to the terminal / console.

Modern programs use GUI elements with dialog boxes, windows, text fields, buttons and so on. Java allows us to do either. The programs we write in DrJava can use the DrJava console tools for text input and output, or create and use GUI elements like dialog boxes.

Unix Commands

1. Start the `Terminal` application from the **Applications> Utilities** directory. The window that opens represents a terminal / console (what used to be visible on a computer monitor in the days before graphical user interfaces and mice). You will see

   ```
   oucsXXX:~ yourShortName$
   ```

   The `$` is a "prompt" after which you may type a Unix command. You are now using a Unix command line interface (CLI). The path of your current working directory is shown at this prompt. In this case you are at the top level. The ~ symbol is called the **tilde** and can be found on the top left of your keyboard. It represents the top level of your **home** directory.

2. Type `pwd` (short for "print working directory") and press <return>. The complete path from the top level of the server to your **home** directory should now be shown. The top level of the machine is represented by the first `/` (back-slash) symbol.
3. Type `ls` (short for "list" – Unix people don't like typing long commands!) and press <return>. You will see a listing of all the directories and files in your **home** directory. By default when you open Terminal you are looking at your **home** directory.

4. Type `ls -al` and you will see more information about these files and directories, including size, ownership and date modified.
The characters at the beginning of each line e.g. `drwxr-x--x` tell you

a) whether this is a directory or a file (leading `d` means directory, leading `-` means file).

b) the reading, writing and executing permission for the owner - you (`rwx`), your group (`r-x`) and other users not in your group (`--x`). (Please don't try to change these permissions.)

5. Suppose you want to see what is in your **COMP160> Lab16** directory (assuming you have one).

> If you have a space in your directory name (e.g. Lab 16) type cd Lab\ 16 rather than cd Lab 16. The backslash tells unix that the next character has no special meaning. The space therefore will not be interpreted as a delimiter, separating instructions and arguments from each other, but is part of the string of characters.

Type `cd COMP160/Lab16` <return> (short for "change directory").
Now you have made the Lab16 directory your current working directory. Type `ls` again, and you will see a listing of the directories and files within it. See how the prompt changes to reflect your current directory

```
oucsXXX:~/COMP160/Lab16 yourShortName$
```

Type `pwd` again. **Note**: to repeat a command you have already used, you can use the up and down arrow (as for the Interactions Pane).

6. To move to your **home** directory from anywhere, type `cd ~` (the tilda can be found on the top left of your keyboard).

To move to your **Lab16** directory (assuming one exists) from anywhere, type `cd ~/COMP160/Lab16`. Because this gives a complete filepath, your starting point is irrelevant.

`cd ..` will move you back up one level in the structure (filepath) from wherever you are.

`cd ../Lab12` will move you back up one level then down into the **Lab12** folder, assuming one exists..

7. You can type the first letter or two of a filepath then press the <tab> key - if the letters uniquely identify a file or folder, unix will fill in the rest for you. If there is a choice, it will fill in as much as it can and wait for you to choose the next letter(s). Note that Unix, like Java, is case sensitive: desktop is not the same as Desktop.

8. Change directory back to your **COMP160** directory. Make a new directory called **Lab17** here using the command `mkdir Lab17`. Open your **COMP160** folder from the Mac OS 10 GUI and you will see a folder/directory there called **Lab17**.

9. Let's create a copy of your **Lab2> TwoNumbersApp.java** file to your **Lab17** directory. Type

```
    cp ~/COMP160/Lab2/TwoNumbersApp.java ~/COMP160/Lab17/TwoNumbersApp.java
OR  cp ~/COMP160/Lab2/TwoNumbersApp.java ~/COMP160/Lab17
```

These commands contain two complete filepaths, so it doesn't matter where your current working directory is. The first version gives you the opportunity to change the name of your file. The second version assumes you don't want to change the name of your file.

If your current working directory is **Lab17**, you could type a simpler instruction:

```
    cp ~/COMP160/Lab2/TwoNumbersApp.java TwoNumbersApp.java
OR  cp ~/COMP160/Lab2/TwoNumbersApp.java .      (the dot is short for current working directory)
```

or if your current working directory is **Lab2**, you could type:

```
    cp TwoNumbersApp.java ~/COMP160/Lab17/TwoNumbersApp.java
OR  cp TwoNumbersApp.java ~/COMP160/Lab17
```

Don't forget to use the <tab> key for speed and efficiency.

10. If you wish to move a file rather than copy it, use the `mv` command. You can rename a file by moving it within the current directory like this:

```
mv oldfilename newfilename
```

11. If you wish to delete a file or directory you use `rm` (remove) or `rmdir` (remove directory). Be warned - there is no Trash / Recycle Bin half-way house. When your file is removed, it is gone.

```
rm filename
rmdir directoryname
```

Java

1. With **`Lab17`** as your current directory, type `javac TwoNumbersApp.java`

This will compile the code from `TwoNumbersApp.java` and produce a class file called `TwoNumbersApp.class`. (Take a look with `ls`).

2. Type `java TwoNumbersApp` - this will run your java class called `TwoNumbersApp.class`. You should see the output listed in the Terminal window.

Nano

1. Now let's create and run a java program with the basic command line tools. Type `nano`.

This is an editor that can be used in terminal. The mouse doesn't work here (except for cut & paste). You move the cursor around with the arrow keys. There is a menu at the bottom of the window which gives you commands for opening (ReadFile) and saving (WriteOut) a file. The ^ stands for the <control> key.

2. In `nano`, type the code for `HelloWorld` from Lab 1. Save your code with `Control O`, giving it the usual file name, and exit from `nano` with `Control X`.

3. Compile and Run your program from the Terminal.

These basic commands are only a small introduction into the world of Unix commands. There is an online manual that explains commands in more detail: type `man` then the command to access this, e.g. `man ls`.

Basic Unix commands

ls = list	cd = change directory
mkdir = make directory	cp = copy
mv = move	rm = remove
man = manual	rmdir = remove directory

If you want to break out of any Unix process i.e. if you want your prompt back, type <control> c (together).

To quit from the `man` pages, press `q`.

Javadoc

In `Terminal`, `cd` to a java file you have written recently e.g. Lab14's `Diner.java` and type

```
javadoc yourFileName.java
```

In the Mac OS GUI, navigate to the file's directory and double-click on `index.html` - you have now made your own java documentation.

If the methods do not show descriptions matching the block comment lines in the file: in DrJava, open the file and make sure the block comment start tags are `/**` rather than `/*` for the header and method comments of your chosen file. Repeat the `javadoc` command, and open `index.html` again. This time, you should see your comments recorded in the documentation.

Command Line Arguments (optional)

Those of you who have wondered what `String[] args` is all about may like to read L,D&C Chapter 7, Section 7.4 (pages 361 – 363 [374 – 376]).

Lab Completed

☐ CLI tasks performed OR ☐ mid-semester exam corrections discussed

Date **Demonstrator's Initials**

Laboratory 18 Graphical User Interfaces

Notes

Readings: L,D&C Chapter 6, Section 6.1 (pages 246 – 256 [262 – 271]). Section 6.2, Text Fields (pages 257 – 260 [271 - 275])

Preparation

1. What three kinds of objects are needed for user interaction? Give some examples of each.

i)

ii)

iii)

2. What method **must** you write if you `implement ActionListener`?

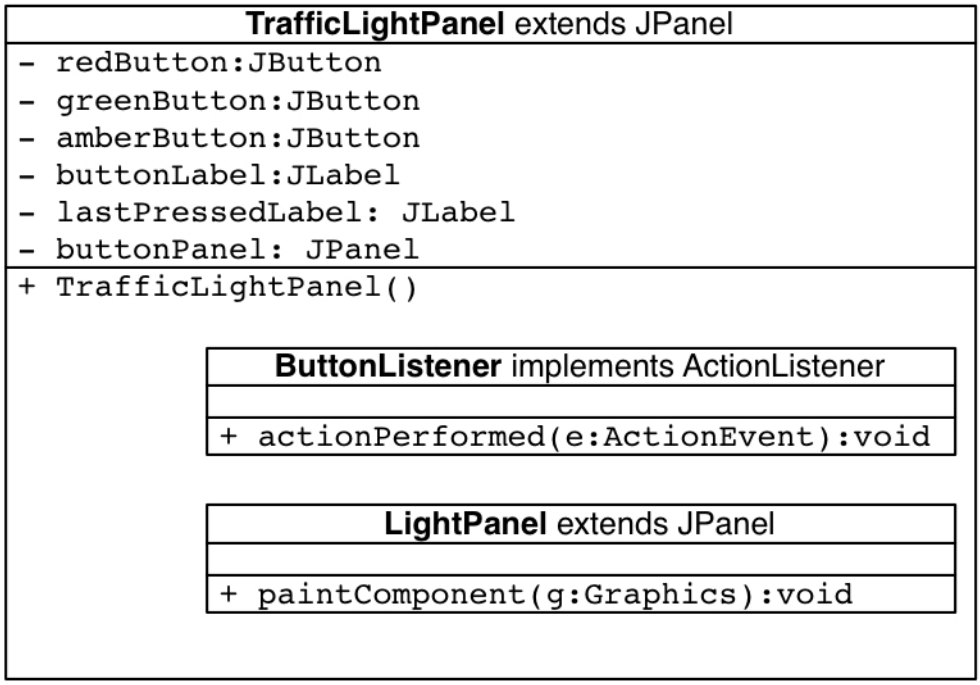
3. In an `actionPerformed` method, with argument `ActionEvent aE`, what does `aE.getSource()` return?

4. Which of the following classes belong to the `java.lang` package, so are imported automatically? (Look up the Java API using the instructions on page 19)

`Color`	`Graphics`
`JFrame`	`Math`
`Object`	`Scanner`
`String`	`System`
`Double`	`Random`
`JPanel`	`Integer`

5. Take a look at code listing 6.6 (LD&C page 273-4), with particular reference to the `getText` and `setText` methods for text fields.

Plans for Lab Work



Lab Work

1. Using listing 6.3 (L,D&C page 253 [269]) as a guide, write an application class which makes a `JFrame` containing an instance of a class called `TrafficLightPanel`.

2. Using listing 6.4 (L,D&C page 255 [270]) and the UML diagram on the previous page as a guide write a `TrafficLightPanel` class which:

- extends `JPanel`
- imports the 3 packages necessary for GUIs, graphics and events
- has 3 `JButton` data fields (which will show "Red" , "Amber" and "Green")
- has 2 `JLabel` data fields (which will show "Button Panel" and "last pressed")
- has 1 `JPanel` data field called `buttonPanel`
- has a constructor which:
  - ♦ instantiates any data fields that still don't exist
  - * sets the size of the `TrafficLightPanel` to 200 by 300
  - * sets the background colour of `TrafficLightPanel` to blue.
  - ♦ sets `buttonPanel's` preferred size to 80 by 290
  - ♦ sets `buttonPanel's` background to white
  - ♦ adds the buttons and labels to `buttonPanel`
  - * adds the buttonPanel to `TrafficLightPanel`

** in the TrafficLightPanel constructor, any instruction to apply to the TrafficLightPanel itself does not need to use dot notation.*

3. Compile and run. You should see something like the frame shown in Figure 18.1.

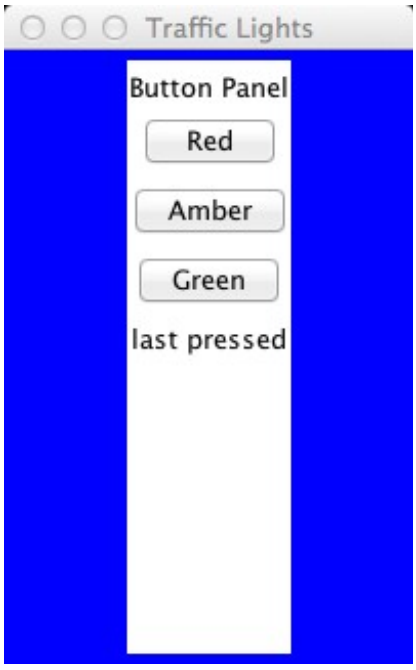


Figure 18.1

Those buttons are no use unless they do something.

Time for action!

4. Still using listing 6.4 as a guide, write an inner private `ButtonListener` class which `implements ActionListener` and has an `actionPerformed` method (it can be empty for now). In the `TrafficLightPanel` constructor make an instance of this class. Register each button to this listener object.

5. In the `actionPerformed` method, if the source of the event is the red button, set the text on the "last pressed" label to "red" and set the background colour of the button panel to red.

6. Repeat this for the other buttons (use `Color.orange` for amber).

7. Compile and run. Your frame will look the same initially, but now should respond to the press of each button.

Graphics on the JPanel

Now we want to draw a graphical representation of a traffic light and have the lights changing colour rather than the background. We could do it in the `TrafficLightPanel`, but it's not the ideal solution. Step 8 will illustrate why.

8. Write a `paintComponent` method in the `TrafficLightPanel` class. Don't forget it needs a `Graphics` object as a parameter (refer back to page 836, `SnowmanPanel`). In this method, set the graphics colour to red and draw a filled circle at `(0,30,40,40)`. Compile and run. Now change the position of the circle to `(40,30,40,40)`.

If your dimensions are the same as those suggested above, the circle will disappear behind the button panel because it is being drawn on the `TrafficLightPanel`. A better solution is to make another panel to put the lights on (Step 9). This panel will sit neatly beside `buttonPanel`.

9. Make another inner private class called `LightPanel` which `extends JPanel`. (This means it will have all the functionality of a `JPanel` as well as anything else we care to add on top, just like `TrafficLightPanel`.)

10. Shift the `paintComponent` method into the `LightPanel` class. Make the first statement of the method a call to `super.paintComponent(page);` (assuming `page` is the name of your `Graphics` parameter)

11. In a constructor for `LightPanel`, set the preferred dimension to 80 by 290 and set the background colour to cyan.

12. In the `TrafficLightPanel` constructor, make an instance of `LightPanel` and `add` it to the traffic light panel.

13. Compile and run.

14. The red circle will now be drawn with reference to the `LightPanel`, so you may need to readjust the `x` position. Now draw the other 2 circles. Your frame should look something like that in Figure 18.2.

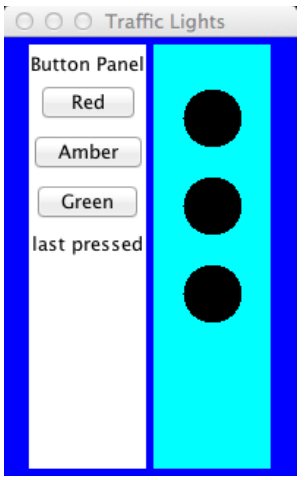


Figure 18.2

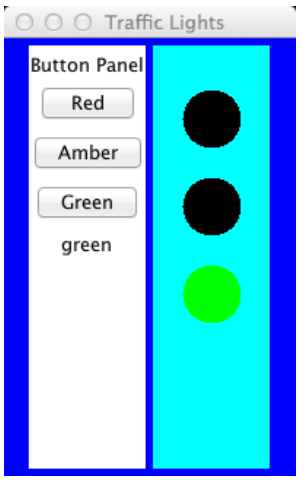


Figure 18.3

15. Make your "lights" change colour with the buttons rather than the background: call `repaint()` at the end of the `actionPerformed` method (no matter what happens in the method, it will always refresh the panel). In the `paintComponent` method, start by drawing all the circles in black. Then if the "last pressed" `JLabel` shows "red", colour the top circle red, etc. (Use the `getText` method of the `JTextField` / `JTextComponent` class - see LD&C page 275)

16. The `lastPressed` JLabel doesn't need to be added to the panel in order to function. Once your program is working and you no longer need to check what is going on, you may remove it from `buttonPanel`.

Lab Completed

☐ preparation exercises complete ☐ comments
☐ sensible variable names ☐ working ☐ submitted

Date **Demonstrator's Initials**

Laboratory 19 Calculator

Notes

Readings: L,D&C Chapter 6, Section 6.2 & 6.3 (pages 256 – 296 [271 – 312]), Chapter 2, Figure 2.2 (page 47 [72]).
The Java `awt` class has an `event` package which contains classes required for user interaction. In a GUI, a user's choice can be defined by `JButtons`, `JRadioButtons` and `JCheckboxes` (among other `JComponents`).

This lab is based on a very good strategy for designing GUIs, where the system of interest (e.g. a calculator) is kept separate from the GUI that forms the "front end" (instead of, for example, trying to mix up both aspects of the program into one class). Otherwise, there are no new concepts introduced in this lab.

Preparation

1. What kind of event is generated by a `JCheckBox`?

2. What is the default layout for the class `JPanel`?

3. Explain why the JRadioButtons in `QuoteOptionsPanel` (page 280-281) are in a `ButtonGroup`.

4. Explain when you should use the data type `long`.

Lab Work

Part 1

Take the `StyleOptionsPanel` and `StyleOptions` classes from the **coursefiles160> Lab19** folder. Add three functioning `JRadioButtons` which will offer three different typefaces.

Use a `GridLayout` to put all the items on the panel in a single column.

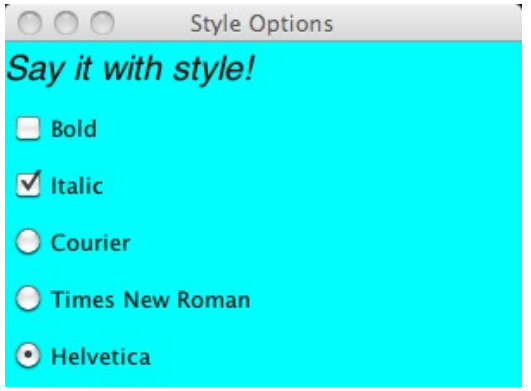


Figure 19.1

Calculator class code listing

You will need to understand what the methods do in order to complete part 2 of your lab work.

```
/**
 * Calculator.java Lab 19, COMP160, 2014
 * A calculator class - for SIMPLE calculations like 5 + 20 =
 * Large inputs will overload int, should convert to long
 */
public class Calculator {

   private int currentInput;                //current input
   private int previousInput;               // previous input
   private int result;                      // result of calculation
   private String lastOperator = "";        // keeps track of the last operator entered

   /** New digit entered as integer value i - moves currentInput 1 decimal place to the left and adds i in "one's column"*/
   public void inDigit(int i) {
     currentInput = (currentInput * 10) + i;
   }

   /** Operator entered + - or * */
   public void inOperator(String op) {
     previousInput = currentInput;   // save the new input as previous to get ready for next input
     currentInput = 0;
     lastOperator = op;              // remember which operator was entered
   }

   /** Equals operation sets result to previousInput + - or * currentInput (depending on lastOperator) */
   public void inEquals() {
     if (lastOperator.equals("+")) {
       result = previousInput + currentInput;
     } else if (lastOperator.equals("-")) {
       result = previousInput - currentInput;
     } else if (lastOperator.equals("*"))  {
       result = previousInput * currentInput;
     }
     lastOperator = "";              // reset last operator to "nothing"
   }

   /** Clear operation */
   public void inClear() {
     currentInput = 0;
     previousInput = 0;
     result = 0;
     lastOperator = "";
   }

   /** returns the current result */
   public String getResult() {
     return Integer.toString(result);     //converts int to String
    }

   /** returns the previous input value */
   public String getPreviousInput() {
     return Integer.toString(previousInput);
   }

   /** returns the current input value */
   public String getCurrentInput() {
     return Integer.toString(currentInput);
   }
}
```

Part 2

In this lab we're going to build a GUI based application, a simple calculator for integer arithmetic. (It will have a few limited functions, not the full range that you are used to in a real calculator). There are several ways to approach such a design task. One very good way is to keep the functionality of the system (in this case a calculator) separate from the GUI "front end".

We will start with a `Calculator` class, and a simple text based "front end" application. Without changing the `Calculator` class at all, we will replace the front end with a GUI based application.

1. Copy the files `Calculator.java` and `CalcTextApp.java` from **coursefiles160** into your working folder. Explore them to see how they work.

> `Calculator` has three data fields: `input` (to hold the current number being input), `result` (to hold the previous input and the result of any operation) and `lastOperator` (to hold a `String` representing the last operator entered).
>
> **Note** that the `Calculator` class only has `+` `-` and `*` operations, an "equals" operation and a "clear" operation. Numbers are entered a digit at a time. If `4` has been entered so far and `2` is entered next then the current value of input is set to `(4 * 10) + 2`, i.e. `42`.
>
> The `CalcTextApp` application front end to `Calculator` is very simple. It makes a `Calculator` object, and codes a single simple calculation, `50 - 26 =`.

2. Compile and run the `CalcTextApp` application. Explore a few more calculations. Calculator can only handle calculations of the form "number operator number =". Other sequences will do strange things!

Time to build a more user-friendly GUI front end. We will do this by modifying an existing example.

3. Copy the file `CalculatorPanel.java` from **coursefiles160** into your working folder. Compile and run.

We need to make a few initial changes to `CalculatorPanel`.

4. Modify the code so that when it runs it looks like Figure 19.2 (the operator keys are `=` `+` `-` and `*`).

When run, it should still behave exactly as before, e.g. the picture shows the panel after the `2` button has been pressed. (How many rows are in the grid now?)

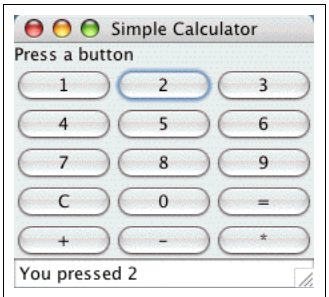


Figure 19.2

5. Now we need to connect the GUI to `Calculator`. `CalculatorPanel` needs a data field `calc` holding a (reference to) a new `Calculator` object.

6. All of the rest of the work happens in the `actionPerformed()` method, making button presses call methods on `calc`. Use "`if`" to work out what button has been pressed, and deal with each one appropriately (the partial example below shows one approach). Think about how and where to display the result. You will need to refer to the listing of `Calculator.java` on the previous page.

```
if ("+".equals(whichButton.getText())) {
   calc.inOperator("+");
} else if
...          // Stuff to add!
} else {     // if the button pressed hasn't been taken care of already, then the button must be a digit
   int i = 0;
   Scanner scan = new Scanner(whichButton.getText());
   i = scan.nextInt();
   calc.inDigit(i);
   display.setText(calc.getCurrentInput());
}
```

Note that all values displayed in the GUI should be values read from the `Calculator` object `calc` (via the accessor methods), not values created within `CalculatorPanel`. The GUI is supposed to reflect data in `calc` only. If we display values created elsewhere then they may be inconsistent with the actual state of `calc`.

7. Complete and test the application so that the calculator works correctly for sums like "234 – 72 =". We now have two different front ends for the same `Calculator` class, a GUI application and a text application!

8. If you enter very long numbers `Calculator` will break (notice that a long number suddenly becomes an odd negative number). This is because `int` only has a limited range. Convert the appropriate `int` to `long` to handle bigger numbers. Note that the conversions `Integer.toString` will need to be changed to `Long.toString`.

`Calculator` can only handle calculations of one fixed simple form. If you finished early, modify `Calculator` to correctly process input sequences like:

```
6 =
8 + 2 + 4 =
3 + 10 =  -  4 =
-6 + 9 =
```

and so on…

Getting the logic of a real calculator correct is very tricky!

Lab Completed

☐ preparation exercises complete ☐ comments altered to suit ☐ Part 1 (radio buttons)
☐ Part 2 clear, equals working ☐ digits, operations working ☐ submitted

Date **Demonstrator's Initials**

Laboratory 20 Reading from Files

Notes

Readings: - L,D&C Chapter 10 (pages 442 – 459 [452 - 469]). You should also read the sections "Tests you should make with try ..catch" (page 117) and "Java Input and Output (I/O)" (page 118) from the Readings at the back of this book.

This lab covers a common task - reading data from a file. There are a few potential hazards which need to be avoided.

Preparation

```
System.out.println("Enter an integer");
Scanner scan = new Scanner(System.in);
int newInt = scan.nextInt();
```

1. What will happen if the user enters a character which can not be stored as an integer e.g. 'w' in response to the code above?

```
public static int readInt() {
     boolean success;
     int input = 0;
     do {
        success = true;
        System.out.println("Please enter an integer");
        try {
           Scanner scan = new Scanner(System.in);
           input = scan.nextInt();
        } catch (java.util.InputMismatchException e) {
           System.out.println("Unexpected input, please try again.");
           success = false;
        }
     } while (!success);
     return input;
  }
```

2. **a.** What will happen if the user accidentally enters a `'w'` in response to the code above?

b. What is the purpose of the `do while` loop?

c. Why do the `boolean` and `int` variables need to be declared outside of the try/catch block?

Planning for Lab Work

Lab Work

The purpose of this lab is to read in integer values from a line in a text file, and use these values to create and draw graphical Rectangle objects. You are given a program as a starting point which follows the structure of the Lab 14 `Diner` / `Splat` files. The program given to you doesn't read its data from a file.

Part 1

1. Copy the files `FileApp.java`, `FilePanel.java`, `Rectangle.java` and `Lab20data.txt` from the **`coursefiles160`** folder into your **`Lab20`** folder.

You have been given a working program which draws 2 rectangles on a `JPanel`. The structure is very similar to the `Splat` and `Diner` programs from lab 14. Take a minute to understand how the program works. Run it.

Your task is to adapt this program so it reads the data for creating `Rectangle` objects from a file. Each line of the file will contain the data for one Rectangle.

2. Open **`Lab20data.txt`** in `DrJava` (File Format : All Files). Look at the contents.

- The first digit of each line is for fill: 0 for `drawRect` or 1 for `fillRect`
- The second digit represents colour: 1 for `Color.red`, 2 for `Color.blue` or 3 for `Color.green`
- The next 4 digits represent the `Rectangle`'s position and size: x, y, width and height.

These 6 items contain the information the `Rectangle` constructor needs. When one line of the file has been read, item by item, a new `Rectangle` object can be instantiated and stored in the `array`. Then the next line of the file is read. And so on until the end of the file. Refer to the example from the readings page 119.

`FilePanel` is the only class you will need to alter, and most of the changes are in the constructor.

3. In order to read from a file, import `java.io.*` at the top of the program. You will be using `Scanner` too, so import that while you are at it.

4. In the `FilePanel` constructor, comment out the lines which create the `Rectangle` objects. Declare a `String` variable called `fileName` and set it to the name of the data file you wish to open.

Inside a `try .. catch` block, declare a `Scanner` object which will get its input from the file:

```
try{
   Scanner fileScan = new Scanner(new File( fileName));

      // any code for reading from the file goes here

} catch (FileNotFoundException e){
   System.out.println("File not found. Check file name and location.");
   System.exit(1);//exit from program if no file to read
}//catch
```

5. Inside the `try .. catch` block you can use `Scanner`'s methods (`next()`, `nextInt()`, `nextLine()` etc) to read data from the file. Try the line:

```
System.out.println(fileScan.nextInt());
```

If you run the application class, you should see the first number of the file in the Interactions pane. So far so good. You could repeat this line 15 times and see every piece of data in the file, but the program does not usually know how much data is in the file. Reading past the end of the file will cause a run-time error. It is better to check whether you have reached the end of the file and then stop.

One way to do this is to use a `while` loop to check that there is still some data left in the file before you attempt to read from it.

6. Write a `while` block around the line which reads and displays the next integer in the file:

```
while (fileScan.hasNext()){
       System.out.println(fileScan.nextInt());
}//while
```

Now that you can read the data safely, let's think about reading it in meaningful chunks, and using it to create `Rectangle` objects to store in the `array`.

7. Get the `while` loop to read 6 `int` values at each repetition, representing all the data for one `Rectangle`. Each piece of data should be stored in a local variable e.g `int fill = fileScan.nextInt();`

8. Now that the `while` loop is reading 6 values at a time, checking that it has one more integer is not ideal. Instead, check that it has another line:

```
while (fileScan.hasNextLine()){
```

Part 2 builds in further checking, to make sure each line is of the correct pattern.

Two of the data items need to be interpreted before the `Rectangle` object can be created.

9. The first integer from each line, representing fill/draw, needs to be turned into a `boolean` true / false. Use an `if .. else` statement to set a `boolean` value to `true` for a 1 (fill) or `false` for a `0` (draw).

10. The second (colour) variable needs to be turned into a `Color`. Set a `Color` variable to `Color.red` for a `1`, `Color.blue` for a 2 or `Color.green` for a `3`.

11. You now have all the data required to create a `Rectangle` object. Use a statement pair similar to those that have been commented out to create a `Rectangle` and store it in the `array`, but use your variables instead of literal values. Don't forget to increment the count.

12. Compile and run your code. You should see a rectangle for each line of the file.

13. To illustrate the use of a static variable, add two **int** data fields to the `Rectangle` class. Make one of them `static`:

```
private static int totalCount;
private int thisCount;
```

In the `Rectangle` constructor, increment `totalCount` by 1 and assign to `thisCount` the value of `totalCount`.

In the `draw` method, use a `drawString` statement to display the `thisCount` of `totalCount` information at position `x,y` of each `Rectangle`.

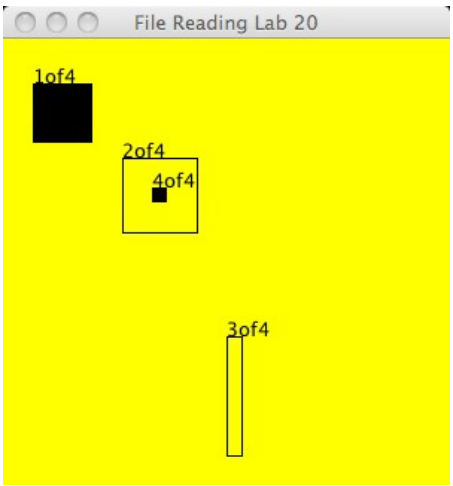


Figure 20.1

The static data field `totalCount` is shared by all instances of the Rectangle class.

Part 2

The program you wrote in Part1 has assumed the data will be 'clean' and will not make more than a particular number (10) of Rectangles. This is not always a sensible assumption for a programmer to make. There is a simple check that could be added to the code which would throw away any line of data which does not conform to the required pattern of 6 integers separated by a delimiter (space/tab).

1. First thing in the `while` loop, declare another `String` variable to hold the next input line temporarily - call it `inputLine`:

```
String inputLine = fileScan.nextLine();
```

2. To check whether this line contains the data that you are requiring, you can match the line with a delimiter pattern. (This avoids the need for an `InputMismatchException try .. catch` block.)

```
if (inputLine.matches("\\d+ \\d+ \\d+ \\d+ \\d+ \\d+")){
    //go ahead, create a new Scan object, use it to read the 6 integers into the variables – see Step 3 below
}  //bad input will just be ignored - no exception handling required
```

`\\w` means a character is expected	`\\w+` means any number of characters (a word) is expected
`\\d` means one digit is expected	`\\d+` means any number of digits is expected

the **single space** between each delimiter is essential - each refers to one space " " between each token (item)

These are called regular expressions, and their documentation can be found in the Java API under `java.util.regex.Pattern` and also in L,D&C Appendix H page 1004 [865-866]

3. Declare and instantiate another new `Scanner` object. This time, rather than getting its input from `System.in` (the keyboard) or `new File` (the file), it wants to get its input from `inputLine`.

Use **this** `Scanner` object to read the individual values in `inputLine`. As each value is read, store it in the appropriate variable as before. Now try your code on a data file containing corrupt data e.g. `"BadData.txt"`

4. Limit the number of Rectangles which you may attempt to store in the array – while there is another line **and** there is room for one more Rectangle in the array . . . Try this out on the file `LongBadData.txt`

Lab Completed

☐ preparation exercises complete	☐ comments	☐ Part 1
☐ static	☐ Part 2	☐ submitted

Date **Demonstrator's Initials**

Laboratory 21 Shapes 1: Building the Structure

Notes

Readings: L,D&C Chapter 8, Sections 8.1, 8.2, 8.3 (pages 380 – 398 [392 – 409]).
This lab is the start of a sequence of four labs that build incrementally to a fairly complex final program. The sequence of labs will illustrate important OO concepts like hierarchies and abstract classes in the context of building an interactive graphical animation. We hope that you will find it both useful and fun!

In this lab you will make an array of (references to) graphical objects. There is little that is new in this lab - it is drawing together all your previous knowledge of arrays, objects, events and graphics. You will be using the keywords `extends` and `super`, and may be able to develop a better understanding now of how these words help you navigate around the class hierarchy that sometimes is provided for you by the APIs, and at other times you create for yourself.

Preparation

1. For each of the classes listed, write the name of the class beside the `import` statement which applies to it:

`JPanel` `JFrame` `Graphics` `FlowLayout` `Color` `Random` `ActionListener`

`import javax.swing.*` _____

`import java.awt.*` _____

`import java.awt.event.*` _____

`import java.util.*` _____

2. Look up the class `Object` in the Java API. Can you explain why every class has a default `toString` method?

3. There is one error in class B, and one error in class C which will stop the code below from compiling. What are the errors?

```
public class Lab21App{
  public static void main(String [] args){
    B b1 = new C(3, "One", "Two");
  }
}
```

```
public class B{
  private String s;

  public B(String s){
    this.s = s;
  }
  public String getS(){
    return s;
  }
}
```

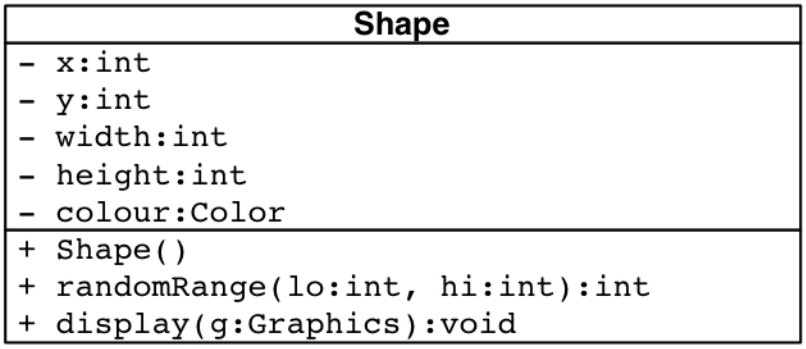
```
public class C extends B{
  private int x;
  private String a;

  public C (int x, String a, String s){
    this.x = x;
    this.a = a;
    super(s);
    s = "Five";
    System.out.println("C: x is " + x);
    System.out.println("C: a is " + a);
    System.out.println("B: s is " + super.s);
    System.out.println("B: s is "  + getS());
    System.out.println("C: s is " + s);
  }
}
```

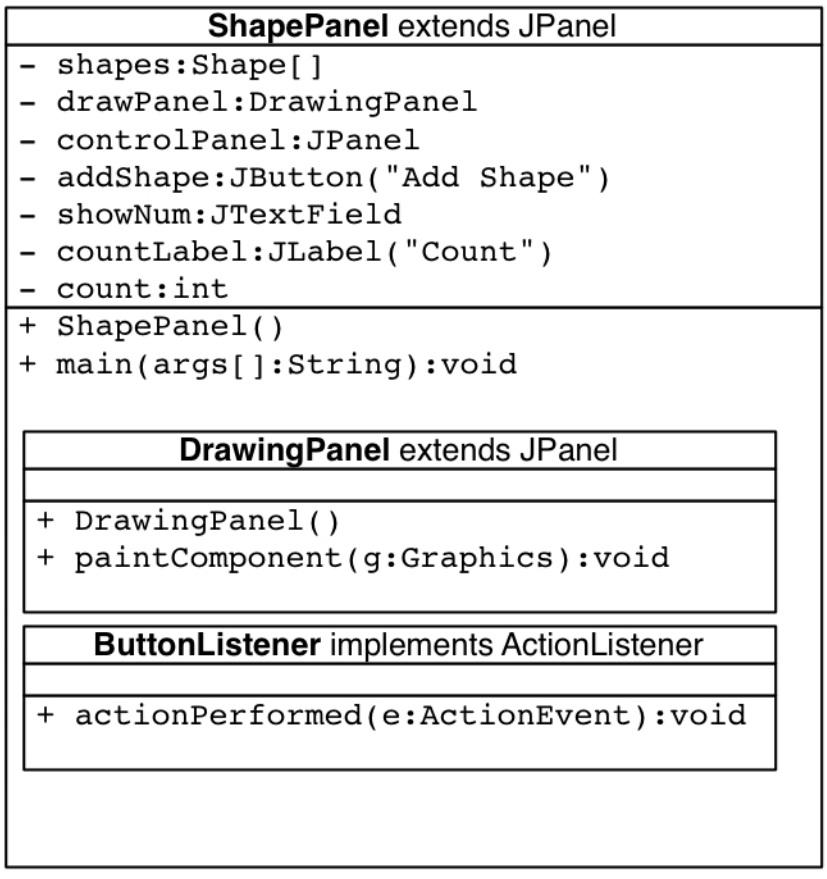
4. What will be printed out by the code if the two errors are fixed?

```
C: x is _____

C: a is _____

B: s is _____

B: s is _____

C: s is _____
```

5. Remind yourself how the `Random` class works (Lab 8, page 38)

Planning for Lab Work





Lab Work

In this lab, we are going to set the scene for the next 3 labs as well. The work in each lab will have well-defined goals, so the program can be 'finished' during the lab as usual, but each exercise will build on the previous ones so that the program will grow and develop over time.

The structure we wish to establish is a `JPanel` which contains two more `JPanels`. The "control panel" is to the left, and will contain buttons, labels etc. For this lab, it will need one `JButton` (labelled `Add Shape`), one `JLabel` and one `JTextField`. A square panel on the right will display graphical objects. We will call this the "drawing panel". The drawing panel displays each element in an array of `Shape` objects.

The `Shape` class represents an object with an `x` and `y` location, width, height and colour (very similar to your `Diner` circles from Lab 14). But now that we know about events and arrays, we are going to add a `Shape` to an array on the press of a button.

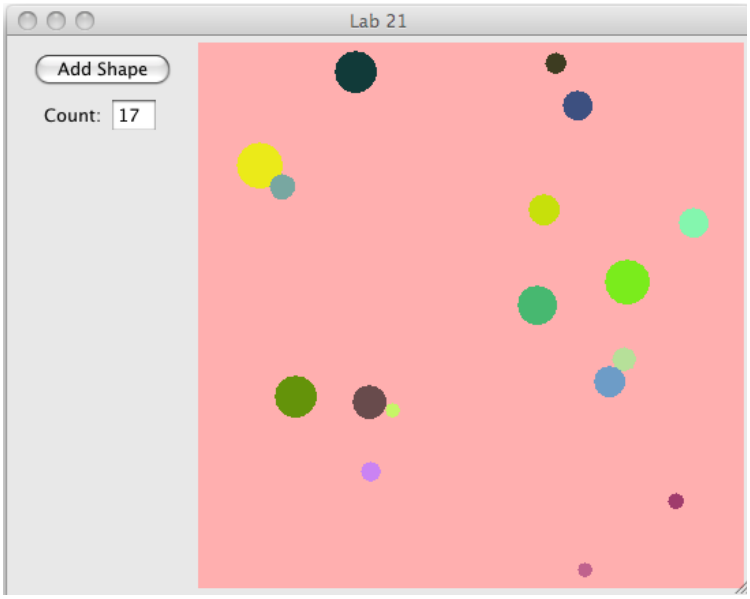


Figure 21.1 controlPanel and drawPanel in the ShapePanel

Figure 21.1 shows the structure we wish to set up by the end of today's lab. The `contentPane` of the Lab 21 `JFrame` contains a panel, which contains two sub-panels.

To the left is `controlPanel`, which contains the button, label and text field. To the right is `drawPanel`, on which the `Shape` objects are drawn.

Each circle is an instance of the `Shape` class. These instances are collected together in an `array`, so they can be managed as a group rather than individually.

The text field displays the number of `Shape` objects which have been created.

The Shape Class

The `Shape` class (for now) is going to represent a circle at a random x, y location, of a random colour, and a random size between 10 and 30. The circles will be drawn on the `DrawingPanel` which is 400 by 400 pixels.

1. Write a class called `Shape` with four `int` data fields to store `x`, `y`, `width` and `height`, and a `Color` data field to store the colour of the shape.
2. These data fields are going to be set to random values with particular limits, so a random range method would be really useful to start with. Write a method which takes two `int` parameters representing the lowest and the highest limit, and returns a random integer within that range. (You used one in Lab 8.)
3. Write a `Shape` constructor which sets the data field `width` to a random value between 10 and 30. Set `height` to equal `width` (we're talking circles here for now).
4. In the `Shape` constructor, set data field `x` to a random value between 0 and (`400 - width`) and `y` to a random value between 0 and (`400 - height`) (to keep all of the shape on the panel).
5. In the `Shape` constructor, set the `Color` data field to random RGB values. A new `Color` can be created using the syntax `new Color(red, green, blue)` where red, green and blue are each integer values between 0 and 255.
6. Write a `display` method for `Shape` which is passed a `Graphics` object. Use the `Graphics` object to set the `Graphics` colour to the value stored in the `Color` data field and draw a filled oval using the values in the other data fields.

The ShapePanel class

The `ShapePanel` class contains the control panel (with its button, label and text field) and the drawing panel. The drawing panel will be written in its own (inner) `DrawingPanel` class. Inner classes should always be declared as `private`.

1. Write a class `ShapePanel` which `extends JPanel`. The main data structure of the program is an array of (references to) as many as 20 Shape objects. Declare this array.
2. Write a `main` method which makes a new instance of `JFrame`, adds a new `ShapePanel` object to it, and calls all the necessary `JFrame` methods - s`etDefaultCloseOperation`, `pack`, `setVisible`.
3. Write a `ShapePanel` constructor which creates a new `JPanel` called `controlPanel`. Create three data fields: a `JButton` (labelled `"Add Shape"`), a `JTextField` with room for two digits, and a `JLabel` showing `"Count:"`. The `JButton` will need an `actionListener` added, so write an inner (private) `ButtonListener` class which `implements ActionListener`. It will need an a`ctionPerformed` method, which can be an empty method for now. In the `ShapePanel` constructor, create a new instance of `ButtonListener` and add it to the button. (You will need to `import swing`, `awt` and `event`).
4. Set the preferred size of the `controlPanel` to dimensions 100 by 400 pixels.
5. Add the button, label and text field to `controlPanel` and add `controlPanel` to `ShapePanel`. If you compile your class and run it, it should display a window like the left-hand `"column"` of Figure 21.1.

The DrawingPanel Class

1. Write another inner (private) class for `ShapePanel` called `DrawingPanel`, which `extends JPanel`.
2. Write a constructor for `DrawingPanel` which sets the preferred size to dimension 400 by 400 pixels, and the background colour to `Color.pink`.
3. Write your own `paintComponent` method for `DrawingPanel` (with parameter `Graphics g`) which calls the `JPanel` `paintComponent` method (using `super`) sending it the `Graphics` object. (Later, this method will also call a `display` method on each `Shape`.)
4. Now you have a `DrawingPanel` class, but you won't be able to see it yet. In the `ShapePanel` class, create an instance of the `DrawingPanel` class as a data field called `drawPanel`. In the `ShapePanel` constructor, add `drawPanel` to `ShapePanel`.

To finish

1. You will need some way of counting the number of references to `Shape` objects which have been stored in your array, both for knowing where to store the next one, and for displaying the number of elements of the array. Make an integer data field to keep a count of the number of valid `Shape` objects in the array.
2. In the `actionPerformed` method of `ButtonListener`, if the source of the `ActionEvent` is the "Add Shape" button, check that the counter is smaller than the length of the array. If it is, add a new `Shape` to the array and increment the counter. No matter what the source of the event, set the text of the `JTextField` to display the `count` and call the `repaint` method on the `drawPanel` in order to update the panel displayed on the screen.

   Note: the `repaint` method automatically calls `paintComponent`.

3. Make the `paintComponent` method of `DrawingPanel` loop through each valid element of the `array`, calling the `display` method on each `Shape` (sending it the `Graphics` object).

   Note: All drawing needs to be managed by the `paintComponent` method, but it needs to let other objects see its `Graphics` object.

4. Now run your code and make some spots. If your code doesn't appear to be working, try selecting the Interactions Pane before running your code - the error messages there may be enlightening.

Lab Completed

☐ preparation exercises complete ☐ stops at 20 with no errors
☐ comments ☐ working ☐ submitted

Date **Demonstrator's Initials**

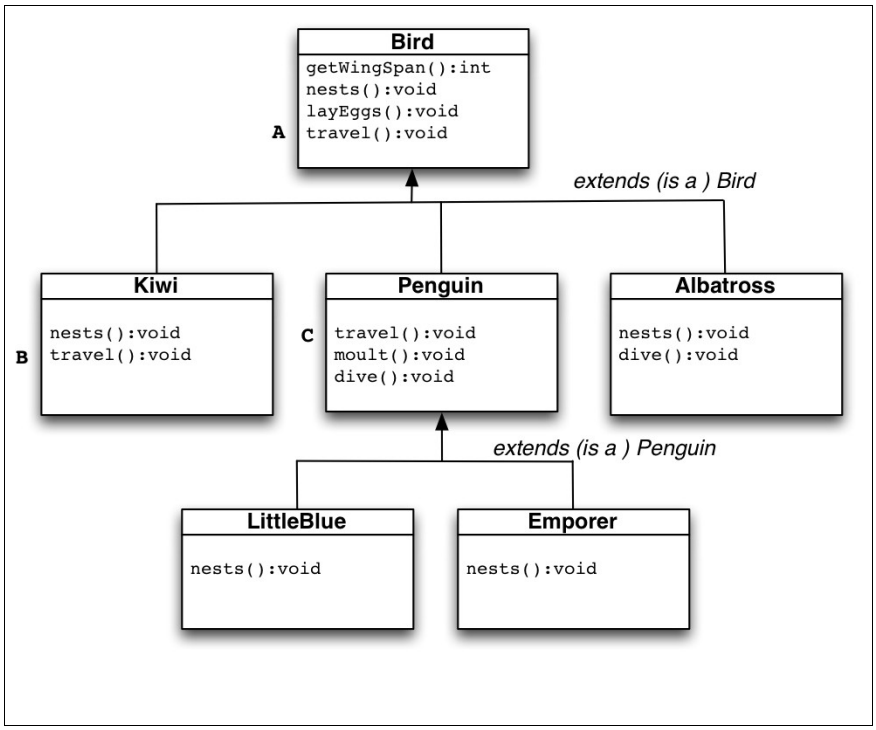
Laboratory 22 Shapes 2: Animation

Notes

Readings: L,D&C Chapter 6, Section 6.2 (pages 277 – 282 [293 – 297]). L,D&C Chapter 8, Sections 8.4 & 8.5 (pages 399 – 403 [410 – 414]).
The concept of inheritance is fundamental to object oriented programming. Inheritance should be used when a behaviour is shared among classes of the same general type. The preparation exercises cover some of the concepts involved. The lab work continues the project started in Lab 21, using the `Timer` class to create action on your `DrawingPanel`.

Preparation

1. Examine the class hierarchy in the diagram, then answer the questions below.



a) There are three different `travel` methods, **A B** and **C**.

Which one does `Kiwi` use?

Which one does `LittleBlue` use?

Which one does `Albatross` use?

b) List all the methods that `Kiwi` can use?

c) Can `LittleBlue` use the `getWingSpan()` method?

d) Can `Albatross` `moult()`?

e) How can `Kiwi` access `Bird`'s `nests()` method?

f) Declare an array which may hold up to 5 elements, of type `Kiwi` **OR** `Albatross` **OR** `Penguin`.

Write another statement which makes the first element of the `array` an instance of `LittleBlue`.

2. Why is inheritance useful?

3. What is over-riding? Give an example from the `Bird` hierarchy.

Planning for Lab Work

Lab Work

In this lab, you are going to get your spots moving. The `Shape` class will have a `move` method which updates the `x` and `y` (location) data fields by some formula each time it is called.

In order to get the animation working, the `ShapePanel` class will declare an instance of `Timer` which automatically generates an `ActionEvent` at regular intervals (possibly fast). This means that the `actionPerformed` method will get called at regular intervals. If we put code that updates and redraws the locations of `Shapes` inside the `actionPerformed` method, each time the `Shapes` are redrawn they will be in a different place, creating the illusion of movement (in the same way that motion pictures display the individual frames of a film).

The `Timer` class has `start` and a `stop` method which can be used to control the action. You will add a button to start them moving, and a button to stop them moving.

Putting your files in a package

1. Make a copy of your **`Lab21`** directory and rename it **`Lab22`**. Open the files in `DrJava`.
2. At the top of each class (after the comments, before the import statements) write the line
   ```
   package shapes;
   ```
3. Save the files and close them. Make a new directory called **`shapes`** in your **`Lab22`** directory. Move your files into this new directory. Open them with `DrJava`.

   Your files are now in a package. Package visibility is the default visibility for Java – the visibility you get if you don't specify public, private or protected. Data fields with package visibility are available to all classes in the same package. Files in a package must be stored in a directory of the same name as the package (**`shapes`** in our case).

The Shape class

1. Declare two `int` data fields called `moveX` and `moveY` in the `Shape` class. Set their initial values to 1.
2. Write a `void` method called `move`. In this method, add `moveX` to the value of `x`, and add `moveY` to the value of `y`. This changes the location at which the shape would be drawn by 1 pixel to the right and 1 pixel down every time it is called.

The ShapePanel class

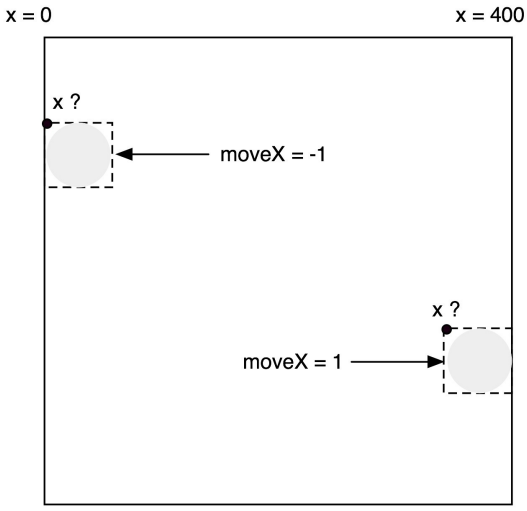
1. Declare two new `JButtons` as data fields in the `ShapePanel` class, one to `Start` and one to `Stop`. Make these buttons capable of event handling. Add them to the `controlPanel`.
2. Declare a data field of type `Timer` called `timer`, and declare a `final int` data field DELAY initialised to 10 (milliseconds). There are three `Timer` classes listed in the Java API, but the `Timer` we are using is a class in the `javax.swing` package.
   ```
   Timer timer;
   private final int DELAY = 10;
   ```
3. In the `ShapePanel` constructor, set the variable `timer` to be a new instance of `Timer`, and send it the DELAY and the `ButtonListener` as parameters.
   ```
   timer = new Timer(DELAY, listener);
   ```
   // or whatever you have called your ActionListener
4. In the `actionPerformed` method, add another `if` block for when the source of the event is `timer`. In this case, each valid element in the array should have its `move` method called in turn.

   This will repeat the `x`,`y` updating of each `Shape` at a rate controlled by the DELAY parameter. Make sure the `repaint` is being called at the end of the `actionPerformed` method, not in an `if` block. The `repaint` will redraw the panel (every 10 milliseconds) with the `Shapes` in their new position.
5. In the `actionPerformed` method, call the method `timer.stop()` if the `stop` button is the source, and `timer.start()` if the `start` button is the source.
6. Compile and run your code.

   If all is well, your shapes will quietly slip off the right or bottom of the panel, never to be seen again. You need to keep them in view by bouncing them off the sides of the panel.

The `move` method will need to check whether the current `x` or `y` location has put the shape on the edge of the drawing panel. If it is on the edge, either the horizontal (moveX) or vertical (moveY) direction should be reversed.

How can you determine whether the shape is on the edge? Take a look at the diagram below, imagining that the shape on the left edge is travelling left (`moveX = -1`), and the shape on the right edge is travelling right (`moveX = 1`).



7. Write an `if` statement in the `move` method which checks to see whether `x` is on or over the edge of the panel. If so, change `moveX` to `-moveX`, toggling the direction of the horizontal movement. Then (whether or not you have had to change direction) carry on and make the change to `x`.

8. Write another statement to look after the vertical movement. Run your code again. Hopefully now your shapes are contained within the panel.

When you're ready, go and change some things.

- Try changing the delay to 5, then to 20. See what happens. (After each alteration, you should return the code to its original state.)
- In the `move` method, comment out the line which increments `y`. Can you predict what will happen?
- In the `Shape` class, set the value of data field `moveX` to 5.

Some challenges.

- Make shapes which are wider than 15 pixels travel straight up and down, while all other shapes travel sideways.
- Get the shapes which are initially drawn in the lower half of the screen to start their travel in an upwards rather than downwards direction. (There is a very simple solution to this one.)

Your mission:

- Get your shapes to change over time according to some criteria of your own design. Leave your code in this state for marking.

Lab Completed

☐ preparation exercises complete ☐ comments
☐ working, in a package ☐ shape change ☐ submitted

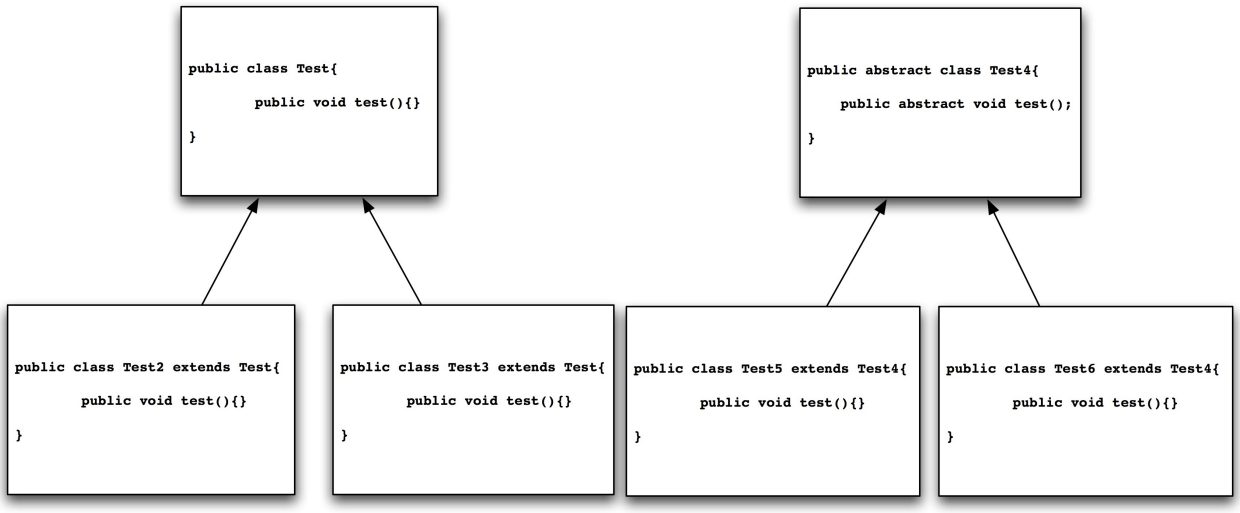
Date **Demonstrator's Initials**

Laboratory 23 Shapes 3: Abstract

Notes

Readings: L,D&C Chapter 8, Section 8.3 (pages 394 – 398 [404 – 409]) and Chapter 9, Section 9.2 (pages 413 – 425 [422 - 435]).
The text book gives an example of an abstract class in Section 9.2, the `StaffMember` class. See if you can follow how the class hierarchy works.

Preparation



1. Which two of the classes above could legally call a method named `test` even if you removed the method definition from that class?

2. Which one of the classes above can not be instantiated?

3. Which two classes' `test` method could legally contain the statement `super.test()` ?

4. Which two classes **must** each legally contain a method called `test`?

5. In an `actionPerformed` method, the two statements below would each store a reference to the component that generated the event in a variable named `source`

```
Object source = event.getSource();              //line 1
JButton source = (JButton) event.getSource();   //line 2
```

True or false:

line 1's `source` could be a `JButton`

line 2's `source` could be a `JButton`

line 1's `source` could be a `JCheckBox`

line 2's `source` could be a `JCheckBox`

line 1 will fail if `source` is not a `JButton`

line 2 will fail if `source` is not a `JButton`

could you call `JButton's getText` method on line 1's `source`?

could you call `JButton's getText` method on line 2's `source`?

6. What is the name for the operation which is being performed by the `(JButton)` instruction in line 2?

7. Could both of the statements in **5**. above appear in the same `actionPerformed` method?

Planning for Lab Work

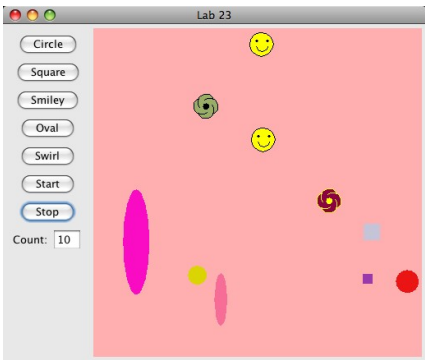


Figure 23.1 Shape window after Lab 23

Lab Work

In this lab, we are going to explore some of the possibilities arising when our `Shape` class is made `abstract`, and our `Shape` can be something more than just a circle. With `Shape` as an abstract class we can have any number of classes which extend `Shape`, each of which can represent a different sort of `Shape` (e.g. a square, an oval, or a more complicated image).

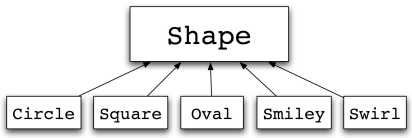


Fig 23.2 Planned Shape hierarchy

Shape to abstract

We can very quickly adapt the finished Lab 22 code to the abstract format.

To divide the `Shape` class into its abstract and non-abstract components, we need to think about the things **all** `Shape` objects will want. Since all `Shape` objects will want to be drawn somewhere, and move, they will all want the data fields `x`, `y`, `width`, `height`, `shade`, `moveX` and `moveY`.

The constructor method must remain with the abstract class. The `move` method is also common to all `Shape` objects, and can stay with the abstract class.

The method which needs to be different for each of our yet-to-be-written classes is the `display` method. At present, it draws a circle.

1. Make a copy of your **`Lab22`** folder and rename it **`Lab23`**. Open the files in `DrJava`.
2. Make `Shape` an `abstract` class. It can not now be instantiated.
3. Make a new class called `Circle` which extends `Shape` and put it in the `shapes` package. This class will also need to `import java.awt.*` in order to easily access `Graphics` methods and `Color`. (Importing is not an inherited characteristic - it just saves you from typing the full name of the class each time you use it e.g. `java.util.Scanner scan = new java.util.Scanner(System.in)`)
4. Copy the existing `display` method from `Shape` into the new `Circle` class.
5. Turn the `display` method of `Shape` into an `abstract` method (with no method body). This forces all classes that extend `Shape` to supply a `display` method.
6. Compile. If your data fields are private, you will get an error message. `Circle` needs to access the inherited data fields of `Shape (x, y, height etc.)`, so they will need to have `protected` rather than `private` visibility.
7. Compile. Another error message: the abstract class Shape cannot be instantiated. In `ShapePanel`, instead of adding a new `Shape` when the "Add Shape" button is pressed, add a new `Circle`. There is no need to change the array declaration - it still holds references to `Shape` objects. Each `Circle` is also a `Shape` because of the type hierarchy. Neat!

   Compile, fix any other errors, and run. Now you have code which will exhibit exactly the same behaviour as before, but the scene has been set for growth.

8. Write another class called `Square` which also extends `Shape`. Write a `display` method in `Square` which draws a square rather than a circle. This is all very well, but you haven't got the button structure set up yet to be able to draw a `Square`.

Array of JButtons

It's time to set up a few extra buttons on the `controlPanel`, so that you can draw your `Square` as well as some other shapes when you have written some more classes. There will be at least 7 buttons (see Figure 23.1 on the previous page), so let's consolidate the code by dealing with them in an array. Repeated tasks can be performed using a loop.

1. In `ShapePanel`, declare an array of `JButtons` that will hold 7 elements.
2. Fill the array with new `JButtons` labelled Circle, Square, Oval, Smiley, Swirl, Start and Stop.
3. In the `ShapePanel` constructor, write a `foreach` loop which accesses each element in the array and

   - adds a listener

   - adds the button to the control panel.

4. Remove all references to the "Add Shape", "Start" and "Stop" `JButtons` from the constructor.

Where the `actionPerformed` method before was using (hopefully) meaningful names for the buttons, it would now need to be referring to something like buttons[0], buttons[1] etc. Matching the right element of the button array to the right event will be prone to error. A neater solution would be to access the (meaningful) text on the button itself. The timer (which is not a button) needs to be dealt with separately.

Make the `if . . timer` block the first block in the actionPerformed method. Follow it with an `else` block which includes all the remaining code in the method EXCEPT the call to `repaint()`. This `else` block will deal just with button presses, so in it you can safely cast the object source into a `JButton` variable.

```
JButton button = (JButton) e.getSource();
```

Now each `if` statement for which button was pressed can access the text on the button itself e.g.

```
if(button.getText().equals("Circle")){
```

It is now much easier to match the event to the right process than it would have been with the code `if(e.getSource() == buttons[0]){`

5. In the `actionPerformed` method, alter the Circle, Start and Stop buttons to the pattern described above. Add the required number of `if..else` statements to cope with all the new buttons. Two of the classes to be called have been written already (`Circle` and `Square`), so test your code now using these two buttons. We need to make more classes to extend Shape . . .

More Shapes

1. Class `Oval` will look very similar to `Circle`, except it needs a constructor which sets `height` to `4 * width`. This will muck up the `y` location, which may draw the oval initially below the bottom edge of the panel, so the `Oval` constructor should also calculate a new random value for `y` using the new value of `height`.
Note: `Oval` can use the "random" method of its parent (`Shape`) class because the method has `public` visibility.

2. To make `Smiley` easier to draw, it is always going to be a standard size - 30 by 30 pixels. A `Smiley` constructor can set `height` and `width` to 30, and recalculate both `x` and `y`.

3. In `Smiley`'s `display` method, draw a filled yellow circle, an unfilled black circle, 2 black eyes and an arc mouth. To save you time try these parameters:

draw the left eye at `x + 7, y + 8,` 4 pixels across.

draw the right eye at `x + 20, y + 8`

try the arc at `x + 8, y + 10, 15, 13, 190, 160`

4. A file called `Swirl.java` is in the **coursefiles160** folder.

5. Link all your button presses to calls to the correct class constructors, and off you go. Your various shapes (Circle, Square, Smiley, Oval, Swirl) can look different, but are behaving in the same way because they are all extensions of the `Shape` class.

If you want to be really clever, see if you can make `Smiley` smile on the way up and frown on the way down.

If you have still got some size changing going on from Lab 22, this may not be appropriate for Swirl and Smiley. There are three ways you could deal with this : 1) remove it 2) make `move` an `abstract` method in `Shape` and have different versions of it in each child class, or 3) in the `Shape` class, use the expression `if (!(this instanceof Swirl) && !(this instanceof Smiley)){` *//leave out smiley & swirl from the size changing*

Lab Completed

☐ preparation exercises complete ☐ foreach JButton loop
☐ comments ☐ getText to identify event source ☐ submitted

Date **Demonstrator's Initials**

Laboratory 24 Shapes 4: ArrayLists

Notes:

Readings: The `ArrayList` reading (page 123 in the Readings section at the back of this lab book).

This is the last of the Shapes labs. You are already familiar with an array, which may hold references to objects. An `ArrayList` is an ordered list of objects of flexible length. This lab illustrates graphically the dynamic nature of `ArrayLists`.

An `ArrayList` must hold references to objects. Unlike an `array`, an `ArrayList` has no fixed size. It can grow and shrink as required. Data (objects) may be inserted and removed at any position, but the process is most efficient if new objects are added to the end of the `ArrayList`.

In this lab you will convert your `array` of Shapes to an `ArrayList` of Shapes.

Preparation

1. **a)** Write a statement to declare an `ArrayList` of objects called `list`.

b) Write a statement to declare an `ArrayList` of objects called `list` which may only hold `String` objects.

c) Write a statement to add "jam" to `list`.

d) Write a statement to add "juice" to `list`.

e) Write a statement to insert "bread" as the first element (index 0) of `list`.

f) Write a statement to store the number of elements in `list` in a variable called `listSize`.

g) Write a statement to remove the element of `list` at index 1.

h) Write, in order, all of the values held in `list` after the statements above have been executed.

i) Write a statement which assigns `true` to a `boolean` variable `oj` if list contains "juice".

j) Write a statement to find out where "juice" is stored (i.e. its index), and store this value in an `int` variable `index`

2. The 2 questions below refer to an `ArrayList` called `zoo` which contains objects of type `Animal`.

```
<Animal> zoo
```

a. Write a `foreach loop` which prints out the value returned by the `toString` method of **each** element of `zoo`.

b. Write a statement which prints out a the value returned by the `toString` method of the **third** element of `zoo`.

3. An ArrayList can only hold (references to) objects and yet both of the following statements will add an object representing the integer value 3 to an ArrayList called `myList`. The first statement can be seen to be using the `Integer` wrapper class.

```
myList.add(new Integer(3));
myList.add(3);
```

What is the term for the process which is happening in the second statement?

4. The `Integer` wrapper class has many useful methods. You have seen `Integer.toString(int num)` used in order to display an `int` on a `JLabel`. There is a reverse method, `Integer.parseInt(String str)`, which converts a `String` into an `int`. Write a statement which will covert the text showing in a `JTextField` called `jt`, and store it in an `int` called `jtValue`.

5. If the text in the `JTextField jt` in Ex. 4 above could not be converted into an `int` (say it was showing an 'a'), what would happen?

Lab Work

In this lab you will convert your array of Shapes to an `ArrayList` of Shapes, then display beside each `Shape` the index number representing that `Shape` object's position in the `ArrayList`. A `Remove` button will remove a `Shape` at a chosen position. You will be able to see the `ArrayList` growing and shrinking as you add and remove `Shape` objects.

1. Make a copy of your Lab23 folder and rename it Lab24. Open the files in DrJava.

2. Change the declaration of the array of Shapes into an ArrayList declaration (you will need to import `java.util.ArrayList`) using the following syntax (but change *shapes* to whatever your array was called):

```
ArrayList<Shape> shapes = new ArrayList<Shape>();
```

3. Change the syntax for adding new `Shape` objects

from `shapes[count] = new Circle();`
to `shapes.add(new Circle());` etc.

There is no longer any need to check the number of elements stored. An `ArrayList` can grow as required.

The call to the display method now needs to use `ArrayList` syntax.

4. In `Shape`, write a `showIndex` method which takes a `Graphics` object and an integer as input, and draws the integer, in black, at position `x`, `y`. (Remember the `drawString` method in Lab 14 - Diner?)

5. In `ShapePanel`, call this method in the for loop which draws each shape, directly after the call to the `display` method. Send it a `Graphics` object and the index number of that `Shape`.

6. Run your code. You should see a number beside each Shape.

7. Make a "Remove" button.

8. Change the `JLabel` so it shows "`Remove which?`" and make it display the index of the last element of the `ArrayList`.

(Remember, if `size()` is 10, the index runs from 0 to 9).

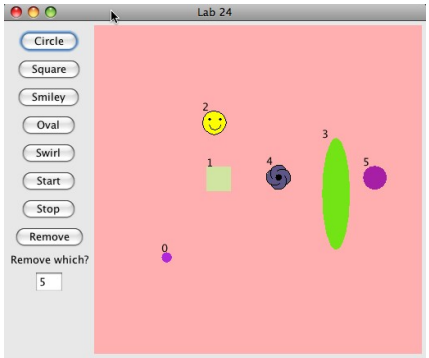


Fig 24.1 The finished Shape window

9. In the `actionPerformed` method, if the `Remove` button is pressed, store the number showing in the `JTextField` in an `int` variable. The text will need to be converted to an `int` - see preparation exercise 2. Add another statement to remove the element at this position. (This will be the last element unless the user changes it).

10. Compile and run. Add three shapes, then remove the shape at position 0. Can you see the other two shapes moving up one index position?

11. Click on the Interactions Pane then remove **all** your shapes and keep clicking the Remove button. You will see a run-time error.

12. Add a statement so that when there are no elements in the `ArrayList,` the `JTextField` shows a blank rather than `-1`.

13. Add code to catch any `NumberFormatException` when the Remove button is pressed. This would occur if there was an attempt to convert the the blank field (or any non-digit character) into an `int`.

14. One last possibility for error is that the number in the text field is greater then the size of the non-empty `ArrayList.` This would be an `IndexOutOfBoundsException` so catch that too. L,D&C page 446 [456] gives an example of two `catch` statements being used with one `try`.

Now your code should be robust enough to cope with anything a user may type in the text field.

Lab Completed

☐ preparation exercises complete ☐ remove working ☐ exceptions
☐ comments ☐ submitted

Date **Demonstrator's Initials**

Laboratory 25 Options

Notes

This is the last lab for COMP160! Its purpose is to set you free on some code to have some fun.

Lab Work

You may choose one of the following tasks, or discuss any option of your own choosing with a demonstrator:

1. Take the code for Lab 24 (Shapes) and write a different `move` method for each kind of shape.

OR

2. Take the code for Lab 24 (Shapes) and make the shapes do something new. Your imagination is the limit.

OR

3. Take the code for Lab 24 Shapes and add functionality to identify and remove the smallest circle shape.

OR

4. Take the code for Lab 19 (calculator) and improve it so that the calculator can correctly process one or more of the following input sequences :

```
8 + 2 + 4 =
3 + 10 = - 4 =
-6 + 9 =
3 + 4 = + 7 =
```

OR

5. Make a GUI version of your temperature converter program from Lab 4

OR

6. Implement an insertion or selection sorting algorithm on an array or an ArrayList.

7. Any other task which gains demonstrator approval

Lab Completed

☐ comments ☐ working ☐ submitted

Date **Demonstrator's Initials**

You made it - well done! We hope you feel that you have learned a lot about Java, and about programming, without being driven entirely mad in the process. Good luck with the final exam.

Executable Java programs

Any of your Java programs can be used to create a Java archive (jar) file. No matter how many classes are involved in the program, it will make just one jar file. A jar file is executable so is a convenient way for an end-user to use a Java program.

A jar file is transportable to other operating systems, so can run on Microsoft windows as well as other Macs. The operating system of any host machine will need to have Java installed, which keeps iPads out of the picture. Android devices are Java-friendly though.

Making a JAR file in DrJava

1. **Make sure you have compiled your code, so the .class files exist.**
2. **In DrJava, Project > New**

   Save As : – Choose a name and navigate to where you want your package saved. **Save.**

   In the **Project Properties** window which appears:

   a. **Project Root** – Defaults to Project's directory - Navigate to the directory where your `.class` files are saved. If you are working in a package, navigate to the directory **containing** the package directory.

   b. **Working Directory** – Defaults to Project's directory - Navigate to where your `.class` files are saved. If you are working in a package, navigate to the directory **containing** the package directory. Make sure the final directory name isn't doubled when it shows in the field – fix manually if necessary.

   c. **Main Class** – navigate to the application class

   If you are working in a package, it will show as `packageName.applicationClassName`
4. **OK.** Your project is made, and can be tested at this point by compiling and running if you wish.
5. **In DrJava, Project > CreateJarFileFromProject**

   Check ✓ Jar classes

   Check ✓ Make executable

   Main class – should be showing

   If you are working in a package, it will show as `packageName.applicationClassName`
6. **Jar File** – this is the classpath and the filename for your executable file. **Navigate** to where you want the file, and give it a name. **Save. OK.** Your jar file should be ready for use.

Making a JAR file in Terminal

For this example, we will assume your code is in a package, the package is called `brick` and your application class is `BrickApp`. So your `.java` and `.class` files are in a directory called `brick`.

1. Make sure you have **compiled** your code, so the `.class` files exist.
2. **In any application, make a file called `Manifest.txt` which contains the text `Main-Class:` followed by the name of the application class AND a line return at the end of the instruction.**

   e.g. `Main-Class: brick.BrickApp`
3. In order that your setup matches the instruction in Step 5, **save** the `Manifest` file in the same directory as your `.class` files (e.g. inside the directory called `brick`)
4. In `Terminal`, **navigate** to the directory **containing** the package directory.
5. **Type the instruction** `jar cfm outputFilename.jar brick/Manifest.txt brick/*.class`

   Your jar file should be ready for use.

A brief explanation of the `jar` instruction:	
`cfm`	**c**reate a jar file, output goes to a **fi**le, **m**erges info from existing files
`outputFilename.jar`	you may name the output file anything you please, and use directory paths to place it
`brick/Manifest.txt`	where to find the manifest file, and what its exact name is
`brick/*.class`	the class files for the program – can also be listed individually, separated by a comma

Readings

The following Readings cover material that is generally useful, or not described in depth in L,D&C (the text book). The first three (under the heading "Getting Started") focus on introductory topics that it may be useful to read at the start of the course. Readings marked below with a "*" are for information only (they aren't "examinable").

Index

Where do You Begin?

Learning to program can seem hard. Initially it seems like a strange way of thinking. OK – it is a strange way of thinking! But don't let that initial strangeness put you off. You should become comfortable with the ideas and skills of programming fairly quickly, and when you do you will find them very useful and powerful.

Programming is really about problem solving. A programming language, like Java, gives you a set of tools and ways of representing and thinking about problems and their solutions. The good news is that once you have learned your first programming language the worst is over, it's much easier to learn subsequent languages.

Java is an Object–Oriented (OO) programming language. You write the code for the classes in your program. Programs work by creating "instances" of classes which are called objects, and then using these objects to get things done. (So one way of thinking about classes is as "templates" or descriptions of objects that can be created). Your program can create objects which are instances of classes you have written, or instances of pre–existing classes in the Java libraries. To learn Java we need to learn about this OO way of thinking, problem solving and programming.

For most of you, this is your first programming language. The hardest thing about writing your first few programs is getting started – looking at that blank sheet of paper and thinking "now what?". Where do you begin? In COMP160 we try to help by describing labs as a clear sequence of steps to complete. You should also read, and use, the COMP160 Program Development Process described in the Reading "How to Write a Program", page 112.

A good way to begin is by writing down the problem in your own words. Start by paraphrasing the problem and then work on breaking down the task into descriptions of smaller issues. (This method of breaking big problems down into successively smaller problems is often called a top–down approach to problem solving). Doing this will help you see patterns and similarities as well as pinpoint areas that will be more difficult.

You don't need to write all the code of the program in final and perfect form right away. You can develop it from an initial outline, filling in details later on. Steps such as specifying the algorithm, or sketching out parts of the program in rough code (or "pseudocode") are a very good place to start.

L,D&C can't teach everything at once! They could have spent longer at the start on describing what objects are, and how to use them to write programs. So the next Reading is a brief introduction that might help to get you started thinking about OO programming.

Finally, there are lots of ways of solving problems and writing programs. The OO approach is only one of these. It may be that other more general problem solving methods (that are not specifically related to objects) will be useful sometimes or help get you started. There is a Reading ("Problem Solving") below on general problem solving methods.

Last thing. You write programs to do some task or solve some problem. Follow the COMP160 Program Development Process! The first step is always always always to start thinking about the problem first and how to solve it. Think first, write code later. Keep in mind the First Rule of Programming:

> The sooner you start coding the longer the program will take to write.

Seriously! If you think first and develop a good understanding of the problem and a good design for the program then the hard work is done. The code should be easy to write and you should create a clean, well organised program.

Object-Oriented Design

Java is an Object–Oriented (OO) programming language. Writing a Java program involves writing the code for the classes in the program. A running program is a collection of interacting objects which are built from the classes. In Java there are two kinds of objects:

> Class objects: One class object is automatically created for each class.
>
> Instance objects: Any number of instance objects ("instances") may be created for a class. So classes can be thought of as "templates" or descriptions of objects that can be created. Instance objects are the usual objects that people mean when they talk about OO programming.

For the early programs in COMP160 we keep things simple by writing just one class, so our programs consist of the one corresponding class object (and various class and instance objects automatically created from library classes). We don't actually write classes for our own instance objects until later (Lab 6, Lecture 6).

However, OO design issues and discussions are almost always about instance objects, so we need to be thinking about them right from the start. Make sure you read the L,D&C Section 1.5 "Object–Oriented Programming" for an introduction and a hint of topics to come.

What is an (instance) object?

In practical terms an object is an instance of a class that gets created (by the Java interpreter) when your program is run, and used in the ways determined by your program. But what is an object in terms of problem solving? What is the best way to think about objects? There are many definitions, and no one best or correct way. Here is one definition:

> "We define an object as a concept, abstraction or thing with crisp boundaries and meaning for the problem at hand." (J. Rumbaugh *et al.*, Object-Oriented Modeling and Design, Prentice Hall. 1991).

An object is a useful chunk of the problem or its solution.

Many text books suggest that it is useful to think of objects as agents – independent entities that can store information and do various things. They usually present this idea with some kind of analogy, like this one (adapted from T. Budd, Understanding Object-Oriented Programming With Java (updated edition), Addison-Wesley. 2000):

> Suppose I want to send flowers to my friend Sally in Auckland. I can't drop them off myself. But I can stop by and visit Fred the local florist. I tell Fred the kind of flowers I want to send, pay for them, and Fred arranges for them to be delivered to Sally in Auckland.
>
> To solve my flower problem all I had to do was find the right agent, Fred the florist, and give him a message containing my request. It's Fred's job to carry out my request. He has some method of doing this (some process or algorithm that he follows). I don't know, and I don't need to know the details of Fred's method (so long as it works and Sally gets her flowers). Fred's method for carrying out my request is hidden from me.
>
> What Fred really did was probably something like this. Fred probably called up a florist in Auckland and gave her a message about my request. (That florist got flowers for her shop from a wholesaler, who in turn got them from several gardeners). She probably had an assistant make up the bunch of flowers that I asked for and call a delivery person. The delivery person took the flowers to Sally. So, unbeknown to me my request was carried out by a whole community of interacting agents (florists, assistants, gardeners, wholesalers, delivery people). Each of these agents can remember certain things, and can do certain things, and can send messages to other agents.

So how do you think of an OO program? Budd says it very well:

> An object-oriented program is structured as a community of interacting agents, called objects. Each object has a role to play. Each object provides a service, or performs an action, that is used by other members of the community.

Object-oriented concepts

That was all fairly general and introductory. The notes in this section are more detailed, and they won't necessarily make sense early on in the course. But this is a section that it will be worthwhile re-reading from time to time. As you get into OO programming the advice and principles described here should start to make sense, and become very useful.

Behaviour and State: The behaviour of an object is the set of actions it can perform, in effect its methods. The state of an object is all the information held within it, in effect the values stored in its data fields. States can change over time. Almost everything that we need to know about an object is contained in its state and behaviour!

Cohesion and Coupling: Cohesion is the degree to which the responsibilities of a single object form a meaningful unit. The tasks that an object performs should all be related in some way. Probably the most frequent way in which tasks are related is by the shared need to access the same data. Bundle data and the tasks that relate to it together into an object. This cohesion is a Good Thing. Coupling describes the complexity of the relationship between objects. In particular coupling is increased when one object must access the state (the data fields) of another object. Coupling is a Bad Thing – to be avoided. Try to move the responsibility for working with a certain piece of data in to the object that actually holds the data.

Interface and Implementation (Parnas's Principles): The sum of the ways a software component (such as an object) behaves is also called its interface. (A more technical definition is the set of exported names and signatures). Describing a component according to its interface has an important benefit – it is possible for one programmer to use a component developed by another programmer without needing to know the details of how it works (how it is implemented). The hiding of implementation details behind the interface is called information hiding, and it is a Good Thing. We say that the component encapsulates the behaviour.

This makes most sense when there are multiple programmers working on the same program. Each programmer has different views of the software components. For the components they develop they have an "implementation view" – they need to see and know all the details. For the components developed by others they have an "interface view" – they only need to know how to use those components (not how they work internally). By interacting with other programmers' components only via their interfaces then all the components should interact in planned, predictable, correct ways. This separation of interface and implementation is often described as the most important concept in software engineering!

This can all be summarised in Parnas's principles:

> The developer of a software component must provide other users with all the information that they need to use the component and no other information.
>
> The developer of a software component must be given all the information necessary to carry out the responsibilities of the component and no other information.

Remember – this does not all need to make sense early on in the course. Re-read this section every now and then, and bits of it will start to make sense. Hopefully by the end it should be clear, and it should help you to think about good ways of designing and structuring your programs.

General Problem Solving

You solve problems every day; you make decisions, prioritise, juggle and plan. A good problem–solver will likely make a good programmer. To solve a problem you need to have a clear goal in mind. To get to the goal, design a sequence of steps / actions (something like for example a recipe, or the instructions for setting up a video or a mobile phone). A formal description of steps is called an **algorithm**. Noted below are some commonly applied strategies from problem solving (adapted from Dale *et al.*, Programming and Problem Solving with Ada, DC Heath. 1994) that may be helpful...

Ask Questions

"I kept six honest serving men, They taught me all I knew, Their names were What and Why and When, And How and Where and Who." – R. Kipling.

Look for familiar things

You are good at recognising patterns and similar situations in everyday life. Try and apply the same skills to problem solving! Effective programmers will identify the elements of a task that must be completed to reach the final goal. They will recognise similar situations and patterns and re-use or modify existing solutions to a problem. So, if you needed to find from a list the highest and lowest mountain peak in the world you could re–use parts of a program that had been written previously to solve the problem of finding the highest and lowest temperature on record.

Solve by analogy

Spend time thinking about the problem at hand before you start typing at the keyboard, it will pay off in the long run. Broaden the strategy of looking for features that are familiar to look for analogies – general concepts that may transfer from other domains. For example, you might think of a warehouse inventory system as being like a library catalogue. The people who designed the first Mac user interface used the analogy of a desktop, and this general concept helped them to visualise, organise and implement the interface.

Means–ends analysis

Often the beginning and end state to a problem are given, and your task is to define how to get from one to the other. Suppose you want to get from Bluff to Kaitaia, that is you know where to begin your journey and where it ends, and you are now faced with the choice of how to get there. The means you chose will depend on your circumstances which is probably a trade–off between cost, time and other factors (how much money, how much time and so on). In general terms a means–ends analysis focuses the choice of intermediate steps on those that are well suited to moving from the start state to the end.

Divide and conquer

Break the large task down into sensible, manageable, achievable pieces. A good strategy to make seemingly overwhelming odds approachable; just one step at a time. Taken as a whole, the task of "preparing for exams" might seem impossibly huge. But it can be broken down into smaller tasks, preparing for each paper, revising each lecture, and so on.

Building-blocks

A combination of look for familiar things and divide and conquer. Divide the large task into smaller units and look for existing solutions that are applicable to these units. It maybe possible to put existing solutions together in some way to solve your current task.

Merging solutions

Sometimes we can save time by merging two tasks (or parts of them) in to one. If I need to bake a cake, and make scones, I could do one (get out ingredients, bake, clean up) and then the other (get out ingredients, bake, clean up). But parts of these tasks can be shared, and it is much more efficient to combine the two processes.

How to Write a Program

There is a lot of research specifically about how people learn to program. There are at least three different kinds of attributes that you need: **Knowledge** of the programming language (e.g. knowing what a "for" loop is); **Strategies** for using the knowledge (e.g. knowing when it is appropriate to use a "for" loop in designing a program); and **Models** of the program (a clear vision of the program that you are trying to write, and if it doesn't work, an understanding of the program that you have actually written instead!).

COMP160 Program Development Process

In practical terms, we recommend the COMP160 Program Development Process described here:

1 **Establish the requirements (understand the task):** We try to explain tasks clearly in the lab book, but it is still vital to understand the task - you can't write a program if you don't know what it's for!

2 **Design the program:** Identify the objects that are required (that naturally match parts of the problem). What data does each object / class need (data fields)? What things does the object /class need to be able to do (methods)? Can you use any existing classes from the libraries (instead of writing everything yourself)?

3 **Implement the design (write the program code):** Write the code in parts / sections, running the code (testing its behaviour) frequently as you go. This makes it much easier to find and fix bugs.

4 **Test the program:** When you think the program is finished, test the whole thing. Test the range of inputs it is supposed to deal with (and also possible "bad inputs"!).

Years of experience show us that following this process results in better progress in labs…

Develop incrementally

Don't try and write a big program all in one go, then expect to compile it and run. There will be errors, and it's hard to find and fix all the errors in a large complicated program! Instead, build the program up incrementally. Write part of the program and test it thoroughly. Write a simple version of a method or class, and when it is working add further detail. In short – try to always build on a firm foundation!

Write safe programs

This topic is so important that it gets its own Reading, page 116.

Think about constraints

One tool for helping with program design is to think about constraints, such as **preconditions** and **postconditions**. Preconditions are things that must be true before a method (or any other logical grouping of code, but here we will use methods as an example) is run, and postconditions are things that must be true after a method has run. A related concept is **invariants**, things that must remain true while a method executes.

It is useful to think about and note preconditions and postconditions when designing methods. For example, in some imaginary toy banking program (where accounts have no overdraft, so must have a balance of $0 or greater), a method for withdrawing an amount from an account might look like this:

```
/* precondition: data field currentBalance must contain a value which is larger
   than or equal to the parameter withdrawAmount
   postcondition: data field currentBalance is set to the old balance, less the
   withdraw amount, this value is not negative
*/
public void withdraw (double withdrawAmount) {
   currentBalance -= withdrawAmount;
}
```

In some languages preconditions and post conditions can be expressed in the code as actual parts of the program. In Java they are usually just constraints to be aware of, and possibly include in comments or other descriptions of the code. Recent versions of Java also provide a mechanism called **assertions** which allow you to code pre and post conditions during program development (they require the Java compiler to be called in a certain way) – see for example: http://www.deitel.com/articles/java_tutorials/20060106/Assertions.html

Look for patterns

You don't have to reinvent every wheel. Lots of programmers have solved lots of problems, and you can use their work! This includes code (see below), but also useful patterns (designs). Useful patterns exist at the level of parts of programs, and at the level of the overall designs of programs.

For example, loops can be described in terms of their mechanics ("for", "do", "while" and so on). Or they can be described in terms of their design, such as counter controlled, state controlled, or sentinel controlled. A counter controlled loop repeats a certain number of times depending on the value of a counter variable. You can use any loop in the language as a counter controlled loop (but "for" loops are particularly convenient). An abstract pattern for a counter controlled loop is:

```
initialise counter variable
test counter variable and repeat loop if necessary {
   loop body
   update counter variable
}
```

A state controlled loop repeats until a certain state is achieved (either true or false as desired). This state is usually represented as a boolean condition / test (e.g. `currentPosition < length`) or the value of a boolean variable (e.g. `success = true`, such a variable is often called a "flag" because it flags an important condition). An abstract pattern for a state controlled loop is:

```
test condition or flag and repeat loop if necessary {
   loop body
   update the value of the test condition or flag if necessary
}
```

A sentinel controlled loop is usually used to read in data, particularly when we don't know in advance how much data to read. A sentinel is a special value that is used to signal the end of input (usually an "impossible" value for the input, e.g. -1 if the inputs are all positive). An abstract pattern for a sentinel controlled loop is:

```
read the first data value
if the data value is not the sentinel repeat the loop {
   loop body / process the data value
   read the next data value
}
```

There are patterns at many levels of program design, including the level of the overall design of programs. Many apparently different tasks have the same underlying structure, and there are well known, good designs for solutions. These design patterns can give a programmer excellent guidance on the structure of a program, and save a lot of time! Google for "design patterns", or see: http://www.javacamp.org/designPattern/ or examples here http://www.javaworld.com/columns/jw-java-design-patterns-index.shtml .

Look for code

There is lots of very useful code out there. The whole philosophy, and many of the practical advantages of the OO approach to programming, are about making code easier to develop and reuse.

A huge amount of useful code is already formalised and available as the Java libraries / APIs (see References at the end of this lab book). You can use the libraries for many common tasks (like searching or sorting) instead of writing your own code. There is also a lot of freely available code out there on the net (but this might be of variable quality, and of course it shouldn't be used for assignments, see the University policies on plagiarism). The Internet and WWW are largely built with free and open programs, and this philosophy of making code available is a strong tradition among programmers.

Debugging Code (and DrJava Tools)

One of the skills of programming is finding and fixing problems (debugging). Programs are complicated, and code that you wrote may not actually be working in the way that you expect! Debugging a program can be a bit like detective work, you may have to look for hints and clues, you may have to conduct a systematic study, you have got to work out what is really going on.

Error messages are your friends

If your program fails to compile, or runs and crashes, you will get an error message (these are called, respectively, compile time and run time messages). At first such messages may seem like gibberish, but they are really extremely useful, and as you gain experience you will find it easier and easier to interpret them. So, read the error messages! They tell you exactly what is wrong (if you can interpret it!).

If you need a bit of help in interpreting error messages, the Web is a great resource. I wish I had a dollar for every problem I have solved by Googling for the text of an error message (almost always you can find a useful discussion somewhere), or see for example resources like http://mindprod.com/jgloss/errormessages.html .

However, the best advice about errors is to protect yourself in advance – Write Safe Programs (see the Reading page 116).

println is your friend

If you have a program that runs but doesn't work as expected (or has a run time crash that you really can't fix using the error message), then you need to find out more about what is going on. The simplest place to start is by printing out relevant information. In Java terms this means adding a few println statements, so we can call this println debugging. (The language C prints with printf, and if you Google for "printf debugging" you will find lots of resources and debate!). You can use `System.out.println`, but Java also supplies `System.err.println` (and DrJava helpfully colours this output red!).

For example, say you have a program with unexpected behaviour. You suspect the problem may be somewhere in `MyMethod`. So add printlns to print out the inputs to `MyMethod` when it is called - are these what you expected? Add printlns to check the outputs - are these what you expected? (You are checking aspects of pre and post conditions, see the Reading "How to Write a Program" page 112). If something is wrong, investigate further. If `MyMethod` uses some variables, print them out at various points in the method. If it has a loop with a counter, maybe check the behaviour of the counter by printing it out at the start and end of the loop (or even every iteration). Often this kind of approach is all that is needed to track down a problem.

Similarly, if your program runs but crashes in `MyMethod`, add printlns at various points ("I got to here 1", "I got to here 2" and so on). Work out which line is causing the crash. What variables or operation is involved with the statement on that line? Before it gets executed, print out relevant details so that you can check them.

Use other tools

There is plenty of debate about debugging. Simple println / printf debugging gets much less useful as programs get large and complicated (and some people think it should never be used at all!). But there are also plenty of other tools you can use. Lots of languages have support programs that help to write and debug code (such as the popular "Lint" program that helps to pick the fluff out of C). In most cases an IDE will provide tools to help develop and debug programs, and here we will very briefly consider some aspects of our DrJava (see also the DrJava userdoc in LabFiles or download it from http://drjava.org/).

The DrJava interactions panel

The DrJava interactions panel is very useful. It lets you test bits of Java code without having to put them in a whole program. This is particularly useful for exploring the behaviour of objects, e.g. see Lab 7 where we use the interactions panel to make instances of various JFrame objects and explore their behaviour. You can use this to test the classes that you write! If you have written some support class, use interactions to make an instance, call the various methods with sample input to test them and make sure that they behave as expected. Then (with confidence that this class is safe) you can go on and write the rest of the program.

You can also use interactions to test most other aspects of Java code. Print out the result of a quick calculation to test the behaviour of a mathematical operator. Type in a "for" loop line by line to see how it will behave. All very useful! (DrJava is actually pretty cunning, it runs two copies of the Java virtual machine / interpreter, one to support most of the application, and one just for the interactions panel).

The DrJava debugger

DrJava supplies a built in debugger (similar examples are provided by many IDEs). This automates and greatly extends the kinds of exploration of a program that it is possible to do with println debugging. It makes it easy to examine the behaviour of a running program and the state of its data.

Common debugger tools are: stepping through code (executing it one line at a time), break points (executing code as normal but pausing at specific selected points), displaying the values of selected variables as the code executes, and even manually changing the values of variables as the code executes (to test the behaviour of the code as desired). This is much more powerful than println debugging, and really lets you see what is going on as the program runs! See the DrJava userdoc, Chapter 9, for more details, or just experiment with a simple program...

Other DrJava

DrJava, like many IDEs, supplies a facility called "projects" for managing very complicated programs with large numbers of files. We won't use it in COMP160.

Writing Safe Programs

A working program is not necessarily a good program. A good program should not only work, it should work safely – by which we mean that the program should be designed to handle possible problems without crashing. Programs crash when they execute commands that damage the operating system or the contents of memory, or try to execute commands that make no sense (like trying to divide a number by zero, or trying to read from a file that isn't there).

Java is a great language for writing safe programs because (1) it provides type checking and other kinds of checking that identify problems when the program is compiled, and (2) it simplifies and automates some of the most dangerous features of some programming languages (instead of pointers and manual memory allocation Java has references and automatic garbage collection), and (3) it provides language features that let the programmer test for and handle problems while the program is running, i.e. "exceptions" and the "try..catch" system (L,D&C Chapter 10, Lec 20, and below). We only cover only a bit of the try..catch system in COMP160 – if you are planning to go on with Java programming read L,D&C Chapter 10 some time. (Read the other chapters too!).

Handling problems

When a Java program encounters a known problem (such as trying to read an `int` value and finding that the input is not an `int`) it generates an "exception" (error), and constructs an exception object (e.g. an "InputMismatchException"), which is a special object describing the problem (much like the Event objects used in the event model). At this point, depending on how the programmer has written the program, one of three things will happen:

(1) If the programmer hasn't set up a `try..catch` for the exception the program will crash.

(2) If the problem occurs inside a `try` block the program will execute the code in the `catch` block (see examples below).

(3) If the method throws (propagates) the exception it can be caught elsewhere in the program (not done in COMP160).

In this Reading we will see some examples of the second case. The code in the `catch` block should handle the exception, possibly by printing a message to the user, and / or taking some action to recover from the problem, and / or stopping the program with `System.exit(1)` (exit with non 0 values indicates that the program is stopping because of a problem). The catch block could also use information in the exception object by calling various methods on it.

To write safe programs there are:

(A) various places where we should think about things that could go wrong

(B) various places that we should think about things that could go wrong and use try..catch

(C) various operations that are so potentially unsafe that the Java compiler forces us to use try..catch.

Let's look at examples:

Case A: Tests you should make

In most programs of reasonable size, there are many places where you can add extra safety checks. For example, if you're accessing an item at position `n` in an array of 100 elements, you could check to see (use an `if`) that `n >= 0` and `n <= 99`. If the user has entered a value `x` that should be in the range 1 to 5, you should check to see that `x >= 1` and `x <= 5`. If you are using a scanner to read `int` tokens from a string, you should check to see (use the `hasNextInt` method) that it has another `int` token before you read it. This is also a great way to process all the ints in a string without needing to know how many there are – the following loop will print (on new lines) every int in the string.

```
Scanner line = new Scanner("1 2 3");
while ( line.hasNextInt() ) {           // hasNextInt returns true or false, if true enter the loop
  System.out.println( line.nextInt() ); // nextInt returns the next int
}
```

In Lab 20 Part 2 there is a very nice example of a check to see that the structure of a string has the correct sequence of tokens before proceeding to process the string. There are many more cases like this where you can make safety checks in your programs!

Case B: Tests you should make with try..catch

Processing data that has been input to the program (e.g. from the user, or from a file) is a very common source of problems. Users and files are outside the program's control, so the program might not get the data it expects!

For example, the `readInt` method below is unsafe. If the user does as asked and inputs an `int` value it works fine, but if the user inputs something else (such as a character or a string) the program crashes with an "InputMismatchException".

```
public static int readInt() {
   Scanner sc = new Scanner(System.in);
   System.out.println("Please enter an integer);
   return sc.nextInt();
}
```

Below is a safer version of `readInt`. It follows a common general pattern for dealing with input safely:

(1) create a loop that will repeat until the input has been tested and found to be safe / correct

(2) within the loop read input and test to see if it is correct

(3) if it is correct exit the loop

(4) if it isn't correct repeat the loop and try to read more input

```
public static int readInt() {
     boolean success;          // indicates whether we have correctly read an int
     int input = 0;      // this is the variable that the method will set, and return the value of
     do {                // loop as often as necessary to successfully read int
        success = true; // assume that this attempt will be successful
        System.out.println("Please enter an integer");
        try {
           Scanner sc = new Scanner(System.in);       // make a scanner
           input = sc.nextInt();                // try to read an int
        } catch (InputMismatchException e) {    // this block will be executed if read fails
           System.out.println("Unexpected input, please try again.");
           success = false;    // indicate failure so that the loop repeats
        }
     } while (!success);       // if success is false, repeat the loop
     return input;             // else exit the loop, input is correct so return it
  }
```

In this example the key testing of the validity of the input (is it an int?) is done by the statement

```
input = sc.nextInt();
```

If the input is invalid Java generates an "InputMismatchException" and executes the catch block (which here doesn't do anything with the exception object `e`, but does take other action).

Note that the variable `input` must be declared and initialised before the `try..catch` block.

Case C: Tests you must make with try..catch

Files are notoriously dangerous sources of input (they might have moved or changed since the program was written), so in general code that accesses a file must be in an appropriate try..catch, or the program will not compile.

The code on page 119 which deals with the file is in a try block. The first line attempts to create a scanner for the given file (identified by the string `fileName`, e.g. `data.txt`):

- if the scanner is successful (the file exists and can be opened) we carry on in this block
- otherwise a FileNotFound exception is generated and we execute the catch block, which prints a message to the user and stops the program.

Note: When working with files see the Reading **"Locating Support Files"** page 120

Java Input and Output (I/O)

Java is descended from a language that was designed for embedded systems (like cell phones and dishwashers). It therefore provides very complex and fine control of input and output (I/O) from various sources (the user, files, various devices, even the Internet). In older versions of Java this flexibility led to complexity in tasks that should be simple, like reading input from the keyboard or writing output to the screen. Fortunately Java 5.0 (i.e. JDK 1.5) introduced the useful new Scanner class, which provides a much more convenient way of doing these tasks.

Streams

Before we get into the details of Scanner, it is useful to have a bit of background on the concept of streams. In computing, "stream" is used in various ways, all referring to a succession of data elements over time (e.g. "streaming media"). Java follows the style of the Unix (and Linux) operating systems, which use the concept of a stream (source or destination of a sequence of data) as an abstraction for dealing uniformly with files and devices. Java (in the Unix tradition) has three "standard streams":

in for reading data from the keyboard, e.g. System.in

out for writing data to the console, e.g. System.out.println

err for writing error messages, e.g. System.err.println (also to the console, in DrJava the output of System.err.println() is coloured red!).

Other streams can be defined as needed, to interact with devices, memory, files, strings, and so on. This used to be complicated, but the Scanner class, simplifies many aspects of reading streams (hides the details).

Reading input from the keyboard

Any class that uses Scanner must `import java.util.Scanner;` (include this statement at the start of the file containing the class). To read from the keyboard (actually via the console, as described in the next section) make a Scanner object as shown below, and call a method on it to read the data type required. This example shows reading an integer (int) value.

```
Scanner in = new Scanner(System.in);
System.out.println("Please enter an int: ");
int i = in.nextInt();
```

To read different data types just call different methods on your scanner object, e.g. to read a double call `nextDouble` (see the other scanner methods listed in L,D&C page 62 [86]).

Reading an input like this is simple, but it means that the programmer must always remember to print a prompt to the user, to tell them what kind of data to enter.

Writing output to the screen / console

Java, like most programming languages, can be run in a command line window / terminal which is supplied by the operating system (you do this, for example, in Lab 15). In this case all text input and output happens in the terminal. When Java is run via a graphical user interface or IDE (Integrated Development Environment) like DrJava, then that

interface or IDE must supply a display for basic text input and output. This display is called the console. DrJava provides a console in a complicated way. Outputs are shown in the interactions tab, inputs are read using a popup window. Messy!

Writing text output to the console is simple, use the method System.out.println. For example:

```
System.out.println("Hello world.");
```

The println method can accept any number of inputs of different data types, which get joined together (concatenated) into an output string, which is then displayed in the console.

Files

When working with support files (data, image or other files used by your program) see the Reading below on "Locating Support Files" page 120. Here we assume that the support files and your program are in the same directory, and that Java is correctly configured to use that directory. Reading and writing to a file are briefly covered in Lecture 20.

We will work only with "text files", where the contents are treated as a sequence of characters. Connected sequences of visible characters are called "tokens", tokens are separated by "white space" (invisible characters like blanks, tabs or new lines). For example, the following illustrations of files all contain three tokens:

```
token1  a  42
```

```
long-example!
42
@@@@
```

```
22      23
24
```

Reading input from a file

Any class that works with files as shown here should `import java.util.Scanner;` and `import java.io.*;` (include these statements at the start of the file containing the class).
In this example we assume a support file called `data.txt`. If we want to read an an integer from the file we can do it as follows:

```
String fileName =  "data.txt";
int nextItem = 0;
try{
  Scanner fileScan = new Scanner(new File( fileName)); // make a scanner for the file
  while (fileScan.hasNextInt()){
    nextItem = fileScan.nextInt(); // read the next token as an int
  }//while
} catch (FileNotFoundException e){
  System.out.println("File not found. Check file name and location.");
  System.exit(1);// stops the program - can't proceed without a file!
}//catch
```

The main steps are to create a scanner for the file, and read the next token in the file as an int. These steps must be contained in a try..catch block – see the Reading "Writing Safe Programs" page 116. Using for example the third of the example files described just above, this code would read each token as an `int`, and write out the values 22, 23 and 24.

The while loop checks that there is another int to read before going ahead and reading it. To read different data types call different methods on your Scanner object, e.g. to read a double call `nextDouble` (see the other Scanner methods listed in L,D&C p. 86). A common technique is to use `nextLine` to read a whole line of the file as a string and then work within the program to get the tokens of the string – see the section "Working with tokens" page 120 (and see for example Lab 20).

If the next token in the file does not match what you are trying to read (e.g. you call `nextInt` when the next token is "Abc") then the read will fail with an "InputMismatchException". You should do the extra work to write programs that are safe from this kind of error, see the Reading "Writing Safe Programs" page 116.

Writing output to a file (and other operations)

We don't write to files in COMP160, but we should note it in passing. Writing output to a file is a bit more complicated than reading (the language does not yet provided a simple tool like Scanner for writing to files).

The example in L,D&C page 458 [468] shows the basics – construct a `FileWriter` that uses a `PrintWriter`, call `print` and `println` methods to write to the file, and always close the file when you are finished (if you don't, you can in some circumstances damage your computer's file system, or more likely you will find that not all of the data "written" to the file is there).

Note that this example does not `try..catch` any possible exceptions, it just "throws" them (see the declaration of the main method), i.e. passes them off for some other part of the program to deal with (see L,D&C Ch 10).

There are many other operations you can do with files. Check if they exist, create them, create new directories, rename them, check to see if they are readable or writable, and so on. For a brief summary see the File class (package java.io) in the Java Libraries / APIs.

Working with tokens

We often want to process the tokens in a string of text (e.g. a string read in from the user or from a file). The most convenient method is to create a Scanner for the string (remember to **`import`** `java.util.Scanner;`). In this example we just use an example string containing a first name, a last name, and an imaginary age:

```
String line = "James Dean 21"; // example input string
Scanner sc = new Scanner(line); // create scanner to process input string
String first = sc.next(); // call methods to return tokens from the string
String last = sc.next();
int age = sc.nextInt();
System.out.println( age + ": " + last + ", " + first);
```

The example shows how to read the first token as a string, the second token as a string, and the last token as an int. The final statement would write the output "`21: Dean, James`". The process does not alter the original string, the value of `line` remains as set by the first statement. See also the Reading "Writing Safe Programs" page 116.

There are many other things that can be done to process tokens. See for example L,D&C 2.6, 4.6, and for a brief summary see the Scanner class (package java.util) in the Java Libraries / APIs.

Locating Support Files

Java is supposed to be "cross platform", i.e. run on different kinds of operating system, computer or device. One of the most problematic aspects of making this work is interacting with files, because different operating systems have different ways of managing files.

To support Java an operating system needs to set a CLASSPATH variable specifying the directories where Java class files can be found. Java is supposed to handle CLASSPATH issues automatically, but problems sometimes arise (in fact they account for most of the problems using Java on different machines). CLASSPATH always includes the Java libraries, and the directories where any source code (".java" files) get compiled. You can find out more about CLASSPATH (e.g. for Windows machines) here:

`http://java.sun.com/javase/6/docs/technotes/tools/windows/classpath.html`

A problem

It is very convenient to put a program (source code) and any other / support files that it uses together in one directory. L,D&C do this and we will do this in COMP160. However, getting a program to locate other files is another source of complication in Java.

The complication arises from the concept of the "current directory", which is where Java looks for support files (like images, or data files) that the program interacts with. In general the current directory is the directory where Java is launched, but different Java compilers, environments or IDEs are launched in different ways and so set the current directory differently. Often Java is launched in the same directory as your source code. In some cases it is the directory where the IDE is installed. In some cases it is the directory that was used to start the IDE (e.g. via the command line or an alias). You can find out what Java thinks the current directory is by using the System property "user.dir". e.g.:

```
System.out.println(System.getProperty("user.dir"));
```

L,D&C has been written for the typical case where the current directory (where Java looks for support files) is always the directory with the source code. So for example the DirectionPanel example in L,D&L Chapter 6 (page 307 [322]) loads an image file by referring to it with a simple name "arrowUp.gif". But this only works if the IDE happens to set the current directory that way! Sometimes (depending on the IDE), this doesn't work, so it is not safe. In short, the L,D&C approach usually works, and it is nice and simple for teaching purposes, but it is not safe, so is teaching bad habits.

For example, this simple approach to file naming doesn't work with some versions of the DrJava IDE, which assumes that the "current directory" is the directory where DrJava is installed (e.g. /Applications on Mac OS). In short, DrJava looks in the wrong place for support files, so none of the L,D&C examples using files work. We have a problem!

How do we get around this in practice?

A practical solution to the problem is to force DrJava to launch in the directory where our source code is located, so that the simple file naming scheme starts working. We can do this in COMP160 labs by launching DrJava from the command line (terminal) in the correct directory as described in Lab 15. On a home Windows machine or Mac the easiest way is to drag the DrJava application / icon (exe or package) into the same directory as your code, and launch it (double click) there.

How do we get around this ideally?

A better solution would be to write a safe program, one that does not assume that the Java current directory happens to be set correctly (i.e. to the directory with your code).

One good solution is to specify files not with their simple name, like

`"data.txt"`

(which is always relative to the current directory), but with their full path name, like (for example)

`"C:\My Documents\MyJavaWork\data.txt"` (which exactly locates the file).

How does that work in practice?

The simplest approach is to just replace all simple file names in the program code with full path names, but that is clumsy (it means the code has to be rewritten every time a directory gets moved - not so good). A better option is to use Java tools to work out the full path names for us! (P.T.O)

The `MyLoader` class shows how to use Java tools to turn a simple name into a full path name for any file anywhere in the CLASSPATH. (Because CLASSPATH always includes the directory with the source code, we can carry on with the convenient practice of putting source code and support files in the same directory).

The `getPath` method takes as input a string representing a simple name, and returns as output a string representing a full path name. Thus anywhere that we can use a string to specify a file name, we can instead use the output of the `getPath` method.

For example, to create a scanner for a data file, where the usual version is:

```
Scanner in = new Scanner(new File ("data.txt") );
```

our safe alternative is:

```
Scanner in = new Scanner(new File (new MyLoader().getPath("data.txt")));
```

Code for the `MyLoader` class is below. It could be used as an inner class. Similar approaches to this are described at: `http://www.bluej.org/help/archive.html#tip10`

```
/* Anthony Robins, June 2006, JDK 1.5
   This class is used to help locate support files.  It has one method, which takes as input
   a string that is a simple name of a file, and returns a string that is the full path name of the file.
*/

import java.net.URL;

class MyLoader extends ClassLoader {  // needs to be static if used from a static context like main

   public String getPath(String fileName) {  // takes as input a string which is a simple file name
      URL url = getClass().getClassLoader().getResource(fileName); // get url for the file
      if(url == null) {    // handle the case of file not found
         System.out.println("File not found.");
         System.exit(1);   // stops program
      }
      return url.getPath(); // return a string which is a section of the url that is the full path name of the
                                                                                                    file
   }

}  // MyLoader
```

The ArrayList class

The `ArrayList` class is part of the Collections API, a group of classes that serve to organise and manage other objects. The `ArrayList` class is part of the `java.util` package of the standard class library. It provides a service similar to an array in that it can store a list of values and reference them by an index. However, whereas an array remains a fixed size throughout its existence, an `ArrayList` object dynamically grows and shrinks as needed. A data element can be inserted into or removed from any location (index) of an `ArrayList` object.

Unless we specify otherwise, an `ArrayList` is not declared to store a particular type. That is, an `ArrayList` object stores a list of references to the `Object` class, which means that any type of object can be added to an `ArrayList`. Because an `ArrayList` stores references, a primitive value must be stored in an appropriate wrapper class in order to be stored in an `ArrayList`. Autoboxing – the automatic conversion of a primitive type to its corresponding wrapper object – will occur when you assign a primitive type to an `ArrayList`, for example an `int` will automatically become an `Integer` object.

Some methods of the `ArrayList` class of the `java.util` package:

`ArrayList()`
Constructor: creates an initially empty list

`void add (Object obj)`
Inserts the specified object to the end of this list

`void add(int index, Object object)`
Inserts the specified element at the specified position in this list

`void clear()`
removes all of the elements from this list

`Object remove (int index)`
Removes the element at the specified position in this list

`Object get (int index)`
Returns the element at the specified position in this list

`int indexOf(Object obj)`
Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element

`boolean contains(Object obj)`
Returns true if this list contains the specified element

`boolean isEmpty`
Returns true if this list contains no elements

`int size()`
Returns the number of elements in this list

The program listing below instantiates an `ArrayList` called `course`. The method `add` is used to add several `String` objects to the `ArrayList` in a specific order. One specified string is deleted, and another is inserted at the same index. As with any object, the `toString` method of the `ArrayList` class is automatically called whenever it is sent to the `println` method i.e. `course.get(1)` is the same as `course.get(1).toString()`.

When an element from an `ArrayList` is deleted, the list of elements "collapses" so that the indexes are kept continuous for the remaining elements. Likewise, when an element is inserted at a particular point, the indexes of the other elements are adjusted accordingly.

```
/*
 * Papers.java
 *
 * Demonstrates the use of an ArrayList object.
 */

import java.util.ArrayList;

public class Papers {

   public static void main (String[] args) {

      ArrayList course = new ArrayList();

      course.add ("COMP160");
      course.add ("ENG127");
      course.add ("BSNS106");
      course.add ("HIST102");
      course.add ("MATH160");

      System.out.println (course);

      System.out.println ("At index 1: " + course.get(1));

      int location = course.indexOf ("HIST102");
      course.remove (location);
      System.out.println (course);

      course.add (location, "HIST104");
      System.out.println (course);

      System.out.println ("Number of papers : " + course.size());
   }
}
```

OUTPUT

```
[COMP160, ENG127, BSNS106, HIST102, MATH160]
At index 1: ENG127
[COMP160, ENG127, BSNS106, MATH160]
[COMP160, ENG127, BSNS106, HIST104, MATH160]
Number of papers : 5
```

Specifying an ArrayList Element type

By default, an `ArrayList` can store any type of object, so in order to retrieve a specific object from the `ArrayList`, the returned object must be cast to its original class.

For an ArrayList called `buttonList` which contains `JButton` objects:

```
ArrayList buttonList = new ArrayList();
buttonList.add(new JButton("Red")); etc.
```

the line `buttonList.get(0).getText()` will fail because the `Object` class does not have a `getText` method. When the object returned is cast into a `JButton`, there is no problem calling the `getText` method on it.

```
//these lines fail
String text =  buttonList.get(0).getText() ;
System.out.println(text);

//these lines work
JButton red = (JButton) buttonList.get(0);
System.out.println( red.getText());
```

An `ArrayList` object can be defined to accept only a particular type of object. The following line of code creates an `ArrayList` object called `transport` that stores `Vehicle` objects.

```
ArrayList<Vehicle> transport = new ArrayList<Vehicle>();
```

The type of the `transport` object is `ArrayList<Vehicle>`. Given this declaration, the compiler will not allow an object to be added to `transport` unless it is a `Vehicle` object (or one of its descendents through inheritance). In the `Papers` program we could have specified that the course was of type `ArrayList<String>`.

Declaring the element type of an `ArrayList` is usually a good idea because it adds a level of type-checking that we otherwise wouldn't have. In fact, the DrJava compiler produces a warning if the `ArrayList` is not type-specified:

```
warning: [unchecked] unchecked call to add(E) as a member of the raw
type java.util.ArrayList
```

Declaring the element type of an `ArrayList` also eliminates the need to cast an object into its true type when it is extracted from the `ArrayList`.

Reference Types

This Reading follows on from Lecture 14. In the notes below we provide more detail about the behaviour of reference types.

Data fields and variables have / "contain" either primitive values (a specific number, truth value or character) or reference values (a reference to a specific object, shown as an arrow on the following pages). Primitive values are of primitive types (e.g. int, double, boolean or char). Reference values are of reference types (classes in the program that we can construct instance objects of, including classes written by the programmer and Library classes).

In other words, to say that a data field / variable is of a primitive type X means that it can have a value that is of the primitive type X. To say that a data field / variable is of a reference type Y means that it can have a value that is a reference to an object which is an instance of class Y.

In this reading we will look at the behaviour of primitive types, reference types, and Strings (a particular kind of reference type) when we pass them as arguments, copy them using assignment, and compare them.

— Primitive types —

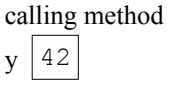
As a starting point for comparison, we will look at the behaviour of primitive types.

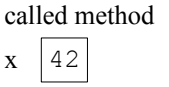
a) *passing arguments:* `x` gets a copy of the value of `y`

In this example we imagine that some method with a variable `y`, which has the value 42, calls another method called `aMethod`, passing it `y` as an input. As a result, the local variable `x` in `aMethod` gets a copy of this value. As x and y are separate variables, changing one does not affect the other.

```
// calling a method
aMethod(y);
```

```
// the method called
public void aMethod(int x){ ... }
```

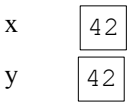




b) *assignment:* `x` gets a copy of the value of `y`

In this example we imagine that some method has two variables, `x` and `y`, and that `x` has the value 42. As a result of the assignment statement shown, the variable `y` gets a copy of this value. Again, x and y are separate variables, changing one does not affect the other.

```
x = y;
```



c) *comparison:* true if `x` and `y` have the same value

In this example we compare the values of the variables x and y. For the picture in case "b" just above, this comparison would be true.

```
x == y;
```

—— Reference types ——

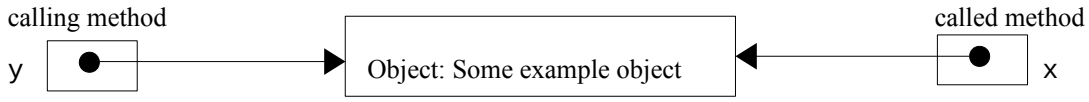
Variables of reference types (classes in the program) have reference values (a reference to a specific object).

An object is distinct and separate from a reference to the object.

For String, see the next page, otherwise...

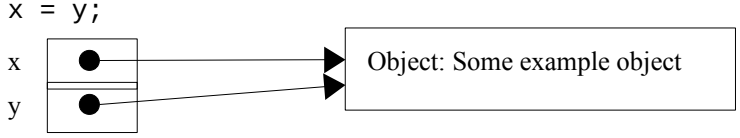
a) *passing arguments:* `x` gets a copy of the value of `y` (they refer to the same object)

```
//calling method                //the method called
aMethod(y);                     public void aMethod(Object x);
```



b) *assignment:* `x` gets a copy of the value of `y` (they refer to the same object)

```
x = y;
```



c) *comparison:* true if `x` and `y` have the same value (refer to the same object)

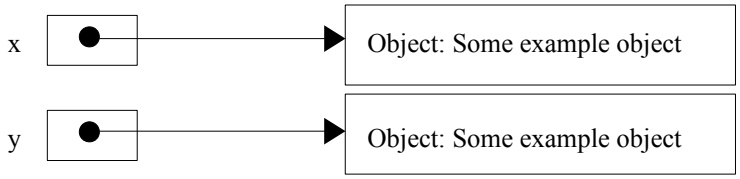
```
x == y;
```

in the pictures above `x == y` is true (in the picture below it is false)

Working with objects

If we want to copy objects themselves (rather than just the references to them) we can use for example the clone method (the example below makes object `x` a copy of object `y`). This can be very complicated.

```
Type x = (Type) y.clone();  // for some classes / types
```



We can compare objects using the equals method:

```
x.equals(y);
```

The default equals method tests for references to the same object (so it is the same as `x == y`). If you want to instead compare the states of objects to see if they are the same, then (1) for your own classes you must override this equals method (write your own!) to test for the states, but (2) most Library classes in (like String) have already done this and equals tests to see if objects are in the same state. (The states of Array objects can be compared with `java.util.Arrays.equals()`).

Most objects have default equals and clone methods inherited from `java.lang.Object`.

— Strings —

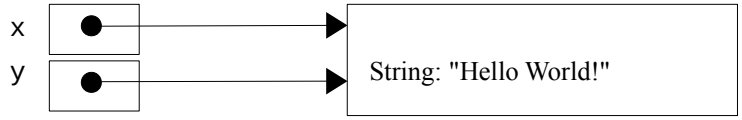
The class String includes several methods for working with string objects. Strings are immutable (see below):

a) *passing arguments*: `x` gets a copy of the value of `y` (they refer to the same string object)

```
//calling method          //the method called
aMethod(y);               public void aMethod(String x);
```

b) *assignment:* `x` gets a copy of the value of `y` (they refer to the same string object)

```
x = y;
```



c) *comparison:* true if `x` and `y` refer to the same string object

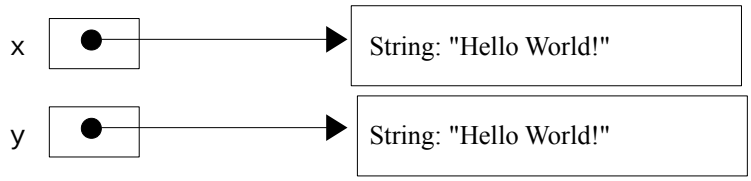
```
x == y;
```

in the pictures above `x == y` is true (in the one below it is false)

Working with String objects:

If we want to copy String objects themselves (rather than just the references to them) we can use for example:

```
String x = new String(y);
```

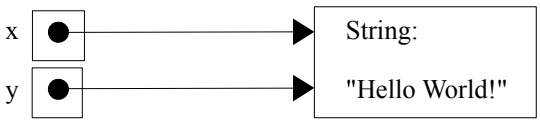


We can compare the states String objects using the equals method (the class String has overridden the default inherited from object):

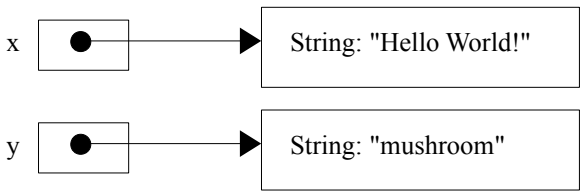
```
x.equals(y);      // is true in all pictures above because all the strings have the  same contents / state
```

NOTE: Strings and (and some other types defined by library classes) are immutable (never change their value). Instead of operations to change Strings, there are only operations to create new String objects. If `x` and `y` refer to the same String object and we assign a different value to `y`, then `x` and `y` now refer to different String objects (because `x` is not changed). For example:

On the left, after `x = y;`



On the right, after `y = "mushroom";`



Further note on the immutability of Strings.

In Java, Strings are handled in Pool format. For example:

```
String str1 = "xyz";
```

This string (`str1`) will be stored into memory in a particular address.

When you define a new String with same array of characters like this

```
String str2 = "xyz";
```

JVM will check in the String Pool whether a string containing those same characters is available. If there is a match the JVM will refer `str1` address to `str2`. Now the `str1` and `str2` refer to the same characters in the same memory location. This is a good for increasing memory efficiency. **But** - if you (could) change the `str1` characters, the changes would be reflected to `str2` because both `str1` and `str2` variables are referring to the same memory location.

To avoid this problem, Strings are immutable.

Style guide

These guidelines are part of a far more comprehensive document available from Sun at the URL:

http://java.sun.com/docs/codeconv/

Line length

Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.

When an expression will not fit on a single line, break after a comma or before an operator.

Each line should contain at most one statement.

Comments

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. Avoid duplicating information that is clear from the code.

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments should be used at the beginning of each file and before each method. All files should begin with a block comment that lists the class name, date, and author. Block comments inside a function or method should be indented to the same level as the code they describe.

```
/**
  This sort of comment should be used to head-up a method,
  explaining what it does, who wrote it, and when.
*/
```

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format.

```
// This sort of comment should be used inside methods for commenting "difficult" bits of code.
```

Very short comments can appear on the same line as the code they describe, but should be shifted far enough right to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same level.

Comments should be preceded by a blank line.

Variables

One declaration per line is recommended since it encourages commenting.

Initialize local variables where they're declared. The only reason not to initialize a variable where it is declared is if the initial value depends on some computation occurring first.

Put declarations only at the beginning of blocks (any code surrounded by braces).

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block.

Braces

The open brace "{" appears at the end of the same line as the declaration statement.

The closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement, when the "}" should appear immediately after the "{"

Statements inside braces should be indented at least one level more than the declaration statement.

Braces are used around all statements, even single statements, when they are part of a control structure, such as an if-else or for statement. This allows code to be easily added later. For example:

```
if (itemsRead == 0) {
  System.out.println("Nothing was read");
}
```

Spaces

One blank line should always be used between methods, between the local variables in a method and its first statement and between logical sections inside a method to improve readability.

A keyword followed by a parenthesis should be separated by a space.

A blank space should appear after commas in argument lists.

All binary operators except . should be separated from their operands by spaces.

A blank space should not be used between a method name and its opening parenthesis.

Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands.

The expressions in a for statement should be separated by blank spaces.

Parentheses

Use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems.

10 Tips For Beginner Programmers

1. Balance the quotes

When you type parentheses, double quote and other balanced symbols, type them in pairs e.g. `System.out.println("");` `public class Tester{}` `public void makeDo{}` then fill in the code required in between.

2. Use meaningful variable names

1. Don't use the same variable name twice in a class, even if they are used in different methods. It will make debugging easier.
2. When you name a class, variable, method, or constant, use a name that is, and will remain, meaningful to those programmers who will read your code.

3. Define symbolic names for constant values

By defining symbolic names for constant values, the program will be easier to read and modify. Suppose later you decide to change a constant: rather than change the value everywhere in your program, you can just change one value on your symbolic name directly.
e.g. use `array.length` as a test for when to end the loop rather than use the actual array size, e.g.10. If you need to change the array size later, you can simply change the size from the data field rather than change the size value everywhere within the program.

4. Initialise local variables

Initialise local variables when you declare them. e.g. `int a = 0;` `double b =0.0;` and `String today = "" ;`

5. Define small methods

Since there are few statements in a small method, it's easy to design, code, test, read, understand and use. Write a method to perform just one task rather than lots of tasks.

6. Leave a space between each section

Leave a space between class header and main method header, method header and method body and so on. It makes other people and you read the code more comfortably later.

7. Break up long lines

If you write a statement like the following, it's too long and complex:

```
System.out.println(adder(howLong("elephant","mouse"),7));
```

Write it in smaller steps like this:

```
int length = howLong("elephant","mouse");
double added = adder(value,7);
System.out.println(added);
```

8. Use `System.out.println()` help you to debug

When you are not getting the result you intended, `System.out.println()` can help you.
e.g.

```
public class TotalValue {
  public static void main(String[]args) {
    SomeClass s = new SomeClass();
    int total = sum(s.getFirst(), s.getSecond());
  }
  public static int sum(int first, int second) {
    //Use System.out.println() to check the parameter values.
    System.out.println("first is" + first);
    System.out.println("second is" + second);
    return first + second;
  }
}
```

9. Make sure your current code works before writing more

Before you do further programming, check your current code. Does it work? e.g. Suppose we want to calculate the sum of two numbers. Before you write your sum method, make sure you can get the right input value from the keyboard. Did you cast the number from String to int or double? It's much faster and easier to solve small simple problems than large complex ones.

10. Write detailed comments

Writing comments can help other readers understand your ideas easily and quickly.

Introduction

COMP160 Lecture 1
Anthony Robins

- Welcome
- Practicalities
- What is a program?
- What is Java?
- A Java program
- The programming process
- Errors
- Data structures and algorithms

Course information:
http://www.cs.otago.ac.nz/comp160/

Readings LDC (the textbook): Chapter 1.
LabBook (readings at back): "Where Do You Begin?", "Object-Oriented Design".

LDC 3e to (2e) p4 (p32)

Welcome!

COMP 160 is about:

- Learning the Java programming language
- Learning general programming principles
- Object oriented programming
- User interfaces and graphics

Programming is a useful skill, a good career, and it's fun!

But it is hard to "teach" – the best way to learn is through experience. **(Go to labs!)**

Practicalities

For all course details see the web page:

http://www.cs.otago.ac.nz/comp160

Note:

- Lecture and Lab timetables
- Assessment (terms requirement on labs, must pass final exam)
- The textbook (LDC) - Lewis, De Pasquale & Chase , *Java Foundations: Introduction to Program Design and Data Structures, 3rd edition*
- More details in the Lab Book

COMP 160 is a beginning course, but not an easy course…

What is a program?

A program is a clear and correct sequence of instructions ("code") written in some language that a computer can execute ("run").

Many programming languages exist, and more are being produced daily.

In COMP160 we use the Java programming language.

What is Java?

The **Java Platform** is a set of tools and resources originally produced by Sun Microsystems, now Oracle. The developer kit (JDK) includes:

- the programming language "**Java**" (originally called "Oak" and developed for embedded processors - small and robust)
- tools such as the **interpreter** (runs programs) and **compiler** (needed for creating programs)
- the standard **class libraries** (classes - parts of Java programs - that other programs can use)

Java is:
- **object oriented**
- widely used
- powerful and flexible
- portable
- secure

Java is not:
- a simple first language
- the answer to everything
- the fastest language

http://en.wikipedia.org/wiki/Java_(programming_language)

A Java program

A Java program consists of **code** specifying one or more **class** definitions. Each public class is saved in a file called "Name.java" ("Name" is the name of the class).

Classes may contain **data fields** (named pieces of data) and **methods** (containing **statements** that describe the steps of the program).

One class must contain a method called **main**. The program runs by executing each statement of main in turn, until the end of the main method is reached.

To use a Java program the code must be **compiled** and then **run**.

A Java program usually works by creating and using **objects** (instances of classes).

The classes from java.lang in the **libraries** are automatically available to every Java program.

Our first example program (similar to LDC p4) has only one class, with the one required main method, containing only one statement…

File name: Hello.java

```
/*  Anthony, June 2014
    Hello World program, COMP160.
*/
public class Hello {

    public static void main(String [ ] args) {

        System.out.println("Hello World!");

    }
}
```

This is a **comment** describing the program.

This is the Java code.

The program has a single **class** called Hello containing a single **method** called main (with a complicated declaration Later!). main contains just a single statement.

The statement: calls a **method** println (on an **object** out in the class System) which writes out a message on the screen. Later!

end of the method "main"
end of the class "Hello"

The program is saved in a file which must be called Hello.java (see Slide 6).

Like word processors for writing documents, there are many tools for writing programs. We use an Integrated Development Environment (IDE) "DrJava".

The formatting / layout of the program (e.g. indenting) is not part of the language, but it is necessary so that people can read and understand programs. HOWEVER – the language is CASE SENSITIVE – "Main" is NOT the same as "main".

We will use the Java standard brace style. (LDC do not!? It makes no difference to the program, but it's better to stick to standards...)

```
/*  Java standard brace style used in 160 */
public class Hello {
  public static void main(String [ ] args) {
     System.out.println("Hello World!");
  }
}
```

```
/*  LDC brace style is not standard */
public class Hello
{
  public static void main(String [ ] args)
  {
     System.out.println("Hello World!");
  }
}
```


Comments

Comments are an important part of every program (LDC 1.1).

Comments and good formatting are required for programs in COMP160.
See the Style Guide in the Lab book.

```
/*  Block comments wherever necessary, especially at the start of a class
    to describe its author and its purpose, and at the top of every method
    (except main)
*/

// single line (or end of line) comments like this to describe data and statements

/** This is a special kind of block comment called a doc comment which can be
 *  read from your program by the javadoc tool.  By default DrJava creates this
 *  kind of block comment.
 */
```

Try this

There are at least four things wrong with this program. What are they?

```
public class Words {

    public static main(String [ ] args) {

        System.out.println("aardvark");
        System.out.println("apple");
        system.out.println("albatross");
        System.out.println("zorro");


}
```


The programming process

In general (see Slide 5)

Write the **source code**. Save in file "Name.java".

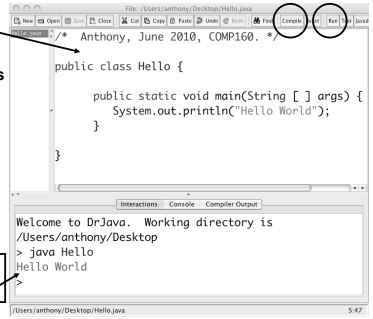
compile compiler "javac" + **library classes**

Class file "Name.class" contains object / byte code (portable - different machines).

run interpreter "java"

Executable instructions on particular machine, program runs.

In Dr Java



Errors

We all make mistakes. There are three kinds of programming error (LDC 1.2).

Syntax errors:

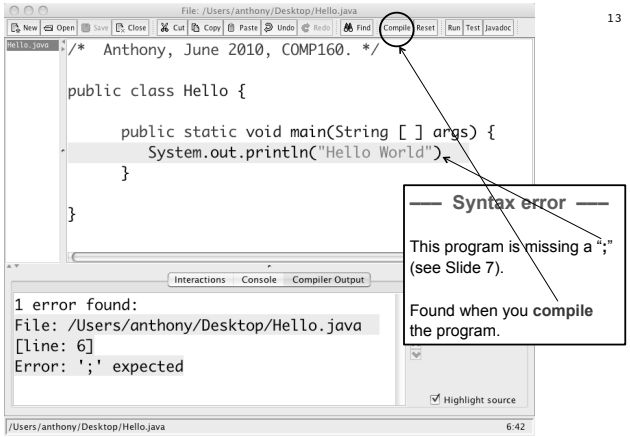
The code is not "legal". It breaks the strict rules of how Java can be written. Picked up when you **compile** the program (error messages).

Run time errors:

The code is legal, it compiles. But it specifies an operation that is impossible to execute, like dividing by zero, or opening a file that doesn't exist. Picked up when you **run** the program (error messages).

Semantic ("meaning") errors:

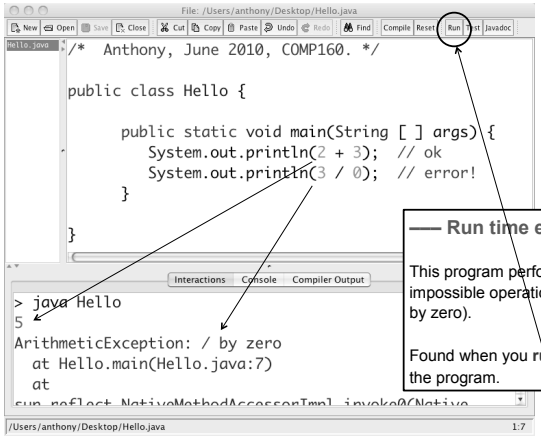
The code works, it compiles and runs. But it doesn't do what you wanted it to do. No error messages! This kind of error can only be noticed by a human.



Syntax error

This program is missing a ";" (see Slide 7).

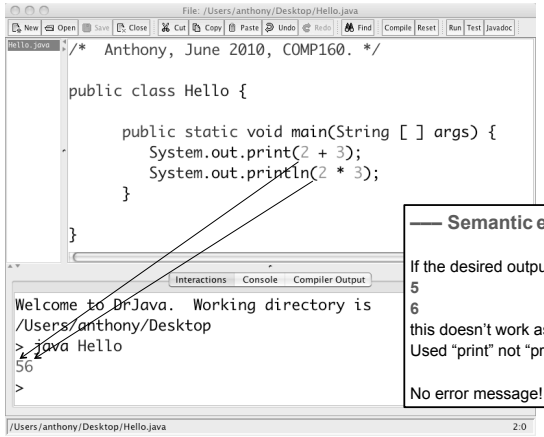
Found when you **compile** the program.



Run time error

This program performs an impossible operation (divide by zero).

Found when you **run** the program.



Semantic error

If the desired output is:
5
6
this doesn't work as planned!
Used "print" not "println".

No error message!

Data structures and algorithms

Programs process information. The information is stored in **data structures**, and an **algorithm** is a description of the processing to be carried out. These concepts lie at the heart of programming and computer science (see COSC242).

A cooking recipe is a good analogy. Most recipes set out the ingredients to be used (the data structures) and then the sequence of operations to follow (the algorithm).

Data structures become more relevant to us later, but algorithms are usually used to introduce the idea of programming:

> An algorithm is a clear sequence of basic steps.

We all know lots of algorithms already, e.g. to store a new number in our mobile phone, or to multiply the numbers 42 and 13.

> Programming is about finding the algorithms that solve problems.

A programming language is just a code for specifying data structures and writing down algorithms (sequence of statements – in Java these are in methods).

Try this

Algorithm 1: The "data structures" are the controls of your mobile phone. Write out an algorithm for saving a new phone number.

Algorithm 2: The "data structures" are a bath (with hot tap at 65˚C, cold tap at 2˚C, and plug), a thermometer, and a ruler. Try to write out an algorithm for filling the bath to a depth of 35cm with water at 32˚C.

Data types and language basics

COMP160 Lecture 2
Anthony Robins

- Introduction
- Variables
- Identifiers
- Primitive data types
- Object oriented programming
- Objects
- Strings
- Output
- Input

Readings LDC: Chapter 1, and Sections 2.1, 2.2, 2.3, 2.6
LabBook: "Where Do You Begin?" and "Object-Oriented Design"

LDC
3e to (2e)
p40 (p65)
p50 (p74)
p64 (p88)

Introduction

In some OO programming languages it is true that **"everything is an object"**.
In Java it is almost true. As well as objects Java also uses some primitive values.

Today we look at variables and data types. Variables of **primitive types** hold primitive values such as numbers or letters. Variables of **reference types** hold references (pointers) to instance objects (made from classes).

Types are checked for **consistency** as variables are used in the program.

This lecture is a quick introduction to a lot of stuff!
We get back to all of it again, so don't panic.

Variables

A variable is a named bit of data. It needs to be **declared** and **initialised**. We can assign a value to a variable using = (the **assignment operator**).

int x; *// **declares** a variable called x holding values of type int (integers)*
x = 1; *// sets x to 1, "**initialises** the variable" (assigns its first value)*
int y = 2; *// we can combine declaration and initialisation in one step*
x = 3 + 5; *// sets x to 8, we can assign values that are calculated*
x = y + 4; *// sets x to 6, calculations can involve other variables*
x = x + 10; *// sets x to 16, calculations can involve the current value!*

If data is declared as "final" and initialised it can never be changed again – this is called a **constant** (e.g. maths constants like PI) and named in CAPITALS:

final int A = 7; *// the value of A can never be changed*
A = 2; *// this would cause an error!*

Identifiers (names)

Each variable, class and method needs an **identifier**.

Thing named	Convention	Examples
variable or method	Start with lowercase, each new word with uppercase.	main, subTotal aMethodName
class	Start with uppercase, each new word with uppercase.	SomeClassName AnotherExample

See LDC 1.1. Some identifiers (e.g. "void") are **reserved words** – part of the Java language – you can't use them to name other things.

Identifiers should be meaningful, not too long and not too short.

Primitive data types

In Java the type of a variable is determined when it is created (static typing). Java has the following types – COMP160 only uses the ones **underlined**:

Value (concept)	Primitive data type
reals (fractional / decimal numbers)	float, **double**
integers ("whole numbers")	byte, short, **int**, long
truth values (true, false)	**boolean**
characters (letters, digits, symbols)	**char**

Why have data types?

- so that the compiler can check for **consistency** / validity (e.g. trying to add a **char** to a **double** makes no sense)
- to use memory efficiently

LDC wrong, no commas

double	decimal / fractional numbers like 3.14159 478911.0 uses 64 bits of memory to represent values $-1.7 * 10^{308}$ to $+1.7 * 10^{308}$ or fractions to $\pm 4.9 * 10^{-324}$ (roughly 15 significant digits) **float** represents a smaller range but otherwise behaves like **double**
int	whole numbers like 0 17 -34 189467 uses 32 bits of memory to represent values -2147483648 to 2147483647 **byte**, **short** and **long** use different amounts of memory and can represent different ranges of values, but otherwise they behave just like **int**.
char	single letters, digits, and symbols like 'X' 'b' '3' '#' '!' ' ' '.' uses 16 bits of memory to represent any character in the international **Unicode** character set (see LDC p50) – almost any language.
boolean	logical values **true** **false** uses 1 bit of memory to represent the two "truth values"

Operators (Lecture 4)

addition	**+**	comparison	**< > == != <= >=**
subtraction	**–**		less than, greater than, equal, not
multiplication	*		equal, etc. Result is boolean e.g.
division	**/** or **%**		3 < 5 is **true**

Scientific notation

Java writes large and small values in **scientific notation** For example:

number	scientific notation	Java scientific notation
49800000.0	4.98×10^{7}	4.98e7 or 4.98e+7
0.000038	3.8×10^{-5}	3.8e-5

Not covered in LDC, see for example http://en.wikipedia.org/wiki/Scientific_notation

Unicode and ASCII

The characters we deal with (typical English keyboard) are the ASCII characters (subset of Unicode). Like numbers, they have an ordering:

' ' **<** digits (0 < 9) symbols punctuation **<** 'A' **<** 'Z' **<** 'a' **<** 'z'

Try this

Assuming the declarations, which of the statements are legal and which are illegal?

declarations	statements
int sum, total;	sum = 5;
double range;	total = sum * 3;
boolean test;	range = 47.248573;
char letter;	letter = 'H';
	test = 2 < 9;
	total = 53.6;
	range = 4.3 * test;
	letter = A;

What declarations are needed to create the variables in these statements?

declarations	statements
	key = 'g';
	result = true;
	average = 4.326;
	games = 22;

Object-oriented programming

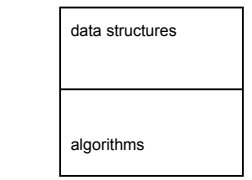
"Imperative" languages are the oldest and most common. They are structured a lot like a cooking recipe, data structures at the start, then code implementing the algorithms.

All the cooks share one kitchen – chaos!

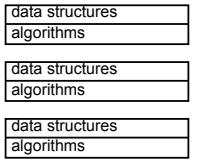
"OO" languages are better for large complex tasks (and teams) because they group specific data and the algorithms that work on it into well managed strutures called **objects**.

Each cook in their own small kitchen.

program (imperative)



program (object-oriented)



Objects

There are two kinds of objects:

Class objects: one created automatically for every class in a program, including your classes and library classes such as those in java.lang (automatically part of every Java program).

Instance objects: any number may be created (from the classes / class objects). Some library instance objects created automatically.

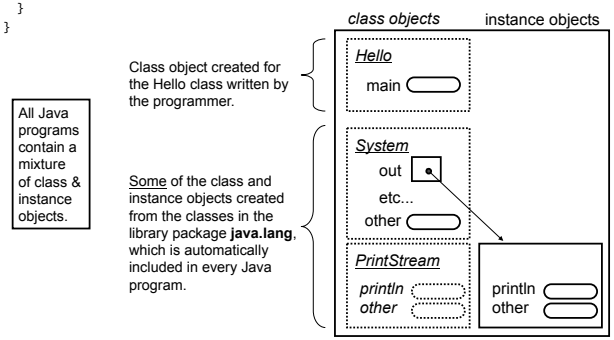
Most of object-oriented programming
is about the creation and use of **instance objects**.

For our early programs we write just one class / **class object.** We don't write multiple classes and make **instance objects** from our own classes until Lecture 6.

A model of (some of the) objects involved in the Hello program from Lecture 1…


```
/*  Hello World program, COMP160. */
public class Hello {
  public static void main(String [ ] args) {
     System.out.println("Hello World!");
  }
}
```

A model of the objects that are part of this program when it runs. You don't need to follow all the details yet - Later!



Strings

A string is a sequence of characters:

"this is a string" ":this! TOO2" "$^^<NN" " $ " "strings can be very long…"

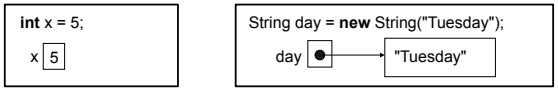
We can declare and initialise variables of type String like this:

String name = **new** String("Anthony");

String name = "Anthony";

The first example is the usual way we make an **instance object** (with **new**). Because strings are so common Java allows the second way as a shortcut.

Strings are -
not primitive values (**primitive types**), they are instance objects (**reference types**).



Strings can be concatenated (joined together) using "+"

```
String s;    // declares a variable s of type String
s = "Hello" + "World";    // initialises s
System.out.println(s);    // prints s
```

Output:

```
HelloWorld
```

We can "call methods" on objects. Strings are objects, so we can call methods on them, e.g. the "length" method which returns the number of characters:

```
String day = "Wednesday";
System.out.println(  day.length()  );
System.out.println(  "Hello".length()  );
```

Output

```
9
5
```

More later!

Try this

What output is created by the following code?

```
String a, b;  // declares two String variables
a = "Apple";
b = "Banana";
System.out.println(  a + b  );
System.out.println(  a + "##" + b  );
System.out.println(  a + "  " + b  );
System.out.println(  a.length()  );
System.out.println(  "The sum " + 5 + 2  );
System.out.println(  "The sum " + (5 + 2)  );
```


Output

Programs need to communicate with the user, and possibly with data files or devices.

For simple text output to the screen use System.out.println(). The println method takes its inputs (the parameters in brackets), turns them into a string, and writes out the string. The print method is similar, but does not move automatically to the next line:

```
System.out.println("Don't");
System.out.println("Panic " + (40 + 2) );
```

```
Don't
Panic 42
```

```
System.out.print("Don't");
System.out.print("Panic " + (40 + 2) );
```

```
Don'tPanic 42
```


We can put special "escape sequences" (LDC p40) into strings that control the way they are printed, e.g. "\n" means print on a new line, "\t" means insert a tab.

```
System.out.println("\tStudents\n\t========\nAnne\nBetty");
```

```
        Students
        ========
Anne
Betty
```


Input

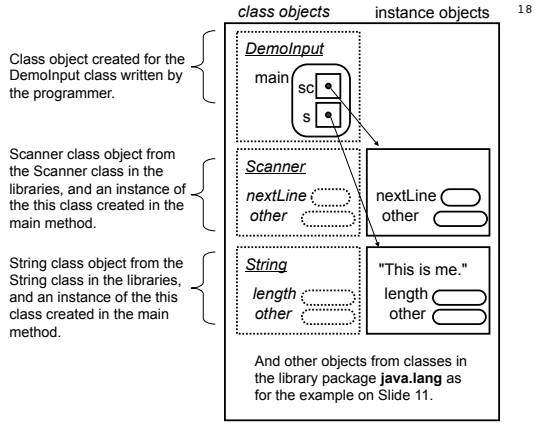
For simple text input from the keyboard make a Scanner object set to System.in as shown. (We're making and using instance objects from library classes already!).

```
/*  Anthony, July 2010, COMP160. */
import java.util.Scanner;   // Scanner class must be imported to be used – later!
public class DemoInput {
    public static void main( String [ ] args ) {
       Scanner sc = new Scanner( System.in );  // make a scanner object sc
       System.out.println( "Please enter some text: " );
       String s = sc.nextLine();   // call a method on sc to read a line from the
                                   // keyboard and store it in String variable s
       System.out.println ("You entered: " + s);  // print out a message and s
    }
}
```

```
Please enter some text:
This is me.
You entered: This is me.
```

Similar example LDC p64.

Model of some of the objects in preceeding program.



Program structure, methods and basics.

COMP160 Lecture 3
Anthony Robins

- Introduction
- Program structure
- Variables again
- Methods
- Libraries / API
- Developing programs

Reading: Revise earlier readings. Optional look ahead to LDC 5.4.
Look at Java libraries, Slide 16.

Introduction

This lecture introduces some basic aspects of how a Java program is structured and run. It gets a bit ahead of LDC Ch 2 (later!).

There are two main kinds of Java program:

Application programs: The most common form, these are "stand alone" programs that can run directly on a machine (i.e. using the Java interpreter).

Applets: These are programs that run within browsers like Firefox / Internet Explorer, or within Applet Viewers.

In COMP160 "program" will always mean an application program, until Lecture 24 when we look at applets briefly.

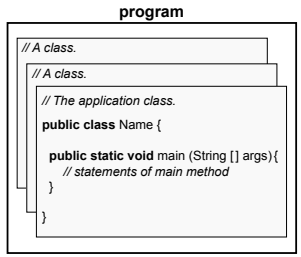
Program structure

A program consists of one or more **classes,** one of which must be the **application class**.

Application class has a main method. The program runs by executing the statements in main.

Classes are like "descriptions" of objects that can be created.

Programs usually run by creating instances of objects (from classes) and using them.



A **class** consists of:
- a declaration / header
- data fields (0 or more)
- methods (0 or more)

A **method** consists of:
- a declaration / header
- local variables (0 or more)
- other statements (0 or more)

Executable statements (usually single lines of code) are the basic building blocks of program. Every step of program execution is specified by some statement.

```
/* Anthony Robins, July 2010, JDK 1.6
   An example of an application class */

public class Example {
    private static int x = 5;
    private static String greet = "Hello";

    public static void main( String [ ] args ) {
        int y = 2;
        String name = "Fred";
        System.out.println( x + y );
        System.out.println( greet + " " + name );
    }

} // end of class
```

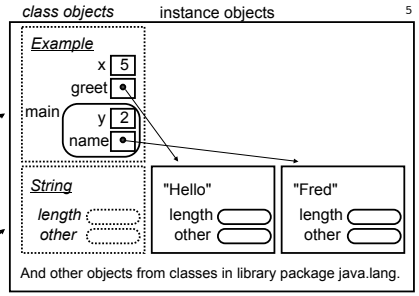
```
7
Hello Fred
```
program output

For now, make classes **public**, data fields **private static**, methods **public static**.

Example has 4 variables = 2 data fields (x, greet) and 2 local variables (y, name).

Some of the objects created (partial models only!). Various **class** and **instance** objects from java.lang are created automatically.

Until Lecture 6 we will stick to programs where we write just an application class (with **static** data fields and methods – later!).



Class String is in java.lang so automatically imported for use. It has a length method (Lecture 2) and many others.

Note: x and y are **primitive types** (hold primitive values), greet and name are **reference types** (hold references to instances of the class String).

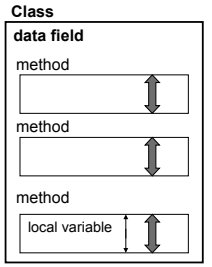
Variables again

Data is stored in named variables. There are two kinds of variables, **data fields** (of classes) and **local variables** (of methods).

They are very similar, except that:

data fields
- can be used by every method in the class
- have a specified visibility, e.g. **private**, and need to be **static** if used in a class object (later)
- may be initialised or are automatically initialised to zero states (0, 0.0, false, ' ', null).

local variables
- can only be used within their method
- so no need to specify visibility or static (later)
- must be initialised (never automatic).



The "**scope**" of a variable is the parts of the program where it can be used (later).

A variable can hold **primitive values** (numbers, letters). or **references to instance objects** (made from classes).

Primitive variables and reference variables are very similar.

For example, they can be declared and / or initialised in the same ways:

```
int sum = 5;  // declare & initialise
```

```
int sum;   // declare
sum = 5;  // initialise
```

sum 5

```
Scanner sc = new Scanner();
```

```
Scanner sc;          // declare
sc = new Scanner(); // initialise
```

sc → Scanner instance object see lecture 2

Try this

What is printed out by this program?

What happens if you delete the statement "y = 4"?

```
/* Anthony Robins, July 2010, test initialisation */
public class SomeData {
  // data field declarations
  private static int a = 1;
  private static int b;

  public static void main(String[ ] args) {
    // local variable declarations
    int x = 3;
    int y;
    // other statements
    y = 4;
    System.out.println("Fields a & b: " + a + "," + b);
    System.out.println("Locals x & y: " + x + "," + y);
  }
}
```


Methods

A method is a named sequence of statements. All the processing done by the program (the algorithm) is specified by the statements in the methods.

Methods can be "called" (executed, run). When a method is called the current method pauses and waits for the called method to finish, then continues…

(If a method is never called, it never runs!)

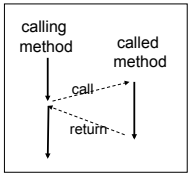
```
One fish
Two fish
A poem by
Dr Seuss
Red fish
Blue fish
```

```
/* Anthony Robins, March 2006, JDK 1.5 */

// The application class
public class Poetry {

  // The main method (program starts here)
  public static void main (String [ ] args) {
    System.out.println("One fish");
    System.out.println("Two fish");
    myMethod(); // call myMethod
    System.out.println("Red fish");
    System.out.println("Blue fish");
  }

  // This method is called by main
  public static void myMethod() {
    System.out.println("A poem by");
    System.out.println("Dr Seuss");
  }

} // End of class
```

This is our first look at the important programming concept "flow of control" - the sequence in which statements are executed.

Unless otherwise specified, statements are executed sequentially.

Sequential execution can pause, move to a called method, then return and resume in the calling method, as shown in the previous example.



We look at other ways of specifying flow of control later (LDC Ch 4, Lectures 9 - 12)…

A related concept is "flow of data" - the "movement" of data around a program, e.g. reading it in, passing it to methods (slides 12 ,13), writing it out.

—— Try this ——

For the two example programs below, assume that the methods shown are in some application class. In each case, what is printed out when the program runs?

```
public static void main (String [ ] args) {
   System.out.println("Apple");
   System.out.println("Banana");
   three();
   two();
}

public static void one() {
   System.out.println("Cherry");
}

public static void two() {
   System.out.println("Date");
}

public static void three() {
   System.out.println("Eggplant");
}
```

```
public static void main (String [ ] args) {
   System.out.println("Apple");
   System.out.println("Banana");
   three();
}

public static void one() {
   System.out.println("Cherry");
}

public static void two() {
   System.out.println("Date");
}

public static void three() {
   one();
   System.out.println("Eggplant");
}
```


—— Passing data to a method ——

We can pass / send data (one or more values of some primitive type or references to objects) to a method when it is called. These are called the "parameters" or "arguments" to a method (Lecture 6).

```
/* Anthony Robins, Illustrate passing data to a method */
public class PassingData {

   public static void main (String [ ] args) {
      aMethod(42);  // call aMethod and pass it the value 42
   }

   public static void aMethod(int myInput) {
      System.out.println(myInput);  // prints 42
      myInput = myInput + 10;       // add 10 to myInput
      System.out.println(myInput);  // prints 52
   }

}
```

```
42
52
```

This example passes the value 42 (an **int**eger) to aMethod, where it is stored in a local variable called myInput.

myInput 42

aMethod prints the value of myInput, adds ten to it, then prints the new value.

—— Returning data from a method ——

We can return data (a single value of some primitive type or a reference to an object) from a method when finishes. Such methods state the type of value returned in their header (or state "void" if no value is returned - e.g. main).

```
/*Anthony Robins, Illustrate returning data from a method */
public class ReturningData {

   public static void main (String [ ] args) {
      int x = aMethod();
      System.out.println(x);
   }

   public static int aMethod() {
      // "return" will end the method and send the specified
      // value (e.g. a variable or the result of a calculation)
      // back to where the method was called from.
      return 2 + 3;
   }

}
```

```
5
```

This example calls aMethod which does nothing except return the **int**eger value 5 as a result. (The method could have had any other statements as usual before the return).

In the main method the returned value is stored as the value of the variable x, which is then printed out.

—— Try this ——

Write a public static method called pMethod which takes a single integer as an input. The method should print out the value and then return a result which is that value plus 5.

We will be using methods in the labs soon, passing and returning values, so this is an important topic! See further examples in LDC, optionally look ahead to Section 5.4.

We have been passing and returning values already!
In this code (Lecture 2 Slide 17) lines 2 and 4 send values to the println method, and line 3 uses a value returned by nextLine method.

```
Scanner sc = new Scanner( System.in );  // make a scanner object sc
System.out.println( "Please enter some text: " );
String s = sc.nextLine();   // call a method on sc to read a line from the
                            // keyboard and store it in String variable s
System.out.println ("You entered: " + s);  // print out a message and s
```


Libraries / API

These are packages of classes that we can use in our programs (classes in package java.lang are used automatically). They are part of the Java Developer Kit (JDK).

They include the **JFC** (Java Foundation Classes) and other packages of classes. The have various names including, the **libraries**, the **class libraries**, and the **Java API** (Application Programming Interface).

They are documented on line in various places, especially:

http://docs.oracle.com/javase/7/docs/api/
(see link from COMP160 home page, resources)

This can be very useful! If you want to see how stuff works you need to explore the libraries.

Developing programs

Think before you code! Writing programs can be complicated, and right from the start it is useful to think about using a systematic development process. This is even more important for big projects (see "software engineering" in later papers). See LDC 1.2 – 1.4.

Steps of a typical development process:

1 Establish the requirements (understand the task).
2 Design the program.
3 Implement the design (write the program).
4 Test the program.
5 Maintain the program over its lifetime of use.

The next slide outlines the development process recommended in COMP160. It expands on some of the steps above, and drops Step 5 (not relevant for us).

—— COMP160 program development process ——

1 Establish the requirements (understand the task).
We try to explain tasks clearly in the lab book, but it is still vital to understand the task - you can't write a program if you don't know what it's for!

2 Design the program.
Identify the objects that are required (that naturally match parts of the problem). What data does each object / class need (data fields)? What things does the object /class need to be able to do (methods)? Can you use any existing classes from the libraries (instead of writing everything yourself)?

3 Implement the design (write the program code).
Write the code in parts / sections, running the code (testing its behaviour) frequently as you go. This makes it much easier to find and fix bugs.

4 Test the program.
When you think the program is finished, test the whole thing. Test the range of inputs it is supposed to deal with (and also possible "bad inputs"!).

Years of experience show us that following this process = better progress in labs…

Expressions. Arithmetic.

COMP160 Lecture 4
Anthony Robins

- Introduction
- readInt
- Expressions
- Arithmetic operators
- Arithmetic constants and functions
- Precedence and brackets
- Conversion and casting
- Composition

Reading: LDC Sections 2.4, 2.5., 3.5.

LDC
3e to (2e)
p52 (p76)
p956 (p820)

Introduction

Today we look at **expressions** – the various ways that values can be represented or calculated. We focus on numbers and **arithmetic expressions** (different kinds of expressions later).

Apart from some different symbols and a couple of shortcuts, arithmetic all works exactly as expected.

In this lecture we assume the following variable declarations:

```
int i = 1;
double db = 14.3;
char ch = 'w';
boolean bob = false;
String str = "Hi there.";
```

We also assume a readInt method…

readInt

This program revises topics from last lecture, passing data to a method (in this case a string) and returning data (in this case an integer value).

It uses a Scanner object to read a single integer from the keyboard with the **nextInt** method (compare with Lec. 2 Slide 17, **nextLine**).

In many lectures from now on I'll assume the existance of this method and use **readInt** in example code.

```
/* Anthony Robins, introduce readInt method */
import java.util.Scanner;

public class DemoReadInt {

  public static void main ( String [ ] args ) {
    int x = readInt( "Please enter an integer:" );
    System.out.println( "You entered: " + x );
  }

  public static int readInt( String message ) {
    Scanner sc = new Scanner( System.in );
    System.out.println( message );
    return sc.nextInt();
  }

}
```

```
Please enter an integer:
17
You entered: 17
```


Expressions

An **expression** is a specification of a value

When a program is run each expression is evaluated, and the resulting value is used. So:

Anywhere we expect a value (of a type)
we can use **an expression** (giving a result of that type)

For example, if myMethod expects an integer as an input, any of these will work:

```
myMethod( 5 );
myMethod( 7 – 2 );
myMethod( i );          // assume i declared & initialised – see Slide 2
myMethod( readInt("Enter integer: ") );
myMethod( 2 * i + readInt("Enter integer: ") – 42 );
```


Kind of expression	Examples					
literal (as written)	1	14.3	true	'v'	"why me?"	Note 1
variable (if initialised)	i	db	bob	ch	str	Slide 2
assignment	i = 22	bob = false	i = readInt("Enter integer:")			Note 2
method call (if not void)	Math.sqrt(25)	readInt("Enter integer:")				Note 3
calculation (uses operators)	2 + 3	"abc" + "2K"	7 + readInt("Enter integer:")			
	i < 22	(db + 11.7) / 1.5				

Notes
1: literals are usually primitive values but strings and arrays (later) can also be literal
2: i = 22 sets i to 22 and also evaluates to the value 22
3: Math.sqrt() see Slide 9

Arithmetic operators

Expressions can combine values using operators. The arithmetic operators are the ones used on numerical values (**int** and **double**):

+	addition	5 + 2 is 7 5.1 + 2.2 is 7.3
–	subtraction	5 – 2 is 3 10.0 – 5.3 is 4.7
*	multiplication	5 * 2 is 10 5.1 * 3.0 is 15.3
/	division	8 / 2 is 4 5 / 2 is 2 5.0 / 2.0 is 2.5
%	remainder	13 % 5 is 3 5 % 2 is 1 10.5 % 3.0 is 1.5

% used on integers is sometimes called "mod" or modulo

/ operator

e.g.: 5 divided by 2 is 2 with 1 remainder

% operator

Note: also unary minus, e.g.: sum = –1;

Assignment operators

It is very common to take the current value of a variable and modify it:

```
i = i + 5;      // i is set to its old value plus 5
```

This is so common that it can be abbreviated to:

```
i += 5;
```

and equivalently for other operations: –= *= /= %=

Increment (and decrement) operators

It is very common to take a current value and add one to it (increment):

```
i = i + 1;        or        i += 1;
```

There are two other ways to increment:

```
i++       // post-increment:   evaluates to old value and increments
++i       // pre-increment:    increments and evaluates to new value
```

and equivalently for subtracting one (decrement): i–– –– i

Try this

What output is produced by this code?

```
int i = 3;
System.out.println( i += 2 );
i = 10;
System.out.println( i++ );
System.out.println( i );
i = 10;
System.out.println( ++i );
System.out.println( i );
i = 10;
System.out.println( i *= 4 );
i = 10;
i += (i * 2);
System.out.println( i );
```


Arithmetic constants and functions

Useful arithmetic constants and functions are available in class Math.

Math is a class in the package java.lang (automatically imported), so every program contains a Math class object with data fields and methods we can use:

Math.PI	is 3.14159...
Math.E	is 2.71828...
Math.sqrt()	returns square root of input
Math.sin()	returns sine of input
Math.max()	returns maximum of two inputs
etc...	

data fields in Math
methods in Math
(both specified with "dot notation")

We can use these in expressions like other values / expressions:

```
db = Math.PI / 2.0;
db = 7.0 + Math.sqrt(100.0);
System.out.println( Math.sqrt(50.0 + 50.0) );
```

Precedence and brackets

Precedence works exaclty as for standard mathematical notation, with operator precedence, left first processing of equal precedence, and brackets.

method call (e.g. Math.sqrt()) — highest
– (unary minus e.g. –5)
*, /, %
+, –
= — lowest

See LDC p52-56, p956-957 (later), Lecture 9.

Examples:

```
i = 5 + 2 * 2;        // i is set to 9, * (higher precedence) before +
i =  12  /  2  *  3;  // i is set to 18, equal precedence left first
i =  (12  /  2)  * 3; // i is set to 18, optional brackets
i =  12  /  (2  * 3); // i is set to 2, brackets can change order of evaluation
```


Try this

What is the value of the integer variable i after each of these statements?

```
i = 5 * 5 + 2;                Value of  i is  27
i = 5 * (5 + 2);
i = 8 / 4 / 2;
i = (4 / –2) + ( 4 % 2);
```

Given i = 10, what is the new value of i?

```
i += 2 + ( 2 * ( i – 1 ) );
```


Conversion and casting

Java has "type checking", so we can only use data of the right type (in assignments, calculations, method calls, etc). But data can be converted from one type to another.

For most primitive types, conversion to a "bigger" type / range of values (widening) happens automatically in two ways, **assignment conversion** and **promotion**.
The only way to convert to a "smaller" type / range (narrowing) is explicit **casting** (with a **cast operator**). See LDC 2.5.

primitive type

widening conversion (automatic) ↓ char (16 bits) int (32 bits) double (64 bits) ↑ narrowing conversion (must be explicit)

Later we will see that we can convert from one reference type (class) to another too.

Assignment conversion: occurs when a value of one type is assigned to a variable of another type, for example:

```
double d = 1;          // assign. conversion of int 1 to double 1.0
```

Promotion: occurs when operators need operands of a certain type, for example if result and sum are **double**, and count is **int**:

```
result = sum / count;  // promotion of int count to double
```

automatic widening

Casting: we can use a cast operator – the name of a type in brackets – to force the conversion / casting of one type into another, for example:

```
int i = (int) 15.32;   // cast of double 15.32 to int 15
```

only way to do narrowing

Other examples:

```
int i = 2.988;          // Illegal narrowing,  causes an error
int i  =  (int) 2.988;  // explicit narrowing cast double to int, sets i to 2
int i = 'a';            // assignment conversion, char to int, sets i to 97
char c = (char) 97;     // explicit narrowing cast int to char, sets c to 'a'
System.out.println( 3 + 'a' );  // promotion of char to int, prints out 100
```

Note: if an expression involves integers and doubles the result is always double:

```
1 + 2 + 3 + 4.0        has the value 10.0
```


Try this

Given these declarations:

```
char c = 'a';     // note that 'a' has the value 97 when cast to int
int i = 2;
double d = 3.6;
```

what happens in each case? Note any casts…

```
d = i;                  assignment conversion, sets d to 2.0
i = c;
i = (int) d;
System.out.println( 2 + (int) d );
```


Composition

The same operations can be expressed in many different ways, e.g. using a sequence of separate steps, or (often) by "composing" the steps into one.
For example, Given: **double** d = 100.0;

```
d = Math.sqrt(d);          // sets d to 10.0
d += 5.0;                  // sets d to 15.0
d *= 2.0;                  // sets d to 30.0
System.out.println(d);     // writes out 30.0
```

Compare with:

```
System.out.println(  d = 2.0 * ( 5.0 + Math.sqrt(d) ) );  // calculate & write 30.0
```

As you gain experience you tend to move from the first style to the second, but don't sacrifice clarity!

Try this

Given
double d = 5.0;

Note: Math.pow(x, y) raises x to the power of y, e.g. Math.pow(4, 2) is 4^2, which is 16.

and the following sequence of statements:

```
d *= 2;
d = Math.pow(d, 2);
d = d + 10.0;
System.out.println(d);
```

write a single statement that performs the same calculations and prints the same result:

Graphics, drawing & GUIs.

COMP160 Lecture 5
Anthony Robins

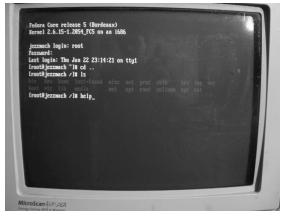
- GUIs
- Graphics in Java
- Drawing
- Simple design
- Einstein example
- A brief history of programming languages

Reading: LDC: Appendix F.

LDC
3e to (2e)
p968 (p832)
p11 (p39)

GUIs

Computers used to have simple text "command line interfaces" (CLIs).



Today it's almost all complex "graphical user interfaces" (GUIs).

By the end of COMP160 you will be able to code a simple GUI with buttons, text fields, actions, and so on, even simple animation.

For teaching, graphics has advantages (interesting and fun!) and disadvantages (complicated, automatic "magic" starts happening).

Graphics in Java

Graphics in Java uses classes in libraries called **Swing** and the **AWT** (Abstract Windowing Toolkit). AWT is from older versions of Java, Swing builds on AWT.

```
import javax.swing.*; // import all classes in these packages - needed to do
import java.awt.*;    // graphics (these imports usually the first lines of the file
```

These contain classes representing **components** such as windows, frames, panels, buttons, text fields, scroll bars check boxes, menus, and so on.

As much as possible Java graphics is self-contained (lightweight) and thus portable, but basic functions must draw on the device's OS / windowing system (heavyweight) which requires customisation of a version of the Java platform for that device.

In COMP160 we mostly do graphics towards the end of the course, but today we introduce the basics of simple drawing in a JFrame (a kind of window).

Drawing

Components like windows, frames and panels have a **paint** method…

All graphics "drawing" code should be in (or called by - later) a component's **paint** (or **paintComponent**) method.

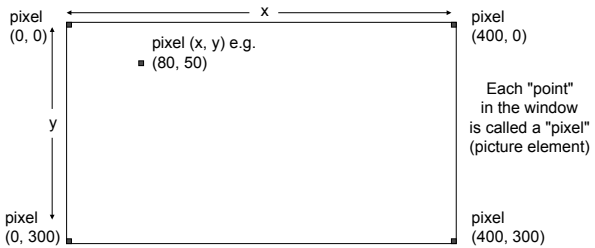
… called automatically when needed, e.g. to display graphics as windows get resized, moved, redrawn etc. It is passed a reference to a Graphics object…

```
public void paint (Graphics g) {
    g.drawLine (0, 0, 80, 50);    // draw a line between two points / pixels
    g.setColor(Color.red);        // set color for drawing
    g.fillRect(50, 50, 20, 30);   // draw a solid rectangle at given points
}
```

... which has various useful methods. Color (American spelling!) is black unless it is changed. What colors are predefined? See documentation…

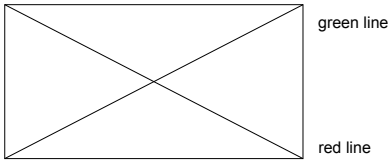
Coordinate system

Graphics are drawn in a coordinate system with the origin at the top left.
For example given a window of size 400 by 300:



Try this

Complete the paint method below (similar to Slide 4) to draw in some window of size 400 by 300 (e.g. Slide 5) the following two lines:

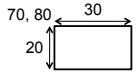



```
public void paint (Graphics g) {



}
```

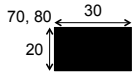

Drawing methods

Graphics objects have useful methods for drawing shapes etc, for example:

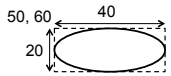
```
drawRect(int x, int y, int width, int height);
drawRect(70, 80, 30, 20);
```




```
fillRect(int x, int y, int width, int height);
fillRect(70, 80, 30, 20);
```




```
drawOval(int x, int y, int width, int height)
drawOval(50, 60, 40, 20);
```

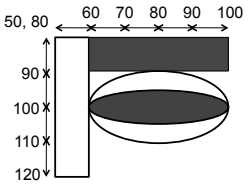



```
drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)
drawArc(10, 10, 60, 30, 20, 90)
```

See LDC p968 – p969!

Try this

Write a paint method that produces the following (without numbers & arrows). The filled box is red, the filled elipse is blue.



Simple design

The following slide shows a complete working program that creates a drawing (the colored cross from Slide 6).

It makes a JFrame (a kind of window) object with a paint method.

This program has a very simple design. It still uses just one (application) class (has main method). In this case we create a single instance object – an instance of the application class.

It is more usual in OO programs to create instances of other "support classes" rather than the application class as in this design.

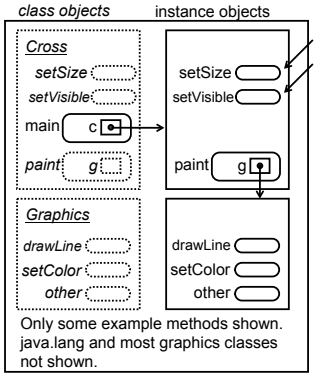
Programs that you use in Lab 5 have this design.

```
/* Anthony Robins, simple working graphics demo */

import javax.swing.*;  // import graphics classes
import java.awt.*;

public class Cross extends JFrame {

  public static void main( String[ ] args ) {
    Cross c = new Cross();
    c.setSize( 400, 300 );
    c.setVisible( true );
  }

  public void paint ( Graphics g ) {
    g.setColor(Color.red);           // set colour for drawing
    g.drawLine (0, 0, 400, 300);  // draw red line  \
    g.setColor(Color.green);         // set colour for drawing
    g.drawLine (400, 0, 0, 300);  // draw green line /
  }

}
```

This class **extends** (is a kind of) JFrame (later!). JFrame is a library class that implements a kind of visible window.

Make an instance of Cross (a JFrame window), set the size, and make it visible.

The paint method is called automatically when the window is drawn (and passed a reference to a Graphics object).



Only some example methods shown. java.lang and most graphics classes not shown.

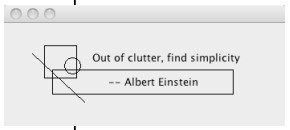
These methods (and many others) are **inherited** from JFrame because Cross **extends** JFrame (later!).

Try this

What are some of the other methods that could be added to the Graphics object in the model?

Einstein example


```
/* Anthony Robins, similar to next lecture examples*/
import javax.swing.*;   import java.awt.*;

public class Einstein extends JFrame {

  public static void main( String[ ] args ) {
    Einstein e = new Einstein();
    e.setSize( 350, 150 );
    e.setVisible( true );
  }

 public void paint(Graphics g) {
   g.drawRect(50,50,40,40);      // square
   g.drawRect(60, 80, 225, 30);  // rectangle
   g.drawOval(75, 65, 20, 20);   // circle
   g.drawLine(35, 60, 100, 120); // line
   g.drawString("Out of clutter, find simplicity", 110, 70);
   g.drawString("-- Albert Einstein", 130, 100);
  }
}
```



Try this

For the Einstein program:

How would you change the code so that the circle is solid / filled?

What are the coordinates of the top right hand corner of the square?

What is the function of the drawString method?

How can you double the height of the rectangle?

What code could be added where to make (only) the text blue?

A brief history of programming languages

Pre 1950's:

- Programs were "machine codes" based very closely on the binary hardware of the computers. Programmers used machine codes, console buttons and switches, even a soldering iron (ENIAC)!

Counting to 10 in machine code

```
23fc 0000 0001 0000 0040
0cb9 0000 000a 0000 0040
6e0c
06b9 0000 0001 0000 0040
60e8
```

Counting to 10 in assembly language

```
        movl    #0x1,n
compare
        cmpl    #0xa,n
        bgt     end_of_loop
        addl    #0x1,n
        bra     compare
end_of_loop
```


1950's:

- Use of assembly language - still tied to specific machine.
- Development of "autocodes" and FORTRAN.

1960's:

- Growth of machine independent "high level" languages, e.g. BASIC, Lisp, Algol 60, Cobol. Most "serious programming" still done in assembly language (slower to write, faster to run!).

1970's:

- "Structured programming" methods to increase program correctness.
- Further development of high level structured languages e.g. Pascal, **C**, Algol 68.

Counting to 10 in Pascal:

```
For n := 1 to 10 do
```


1980's

- Large projects. Focus on reducing the complexity of the task. Program management systems, e.g. Make, APSE, Project Builder.
- New "paradigms" (styles) of programming - object-oriented, functional.
- New languages, e.g. Ada, Modula-2, **C++**.

1990's - 2000's:

- 4000+ programming languages, new ones at a rate of 1 - 4 a month...
- Languages for using parallel and distributed computers?
- Impact of the internet and the Web - exciting times! HTML, XML, network portable applications and "applets", **Java**.

Two ways of classifying languages

By Generation (LDC p11):

- 1GL – First **G**eneration **L**anguage – machine languages
- 2GL – assembly languages
- 3GL – "high level" languages as noted above, general purpose.
- 4GL – application languages (especially for databases)
- 5GL – AI techniques, inference languages
- 6GL – neural networks

By Paradigm (style):

- imperative – describe the operations to perform on data
- object-oriented – operations + data = objects, objects interact
- functional – specify the desired result using functions
- logic – specify properties of the problem, the system solves it

– Java is an object-oriented 3GL –

Objects1 and special methods

COMP160 Lecture 6
Anthony Robins

- Classes and objects
- Data fields and visibility
- Accessors and mutators
- Constructors

See second notes document for this lecture.

Reading: LDC: Sections 3.1, 1.5.
LabBook: "Object-Oriented Design"

Classes and objects

Class objects: one automatically created for every class (including library classes).

Instance objects: any number of instances may be created (from the classes / class objects). For some library classes some instances are created automatically.

So far we have written simple programs with just an **application class** (**static** data fields and methods).

From now on we'll be making instance objects from our own **support classes** (non static data fields and methods – later!).

Classes are like templates / descriptions of objects that can be created.

We can think of objects as **agents** having **state** (values of the data fields) and **behaviour** (the functionality of the methods). We want state to be always **consistent**, with data **encapsulated** (protected by methods – the **interface**). See reading…

Constructing an object

This example creates an instance of the library class JFrame (must be **imported**). Nothing is done with the object, so the program has no output.

```
/* Anthony, April 2012, JDK 1.6, Create a
   JFrame object from JFrame class */

import javax.swing.JFrame; // import class

public class Example0 {

   public static void main(String [ ] args) {
      JFrame obj1 = new JFrame();
   }
}
```

call constructor

class objects instance objects

Example0
main
obj1

JFrame
this class object has various data fields and methods

this object has data fields and methods from the class object

various objects from java.lang not shown

Declares a local variable obj1 (of type JFrame) and initialises it with a reference to an instance object created using a constructor (Slide 13) for the class.

Example 1 (see Second notes)

In this example methods are called on obj1 (the JFrame instance object) to draw a representation of the object (frame / window) on the screen.

JFrame is one of the many classes in java.awt and javax.swing that can be drawn as GUI elements – Lecture 5.

Usually we can't "see" the objects in a program - but this graphical example is a good way to introduce the concept of instance objects…

Example 2 (see Second notes)

In this example we write a class MyFrame which **extends** (later) JFrame, and create an instance of it. MyFrame has whatever data fields and methods class JFrame has (including setSize and setVisible) as well as any that it declares itself.

MyFrame has a paint method, which gets called automatically (Lecture 5).

This is our first program where
we write a support class (as well as the usual application class).
Put each class in its own file, and all the files in the same directory / folder.

Compare this version of Einstein with the one in Lecture 5.

Example 3 (see Second notes)

This example is similar, but we create two instances of MyFrame objects. We can create as many instance objects as we like!

In this case both instance objects are drawn in the same location on the screen, so we have to move one aside to see the other. We could fix this by using another method that MyFrame has (inherited from JFrame like setSize and setVisible) - setLocation. For example, calling:

```
obj2.setLocation(100, 200);
```

would move the frame representing obj2 to the specified x,y coordinates on the screen.

Example 4 (see Second notes)

This example doesn't involve graphics. We write our own support class, Book. We can't see Book objects on screen, but they exist as objects within the program! Most of the programs we write early in COMP160 are like this (non graphics).

Book has two data fields, and methods which take inputs and set the values of the data fields. It also has a method which prints out the information stored in the data fields.

Try this

Write a new main method for Example4 that creates two Book instances, and generates the output below. Sketch a model of the program…

Output:

War and Peace by Leo Tolstoy
The Hobbit by John Tolkien

Data fields and visibility

Variables can be **local** (used only in the method where they are declared) or **data fields**. Data fields can be used by all methods in the class.

Data fields and methods are called the **members** of a class.

There are ways of controlling the **visibility** of members outside the class. The basic rule, **private** members cannot be used outside the class, **public** members can.

E.g. outside the class can get and set **public** data field x. Both of these operations would fail with **private** data field y:
"Error: y has private access in DataFun"

```
public class DataFun {

   public int x = 1;
   private int y = 2;

   // x and y can be used by
   // all methods in DataFun - but
   // ONLY x can be used outside

}
```

In a method in some other class:

```
DataFun df = new DataFun();
// can get the data field x
System.out.println( df.x );
// can set the data field x
df.x = 20;
```

Accessors and mutators

It is good OO design to make data fields private and use methods to get and set. These are ordinary methods that do tasks that are so common that they're called:

accessors / get / getters — to get (return) the value in a data field
mutators / modifiers / set / setters — to set (change) the value in a data field

Mutator

```
public void name(type inputName) {
    dataFieldName = inputName;
}
```

name is usually setDataFieldName
type is usually the type of the data field (e.g. **int**).

Accessor

```
public type name() {
    return dataFieldName;
}
```

name is usually getDataFieldName
type is usually the type of the data field (e.g. **char**).

They may do more than simply get and set values, see examples next slides…


```
public class CurrentTemperature {
   // both must represent the same temperature
   private double celsius = 0.0;
   private double fahrenheit = 32.0;

   public void setCelsius( double inC ) {
      celsius = inC;
      fahrenheit = celsius * 1.8 + 32.0;
   }

   public void setFahrenheit( double inF ) {
      fahrenheit = inF;
      celsius = (fahrenheit - 32.0) * (5.0 / 9.0);
   }

   public double getCelsius() {
      return celsius;
   }

   public double getFahrenheit() {
      return fahrenheit;
   }
}
```

This class holds the same temperature value expressed in both Celsius and Fahrenheit. Mutators can enforce the fact that if either value changes the other is updated too.

Direct access (Slide 9) would allow the programmer to update a single field and forget to update the other (inconsistent state).

Using mutators supports **encapsulation** and allows you to enforce **consistency** in object's state.

See OO design readings.

A related idea for making sure the data we request from an object is always correct is not to store any "derived" values, but calculate them as they are needed. For example...

Try this

Write a class Rectangle with two int data fields, length and width.

Write mutator methods to set length & width.

Write an "accessor" method that returns the area of the rectangle, calculated from length and width.

Note that we never store the area explicitly, it might become out of date if length or width change. Instead, we calculate it as needed!

Constructors

Accessors and mutators are perfectly ordinary methods. Constructors are not! We've already used them, time to look at what they are...

Constructors are special methods that **must** be called, and can **only** be called, in the process of creating an **instance** object.

Constructors can call other methods on the object (class or instance), but other methods cannot call constructors except in the process of creating an object (usually with the keyword **new**).

The default constructor

We have been using default constructors already! E.g. Example 4 (2nd Handout):

```
Book book1 = new Book();
```

type of data (name of class) | name of variable | keyword to create | constructor method call (name of class / subclass)

Where did the constructor method come from (we didn't write it)? If we don't write a constructor Java supplies one (the **default constructor**, taking no input parameters) automatically. It leaves the states of all data fields at zero values (0.0, false, null…).

We can write a constructor that does the same job as the default constructor, e.g. for Book, as follows:

```
public Book() { }
```


Writing constructors

A constructor **may** set the values of data fields, and may have parameters / inputs which specify what those values should be. See the example next 2 slides...

Constructors:

- always have the same name as the class
- may have different numbers and kinds of parameters / inputs
- do **not** specify a return data type **or** void in the header

For example, there are two constructors for Point (next slide):

- one accepts inputs that specify values for x and y, and sets the data fields,
- the other does the same job as the default constructor, it takes no inputs and leaves all data fields at zero values.

(Why bother including it? Because if we declare **any** constructors Java no longer supplies an automatic default, so if we want to have a constructor that works like the default we must include an empty constructor like this. See also constructor chaining, Lec 21).


```
/* Anthony, Aug 2012, JDK 1.6
   Demonstration of constructors. */

public class PointApp {

   public static void main (String[ ] args) {
      Point point1 = new Point();  // default
      Point point2 = new Point( 5, 6 );
      Point point3 = new Point( 9, 2 );
      System.out.println( point1.write() );
      System.out.println( point2.write());
      System.out.println( point3.write() );
   }

} // PointApp
```

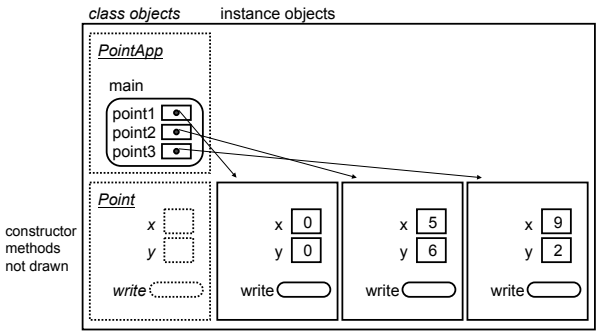
Output:

```
A point 0 0
A point 5 6
A point 9 2
```

```
/* Anthony, Aug 2012, JDK 1.6
   Support class for PointApp */

public class Point {

   private int x, y;

   // equivalent to default constructor
   public Point() { }

   // constructor
   public Point(int inx, int iny) {
      x = inx;
      y = iny;
   }

   // a kind of accessor
   public String write() {
      return "A point " + x + " " + y;
   }

} // Point
```



```
Point point1 = new Point();  // default
Point point2 = new Point( 5, 6 );
Point point3 = new Point( 9, 2 );
```



Constructors let us create objects which are instances of the same class, but in different states (right from the moment they are created). They are a powerful and flexible way of creating objects / "setting up" parts of a program (Lab 13).

We have already seen examples in earlier lectures, for example:

```
String s = new String("Hello");
Scanner sc = new Scanner( System.in );
```

Try this

Write out a class Shape with three integer data fields called length, width and height.

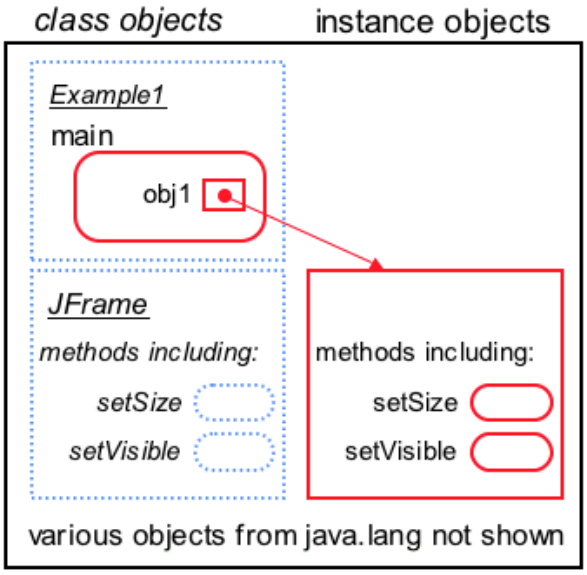
Write a constructor that takes one input and sets all three data fields to that value, and a constructor which takes three different inputs and assigns them to the three data fields.

COMP160 Lecture 6 Second notes

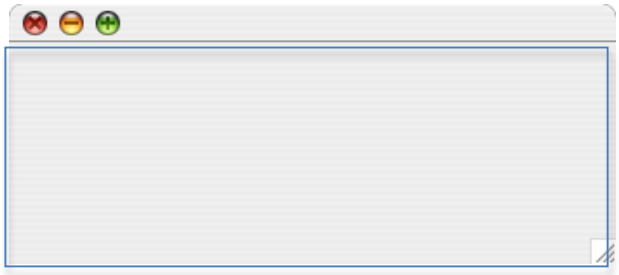
Example 1 (Construct a single frame object and draw on screen)

```
/*  Anthony, April 2012, JDK 1.6,  Create a JFrame object    */

import javax.swing.JFrame;  // import the JFrame class

public class Example1 {

   public static void main(String [ ] args) {
      JFrame obj1 = new JFrame(); // create an instance of a JFrame object
      obj1.setSize(350, 150);          // call a method to set the size
      obj1.setVisible(true);           // call a method to make it visible
   }

}
```

In file Example1.java



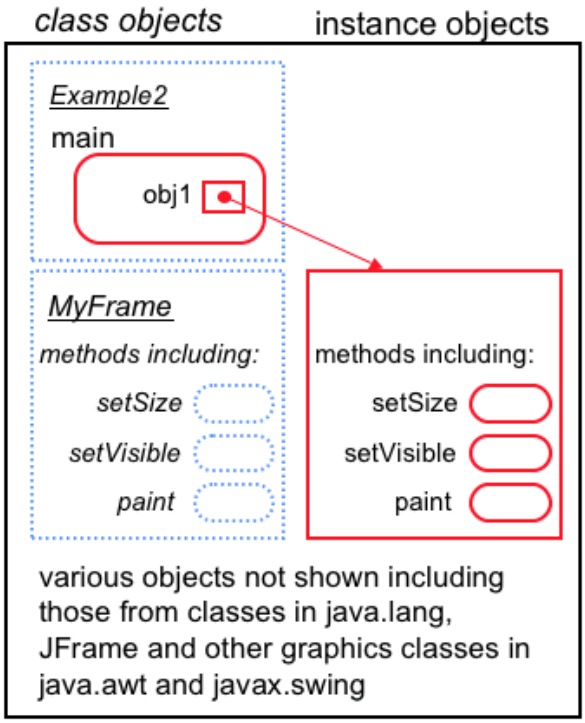
Output:



Example 2 (Draw graphics in a frame, loosely based on LDC p851)

```
/*  Anthony, April 2012, JDK 1.6
    Create a MyFrame object  */

public class Example2 {

  public static void main(String [ ] args) {
     MyFrame obj1 = new MyFrame();
     obj1.setSize(350, 150);
     obj1.setVisible(true);
   }

}
```

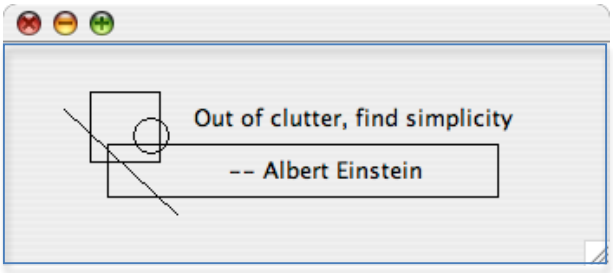
In file Example2.java




```
/*  Anthony, April 2012, JDK 1.6
    A support class for Examples 2 and 3  */

// import graphics classes including JFrame
import javax.swing.*;
import java.awt.*;

// make our own version of a JFrame class
// with our own paint method...
public class MyFrame extends JFrame {

  public void paint(Graphics g) {
    g.drawRect(50,50,40,40);       // square
    g.drawRect(60, 80, 225, 30);  // rectangle
    g.drawOval(75, 65, 20, 20);    // circle
    g.drawLine(35, 60, 100, 120); // line
    g.drawString("Out of clutter, find simplicity", 110, 70);
    g.drawString("-- Albert Einstein", 130, 100);
  }

}
```

In file MyFrame.java

Output:



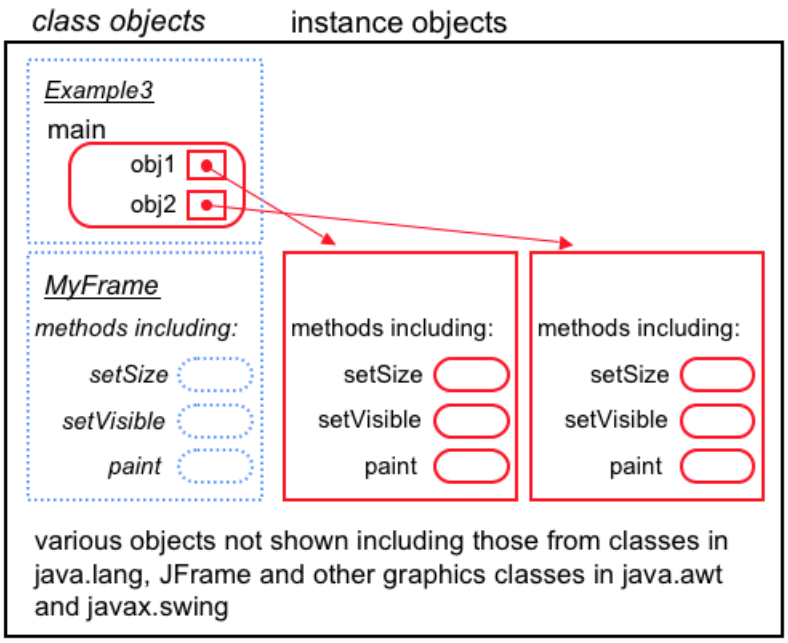
Example 3 (As for Example 2, but two objects)

```
/*  Anthony, April 2012, JDK 1.6
    Create two MyFrame objects  */

public class Example3 {

  public static void main(String [ ] args) {
     MyFrame obj1 = new MyFrame();
     obj1.setSize(350, 150);
     obj1.setVisible(true);
     MyFrame obj2 = new MyFrame();
     obj2.setSize(350, 150);
     obj2.setVisible(true);
   }

}
```

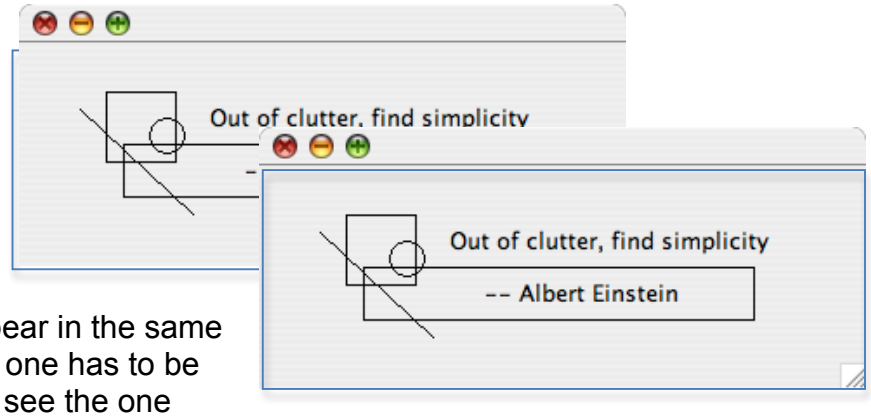
In file Example3.java

MyFrame code is exactly as for Example2

In file MyFrame.java



Output:

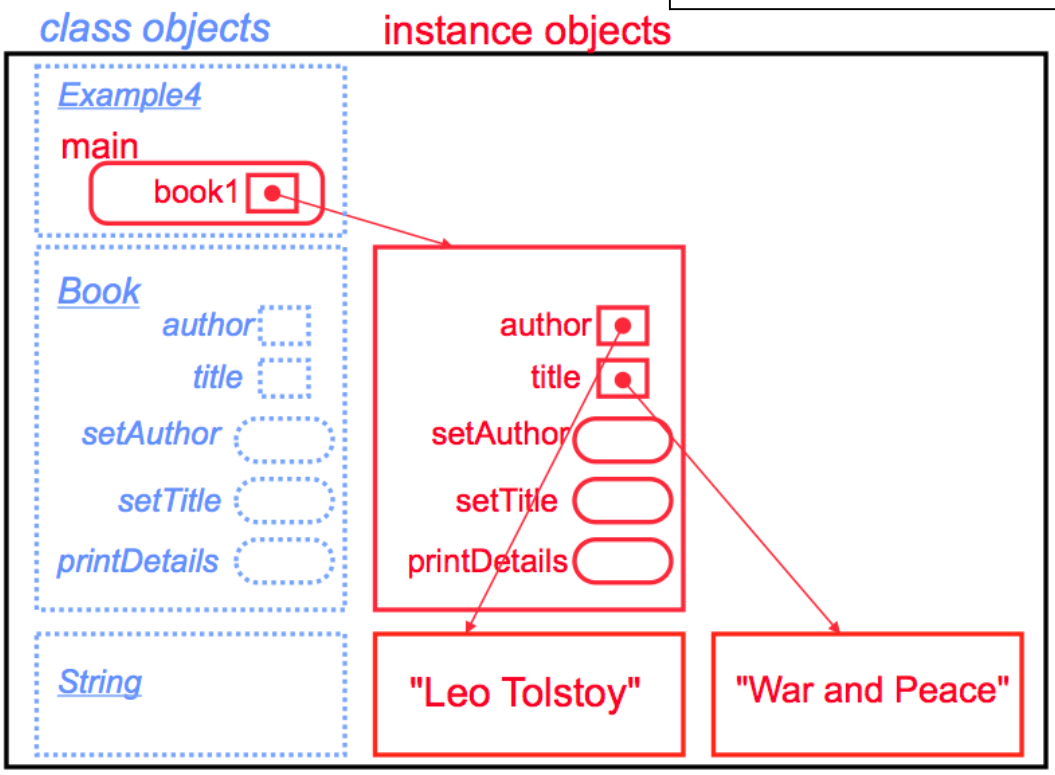


The frames appear in the same spot, so the top one has to be moved aside to see the one underneath.

Example 4 (A non graphical example, construct a single object from a support class)

```
/*  Anthony, April 2012, JDK 1.6
    Create and use a Book object. */

public class Example4 {

  public static void main(String [ ] args) {
     Book book1 = new Book();
     book1.setTitle("War and Peace");
     book1.setAuthor("Leo Tolstoy");
     book1.printDetails();
   }

}
```

```
/*  Anthony, April 2012, JDK 1.6
    A support class for Example 4  */

public class Book {

   private String title;
   private String author;

   public void setTitle(String t) {
      title = t;
   }

   public void setAuthor(String a) {
      author = a;
   }

   public void printDetails() {
      System.out.println( title + " by " + author );
   }

}
```



Output:

War and Peace by Leo Tolstoy

Objects 2. Strings.

COMP160 Lecture 7
Anthony Robins

- Constructors (continued)
- Class and instance (static)
- toString
- References and aliases
- Reference data types
- Strings

LDC note:

On Slides 11 & 12 we look at aliases to objects, and how the state of an object can be changed using either alias.

LDC 3.1 section on "Aliases" makes the same point, but unfortunately they use Strings as their example object. Strings can behave differently from other objects in some circumstances because they are "immutable" – see the Lab Book reading Reference Types (Strings).

So on this point LDC is correct in general but not necessarily correct about Strings.

Reading: LDC: 3.1, 3.2, 3.3
LabBook: Object-oriented design.

LDC 3e to (2e) p40 (p65)

Constructors (continued)

Recall (last lecture) that a constructor is a special method which is called whenever a new instance object is made from a class / class object – usually with the keyword **new**.

The constructor has the same name as the class and no return type. A default constructor is supplied for every class. If we write any of our own constructors the default is not supplied, so we should replace it. For example, for a class called Point the following is a replacement for the default constructor:

```
public Point() { }
```


Which constructor?

All the constructors for a class have the same name. For example, a class Point with data fields x and y might have the constructors:

```
public Point( int input ) {
      x = y = input;
 }
public Point( int inx, int iny ) {
      x = inx;
      y = iny;
}
```

How do we know which one we are calling when we say for example:

```
Point p = new Point( 2, 9 );
```

The constructor which is called is the one where the **formal parameters** match (right number, type and order) the **actual parameters**. In this example, the second one above! See more on method overloading later (Chapter 5).

Class and instance (static)

One **class** object is created for every class (automatically), and various **instance** objects may also be created (from classes / class objects with **constructors**).

A **member** of class (data field or method) can be declared: **static**, then it is part of the **class** object (a **class member**); or without static, then it is part of any **instance** objects from the class (an **instance member**).

Static members are created and usable before the program starts to run (like the main method, Math.PI, Math.sqrt()).

Members of the application class are usually static, support classes usually not.

The program can only use **static** class members and (all) instance members .

These are the solid bits of my models, not the *dashed*

So:

(1) We can use class members in class objects: ClassName.memberName

(2) We can't use instance members in class objects (they don't really exist!)

(3) We can use instance members in instance objects: variableName.memberName

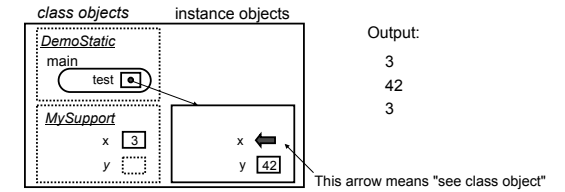
(4) We can seem to use class members in instance objects: variableName.memberName but we are really using the member in the underlying class object instead.

See Cases 1 - 4 on the next slide in an example using data fields…


```
/*  Anthony, April 2012, JDK 1.6, uses a support class
with instance and class data fields  */
public class DemoStatic {
  public static void main(String [ ] args) {
    MySupport test = new MySupport();
    System.out.println( MySupport.x );    // Case 1
    // System.out.println( MySupport.y ); // Case 2 fails
    System.out.println( test.y );         // Case 3
    System.out.println( test.x );         // Case 4
  }
}
```

```
public class MySupport {
  // a class member
  public static int x = 3;
  // an instance member
  public int y = 42;
}
```

fails with error:
"non-static variable y cannot be referenced from a static context"



Output:
3
42
3

Accessing members (and dot notation)

To access class (static) members use ClassName.memberName (if they are public) for example:

Math.PI	*a data field in the class object Math (Lec 5)*
Math.sqrt()	*a method in the class object Math (Lec 5)*
MySupport.x	*a data field in the class object MySupport (Slide 6)*
MySupport.hello()	*a method in the class object MySupport (Slide 6 - if class MySupport had a static method called hello)*

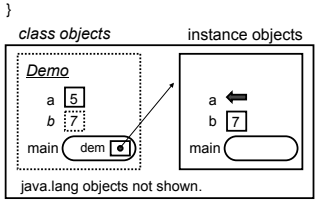
"dot notation"

To access instance members use variableName.memberName for example:

test.y	*a data field in the instance object referred to by variable test (Slide 6)*
test.greet()	*a method in the instance object referred to by variable test (Slide 6 - if class MySupport had a non static method called greet)*

The naming conventions tell you what kind of object is involved!
ClassName = initial capital = class object
variableName = initial small = instance object


```
/* Anthony, What exists – static demo. */
public class Demo {
    public static int a = 5;
    public int  b = 7;
    public static void main(String[ ] args) {
        System.out.println( a );
        System.out.println( b );
        Demo dem = new Demo();
        System.out.println (dem.b );
    }
}
```



Try this

Which of the println's will fail, and why?

toString

Every instance object has a toString method that prints out information about it:

In main:

```
AClass a = new AClass();
System.out.println( a.toString() );
```

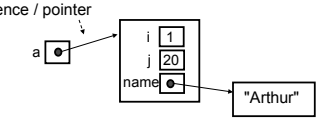
Output:

```
AClass@8fbbaf
```

This is a string that represents a reference / pointer (an arrow in my models). In general:

Type@address in memory

```
public class AClass {
    private int i = 1;
    private int j = 20;
    private String name = "Arthur";
}
```



Java allows a special shortcut for the toString method - it is called automatically if we just use the simple variable name:

```
System.out.println( a );   // same as the println line above
```

You can replace (override) the inherited toString and define your own toString method that returns any string you like! This is useful for printing out a summary or overview of an object.

In main:

```
AClass a = new AClass();
System.out.println( a );
```

Output:

```
Hello, I am an object
and my summary is
Arthur 1
```

```
public class AClass {
    private int i = 1;
    private int j = 20;
    private String name = "Arthur";

    public String toString( ) {
        return  "Hello, I am an object\n" +
        "and my summary is\n" + name + " " + i ;
    }

}
```

\n escape characters LDC p40

References and aliases

References are not well covered in LDC (see note Slide 1, see Lec 14).

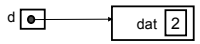
An object is distinct and separate from a reference to the object.

(I draw references as arrows, but they are really an integer value specifying an address in memory, see toString).

An (instance) object might be referenced by one variable in the program, or multiple variables (these are called **aliases**), or by no variables (in which case it will soon be eaten by the **garbage collector** to free up memory). See next slide.

Assume a support class Demo with a constructor (takes an input **int** to set a **public** data field called dat) and a toString() that returns "dat is " + dat. A very simple model:

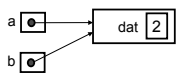
Demo d = **new** Demo(2);




```
Demo a = new Demo(2);
Demo b = a;
System.out.println(a + ", " + b);
```

dat is 2, dat is 2

These examples call toString on a and b



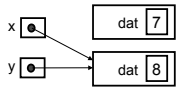
Key concept: a and b are aliases

```
a.dat = 5;  // public dat set to 5
System.out.println(a + ", " + b);
```

dat is 5, dat is 5

Direct access (Lecture 6) to change the state of the object referred to by aliases a and b (change model above dat is 5)

```
Demo x = new Demo(7);
Demo y = new Demo(8);
x = y;
```



Top object is now "lost" and will be cleared by the garbage collector.

The value stored in each variable is a **reference** to an object NOT the object itself!

Reference data types

Every (instance) object is made from a class, the class is the type of the object (as **int** is the type of 3).

Java has **primitive types** (like **int**), and **reference types** (classes, including library classes). Classes are flexible, so if we want to represent things in the world we can define a class / data type for the purpose.

OO theory: Instances of the same class have the same methods (**behaviours**), and have the same data fields, but can have different values for the data fields (**states**). Lab Book reading…

For example, in a database about music, we might want to represent songs:

```
public class Song {
    String title = "Least complicated";
    String artist = "Indigo Girls";
    int time = 252;       // playing time in seconds
    // other data fields and methods
}
```

Each song can be represented by an instance of this class (each instance is of type Song).

Classes in the libraries define many very useful data types already!

For example, there are classes representing: shapes, colours, and drawing surfaces for creating graphical outputs; fonts and text layouts; "streams" of data for dealing with devices; "events" for representing things like mouse clicks in a GUI; and more...

The ability to define classes gives us an infinite number of possible data types!

Strings

In programming languages "string" generally means any sequence of characters, such as "abc1", "#%23a" or "It's a string !".

In Java strings are implemented using String - a class in java.lang. We can make instances of string objects as for any other class:

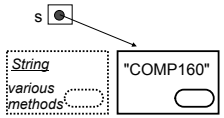
String s = **new** String("COMP160");

or use a shortcut (Java has shortcuts for classes Array and String):

String s = "COMP160";

Both create a new instance (of the class String) and store a reference to the object in the variable s.

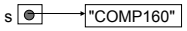
Usual partial model



+ joins (concatenates) strings together and converts data of any other type in to a string as it does this:

"Hello " + "All" + 2 is "Hello All2"

Very simple model



String methods

The class String has (and so String objects have) many useful methods. Assume:

```
String hi = "Hello All!";
            0123456789 – positions in the string (computer scientists number from 0)
```

length() returns the length of a string:

hi.length() is 10 "Anthony".length() is 7

literal strings (Lec 4) like this create a normal string object with a "temporary" reference that is not stored in a variable

indexOf() returns the position of a char (character):

hi.indexOf('e') is 1 hi.indexOf('x') is –1 "wibble".indexOf('w') is 0

not found

substring() returns a specified part of a string:

hi.substring(3, 9) is "lo All" "a1#2c".substring(0, 2) is "a1"

from not including

replace() replaces every occurrence of an old char with a new one:

hi.replace('l','X') is "HeXXo AXX!"

charAt() returns the character at the specified position:

hi.charAt(1) is 'e'

There are many more – see LDC 3.2.

Try this

For each of the following, what is the value that results? In each case (except the third) the result is a string and a reference to the new string is stored in a variable.

String dun = "Dunedin"; "Dunedin"

String weather = "Sunny" + dun;

int i = dun.indexOf('e');

String s = dun.substring(1, 5);

String t = dun.substring(0, dun.indexOf('e') + 1);

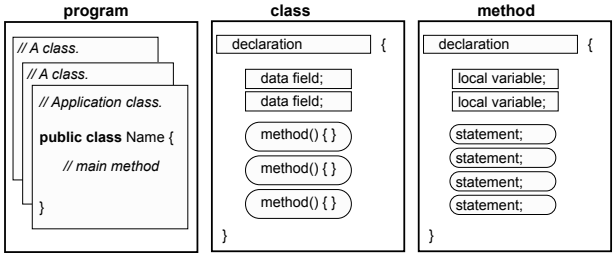
Structured programming, more maths

COMP160 Lecture 8
Anthony Robins

- Program overview
- Case study
- Service and support methods
- More maths
- Formatting
- Wrapper classes

Reading: LDC: 3.4, 3.5, 3.6. 3.8 (later see also 5.3)

Program overview



Program: an application class and any number of other classes (including some library classes).

Class: a declaration and a body (which is a { block } of code containing any number of data fields and any number of methods).

Method: a declaration and a body (a { block } of code containing any number of local variables & any number of statements).

Classes

The **declaration** (header) is the first line, with information about the class such as its:

- visibility (which other classes can see and use this one), e.g. **public**
- name
- if it extends (is a subclass of – later) any other class.

The **data fields** store primitive values or references to objects. The declaration of a data field tells us its visibility (e.g. **private**), its type (e.g. String), its name, and perhaps its initial value. The values of the data fields define the **state** of the object.

The **methods** of a class specify actions / process (usually performed on the data fields of the class) – the **behaviour** of the object. Methods can be in any order in the class.

Static / class members (data fields and methods) are part of class objects, non–static / instance members are part of instance objects (Lec 7).

Methods

The **declaration** (header) is the first line, with information about the method such as its:

- visibility (which other classes can see and use this method), e.g. **public**
- what kind of data it returns (or **void** if it returns nothing)
- name
- what kind of data it takes as inputs / "arguments", in the **( )**

The **local variables** of a method (including its inputs / arguments) store primitive values or references to objects. The declaration of a local variable tells us its type (e.g. String), its name, and perhaps its initial value. (Not visibility, local variables can only be seen / used in the method where they are declared).

Statements are the basic building blocks of the program – executable statements are the single steps of the algorithm in each method. They are usually one per line, and end in ";". The order of the statements is crucial!

Statements

Statements can be:

- **declarations** of classes, variables (data fields, locals) and methods, or
- **executable** statements of various kinds, which can
  - call methods
  - calculate values
  - store values in (assign them to) variables
  - return values at the end of a method
  - and more...

Statements can be grouped together into a **block** using "{" and "}".

- class bodies are blocks
- method bodies are blocks
- there are other ways to use blocks, see selection & repetition Lecs 9 – 12.

Try this

For this example program:

underline any class headers

underline any method headers

oval any data fields

box any local variables

highlight any visibility modifiers

tick ✓ any executable statements

```
/* Anthony, August 2012, JDK 1.6
   Sample program structure */

public class Sample {

   private static String s = "Hello", t = "Bye";

   public static void main (String [ ] args) {
      System.out.println(s + " " + t);
      adder(5);
   }

   public static void adder(int x) {
      int y = 1;
      System.out.println(x + y);
   }

}
```


Case study

This example illustrates many points! The main method constructs one instance of ShoppingList, calls a method on it to set the value of the data field list, and calls a method to print out the information shown (the shopping list one item at a time).

This example only works for a list of three items separated by and ending in " ".

Two statements work on the String stored in data field list:

```
          0123456789…
"Apples Eggs Tea "
```

```
next = list.substring( 0, list.indexOf(' ') ); // store first item in data field next
list  =  list.substring( list.indexOf(' ') + 1 ); // make a new copy of list without first item
```

One input specifies starting point (to the end).


```
/* Holds and prints a three item shopping list */
public class ShoppingList {
  private String list, next;

  // set the list data field (service)
  public void setList(String l) {
    list = l;
  }

  // print the shopping list items (service)
  public void printItems() {
    System.out.println("Initial list: " + list);
    nextItem();
    nextItem();
    nextItem();
  }

  // print first item and remove it from list (support)
  private void nextItem() {
    next = list.substring(0, list.indexOf(' ') ); // store current first item in data field next
    list = list.substring(list.indexOf(' ') + 1 ); // make a new copy of list without first item
    System.out.println("Item: " + next + "  Remaining: " + list);
  }
}
```

```
/* Anthony, August 2012, JDK 1.6 */

public class DemoShoppingList {

  public static void main (String [ ] args) {
    ShoppingList shopList = new ShoppingList();
    shopList.setList("Apples Eggs Tea ");
    shopList.printItems();
  }

}
```

```
Initial list: Apples Eggs Tea
Item: Apples  Remaining: Eggs Tea
Item: Eggs  Remaining: Tea
Item: Tea  Remaining:
```


Service and support methods

These are ideas about program design relating to **structured programming**, an approach prior to OO design (Lec 5). ("Methods" = functions or procedures).

Service methods

Methods which are intended to be called on the object. In other words, they are called by methods elsewhere in the program (not in this class / object) using dot notation. Must be visible (e.g. **public**).

In the class ShoppingList, setList and printItems are service methods. They are called on the instance shopList from the main method (dot notation).

Service methods provide a service (!) to other parts of the program. They often implement the behaviours relating to an object's state / data fields. They may be the ones that spring to mind first as you design a class.

Lots of methods in the libraries supply useful services. For example, the println method (e.g. System.out.println()) is very useful!

Support methods

Methods which are intended to be called by other methods within the class / object (using their simple name). It is good design (Lec 6) to make these methods not visible outside the class (e.g. **private**).

In the class ShoppingList, nextItem is a support method. It is called by another method in the class (printItems) using its simple name: nextItem();

Support methods support (!) other methods in the class. Use them to group together operations that are used frequently. They organise and "modularise" processing.

Without support methods we would duplicate a lot of code, making it:

- hard to read, and
- hard to modify (duplication!)
- especially for lots (500?) of repetitions

Try this

The lower version of printItems uses a support method (Slide 8), the upper version does not. In both cases modify the code so that it would work with a list where items are separated by "-" instead of " ", e.g. "Apples-Eggs-Tea-"

Repetition / loops (Ch 4) will make this much more efficient too - e.g. 500 repetitions with just one line of code!

```
public void printItems() {
    System.out.println("Initial list: " + list);
    next = list.substring(0, list.indexOf(' ') );
    list = list.substring(list.indexOf(' ') + 1 );
    System.out.println("Item: " + next + "  Remaining: " + list);
    next = list.substring(0, list.indexOf(' ') );
    list = list.substring(list.indexOf(' ') + 1 );
    System.out.println("Item: " + next + "  Remaining: " + list);
    next = list.substring(0, list.indexOf(' ') );
    list = list.substring(list.indexOf(' ') + 1 );
    System.out.println("Item: " + next + "  Remaining: " + list);
  }
```

```
public void printItems() {
    System.out.println("Initial list: " + list);
    nextItem();
    nextItem();
    nextItem();
  }

  private void nextItem() {
    next = list.substring(0, list.indexOf(' ') );
    list = list.substring(list.indexOf(' ') + 1 );
    System.out.println("Item: " + next + "  Remaining: " + list);
  }
```


More maths

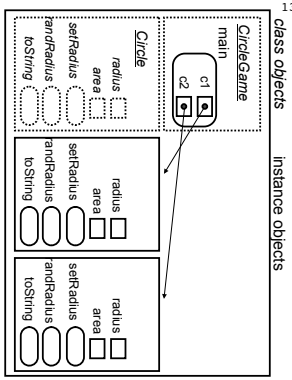
LDC 3.5 discusses the class Math (in java.lang so automatically imported). Math defines useful mathematical:

- constants, which are static data fields in the class object (Lec 4 Slide 9)
- functions, which are static methods in the class object (Lec 4 Slide 9)

Lots of tasks involve random numbers, e.g. games, statistics, scientific modelling. The class Random in the package java.util (utilities) has many methods for generating random numbers in specified ranges (LDC 3.4).

The following example covers much of this. The application class makes instances of the support class, Circle. We can call a method to set the radius of a circle to a specified value, or call another method to set a random radius…


```
/*  Anthony, April 2012, JDK 1.6,
    Demonstrate examples from Math */
public class CircleGame {

  public static void main(String [ ] args) {
    Circle c1 = new Circle();
    c1.setRadius(1.0); // set specified radius
    System.out.println(c1); // calls toString
    Circle c2 = new Circle();
    c2.randRadius();    // set random radius
    System.out.println(c2); // calls toString
  }

}
```



Output:

```
Circle with radius 1.0 and area 3.141592653589793
Circle with radius 0.932781786220709 and area 2.7334427816128684
```

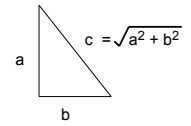


```
/* A class to represent circles, with specified radius or random radius.  Anthony, JDK 1.5 */

import java.util.Random;   // import the class Random used in this class

public class Circle {
  private double radius, area;

  public void setRadius(double r) {  // takes an input value, sets radius, and sets area
    radius = r;
    area = Math.PI * Math.pow(radius, 2);   // PI is a data field and pow is a method in Math
  }

  public void randRadius() {      // takes no inputs value, sets random radius, and sets area
    Random rand = new Random(); // creates an instance of Random called rand
    radius = rand.nextDouble();     // calls a method on rand to return a random double
    area = Math.PI * Math.pow(radius, 2);  // pow raises first input to the power of the 2nd
  }

  public String toString() {        // can be called to return a string describing the circle
    return "Circle with radius " + radius + " and area " + area;
  }
}
```

area of a circle is πr^2

Try this

Assume that we have a readInt method (Lec 4 – returns an integer entered by the user). Complete the method calcHyp so that it uses readInt to read in values for the lengths of two sides (a and b) of a right angle triangle, then calculates and prints the length of the hypotenuse c.



(Note, Math has a method sqrt which returns the square root of its input as a double).

```
public void calcHyp ( ) {
    int a = readInt("Please enter first side:");




}
```


Formatting

By default Java prints out real (double, float) values with lots of decimal places - e.g. Slide 13 - or in scientific notation (Lec 2). This is not always appropriate! LDC 3.6 describe two ways to format real values:

(1) DecimalFormat / NumberFormat

For example, **import** java.text.DecimalFormat. Construct an instance of DecimalFormat. Call methods on the instance to define the format structure, and the format method to return a formatted string.

```
DecimalFormat f1 = new DecimalFormat();
f1.setMaximumFractionDigits(3);
System.out.println(Math.PI);      // print pi in default format
System.out.println( f1.format(Math.PI) );  // in specified format
```

```
3.141592653589793
3.142
```

note rounding!

see also NumberFormat and other examples LDC 3.6

(2) Using the printf method

The printf method takes as input a string specifying the format of the output string, followed by the values to be incorporated into the output string.

This is very much like C and C++, and was only introduced in Java in version 1.5. We won't use printf in COMP160, but see LDC 3.6, and for example:

http://www.cplusplus.com/reference/clibrary/cstdio/printf/

Wrapper classes

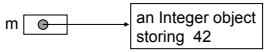
In some languages "everything is an object". In Java we occasionally want to treat primitive values like objects (e.g. to store them in collections of objects - later).

Wrapper classes let us "wrap up" a primitive type and treat it as an object / reference type. The wrapper classes are:

Integer: wraps an int value	Boolean: wraps a boolean value
Double: wraps a double value	Character: wraps a char value

For example:

Integer m = new Integer(42);



See LDC 3.8 for a discussion of wrapper classes and their use, and automatic conversion from a primitive to a wrapped object ("autoboxing"), and from an object to a corresponding primitive ("unboxing").

Booleans, selection 1

COMP160 Lecture 9
Anthony Robins

- Introduction
- Boolean expressions
- Precedence
- Comparing data
- Conditions, selection (if)
- Blocks
- if..else

Reading: LDC: 4.1 – 4.3.

LDC
3e to (2e)
p54 (p78)
p956 (p820)

Introduction

For the next four lectures (Chapter 4) we concentrate on **executable statements** that are used **within methods**.

Most interesting algorithms involve complex orderings of steps (**flow of control**), where we can choose and organise which steps to do next. In Java, like most programming languages:

- choosing is done using boolean expressions to calculate whether conditions are true or false
- organising is done using selection and repetition statements such as: **if**, **switch**, **for**, **while**, and **do**

For this reason we often find boolean expressions in the conditions of selection and repetition statements.

Boolean expressions

We can represent true and false states / conditions using **boolean expressions** (named after logician George Boole).

Recall (Lecture 4) that an expression is a specification of a value. There are different kinds of expression in Java, including: literals; variables; methods that return results; assignments; and calculations involving operators.

A boolean expression is a specification of one of only two possible values, **true** or **false**.

The Boolean data type

Recall (Lecture 2 / Chapter 2) that **boolean** is a primitive data type. It has only two possible values, represented by the literals true and false (which are reserved / keywords in Java).

We can do the same sorts of things with the **boolean** data type that we can do with other types:

```
boolean done = false;          // declare and initialise a boolean variable
boolean alpha, beta, gamma; // declare three boolean variables
done = true;                   // assign a value to a boolean variable
public boolean aMethod() {  // declare a method returning a boolean value
public void alsoMethod(boolean bigInput);    // a boolean parameter
```


Relational / comparison operators

Relational / comparison operators (introduced in Lecture 2) allow us to test values for equality, or compare them. They can be used to form boolean expressions.

<	less than	>	greater than
<=	less than or equal	>=	greater than or equal
==	equal	!=	not equal

only == and != can be used with reference types

For example, given:

```
int x = 5, y = 10;
char aChar = 'Z';
```

all of the following expressions produce the result true:

```
x < y          x == 5          (x + 1) <= (y * 5)
x != 2         20 >= x * 2     aChar > 'A'
```


Logical / boolean operators

As well as the general relational operators there are logical operators which can be used to form boolean expressions:

```
&&   // "and", true if both operands are true
||   // "or", true if either operand is true
!    // "not", reverses truth (not true is false, not false is true)
```

One way of expressing what these operators do is as "truth tables":

and			
true	&&	true	is true
true	&&	false	is false
false	&&	true	is false
false	&&	false	is false

or			
true	||	true	is true
true	||	false	is true
false	||	true	is true
false	||	false	is false

not	
! true	is false
! false	is true

For example, given:

```
int x = 5, y = 10;
boolean tim = true, fred = false;
```

then:

tim && fred	is false	x == 5 && y < 5	is false
tim || fred	is true	x == 5 || y > 5	is true
!tim	is false	!(x < y)	is false

Try this

Given the values of x, y, tim and fred on the previous slide, what does the **boolean** variable result get set to in each case? Which ones are errors / illegal statements?

```
result = (x + 2) < y;
result = x || tim;
result = y != 4;
result = (x == 5) || (y == 8);
result =  (x == 5) && (y == 8);
result = y < fred;
```

&& has higher precedence than || (Slide 10). What do these expressions evaluate to?

```
true || true && false
(true || true) && false
```


An example

We can do quite complex calculations with boolean expressions!

For example, say we have an integer value i, and we want to know if it is an even number between 1 and 500 (inclusive):

```
boolean valid, even, range;
int i = 260;
even = (i % 2) == 0;                  // true if i divides by 2 with no remainder
range = (i >= 1) && (i <= 500);       // true if i is 1 or greater and 500 or less
valid = even && range;                // true if both even and range are true
System.out.println("Validity of " + i + " is:  " + valid);       // print result
```

If i is for example 260 as shown, this will print:

```
Validity of 260 is:  true
```

Note: in Python we could write
1 <= i <= 500
not in Java – must use &&

Precedence

Relational and logical operators have various levels of precedence like the arithmetic operators. LDC have tables showing precedence on pages 54 & 956. For the operators that we know so far:

method call	highest
!, – (unary), + (unary)	
new, type cast	
*, /, %	
+, –	
<, <=, >, >=	
==, !=	
&&	
||	
=, +=, –=, /=, %=	lowest

Equal precedence operators are treated as left associative.

Brackets can be used optionally (e.g. previous slides) or to clarify or force a certain order of evaluation.

Comparing data

LDC Section 4.3 collects various topics together under the general heading of "comparing data". Lectures have / will address these topics in various places:

Comparing floats (doubles) was mentioned in the discussion of Lecture 2. Real valued arithmetic is not precise!

```
System.out.println( 10 * 1.12 );    // prints  11.200000000000001
```

Comparing characters, see the Unicode and ASCII character sets discussed in Lecture 2. See also LDC Appendix C.

Comparing objects will be discussed extensively in later lectures. See also the Lab book reading Reference Types.

Conditions, selection (if)

Boolean expressions often occur in the conditions of selection & control statements. For example, the **if** statement (more to come in the next lectures):

The **if** statement has the form:

if (condition) statement;

where the condition is a boolean expression. If the condition is (expression evaluates to):

- true : the following statement is executed
- false : the folllowing statement is not executed.

Choosing and selecting, what to do next. Flow of control **within methods** – Slide 2.

We can now choose whether or not to execute a statement!

For example (assuming a readInt method as usual – Lec 4):

```
int i;
i = readInt("Enter an integer: ");
if (i < 10) System.out.println("input is small");
System.out.println("next statement");
```

First run

```
Enter an integer:
2
input is small
next statement
```

Second run

```
Enter an integer:
30
next statement
```

The statement does not have to be on the same line - this version is the same…

```
int i;
i = readInt("Enter an integer: ");
if (i < 10)
    System.out.println("input is small");
System.out.println("next statement");
```


Try this

The statement below declares a variable i, and initialises it with a value read in from the user. Add further statements so that if i is less than or equal to10 then its value is doubled, and then the value if i (whatever it is) is printed.

```
int i = readInt("Please enter an integer: ");
```


Blocks (compound statements)

We can group statements together into a **block** or **compound statement** (some people make a distinction between these terms, LDC do not) using braces {…} .

We already know that the body of a class or method is a block, starting with { and ending with }.

Now we will learn other useful places to use blocks...

We can put a block anywhere we can put a single statement!

If with a block

For example, we can use a block instead of a single statement in an **if** statement. The block is treated as a single unit, either we do all the statements in the block, or none of them.

```
if (condition) {
    statements;
}
```

```
if (a + b < limit) {    // note formatting, indent the contents of a block
    a  = 0;
    b = 10;
    System.out.println("a is set to zero, b is set to ten");
}
```

In this example, if the condition is true (sum of a and b is less than limit) then we go on and execute the three statements in the block. If the condition is false we do none of the statements in the block.

if..else

The **if** statement may contain an **else** clause...

```
if (condition)
    statement;
else
    statement;
```

OR

```
if (condition) {
    statements;
} else {
    statements;
}
```

If the condition is
- true : the first statement or block is executed
- false : the second statement or block is executed

```
int i;
i = readInt("Enter an integer: ");
if (i < 10)
    System.out.println("input is small");
else
    System.out.println("input is large");
System.out.println("next statement");
```

```
Enter an integer:
4
input is small
next statement
```

```
Enter an integer:
99
input is large
next statement
```


This method takes two double values as input and returns the larger value as a result.

```
public double max (double d1, double d2) {
    if (d1 > d2)
        return d1;
    else
        return d2;
}
```

Try this

Write a method called test that takes two ints as input. If they are the same print "same" and return the sum. If they are different print "different" and return the product. You will need to use if..else and blocks!

Selection

COMP160 Lecture 10
Anthony Robins

- Introduction
- if..else continued
- Nested and multiple if..else
- Switch
- Short-circuit evaluation
- Scope and design

Reading: LDC: 4.2, 4.4

LDC
3e to (2e)
p53 (p78)

Introduction

See Lecture 9 Slide 2. We are looking at **flow of control** where we can choose and organise which steps to do next.

Today we look at **selection** statements: if (following from last lecture) and switch.

Once again I assume a readInt method (Lec 4).

if..else continued

if and **if..else** statements can be set out in various ways. Some examples:

Enclosing a single statement with { } is optional. The first three examples are all equivalent.

Recall that we can use a {block} of statements anywhere that we can use a single statement

```
if (condition) statement;

if (condition)
    statement;

if (condition) {
    statement;
}

if (condition) {
    statement1;
    statement2;
    statementN;
}
```

Only the first following statement / block is part of the "if", which leads to common errors if we forget the {braces}…

The programmer meant this: →

```
if (i < 10) {
    System.out.println("i is small, add ten");
    i += 10;
    System.out.println( i );
}
System.out.println("Next statement");
```

but wrote this: →

```
if (i < 10)
    System.out.println("i is small, add ten");
    i += 10;
    System.out.println( i );
System.out.println("Next statement");
```

which is effectively this: →

```
if (i < 10) {
    System.out.println("i is small, add ten");
}
i += 10;
System.out.println( i );
System.out.println("Next statement");
```

The indentation makes no difference!

Where the **if..else** clauses contain single statements the {braces} are optional – these two are equivalent.

```
int i;
i = readInt("Enter an integer: ");
if (i < 10)
    System.out.println("input is small");
else
    System.out.println("input is large");
System.out.println("next statement");
```

```
int i;
i = readInt("Enter an integer: ");
if (i < 10) {
    System.out.println("input is small");
} else {
    System.out.println("input is large");
}
System.out.println("next statement");
```

Multiple statements must be grouped together into a block (next slide)…


```
int i;
i = readInt("Enter an int in the range 1 to 10");
if (i >= 1 && i <= 10) {
    System.out.println("That is in the range");
    System.out.println("well done!");
} else {
    System.out.println("That is not in the range");
    System.out.println("thanks anyway.");
}
```

Enter an int in the range 1 to 10 *1* That is in the range well done!	Enter an int in the range 1 to 10 *42* That is not in the range thanks anyway.

Another example (assume a readChar method that reads input characters):

```
char c;
int vowelCount = 0, consonantCount = 0;

c = readChar("Enter a lowercase letter: ");
if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') {
    System.out.println("That was a vowel");
    vowelCount++;
} else {
    System.out.println("That was a consonant");
    consonantCount++;
}
System.out.println("Vowels so far: " + vowelCount
        +"\nConsonants so far: " + consonantCount);
```

```
Enter a lowercase letter:
k
That was a consonant
Vowels so far: 0
Consonants so far: 1
```

```
Enter a lowercase letter:
e
That was a vowel
Vowels so far: 1
Consonants so far: 0
```

In English the letters
a, e, i, o, u
are considered vowels,
all others are consonants

Try this

The statements below declare variables x and y, and initialise them with values read in from the user. Add further statements so that if x is less than 10 and y is less than 10, then 2 is added to x and 3 is added to y. Otherwise, 2 is subtracted from x and 3 is subtracted from y. Then print the values of x and y (whatever they are).

```
int x = readInt("Please enter an integer: ");
int y = readInt("Please enter an integer: ");
```


Nested and multiple if..else

true if i is even (divides by 2 with remainder 0)

```
if ( i > 0 && (i % 2) == 0 ) {
    System.out.println(i + " is greater than 0, and even");
}
```

equivalent

```
if ( i > 0 ) {
    if ( (i % 2) == 0 )
        System.out.println(i + " is greater than 0, and even");
}
```

```
if ( i > 0 )
    if ( (i % 2) == 0 )
        System.out.println(i + " is greater than 0, and even ");
    else
        System.out.println(i + " is greater than 0, but not even");
```

Without { } to set the structure, an **else** goes with the "nearest" **if**

Use { }, or at least (as shown here) use indenting as a clue!

Either part of an if..else can contain any typical executable statements - including for example another if or if..else (which is called a "nested if").

Try this

For the code on the right (on four separate runs), what is printed out when i is:

10

65

98

42

```
if (i <= 50) {
    if (i <= 25)
        System.out.println("one quarter");
    else
        System.out.println("two quarters");
} else {
    if (i <= 75)
        System.out.println("three quarters");
    else
        System.out.println("four quarters");
}
System.out.println("done!");
```



```
int mark;
char grade;
mark = readInt("Enter mark between 1 and 100: ");

if (mark >= 80)
    grade = 'A';
else if (mark >= 65)
    grade = 'B';
else if (mark >= 50)
    grade = 'C';
else
    grade = 'D';

System.out.println("Mark: " + mark + "  Grade: " + grade);
```

This kind of multiple if structure is common.

It executes **only** the statement that follows **the first true condition**

(or the last statement if no condition is true).

```
Enter mark between 1 and 100:
76
Mark: 76  Grade: B
```

```
Enter mark between 1 and 100:
48
Mark: 48  Grade: D
```

Two different runs of the program!

Switch

Switch lets us choose from a finite number of possible cases depending on the value of a **selector** variable of type **int**, **char** or **String** (or enumerated type - later). **Case labels** must be the same type as the selector.

In this example the selector is **int** n, and we have cases to handle some possible values of n. (The default case is optional.)

Note use of multiple labels for same case (n = 2 **or** 3 choose same case).

Common error! If a case is missing the **break**, execution will carry on into the next case!

```
switch( n ) {
   case 1:
      System.out.println("n is 1");
      break;  // exit the switch statement
   case 2:
   case 3:
      System.out.println("n is 2 or 3");
      break; // exit the switch statement
   case 10:
      System.out.println("n is 10");
      System.out.println("hooray!");
      break; // exit the switch statement
   default:
      System.out.println("n is another value");
} // end of switch
```


Assume well defined variables, and a readChar() method.

In this example we are processing an input character. We want to update counter variables to reflect whether the character is an 'a', a vowel ('a' is also a vowel), or a consonant.

After counting the input character we go on and print out some results (e.g. the values of the counters).

```
ch = readChar("Enter lowercase letter: ");
switch( ch ) {
    case 'a':
        aCount++;
        vowelCount++;
        break;
    case 'e': case 'i': case 'o': case 'u':
        vowelCount++;
        break;
    default:
        consonantCount++;
} // end of switch
System.out.println("Results are..." );
```


Short–circuit evaluation

Miscellaneous topic!! We can view expression evaluation as an "expression tree". LDC has only one example (p53).

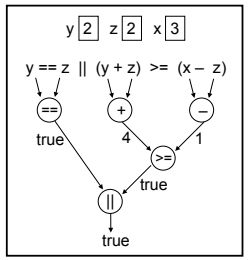
Assuming variables with the values shown, the box shows an example expression tree. In most cases Java evaluates the full tree.

In the case of && and ||, there is a slight exception. The left hand side is evaluated first, and the right hand side only if necessary.

This is because we know that:

false && something? is always false

true || something? is always true



In these cases there is **no need** to evaluate the right hand side. This **short–circuit evaluation** speeds up the program, but can have occasional consequences (that are hard to understand if you've never heard of this!).

Scope and design

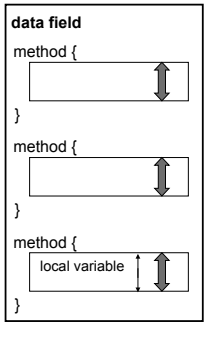
Where a variable can be "accessed" is called the **scope** of the variable.

A variable's scope is:
the block in which it is declared
(including any blocks inside the declaring block)

For example:

Data fields are declared in a class body / block, and can be accessed within it (including all the method body / blocks inside it).

Local variables are declared in a method body / block, and can be accessed only within it (including any other blocks inside it).



Scope within a method

It is good practice to declare local variables at the start of a method so that they are available to the whole method (also easy to find!). In this example **x** and **d** can be used anywhere in the method.

Some structures (e.g. loops, next lectures) declare variables that only exist within the block that makes up their body. Or in this example variable **y** is declared inside a block. In such cases the variable can only be accessed inside the block, not in the rest of the method.

Trying to use a variable out of scope gives a compile error, e.g. trying to use **y** outside its block:

Error: cannot find symbol
symbol : variable y

```
public void myMethod() {
    int x = 2;
    double d = 4.2;


    // block within method
    if (x == 2) {
        int y = 67;

    }
}
```


Try this

This class declares 3 variables, df, loc and x.

Which variables are in scope (can be seen / used, e.g. printed out by println) at each of the indicated points:

Point 1:

Point 2:

Point 3:

Point 4:

```
public class DemoScope {
  private int df = 4;

  public void myMethod1() {
     int loc = 10;
     System.out.println("Point 1");
     if (loc < 100) {
        int x = 5;
        System.out.println("Point 2");
     }
     System.out.println("Point 3");
  }

  public void myMethod2() {
     System.out.println("Point 4");
  }

}
```


Design principles

It is a good idea to make variables as **local** as they can be (same principle as **encapsulation** for objects, and data fields vs. method variables – protection from accidental change). If it is only needed in a block you can declare it in the block.

Instead of scattering variable declarations all over the place it can be very good practice to do all declarations at the start of a block (e.g. declare all variables in a method right at the start – then you always know where to look to find out what variables are used in the method).

Be very careful if you use the same name for different variables with different scope (in different blocks). It's asking for confusion!

Don't use too many variables. Before you declare one you should know exactly what it will be used for and exactly what its scope should be.

Repetition (loops) 1, iterators, iterable

COMP160 Lecture 11
Anthony Robins

- Introduction
- while loop
- do...while loop
- Iterators and Scanner
- for-each and Iterable
- Overview

Reading: LDC: 4.5, 4.6, 4.7.

LDC
3e to (2e)
p62 (p86)
p156 (p177)

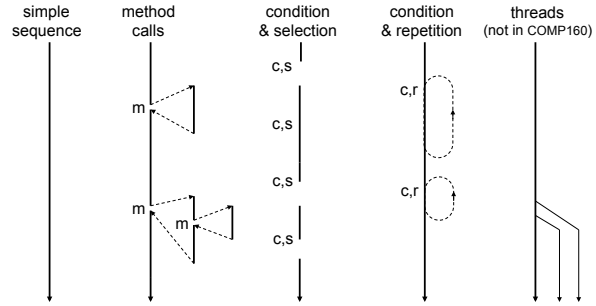
Introduction

This week we continue to look at processing within methods – flow of control (choosing and organising what to do next).

Java statements for selection (**if**..**else**, **switch**), and repetition (loops: **while**, **do**..**while** and **for)** are almost identical in C and C++. Most other languages have similar statements.

Java concepts **Iterator** and **Iterable** are similar to the concept of enumeration in some languages.

Flow of control



m = call a method
c,s = boolean condition, selection statement
c,r = boolean condition, repetition (loop) statement

while loop

while (condition) statement;

or more usually:

```
while(condition) {
    statement1;
    statementN;
}
```

Check the condition. If the condition is true, execute the **body** (statement or block).

Use a while loop when you want to execute the body only if the condition is true, and then keep executing it while the condition remains true.

Note:
- If the condition is false move on to the next statement after the loop.
- If the condition is true and the loop starts, something in the body had better change the value of the condition or the loop will never end!
- The condition might never be true (so the body might never execute).

Examples:

(assume a readInt())

```
String s = "COMP160";
// while not s is empty
while ( ! s.equals("") ) {
   System.out.println(s);
   // remove first character
   s = s.substring(1);
}
System.out.println("all done");
```

```
COMP160
OMP160
MP160
P160
160
60
0
all done
```

```
int i;
i = readInt("Enter an Integer: ");
System.out.println("Incrementing to 10");
while (i > 0 && i <= 10) {
    System.out.println(i);
    i++;
}
System.out.println("all done");
```

Enter an Integer:	Enter an Integer:
6	*12*
Incrementing to 10	Incrementing to 10
6	all done
7	
8	
9	
10	
all done	

Try This

What do these loops print out? The left hand style (update variables used in the condition at the **end** of the loop) is usually better.

```
int i = 1;
while (i <= 100) {
    System.out.println(i);
    i *= 2;
}
```

```
int i = 1;
while (i <= 100) {
    i *= 2;
    System.out.println(i);
}
```


do..while loop

do statement **while** (condition);

or more usually:

```
do {
    statement1;
    statementN;
} while (condition);
```

Execute the **body** of the loop (statement or block). Check the condition, if it is true

Use a do..while loop when you want to execute the body at least once, and then keep executing it while the condition remains true.

Note:
- If the condition is false move on to the next statement after the loop.
- If the condition is true and the loop repeats, something in the body had better change the value of the condition or the loop will never end!
- Even if the condition is never true the body will execute at least once.

Examples:

```
char c = 'a';
do {
    System.out.println(c);
    c++;  // next char
} while (c <= 'e');
System.out.println("all done");
```

```
a
b
c
d
e
all done
```

```
int i;
i = readInt("Enter an Integer: ");
System.out.println("Incrementing to 10");
do {
    System.out.println(i);
    i++;
} while (i > 0 && i <= 10);
System.out.println("all done");
```

Enter an Integer:	Enter an Integer:
6	*12*
Incrementing to 10	Incrementing to 10
6	12
7	all done
8	
9	
10	
all done	

compare with Slide 5

Try This

What does this loop do? What does it print out if the user enters the values 100, 20, 0 ? Why is **do..while** a better loop than **while** for this task?

```
int i = 0, sum = 0;
do {
    i = readInt("Enter value or 0 to finish");
    sum += i;
} while ( i  != 0 );
System.out.println ("Sum " + sum);
```

Iterators and Scanner

Iterate means "perform repeatedly". Iteration goes with loops!

An iterator is an object which implements the **Iterator** interface (interfaces – later). In practical terms it is an object set up for dealing with **collections** (Lec 24) of items / elements (really of **objects**).

Iterators have many useful methods for dealing with collections, the main ones are:

hasNext()	returns true if the collection contains another item
next()	returns (a reference to) the next item (object) in the collection

Scanner is a useful iterator which deals not with collections of objects (Lec 24), but with various data sources that can be treated in a similar way, such as (LDC p62):

- input streams (Lec 20 – like the keyboard / System.in in readInt)
- files (on disc, Lec 20)
- Strings (next slide!)


```
Scanner sc = new Scanner("Axe Bag Cat");
System.out.println( sc.next() );
System.out.println( sc.next() );
System.out.println( sc.next() );
```

The next method returns the next token[1] from the scanner as a String.

```
Scanner sc = new Scanner("Axe Bag Cat");
while ( sc.hasNext() ) {
   System.out.println( sc.next() );
}
```

In this example a loop is used to process each token from the scanner (until the hasNext method returns false).

In both cases above the output is:

```
Axe
Bag
Cat
```

Scanner also has many other methods, e.g. nextInt (returns the next token as an int), nextDouble (returns a double), and nextLine (return all the remaining text on the line as a single String). See LDC 2.6, 4.6.

[1]"Tokens" are text items that are separated by "whitespace" (blanks and tabs).

—— Try This ——

What is the output of this code?

```
String poem = "Roses are red, violets are blue";
String word;
Scanner sc = new Scanner( poem );
while ( sc.hasNext() ) {
   word = sc.next();
   if ( word.charAt(0) != 'a' ) {
      System.out.println( word );
   }
}
```


for-each and Iterable

The **for** loop is another kind of basic loop in Java (next lec). A particular kind of for loop is called "for-each".

for-each loops can process each object in an array (of objects) or collection that implements the **Iterable** interface (interfaces - Lec 17).

> **Iterator** and **Iterable** are similar but not the same!
> For example Scanner is an Iterator but is not Iterable - see LDC p156.

We are dealing with arrays in Lecs 15 & 16, but for now will introduce a simple array of strings as an example to use:

Data type of variable words is array of String

```
String [ ]  words  =  { "The", "quick", "brown", "fox" };
```

In this example you can think of words as just a simple list of strings (it is really an array of references to objects of type String – later!).

Each repetition the variable s is set to the current string in the array.
The variable s is then used in the body of the loop.

```
String [ ] words = {"The", "quick", "brown", "fox"};
for (String s : words) {
   System.out.println(s + "!");
}
```

```
The!
quick!
brown!
fox!
```

The **for** loop used in this way is called for-each.

In general it repeats for each item (setting a variable of that type) in an array or collection of items (of that type).

In this example it repeats for each String (setting variable s) in the array of String (referred to by the variable words).

For another example see LDC p156.

—— Try This ——

Modify the code from the previous example so that as well as printing out the strings (with !) it also counts the number of strings and prints out this total at the end.

```
String [ ] words = {"The", "quick", "brown", "fox"};

for (String s : words) {
   System.out.println(s + "!");

}
```


Overview

Loops are useful for things like:

- processing every item in an array / collection / stream (like a "list") of items
- repeated calculations
- reading an unknown amount of input
- repeating operations to check conditions are true or false before proceeding
- and so on...

Java has **while**, **do..while** and **for** loops (next lec), very similar in many languages!

Java has other functionality (similar to enumeration in some languages):

- **Iterators** (used with loops) to process collections, including Scanner for streams, files and strings
- for-each loop (version of for loop) to process **Iterable** arrays and collections

See more on loops next lecture and other topics later!

—— Try This ——

Given the array of String below, write a for-each loop that will print out every fruit that does not start with the character 'p'. Hints on Slide 12.

```
String [ ] fruit = {"apple", "peach", "plum", "cherry"};
```

```
apple
cherry
```

Repetition (loops) 2

COMP160 Lecture 12
Anthony Robins

- Introduction
- for loop
- Nested for
- Other combinations
- Design issues

Reading: LDC: 4.8, revise Chapter 4
LabBook: "How to Write a Program".

Introduction

By the end of this lecture we have looked at **selection** (if..else and switch), **repetition** (loops: while, do..while and for), and how to combine them.

Selection and repetition statements can contain other selection and repetition in any combinations – complex flow of control structures can be built to implement complex algorithms.

One of the main uses of repetition is to process groups / lists / sets / collections of items. See Slide 7, and more in Lecs 16, 17.

As usual I assume a readInt method.

for loop

```
for (initialisation; condition; update) statement;
```

or more usually:

```
for (initialisation; condition; update) {
    statement1;
    statementN;
}
```

The **header** describes the process for repeating the **body** of the loop (statement or block).

The header specifies an initial state, a condition for when to end, and a way of updating the state each repetition of the loop. All three can be flexible/complicated, in COMP160 we use only the commmon case of initialise, test & update a single "counter" variable.

Use this kind of for loop when you want to repeat the body a known / "definite" number of times (possibly using the different values of the counter in the body).


```
int sum = 0;
for (int i = 1; i <= 5; i++) {
    System.out.println("Hello " + i);
    sum += i;
}
System.out.println("Sum " + sum);
// System.out.println("i " + i);
// above line would fail, i undefined
```

```
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
Sum 15
```

```
int i;
int sum = 0;
for (i = 1; i < 5; i++) {
    System.out.println("Hi " + i);
    sum += i;
}
System.out.println("Sum " + sum);
System.out.println("i is " + i);
// note value of i !
```

```
Hi 1
Hi 2
Hi 3
Hi 4
Sum 10
i is 5
```

If declared (as on the left) in the loop header the counter variable **i** is **local** to the body / block (like a variable declared inside a block – last lecture). Alternatively we could (as on the right) use an existing variable as a counter...

`for (int i = 1; i <= 4; i++) { // statements in the body }`	**The body of this loop will be:** executed four times, with values of i 1, 2, 3, 4
`for (int x = 10; x >= 8; x--) { // statements in the body }`	executed three times, with values of x 10, 9, 8
`for (double d = 1.0; d <= 5.0; d += 1.5) { // statements in the body }`	executed three times, with values of d 1.0, 2.5, 4.0
`for (char c = 'a'; c <= 'g'; c += 2) { // statements in the body }`	executed four times, with values of c 'a', 'c', 'e', 'g' (see Lec 2)

Note: everything except the ";" is optional and the default condition is **true**.
So here is an infinite for loop that does nothing: for (;;) { }

Try This

What do these examples do?

```
int i;
for (i = 1;  i <= 4;  i++) {
      System.out.println("i is : " + i);
      if (i % 2 == 0) System.out.println("even");
}
System.out.println("Finished");
```

```
String s = "Apple";
for (int i = 0; i < s.length(); i++){
      System.out.println( s.charAt( i ) );
}
```


Processing groups of items

For loops are commonly used for processing groups of items, e.g. items in an array (more details Lecs 16, 17):

```
String [ ] words = {"The", "quick", "brown", "fox"};
for (int i = 0; i < words.length; i++) {
    System.out.println(words[ i ] + "!");
}
```

```
The!
quick!
brown!
fox!
```

for loop example common in many languages.

words.length has value 4.

```
String [ ] words = {"The", "quick", "brown", "fox"};
for (String s : words) {
    System.out.println(s + "!");
}
```

```
The!
quick!
brown!
fox!
```

for-each loop introduced to Java version 1.5, simplifies the task. Lecture 11 Slide 14.

Nested for

It is common to **nest** for loops inside each other – for example when printing out any kind of "two dimensional" table or dealing with a multidimensional data structure like (some) arrays.

```
for (int y = 1; y <= 3; y++) {        // outer loop (row)
    for (int x = 1; x <= 4; x++) {    // inner loop (col.)
        System.out.print( (y * x) + " ");
    } // end inner
    System.out.println("end of line " + y);
} // end outer
```

```
1 2 3 4 end of line 1
2 4 6 8 end of line 2
3 6 9 12 end of line 3
```

Note: The inner loop contains just one statement so the { } are optional, but make it clearer.

Try this

What do these print (not println)? The one on the left is harder!

```
for (int y = 1; y <= 4; y ++) {
    for (int x = 1; x <= y; x++) {
        System.out.print("O");
    } // end inner
    System.out.println();
} // end outer
```

```
for (int y = 1; y <= 3; y ++) {
    for (int x = 1; x <= 3; x++) {
        System.out.print( (x + y) + " " );
    } // end inner
    System.out.println(">");
} // end outer
```

Other combinations

Any repetition (loop) or selection statement can be used inside any other repetition or selection statement, as long as it is **completely contained** ("nested").

OK

```
do {
    // stuff
    if (input == 100) {
        // stuff
        // stuff
    } // end if
} while (input != 0);
```

Not legal

```
do {
    // stuff
    if (input == 100) {
        // stuff
} while (input != 0);
        // stuff
    } // end if
```


Try this

Write code to:

(1) Read in 10 integer values from the user and accumulate the total in a variable called sum. Any time the value 100 is entered, print out the message "OK".

```
int sum = 0, input = 0;
for (int x = 1; x <= 10; x++) {
    input = readInt("Enter integer:");
    sum += input;
    if (input == 100) {
        System.out.println("OK");
    } // end if
} // end for
```

(2) Read in a char (assume a readChar method that reads a char from the user). If the input char is "Y", print out 100 times "Are you sure?", otherwise print out "Why not?" once.

(3) Assume a variable called count with an unknown starting value. As long as count is less than 10, we want to repeat the following process: read an integer value, if the value is greater than 100 print "big", otherwise print "small", then (in either case) increase count by 1.

(4) Read in an integer value. If it is positive, print out the numbers from 1 up to the value.

Design issues

Java has three kinds of loops (**while**, **do..while** or **for**). Relating to their use are various design issues. We can classify loops as, for example, **counter** controlled, **state** controlled, **sentinel** controlled, or **menu** controlled.

Counter controlled: repeat for a set number of times as determined by a counter.

The typical example is a typical **for** loop:

```
for (int y = 1; y <= 5; y ++) {
    // loop body
}
```

initialisation
condition / test
update

But other loops can also be counter controlled:

```
int y = 1;
while (y <= 5) {
    // loop body
    y++;
}
```

Both loops repeat for values of y: 1, 2, 3, 4, and 5.

Be careful with changes to a loop counter. Apart from the required update, it is almost always a mistake to change the value of a counter elsewhere in the loop.

```
for (int i = 1; i <= 10; i++) {
    // loop body
    i = 1;  // Don't do this!  Infinite loop!
}
```


Patterns

Each of these different loop designs can be summarised as an abstract **pattern**. For example, the pattern of a counter controlled loop is:

```
initialise counter variable
test counter variable and repeat loop if necessary {
    loop body
    update counter variable
}
```

Other loop patterns (state, sentinel) are discussed in the Lab book reading How to Write a Program. In general there are many patterns at different "levels" to help us design programs (e.g. whole books on "Java patterns") – see the reading.

Options

Programming languages usually have lots of options for achieving the same result! On the next slide all three loops produce the same output…

Output:

```
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
Sum 15
```

The for loop is best suited to this simple counting based task.

```
int sum = 0;
for (int i = 1;  i <= 5;  i++ ) {
    System.out.println("Hello " + i);
    sum += i;
}
System.out.println("Sum " + sum);
```

```
int i = 1;
int sum = 0;
do {
    System.out.println("Hello " + i);
    sum += i;
    i++;
} while(i <= 5);
System.out.println("Sum " + sum);
```

```
int i = 1;
int sum = 0;
while(i <= 5) {
    System.out.println("Hello " + i);
    sum += i;
    i++;
}
System.out.println("Sum " + sum);
```


Scope

Where a loop body is a {block}, variables declared inside the block are **not visible** outside it. Remember issues of scope and design (Lec 10). Example problem:

```
do {
    int i = readInt("Input:");
    // do some stuff
} while (i  != -1);
```

Fails because i is declared inside the loop body (block), so undefined outside it.

```
int i;
do {
    i = readInt("Input:");
    // do some stuff
} while (i  != -1);
```

Succeeds. i is not declared inside the loop body (block), so not restricted to it.

Try This Answers Slides 11, 12

(no peeking during the lecture!)

Slide 12

```
while (count < 10) {
    int input = readInt("Enter an integer:");
    if (input > 100) {
        System.out.println("big");
    } else {
        System.out.println("small");
    } // end if
    count++;
}
```

```
int input = readInt("Enter an integer:");
if (input > 0) {
    for (int i = 1; i <= input; i++) {
        System.out.println( i );
    }
} // end if
```

Slide 11

```
char c = readChar("Enter a char");
if ( c == 'Y' ) {
    for (int x = 1; x <= 100; x++) {
        System.out.println("Are you sure?");
    } // end for
} else {
    System.out.println("Why not?");
} // end if
```

Objects 3: Classes and methods

COMP160 Lecture 13
Anthony Robins

- Introduction
- Methods revisited
- Design issues
- Visibility, encapsulation, design
- Class and instance (static)

Reading: LDC: 5.1 – 5.5
LabBook: "Object–Oriented Design"

LDC
3e to (2e)
p183 (p203)
p201 (p221)

Introduction

There's no strong theme to LDC Chapter 5, just a variety of topics relating to classes, objects, and program design. Much of it should be revision. This lecture covers:

Methods revisited (5.3, 5.4)

Design issues (5.1, 5.2)

Visibility (5.3)

Class and instance (static) (5.5)

For lab work coming up revise Lectures 6 & 7 (see Slide 7).

Methods revisited

Methods can use their own local variables, and data fields of their class. There are also ways of sending data directly to or back from a method.

Inputs

We can send values to a method by enclosing them in the "()" that make up part of the method call. The values we send are called **actual parameters** (also known as **arguments**).

The method must be defined so as to "expect" these values by specifying **formal parameters** (also known as the **parameter list** or **argument list**).

> The number, type and order of the actual parameters must match the number type and order of the formal parameters of (some version of) the method (see polymorphism, later!)

Output

A method can **return** a single value / result of a specified type to wherever it was called from. If it returns no value its return type is **void**.

In a class main method:

```
System.out.println( doubler(10) );
System.out.println( tester(5.1, 12.3) );
```

Output

```
20
true
```

Other methods in the class:

```
public static int doubler (int inValue) {
        return  inValue * 2;
}

public static boolean tester (double in1, double in2) {
        return  in1 < in2;
}
```

In each case the actual parameters match the formal parameters, and a value of the specified return type is returned and printed.

The variables which hold input parameters / arguments are local variables for the method. Apart from the fact that their values are set when the method is called, they behave and can be used just like any other local variable.

In this method a, b, x and y are all local variables of type int, they can all be used in the same ways.

```
public void MyMethod(int a, int b){
    int x = 1, y = 7;
    …
}
```

If inputs are used to set the value of a data field it is common to see code like this…

```
public void MyMethod(int a) {
    dataFieldName = a;
}
```

but if the input will be used as a local variable there is no need for this (often seen in COMP160)…

```
public void MyMethod(int a) {
    int x = a;
    …
}
```

just use the input argument variable!

```
public void MyMethod(int a) {
    // just use a
}
```


Try this

Write a method declaration / header for each of the following cases.

Takes as input two integers and returns a double:

```
public double aMethod(int in1, int in2) {  // start of body
```

Takes as input an integer and a double, returns an integer:

Takes as input three integers and returns a boolean:

Takes as input two doubles and returns no result:

Takes as input a (ref. to a) MyList object and returns a (ref. to a) MyList object:

Try this

Special methods:

What is a toString method?
Revise Lecture 7!

What is an accessor?
What is a mutator?
What is a constructor?
What is a default constructor?

If you don't remember, revise Lecture 6!

```
public Name( //inputs if any ) {
    // statements initialising object
}
```

```
public Name() { }
```

```
public void name(type inputName) {
      dataFieldName = inputName;
}
```

```
public type name() {
      return dataFieldName;
}
```


Design issues

LDC 5.1 & 5.2 cover many issues relating to design – most of it should be revision!

Designing a program involves working out what **classes** are needed. LDC suggest a process based on analysing a description of the task (Figure 5.2). (In practical terms see also "Developing programs" in Lecture 3.) Reuse library classes where possible and for the rest write your own!

Designing a class involves thinking about its attributes / state / data structures (**data fields**) and operations / behaviour / algorithms (**methods**). LDC have a good example Figure 5.1.

Work through the example in Section 5.2. including "UML Class Diagrams" as used in COMP160 labs.

Input – process – output

There is an underlying similarity that unifies many of the "levels" at which we can look at a program.

program
class / object
method
statement

At each of these levels the program "does something". Usually this will involve:

- taking in or accessing some input values
- performing some operation on them
- doing something with the result (saving it in some data structure or sending it somewhere).

Look for this **input** → **process** → **output** sequence when trying to understand a chunk of a Java program.

Visibility, encapsulation, design

There are two ways to access (**get** or **set**) values in the data fields of an object:

- access the data field directly (if public)
- use methods (accessors, mutators)

```
/* Anthony Robins,  August 2010
   Two ways to get values from data fields */
public class Hamlet {
   public static void main (String[ ] args) {
      Store s = new Store();
      // get the values directly
      System.out.println( s.i + " " + s.c );
       // accessors to get values
      System.out.println( s.getI() + " " + s.getC() );
   }
}
```

```
2 b
2 b
```

```
/* See application class Hamlet */
public class Store {

    public int i = 2;
    public char c = 'b';

    public int getI() {
        return i;
    }

    public char getC() {
        return c;
    }
}
```


Try this

For the example on the previous slide, write a mutator for i in class Store, and write statements (for the main method) to set i to 22 directly and to set it with the mutator.

Main method:

Class Store:

Direct access might seem more convenient, but it is more dangerous (different users of a data field might make mistakes). It is better design to use restricted visibility (make data fields **private**) and write methods for access.

See Fahrenheit Celsius example Lecture 6 Slide 11.

Visibility modifiers

Public and private are "visibility modifiers" (see others later). We so far recommend that data fields be **private**, service methods be **public**, and support methods **private**. (Service and support – Lec 8.)

Private data fields enforce **encapsulation**, i.e. data can be accessed only via the **interface** (public accessor and mutator methods), not directly. This protects data. For example, in Fahrenheit Celsius example (Lecture 6 Slide 11) the temperature data is encapsulated, so we can ensure that the state of the object is always consistent (Celsius and Fahrenheit always represent the same actual temperature).

A good summary (LDC p183):

	public	private
data fields	~~violates encapsulation~~	enforces encapsulation
methods	for service methods	for support methods

Design

The concept of encapsulation connects with a lot of other OO design issues:

The focus of OO design is to group data, and the methods that operate on data, into **objects**. The values stored in the data fields are the **state** of the object, its methods are its **behaviours**.

Make data fields **private** and access them with **public** accessor and mutator methods (the **interface** of the object). This is good design in terms of **encapsulation** and **information hiding** (see Lab book reading).

The members (data fields and methods) of an object should "belong together" (have high **cohesion**), direct (non OO) linkages between objects (**coupling**) should be reduced.

See the Lab book reading, Object–Oriented Design, and later (LDC Chapter 8).

Class and instance (static)

The **members** of a class are its data fields and methods.
Class members belong to the class object (declared as **static**),
instance members belong to instances made from the class / class object.

See Lecture 7:

(1) We can use class members in class objects: ClassName.memberName

(2) We can't use instance members in class objects (they don't really exist!)

(3) We can use instance members in instance objects: variableName.memberName

(4) We can seem to use class members in instance objects: variableName.memberName but we are really using the member in the underlying class object instead!

For a class data field the one value is shared by all instances. This can appear confusing at first…

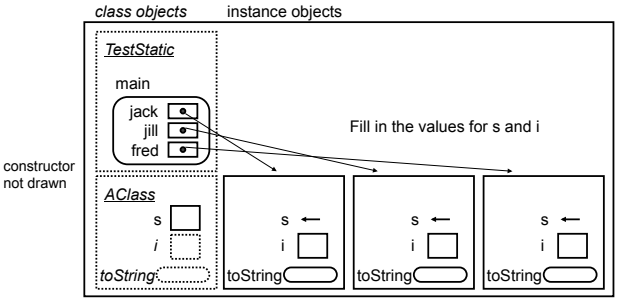
Try this

What is printed out?

```
/* Anthony, support has a class data field */

public class TestStatic {

    public static void main (String[ ] args) {
        AClass jack, jill, fred;
        jack = new AClass(1, 2);
        jill = new AClass(3, 4);
        fred = new AClass(5, 6);
        System.out.println( jack );   //toString
        System.out.println( jill );   //toString
        System.out.println( fred );   //toString
    }

}
```

```
public class AClass {

    public static int s;
    public int i;

    public AClass(int ins, int ini) {
        s = ins;
        i = ini;
    }

    public String toString() {
        return "Data " + s + ", " + i;
    }

}
```


The output is:



Class data fields are useful for:

Sharing information between instance objects, or representing data which is a **property of the whole class / type** rather than any individual. Design! See LDC SloganCounter (p201).

Defining values that can be used without creating instance objects, e.g. constants like Math.PI.

Class methods can only access class data, so they are **not** able to implement behaviours that work with the states of instance objects. They are useful for:

Accessing class data.

Defining methods that can be used without creating instance objects, e.g. methods like Math.sqrt().

In general you should refer to class members with ClassName.name, not (if you have created instance objects) variableName.name (in other words, avoid Slide 15 Case 4).

Objects 4
References

COMP160 Lecture 14
Anthony Robins

- Introduction
- References
- this
- Examples

Reading: LDC: 5.6 – 5.10.
LabBook: "Reference Types".

LDC
3e to (2e)
p78 (p102)
p218 (p237)

Introduction

LDC 5.6 – 5.10 covers lots of good topics – read them!

5.6: Class relationships and **this**
5.7: Method design
5.8: Method overloading (Lec 22)
5.9: Testing
5.10: Debugging
} Very useful practical advice!

However in this lecture I focus on references (including **this**) which are:
- not covered well in LDC, and
- very important to understanding Java

See the Lab book reading - just do it!

References

Variables hold values which are of **primitive types** (int, double etc) or **reference types** (references / handles / pointers) to objects.

Reference types include String, Array (later) and other classes (the **class** is the **type** of any instance objects made from the class).

In some languages (C, C++) you can manipulate **pointers** directly. In Java we call them **references**, they are slightly simplified and automated (garbage collection).

First revise Lec 7 "References and aliases".

Now let's see how this works as references are copied, passed and compared…

support class used this lec.

```
public class Point {
  private int x, y;

  // constructor
  public Point ( int inX, int inY ) {
    x = inX;
    y = inY;
  }

  public void setX( int inX ) {
    x = inX;
  }

  public int getX() {
    return x;
  }

  public String toString() {
    return "A Point " + x + " " + y;
  }

} // Point
```


Copying a value

Primitive type: results in two separate copies of the same value (e.g. a number):

```
int a, b;
a = 3;
b = a;
```

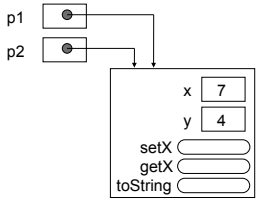
a 3
b 3

Reference type: results in two copies of the same value (a reference) – these are aliases (LDC p78) – NOT copies of the object itself:

```
Point p1, p2;
p1 = new Point( 7, 4 );
p2 = p1;
```

```
System.out.println( p1 );
System.out.println( p2 );
```

```
A Point 7 4
A Point 7 4
```



Passing a value to a method

Primitive type: results in two separate copies of the same value (e.g. a number):

```
int a = 3;
aMethod( a );
System.out.println( "a is " + a );
```

```
b is 3
a is 3
```

```
public void aMethod (int b) {
    System.out.println("b is " + b);
}
```

a 3 b 3

Reference type: results in two copies of the same value (a reference) (LDC p218):

```
Point p1 = new Point( 7, 4 );
aMethod( p1 );
System.out.println( p1 );
```

```
A Point 7 4
A Point 7 4
```

```
public void aMethod (Point p2) {
    System.out.println( p2 );
}
```

p1 and p2 refer to the same object - as on the previous slide

Comparing values

Primitive type: compares values (checks if they are e.g. the same number):

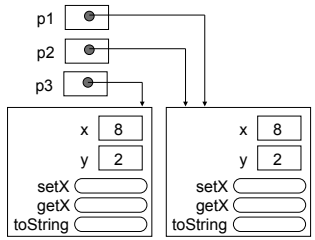
```
int a = 3, b = 3;
// a == b is true
```

a 3
b 3

Reference type: compares values (checks if they are aliases / references to the same object):

```
Point p1, p2, p3;
p1 = new Point( 8, 2 );
p2 = p1;
p3 = new Point( 8, 2 );
// p1 == p2 is true
// p1 == p3 is false
```

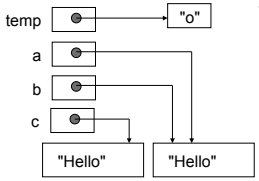
Objects with the same state (values for their data fields) are **not** the same object!



Strings

Strings are objects, so variables hold references to strings.

```
String temp = "o";
String a = "Hello";
String b = a;
String c = "Hell" + temp;
```




```
// a == b      is true  (aliases, references are equal)
// a == c      is false (not aliases)
// a > c       illegal!  makes no sense

// a.equals(b)      is true (objects hold same contents / state)
// a.equals(c)      is true (objects hold same contents / state)
// a.compareTo(c)   would be negative if string a was ordered before string c
```

The class String has various methods, including equals which returns true if the two string objects have the same contents / state. To use equals a reference to a string object is passed as input to the equals method called on another string object.

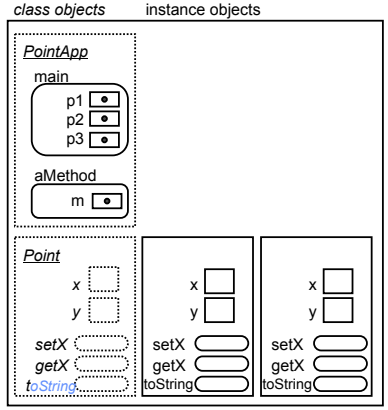
Try this

Consider this program (uses the Point class from Slide 3).

Work through the program, and complete the model on the next slide by adding the references and the values of the data fields. Show the output.

```
/* Anthony Robins, August 2006, JDK 1.5 */

public class PointApp {

    public static void main(String[ ] args) {
        Point p1, p2, p3;
        p1 = new Point( 1, 3 );
        p2 = p1;
        p3 = new Point( 2, 4 );
        p2.setX( 5 );
        System.out.println( p1 );  // toString
        System.out.println( p2 );  // toString
        System.out.println( p3 );  // toString
        aMethod( p3 );  // more difficult!
        System.out.println( p3 );  // toString
    }

    public static void aMethod( Point m ) {
        m.setX( 26 );
        System.out.println( m );  // toString
    }

} // PointApp
```

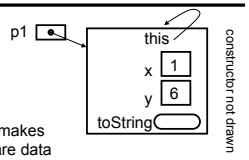

class objects instance objects



Output:

this

The keyword **this** is like a variable that always refers to the current (instance) object. We can use usual dot notation e.g. this.x.



Usual shortcut way to name data fields.

Equivalent way makes it clear x and y are data fields in this class / object.

```
public class Point {
   private int x, y;  // data fields

   // constructor
   public Point (int inX, int inY) {
      x = inX;
      y = inY;
   }

   public String toString() {
      return "A Point " + x + ", " + y;
   }
} // Point
```

```
public class Point {
   private int x, y;  // data fields

   // constructor
   public Point (int inX, int inY) {
      this.x = inX;
      this.y = inY;
   }

   public String toString() {
      return "A Point " + this.x + ", " + this.y;
   }
} // Point
```


Now (for clarity) **parameters can be given the same name as data fields** in a method or constructor. (No need for odd names like "inX" or "inY", e.g. last slide).

Examples like this raise issues of the "scope" of identifiers (names). What do the identifiers "x" and "y" mean at different points in the program?

```
public class Point {
   private int x, y ;  // data fields

   // constructor
   public Point (int x, int y) {
      this.x = x;
      this.y = y;
   }

   public String toString() {
      return "A point " + x + ", " + y;
   }
} // Point
```

or this.x this.y

Within the constructor x and y mean the local variables defined in the parameter list, not the data fields.

this is a reference the current object, so this.x and this.y **always** mean the data fields x and y.

In this method there are no local variables with the same names so x and y mean the data fields.

Try this

What is printed out?

```
/* Anthony, "this", scope of variables */
public class Hello {

  public static void main(String[ ] args) {
    MySupport m = new MySupport( 5 );
    m.one();
    m.two();
  }
}
```

```
public class MySupport {
  private int x = 1;

  public MySupport(int x) {  // constructor
    System.out.println("cons a " + x);
    System.out.println("cons b " + this.x);
    this.x = x;
    System.out.println("cons c " + x);
    System.out.println("cons d " + this.x);
  }

  public void one() {
    int x = 7;
    System.out.println("one  a " + x);
    System.out.println("one  b " + this.x);
  }

  public void two() {
    System.out.println("two  a " + x);
    System.out.println("two  b " + this.x);
  }
}
```


Examples

Both examples combine the use of **this** and the use of class members (**static**).

Example 1 compares two cases, the use of a class method and the use of an instance method, as alternative ways of doing exactly the same task (testing the "lengths" of two "lines" to see if they are equal). Different library classes use class and instance methods like this.

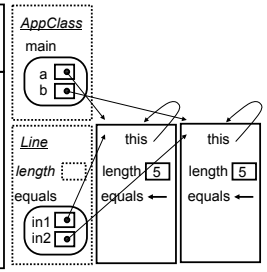
Example 2 shows the use of a class data field to represent information about the whole class / type (all the instances). The imagined task has the simplified structure of some graphics application where there are three individual "button" objects. The underlying class object has a data field which is to contain a reference to whichever button was most recently "pressed". Individual button objects set the class data field (when "pressed") using a **this** reference to themselves.

Example 1 A

Comparing the "lengths" of two "lines" using a class method.
A similar example from a library class is Arrays.equals() - see arrays in Ch 7.

```
Line a = new Line(5);
Line b = new Line(5);
System.out.println( Line.equals(a, b) );  // prints "true"
```

```
public class Line {
   public int length;

   public Line ( int length ) {  // constructor
      this.length = length;
   }

   public static boolean equals( Line in1, Line in2 ) {
      return in1.length == in2.length;
   }
}
```



uses direct access of a public data field to keep example simple - better design to use an accessor!

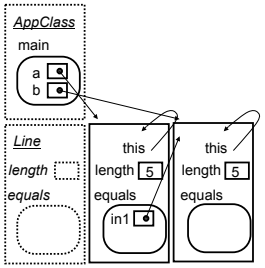
constructors not drawn

Example 1 B

Comparing the "lengths" of two "lines" using an instance method.
A similar example from a library class is equals() for comparing strings (Slide 7).

```
Line a = new Line(5);
Line b = new Line(5);
System.out.println( a.equals(b) );  // prints "true"
```

```
public Line {
   public int length;

   public Line ( int length ) {  // constructor
      this.length = length;
   }

   public boolean equals( Line in1 ) {
      return this.length == in1.length;
   }
}
```



uses direct access of a public data field to keep example simple - better design to use an accessor!

constructor not drawn

Example 2

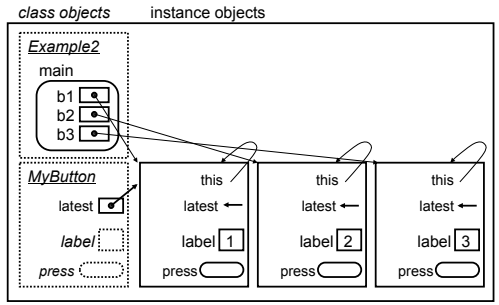
```
/*  Anthony, Sept 2008, this and class */

public class Example2 {

  public static void main(String [ ] args) {
    MyButton b1 = new MyButton(1);
    MyButton b2 = new MyButton(2);
    MyButton b3 = new MyButton(3);
    b2.press();  // "pressing" the button
    b3.press();  // "pressing" the button
    System.out.println("Latest " + MyButton.latest.label);
    b1.press();  // "pressing" the button
    System.out.println("Latest " + MyButton.latest.label);
  }
}
```

```
public class MyButton {
   public static MyButton latest;
   public int label;

   public MyButton(int label) { // constructor
      this.label = label;
   }

   public void press() {
      System.out.println("Pressed " + label);
      latest = this; // sets class data field
   }                 // to refer to this instance
}
```

```
Pressed 2
Pressed 3
Latest 3
Pressed 1
Latest 1
```

MyButton.latest always refers to the object representing the latest pressed button. (Again, direct access of public data fields - poor encapsulation).

State of the model at the end of the program



(constructor not shown)

Arrays 1

COMP160 Lecture 15
Anthony Robins

- Introduction
- Collections of data
- Arrays
- Array types
- Using arrays
- Multidimensional arrays

Reading: LDC: 7.1, 7.2, 7.6

LDC
3e to (2e)
p350 (p363)

Introduction

In the next two lectures we will be looking at arrays, an absolutely core part of most programming languages. Arrays are usually used with loops, especially **for** and **for-each** loops.

LDC 7.1, 7.2, 7.6 – this lecture

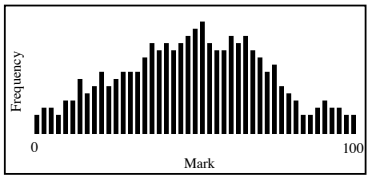
LDC 7.3 – next lecture

LDC 7.4, 7.5 – your own reading (7.4 is relevant to Lab 15)

Collections of data

So far we have dealt with representing individual items of information. What do we do to represent collections of data?

For example, 500 exam marks – integer values between 0 and 100. Each mark will occur a number of times. How can we store information about the frequency of each mark?



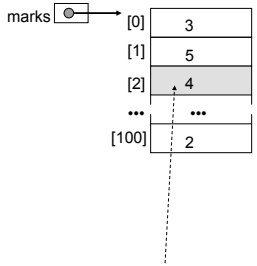
We could use 101 variables, mark0, mark1, mark2 and so on – but that's not pretty!

Lecs 15 and 16 on arrays, Lec 24 on other Java ways of representing collections.

Arrays

An **array** is a named collection of data (of the same type, of a fixed size). An array is like a "table" or "list".

Our 101 mark frequency values could be stored in an **array** called marks. This array has an int **index** from 0 to 100 and holds int values.



For arrays in general:

- the index can only be int (or char, which is cast to int), starting at 0.
- they can hold values of any primitive type, or references / handles / pointers to objects.

We can refer to an individual element / "cell" in the array, for example, marks[2]. (marks[i] will point to different cells depending on the value of i)

Indexed variables

Each cell in an array functions like a normal variable of the type (e.g. int), in fact they are sometimes called **indexed variables**.

We can use the indexed variables in an array in the same way that we could use any variable of the type:

```
marks[1] = 5;
sum += marks[42];
marks[27]++;
marks[i] = x + 12;
marks[2 + i] = readInt("Enter a value");
aMethod( marks[99] );
```

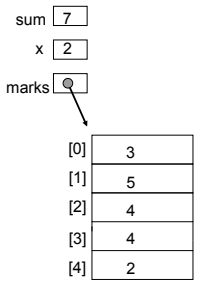
If you try to use an index that is **out of range** the interpreter will generate an error:

```
marks[200] = 5;      // run time error, message in console window
```


Try this

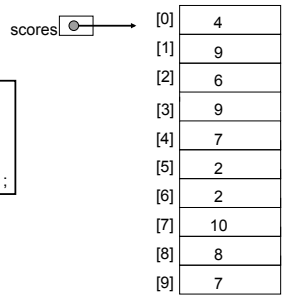
What is the effect of the statements below, given the array and the int variables with initial values shown on the right:

```
marks[1] = 9;
marks[0]++;
marks[x] = 6;
marks[x + 2] = sum;
marks[3] = sum + 2;
sum += marks[1];
sum = marks[1] + marks[x];
marks[5] = 5;

for (int i = 0; i < 5; i++) {
    System.out.println( marks[ i ] );
}
```



For example

A typical use of an array is to hold a "table" of data. We might want to find e.g. the average. Given an array scores of 10 integers:



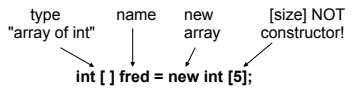

```
double sum = 0.0;
for (int i = 0;  i < 10;  i++) {
      sum += scores[ i ];
}
System.out.println("Average: " + (sum / 10) ) ;
```

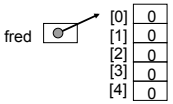

Array types

Arrays are objects (almost).
Variables can contain a reference (Lecture 14) to the array object.
Any type T in the program can be used in an array (of type "T [ ]", array of T).

"Any type" includes primitives and classes (which are the type of instances of that class). An array of a class/ reference type contains references to instances (Lec 16).

The parts of an array declaration and initialisation:





Unless otherwise specified an array has all cells initialised to "zero states": 0, 0.0, null, false, etc.

Declaring and initialising arrays

Example declarations:

can be written – int x [ ]; – but we won't

```
int i;        // primitive type used for comparison, i is of type int
int [ ] x;    // x is of type int[ ] (array of int - can hold integers)
```

Before we can use them we must initialise these arrays, e.g.:

```
i = 5;        // primitive type used for comparison
x = new int [50];    // x holds 50 integers all initialised to 0
```

As for other types we can combine these steps:

```
int i = 5;                  // as above
int [ ] x = new int [50];   // as above
```

Initialising the elements

Initialising arrays as shown so far initialises the cells to zero states (0, 0.0, false etc.). Usually you will want to assign some other value to each element / cell.

For example to assign values (read in from the user) to the cells of **x** (previous slide):

```
for (int i = 0; i < 50; i++) {
    x[ i ] = readInt("Enter an integer for cell " + i ); // assume readInt()
}
```

So setting up an array is a three step process:

(1) declare the array,

(2) initialise the array,

(3) initialise the elements (if not using the default values e.g. 0).

Initialiser lists

Arrays (like Strings) have special shortcuts built into the language. For example, we can, when an array is declared, initialise it with an initialiser list (LDC p350). (This is similar to the use of literals – Lec 4). Enclose the values to be stored in { }…

```
int [ ] intAr = { 3, 7, 9, 23, 2 };    // initialiser list of 5 integer values shown
```

This achieves all three steps (previous slide) at once!

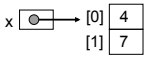
For example, the following definitions of x are equivalent:

comments show the steps (last slide) done by this statement

```
int [ ] x = {4, 7};  //1 2 3
```

```
int [ ] x = new int[2]; // 1 2
x[0] = 4;   // 3
x[1] = 7;   // 3
```

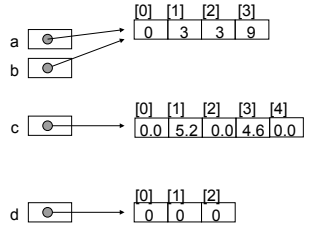
```
int [ ] x;  // 1
x = new int[2];  // 2
x[0] = 4;  // 3
x[1] = 7;  // 3
```



For example

After the declarations and statements on the left have executed, we have the arrays shown on the right:

```
int [ ] a = {4, 3, 3, 2};
int [ ] b = a;
double [ ] c = new double[5];
c[1] = 5.2;
c[3] = 4.6;
a[0] = 0;
b[3] = 9;
int [ ] d;
d = new int[3];
```



Try this

After the declarations and statements on the left have executed, show the arrays on the right.

```
double [ ] a = { 3.3, 2.7, 1.4 };
int [ ] b;
b = new int [5];
b[3] = 42;
double [ ] c = a;
c[0] = 9.0;
```


Using arrays

Arrays can be used (declared, assigned to, compared, passed as parameters, returned as results) like any other variable.

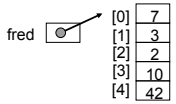
Here's an example use, a method which takes an array of integers as input and returns the average of the values stored in the array:

```
public double avIntArray( int [ ] a ) {
    // accepts any integer array as input and returns
    // the average of the values stored
    double sum = 0.0;
    for (int i = 0;  i < a.length;  i++) {
        sum += a[ i ];
    }
    return sum / a.length;
} // avIntArray
```

the length "field" tells us how many cells in the array, see next slide

Length

The length of (number of elements in) every array is stored in a "data field" **length**.



fred.length is 5

An array always starts at position 0 and goes up to (length - 1)

This information can be very useful in working with arrays. For example, to write out every value in fred:

so use < not <=

```
for (int i = 0;  i < fred.length;  i++)
    System.out.println( fred[ i ] ) ;
```

Writes out:

```
7
3
2
10
42
```

Length is not a true data field. You can read the length value but you cannot set it: fred.length = 10; // can't!

Multidimensional arrays

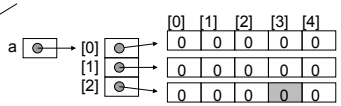
The arrays we have seen so far have all been "**one dimensional**" (1D), a single list of data (e.g. previous slide).

We can can also have **two dimensional** (2D) arrays, which is like a table of data with (when drawn as here) rows and columns.

For example:

```
int [ ] [ ] a = new int [3] [5];
```

type array of array of int



To refer to an element:

a[2][3]

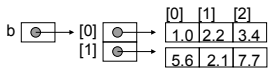
The length of the array, a.length, is the number of rows, 3.
The length of a row, e.g. a[0].length, is the number of columns, 5.

We can also have **multidimensional** arrays, e.g. a 3D cube or oblong of data. COMP160 will stick to 1D and 2D. 2D arrays are implemented as arrays of array objects. (As we see in Lecture 16, we can have an array of any object).

For example

As for 1D, we can write 2D initialiser lists. The following declaration creates:

```
double [ ] [ ] b = { {1.0, 2.2, 3.4}, {5.6, 2.1, 7.7} };
```



To print out the values stored in this array:

length – see last 2 slides

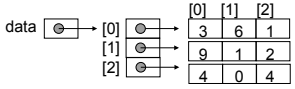
```
// print a 2D array called b
for (int row = 0;  row < b.length;  row++) {        // for each row
    for (int col = 0; col < b[row].length; col++) { // for each column in row
        System.out.print( b[row] [col ] + " " );    // print element
    }
    System.out.println();                           // move to new line
}
```

Output:

```
1.0  2.2  3.4
5.6  2.1  7.7
```


Try this

Given the following 2D array, write code which will print out the total of the values stored in each row, as shown. (The nested for loops on the previous page would be a good place to start…).



Output:

```
Row 0:  10
Row 1:  12
Row 2:  8
```

Arrays 2
references to objects

COMP160 Lecture 16
Anthony Robins

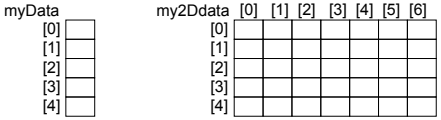
- Arrays are flexible
- Arrays are objects
- Arrays of (references to) objects
- More on using arrays
- Main explained

Reading: LDC: 7.3
LabBook: Reference Types

LDC
3e to (2e)
p156 (p177)
p219 (p239)
p350 (p363/4)
p354 (p367)

Arrays are flexible

In most programming languages arrays can only hold primitive values. They have no "internal structure" (references), so they are very fixed, rigid structures like:



Arrays in Java are more flexible.

They are "objects".

They can hold references to objects.

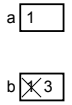
Their internal structure (references to arrays within arrays) lets us access and modify them (and parts of them) in a variety of ways.

Arrays are objects

A reference to an object is distinct from the object itself (Lecs 7, 14). Arrays are objects (almost). See Examples 1 - 3, the first is a primitive type for comparison.

```
/* Example with primitive data type */
public class Example1 {

  public static void main (String [ ] args) {
    int a = 1;
    System.out.println("a is " + a);
    myMethod( a );
    System.out.println("a is " + a);
  }

  public static void myMethod(int b) {
    b = 3;
    System.out.println("b is " + b);
  }
}
```

```
a is 1
b is 3
a is 1
```



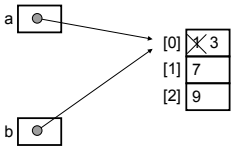
a and b are separate values (ints).

myMethod changes the value of b, this does not affect the value of a.

LDC have a similar example (far too detailed) starting p219.


```
/* Example with an array object */
public class Example2 {

  public static void main (String [ ] args) {
    int [ ] a = {1, 7, 9};
    System.out.println("a[0] is " + a[0]);
    myMethod( a );
    System.out.println("a[0] is " + a[0]);
  }

  public static void myMethod( int[ ] b ) {
    b[0] = 3;
    System.out.println("b[0] is " + b[0]);
  }
}
```

```
a[0] is 1
b[0] is 3
a[0] is 3
```



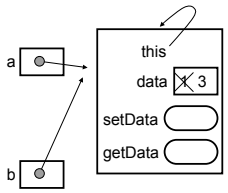
a and b are separate values (references to an array, in this case aliases for the same array - Lec 7).

myMethod does nothing to the value of b, but it changes the value of cell b[0] (changes the state of the array that b currently refers to).


```
/* Example with an instance object */
public class Example3 {

  public static void main (String [ ] args) {
    MyClass a = new MyClass(1);
    System.out.println("a.data is " + a.getData() );
    myMethod( a );
    System.out.println("a.data is " + a.getData() );
  }

  public static void myMethod(MyClass b) {
    b.setData( 3 );
    System.out.println("b.data is " + b.getData() );
  }
}
```

```
a.data is 1
b.data is 3
a.data is 3
```

```
public class MyClass {
  private int data;
  public MyClass(int data) { this.data = data; }
  public void setData(int data) { this.data = data; }
  public int getData() { return data; }
}
```



a and b are separate values (references to an object, in this case aliases for same object).

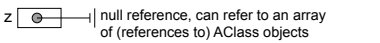
myMethod does nothing to the value of b, but it changes the value of b.data (changes the state of the object b refers to).

Arrays of (references to) objects

Arrays can hold (references to) objects. Completely setting up one of these involves the same three steps as for an array of primitives: declaring the array, initialising the array, and initialising the elements.

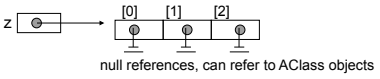
(1) Declaring:

```
int [ ] x;          // x is of type int[ ] (array of int). The array can hold integers
AClass [ ] z;       // z is of type AClass[ ] (array of AClass). The array can hold
                    // (references to) objects that are instances of AClass.
```



(2) Initialising:

```
x = new int [5];    // x holds 5 integers all initialised to 0.
z = new AClass [3]; // z holds 3 references to AClass objects all initialised to null.
```

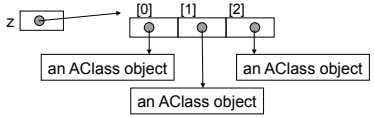


As usual we can combine these steps in one statement:

```
int [ ] x = new int [5];          // as above
AClass [ ] z = new AClass[3];     // as above
```

(3) Initialising elements:

```
z[0] = new AClass( );    // the value assigned is a reference to an object
z[1] = new AClass( );
z[2] = new AClass( );
```



All at once!

We can do all three steps at once by writing an initialiser list (Lec 15, LDC p350). See LDC GradeRange example (p354).

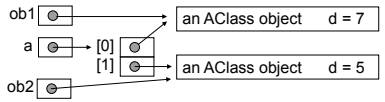
```
AClass [ ] z = { new AClass(), new AClass(), new AClass() };   // same model
                                                               // as above
```


More on using arrays

Other variables can reference the objects in an array (this kind of flexibility is very unusual in other languages!).

Assume a class AClass with a constructor that takes a single input and uses it to set the value of a data field d:

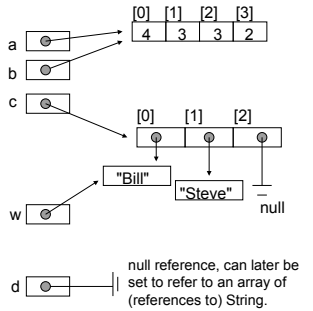
```
AClass ob1 = new AClass(7);  // ob1 refers to a new object with d = 7
AClass [ ] a = new AClass[2]; // create array of (null) refs to 2 AClass objects
a[0] = ob1;                  // a[0] refers to the same object as ob1
a[1] = new AClass(5);        // a[1] refers to a new object with d = 5
AClass ob2 = a[1];           // ob2 refers to the same object as a[1]
```



Examples

After the declarations and statements on the left have executed, we have the arrays shown on the right:

```
int[ ] a = {4, 3, 3, 2};
int[ ] b = a;
String[ ] c;

c = new String[3];
c[0] = "Bill";
c[1] = "Steve";
String w = c[0];

String[ ] d;
```



Try this

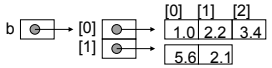
After the declarations and statements on the left have executed, show the arrays on the right. Assume that AClass is a class defined in the program.

```
String [ ] a = {"Hello", "Goodbye"};
double [ ] b = {3.3, 2.7, 1.4};
double[ ] c = b;
AClass[ ] d = new AClass [3];

b[0] = 2.0;
c[0] = 9.0;
d[0] = new AClass();
AClass t = new AClass();
d[2] = t;
```


2D arrays again

In most languages 2D arrays are "rectangular" (all rows the same length). In Java this is not necessarily the case…

```
double [ ] [ ] b = { {1.0, 2.2, 3.4}, {5.6, 2.1} };
```



This is why any processing of the elements in a row should be based on the length of that row…

length of b is 2 (number of rows)

length of row b[0] is 3
length of row b[1] is 2

```
// print a 2D array called b
for (int row = 0;  row < b.length;  row++) {        // counter for each row
   for (int col = 0; col < b[row].length; col++) {  // counter for each column
      System.out.print( b[row] [col ] + " " );      // print element at row, col
   }
   System.out.println();    // at the end of each row, move to new line
}
```

Output:

```
1.0  2.2  3.4
5.6  2.1
```


for-each loops

Recall the for-each loop (Lec 11, LDC p156): **for** (type var : collection) { }

See also LDC Primes (p350), GradeRange (p354).

In Lec 11 we used it to process every string in an array of strings. Here we use nested for-each loops to process the same 2D array b as the previous slide (creating the same output):

- outer loop for each value (an array of double) in b (an array of arrays of double)
- inner loop for each value (a double) in row (an array of double)

```
// print a 2D array called b
for (double[ ] row : b ) {              // for row set to each value in b
   for (double x : row ) {              // for x set to each value in row
      System.out.print( x + " " );      // print x (a copy of value in row)
   }
   System.out.println();    // at end of each row, move to new line
}
```

Variable length of rows handled automatically!

But note: Changes to x do not change the underlying array. To make changes use basic for loop.

Comparing and copying arrays

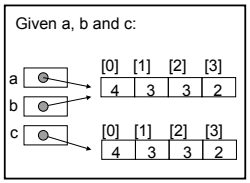
See Lec 14, and the Lab book reading **Reference Types** for background.

As for other reference types the assignment operator "=" assigns / copies **references**:

b = a;

and the relational operator "==" compares **references**:

a == b is true
a == c is false



To compare the **contents** of arrays use:

Arrays.equals(a, b) is true *// Arrays is in the package java.util*
Arrays.equals(a, c) is true

See Lec 14 Slide 14 for a similar model.

or write your own method.

To copy the **contents** of arrays use System.arraycopy().

Accessing members of objects in an array

We can access the data fields and methods of an object in an array just like any other object using dot notation: variableName.memberName.

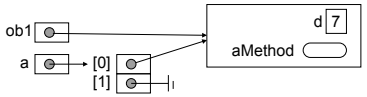
The variable can be an indexed variable (cell of an array).

To access a data field:

ob1.d or
a[0].d

To access a method:

ob1.aMethod() or
a[0].aMethod()



The example on the next slide shows the use of a for loop to call a method on every object in an array. (The array has the same structure as the model on Slide 7).


```
/* Calling a method on every object in an array  - see model on Slide 7 */
public class Example4 {

   public static void main (String [ ] args) {
      AClass [ ] z = { new AClass(7), new AClass(6), new AClass(5) };
      for (int i = 0; i < z.length; i++) {
         System.out.println( z[ i ].getData() );  // call method on object referred
      }                                           // to by this cell in the array
   }
}
```

Output:

```
7
6
5
```

```
/* Support class */
public class AClass {
   private int data;
   public AClass(int data) { this.data = data; }  // constructor
   public int getData() { return data; }
}
```


Try this

Assume a support class Point, and code in a main method that creates an array of Point (see the model on the next slide).

Write a for loop that will access every object in the array, calling the writer method on the object (to create the output at the bottom right).

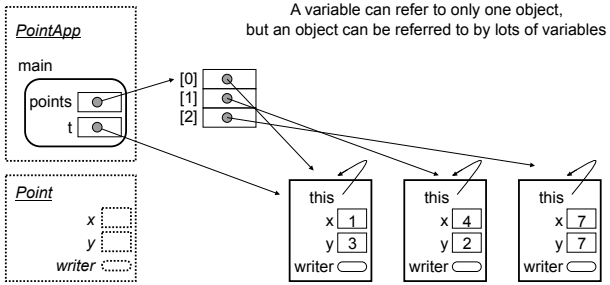
Write a for-each loop (Slide 12) that does the same task.

```
Point [ ] points = new Point [3];
points[0] = new Point(1, 3);
points[1] = new Point(4, 2);
points[2] = new Point(7, 7);
Point t = points[0];
```

```
public class Point {
   private int x, y;

   public Point (int x, int y) {
      this.x = x;   // refers to this object
      this.y = y;   // see Lecture 14
   }

   public void writer( ) {
      System.out.println("At: " + x + " " + y);
   }
}
```

```
At: 1 3
At: 4 2
At: 7 7
```


A useful rule:
A variable can refer to only one object,
but an object can be referred to by lots of variables



Main explained

When we run a Java program the main method can (in command line environments) be passed as input a reference to an array of strings. Not recommended as not all platforms have a command line.

```
public class Demo {
   public static void main ( String [ ] args ) {
      System.out.println( args[0] + " *** " + args[1] );
   }
}
```

The main method declaration is now fully explained!

Output:

```
prompt% java Demo apple pear
```

```
apple *** pear
```

system prompt, java interpreter, class name, arguments

See LDC 7.4, and don't forget 7.5 on variable length parameter lists.

Graphics 1
components

COMP160 Lecture 17
Anthony Robins

- Introduction
- Elements of a GUI
- A graphics design
- Components
- Containment hierarchies
- Inner classes
- Interfaces

Reading: LDC: 6.1 – 6.3 (will be same for next lecture too)

LDC
3e to (2e)
p46 (p70)
p248 (p264)

Introduction

Our programs so far have been mostly text based, not graphical user interfaces (GUIs) – Lec 5 Slide 2. The only "graphics" we have done is drawing pictures in a frame (Lec 5, Lab 5) or in a panel added to a frame (Lab 14).

In the next 3 lectures we cover GUIs – constructing a graphical interface within a window, then making it "come alive" by responding to events (and even simple animation!). LDC 6.1 – 6.3 mix all this together, I'm going to separate it out in lectures. So you probably can't follow everything in the LDC reading yet!

Our programs so far all involve the programmer completely specifying the sequence of operations in the program. GUI programs are ultimately a different way of thinking about program design – event driven programming – where the program is set up to respond to user generated events in the graphical interface.

Elements of a GUI

To build an interactive GUI we need three main kinds of classes:

- **components**: objects that are drawn in the GUI, e.g. buttons, labels, text fields, including **containers** like frames and panels that organise other components.
- **events**: objects (automatically constructed by components) that represent events in the GUI, such as button clicks, mouse clicks, text input, and so on.
- **listeners**: objects which get sent events (references to event objects) and have methods set up to process them.

Other kinds of classes involved in GUIs include **graphics** (to draw graphics), **layout managers** (help organise components in a container), and **menu components** (for building menus).

We will cover all this in the next three lectures, but let's start with a simple design…

A graphics design

All our graphics programs will have the same basic design. Most of the work is done in the support class (next slide) which defines a panel (JPanel). The application class just creates a frame (JFrame) / window, and adds the panel to it.

Application class always has the same 5 steps:

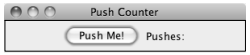
```
import javax.swing.JFrame;   // import required class

public class MyAppClass {

  public static void main(String [ ] args) {                  // Steps:
    JFrame frame = new JFrame("Push Counter");                // 1 make frame / window
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);     // 2 define close behaviour
    frame.getContentPane().add( new MySupportPanel() );       // 3 add support panel *
    frame.pack();                                             // 4 auto size the frame
    frame.setVisible(true);                                   // 5 make frame visible
  }
}
```

panel added to the "content pane" of the window

The steps are the same every time except for the name of the support class in step 3.

* Step 3 is the same as but a bit more efficient than e.g. LDC p 248

Support class. The constructor sets the contents and details of the panel creating (within the window / frame) this "GUI":



Support class always has the same 3 steps:

```
import java.awt.*;       // import standard graphics classes
import javax.swing.*;

public class MySupportPanel extends JPanel {

  public MySupportPanel() {       // constructor     // Steps:
    JButton push = new JButton("Push Me!");         // 1 define components (this eg two only)
    JLabel label = new JLabel("Pushes: ");
    add(push);                                      // 2 add components to panel
    add(label);
    setPreferredSize(new Dimension(300,40));        // 3 set panel details (this eg size only) *
  }
}
```

add and set methods inherited from JPanel

Exact details of steps 1 – 3 will vary depending on the GUI to be built! This simple example based on LDC p 248.

* The size of the panel determines the size of the frame.

Components

A GUI **component** is an object that is drawn on screen to display information or allow the user to interact with the program (e.g. buttons, labels).

Containers

A **container** is a special type of component that holds and organises other components. The main ones are:

Frame (class JFrame): a container that is drawn as a separate window. It is a "heavyweight" component (managed by the underlying operating system, graphics use a paint method). Consists of several layers of **panes**, the main one (the only one used in COMP160) is the **content pane** (class Container).

Panel (class JPanel): a container which can only exist as a subpart of another container (such as a frame). It is used to hold and organise other components like buttons and labels. It is a "lightweight" component (managed within Java, no direct links to the underlying OS, graphics use a paintComponent method).

Other components

Button (JButton): registers a mouse press.

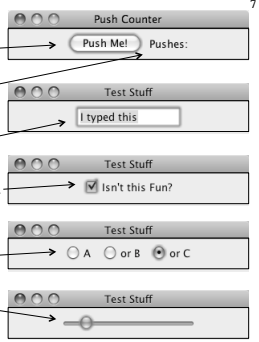
Label (JLabel): displays a line of text.

Text field (JTextField): user can type in text.

Check box (JCheckBox): can be checked or not.

Radio button (JRadioButton): can check (only) one of a group.

Slider (JSlider): input values from a slider.



There are many others that we don't use in COMP160, such as combo boxes, menus, different "panes" in windows, scroll bars, scrollable panes…

Layout

Exact details of how components are arranged are automatic, especially as windows get resized and so on. We have some control using layout managers :

FlowLayout	placed left–to–right then top–to–bottom
BorderLayout	five fixed positions, north, south, east, west, centre
BoxLayout	in a single row or column
GridLayout	within cells of a grid with standard sized rows and columns

If a layout is not specified the default is usually FlowLayout.

The examples on the next slide (using buttons) are constructors for panel classes. To use in a program replace the constructor on Slide 5 with one of these versions.

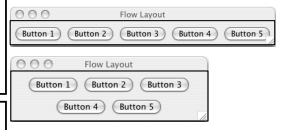
The first example has no explicit layout manager, so the default flow is used. The second example uses border layout. For more see LDC 6.3.


```
public MySupportPanel() { // constructor
  JButton b1 = new JButton("Button 1");
  JButton b2 = new JButton("Button 2");
  JButton b3 = new JButton("Button 3");
  JButton b4 = new JButton("Button 4");
  JButton b5 = new JButton("Button 5");
  add(b1);  add(b2);  add(b3);  add(b4); add(b5);
}
```

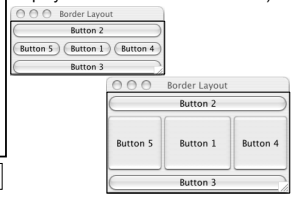
```
public MySupportPanel () { // constructor
  setLayout( new BorderLayout() );
  JButton b1 = new JButton("Button 1");
  JButton b2 = new JButton("Button 2");
  JButton b3 = new JButton("Button 3");
  JButton b4 = new JButton("Button 4");
  JButton b5 = new JButton("Button 5");
  add(b1, BorderLayout.CENTER);
  add(b2, BorderLayout.NORTH);
  add(b3, BorderLayout.SOUTH);
  add(b4, BorderLayout.EAST);
  add(b5, BorderLayout.WEST);
}
```

constants in the BorderLayout class

Example with default flow layout (as displayed in different sized windows)



Example with border layout (as displayed in different sized windows)



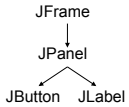
Containment hierarchies

To build a realistic GUI we need to group and organise components using panels (or other containers). The way they are grouped is called a containment hierarchy. The steps (and recommended order) for creating a containment hierarchy are:

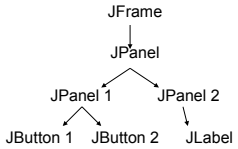
1: Create components
2: Add components to panels
3: Add inner panels (if any) to the main / outer panel
4: Add the outer panel to the (content pane of the) frame (or other window)

There is considerable flexibility in the way actual code can be written as long as overall it follows this process!

The containment hierarchy for the example on slides 4 and 5 is to the right (there are no inner panels).



For example, if we add 2 buttons to a panel, a label to a second panel, and add both panels to a main / outer panel (and that to a frame's content pane as usual)…




```
// replace version on Slide 5 with this
public MySupportPanel() {   // constructor
   JPanel pan1 = new JPanel();
   JPanel pan2 = new JPanel();
   JButton b1 = new JButton("button1");
   JButton b2 = new JButton("button2");
   JLabel label = new JLabel("label");
   pan1.add( b1 );
   pan1.add( b2 );
   pan2.add( label );
   add( pan1 );
   add( pan2 );
   setPreferredSize(new Dimension(300,40));
}
```

Note that by default panels are not visible, but we can color them to see them, e.g.

pan1.setBackground(Color.yellow);

This is an alternative version of the example on the previous slide – it creates the same containment hierarchy and the same output.

The individual operations (lines of code) are in a different order, but for each component and panel the required steps (Slide 10) are in order, so all is well.

I prefer the previous version, make all components then add them.

```
// replace version on Slide 5 with this
public MySupportPanel() {   // constructor
   JPanel pan1 = new JPanel();
   JButton b1 = new JButton("button1");
   pan1.add( b1 );
   JButton b2 = new JButton("button2");
   pan1.add( b2 );
   JPanel pan2 = new JPanel();
   JLabel label = new JLabel("label");
   pan2.add( label );
   add( pan1 );
   add( pan2 );
   setPreferredSize(new Dimension(300,40));
}
```

See the many good example programs in LDC, but don't worry about events until next lecture!

Try this

Write a panel constructor (e.g. to use in a program like Slides 4, 5) that creates the following GUI. Some invisible panels have been outlined…

Inner classes

To prepare for events (next lecture) we need to briefly note inner classes (a feature for safety / encapsulation).

An inner class is declared inside another class (like a member of the outer). Note:

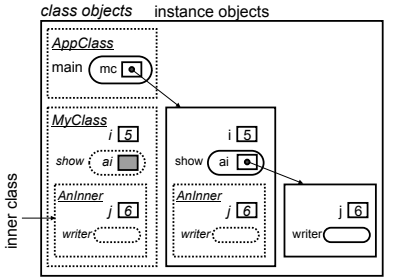
(1) Only the outer can see / create instances of the inner.
(2) Members (e.g. data fields) of outer are in scope for inner (here outer data field i is used by AnInner).

For example, given an application class with a main method containing:

```
MyClass mc = new MyClass();
mc.show();
```

a model is shown on the next slide…

```
public class MyClass {
   private int i = 5;

   public void show () {
      AnInner ai = new AnInner();
      ai.writer();
   }

   // this is an inner class
   private class AnInner {
      private int j = 6;

      public void writer() {
         System.out.println( i + ", " + j );
      }

   }

} // MyClass
```

Output:

5, 6

Interfaces

To prepare for events (next lecture) we need to briefly look at interfaces.

An interface is like a restricted form of a class. They may contain only:

- **abstract** methods: a method with a declaration / header but no body,
- **static final** data fields: a data field which is **static** (a class member) and **final** (may never be assigned a new value).

For example:

```
public interface Stats {
    public static final int DIMENSIONS = 2;
    public abstract double getArea();
    public abstract double getPerimeter();
}
```

A final variable is called a constant, see LDC p46. The naming convention for constants is all CAPS, e.g. Math.PI

Note: in an interface the modifiers abstract, public, static and final are **all optional** because they are assumed by default if they are omitted.

What use is an interface? Other classes may **implement** one or more interfaces. Any class which implements an interface has access to (inherits) the data fields, and must have matching complete methods. (See also abstract classes, Lec 23).

Implementing an interface is like "signing a contract" to provide certain methods.

Interfaces are another way for the language to enforce good **design**.
An interface makes you implement **all** the methods needed to do a task properly.

See the example, next slide…

Interfaces are used a lot in Java graphics. We will be seeing examples like this:

```
public class MyFrame extends JFrame implements WindowListener, ActionListener {
    /* body */
}
```

This class extends (later) JFrame. It implements the interface WindowListener to define all methods required when the window is opened, closed, selected and so on. It implements the interface ActionListener to define all methods required when a button in the window is clicked.

Other examples include the interface **Comparable**, which specifies e.g. that a class must have a compareTo method (implemented by e.g. String); or **Iterator**, specifies methods like hasNext and next (implmented by e.g. Scanner – Lec 11).

This example uses the interface on Slide 16 (which would be in its own file like a typical class).

```
/* Anthony, September 2012, JDK 1.6, An example showing the use of an interface */

public class Square implements Stats {
  private double width = 2.0, height = width;

  public double getArea() {  // the class is required to have a method with this declaration
    return width * height;
  }

  public double getPerimeter() { // the class is required to have a method with this declaration
    return (width * 2) + (height * 2);
  }

  public void showSquare() {
    System.out.println("A Square has " + DIMENSIONS + " dimensions"); // inherited data field
    System.out.println("This one has area " + getArea() + " and perimeter " + getPerimeter() );
    // DIMENSIONS = 3;  // Can't do this, no new value can ever be assigned to this constant.
  }

}
```

Graphics 2
events

COMP160 Lecture 18
Anthony Robins
2014

- Introduction
- Event model
- Finding an event source
- Reading the code

See second notes document for this lecture.

Reading: LDC: 6.1 – 6.3

LDC
3e to (2e)
p252 (p268)
p255 (p269)

Introduction

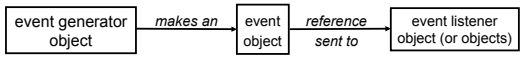
Last lecture we looked at the components of a GUI, how to group and organise them (into a containment hierarchy), how to lay them out, and a design for graphics programs (application class creates a frame with the content pane set to a panel, and the work done in the constructor for the panel object).

Today we look at making the GUI do things – events!

An ordinary program is a *complete specification of a sequence of operations*. When handling input via a GUI a program becomes a *specification of how to handle input events*. This style of programming is called "event driven programming".

Event model

The way a programming language handles events is called its "event model". Java's event model is comparatively simple and tidy!



programmer (1) creates event generator objects / components such as buttons, and (3) "registers" an event listener for each one.

automatically handled when the user generates an event (e.g. clicks a button) - the object holds information about the event.

programmer (2) creates listener objects (using interfaces – mostly from java.awt.event) that must (4) have specific methods that are called to handle input event objects.

Example 1 (Second notes)

Example 1 extends the program Lec 17 Slides 4 and 5, so that it responds to events! This example is almost identical to the LDC PushCounter example Section 6.1 (class names are different).

The application class is on the second handout. Same as version on Lec 17 Slide 4, it just makes the support class frame / window and adds the panel.

The support class is on the second handout. Big changes to Lec 17 Slide 5, the steps for creating the GUI are the same but it has been extended to handle events.

In this example the key steps (Slide 3 Steps 1 – 4) for setting up event handling in MySupportPanel are:

```
push = new JButton("Push Me!");  // Slide 3 Step (1)

push.addActionListener( new MyListener() );  // Slide 3 Steps (2a), (3)

private class MyListener implements ActionListener { // Slide 3 Step (2b)

public void actionPerformed(ActionEvent event) {  // Slide 3 Step (4)
```

ActionListener is an interface. To "implement an interface" a class must have specified methods, in this case an actionPerformed method (with input parameter as shown). Any class can be used as a listener (implement a listener interface), but LDC always use a class within the class containing the event generators, i.e. an **inner class** (Lec 17). This is a common design (see LDC p252).

Event model is the same for all kinds of events

Example generator	The user...	Event	Example listener interface
JButton	clicked a button	ActionEvent	ActionListener
JCheckBox	checked a box	ItemEvent	ItemListener
JScrollBar	moved a scroll bar	AdjustmentEvent	AdjustmentListener
JSlider	moved a slider bar	ChangeEvent	ChangeListener
JWindow	selected a window	FocusEvent	FocusListener
any Component	clicked/moved mouse	MouseEvent	MouseListener
JTextField	entered text	TextEvent	TextListener

For example, in a program handling events from a scroll bar, the four key steps of coding the event model might look like:

```
bar = new JScrollBar();  // Slide 3 Step (1)

bar.addAdjustmentListener( new MyListener() );  // Slide 3 Steps (2a), (3)

private class MyListener implements AdjustmentListener { // Slide 3 Step (2b)

public void adjustmentValueChanged(AdjustmentEvent event) {  // Slide 3 Step (4)
```


Example 2 (Second notes)

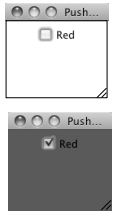
This example builds a GUI with three sliders that are used to set the background colour of the panel displayed in the JFrame. (Java follows a standard convention of representing a colour as a mixture of three primary colours, red, green and blue, with intensity values in the range 0 to 255). The key steps for event handling are…

In SliderPanel:

```
private JSlider rSlider =  new JSlider(**simplified**);  // Slide 3 Step (1)
private JSlider gSlider = new JSlider(**simplified**);
private JSlider bSlider = new JSlider(**simplified**);

 MyListener l = new MyListener();    // Slide 3 Step (2a)
 rSlider.addChangeListener( l );     // Slide 3 Step (3)
 bSlider.addChangeListener( l );
 gSlider.addChangeListener( l );

private class MyListener implements ChangeListener { // Slide 3 Step (2b)

public void stateChanged(ChangeEvent e) {  // Slide 3 Step (4)
```


Try This

Complete code on the next slide to create the GUI and behaviour shown to the right. If the checkbox is selected the color (of the panel) is red, else it is white.



Hints:

The size of the window is 100 x 100.

Use Color.red and Color.white.

Information you need on Slide 6.

The ItemListener interface specifies an itemStateChanged method.

Item listeners are registered with addItemListener()

CheckBox objects have an isSelected() method which returns true if the box is ticked / checked / selected.


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

public class TryThisPanel extends JPanel {
  private JCheckBox box;




}
```

Finding an event source

If many components have one listener we can tell which one generated an event...

Example 3 (see LDC 6.1 p255)

See the LeftRight example from LDC. It displays a text label and two buttons. Clicking on the left or the right button generates an event which is used to set the label text to "Left" or "Right" as appropriate.

What is different about this example is that the listener method (which handles events) has to determine which of the two buttons was pressed.

It finds out by using the getSource method on the event object passed to the listener method. (We have not previously made any use of the event objects). getSource returns a reference to the object that generated the event.

---

getSource may well work a lot like Lec 14 Example 2…

For the code and more discussion make sure you read LDC 6.1 from p255.

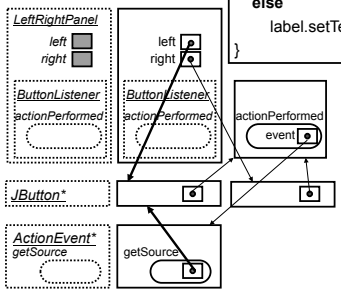
In your reading, try and identify the four steps of setting up event handling (Slide 3).

The following two slides show a very partial model

- when the left button is clicked
- when the right button is clicked

Clicked LEFT button:

```
public void actionPerformed (ActionEvent event) {
   if (event.getSource() == left)
      label.setText("Left");
   else
      label.setText("Right");
}
```



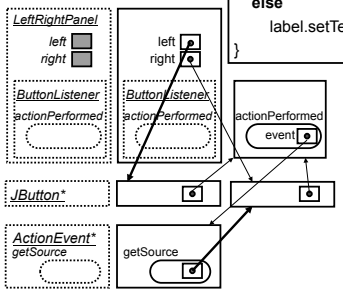
The reference returned by event getSource method and the reference in the data field left are equal (refer to same object)…

…so the condition of the **if..else** statement is true.

We know which button was pushed!

Clicked RIGHT button:

```
public void actionPerformed (ActionEvent event) {
   if (event.getSource() == left)
      label.setText("Left");
   else
      label.setText("Right");
}
```



The reference returned by event getSource method and the reference in the data field left are NOT equal…

…so the condition of the **if..else** statement is false.

We know which button was pushed!

Reading the code

I find it helpful to read graphics code at two "levels":

(1) what it specifies in the GUI (see the *//comments* in this example), and

(2) what it specifies in terms of the processes of the program (see descriptions).

From the application class of Example 1:

frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE); *// specify how to close frame*

On JFrame referred to by the variable frame call the setDefaultCloseOperation method. Pass it as input the value of a (static final) data field called EXIT_ON_CLOSE in the JFrame class object.

frame.getContentPane().add(**new** MySupportPanel()); *// add support class panel to the frame*

On the JFrame referred to by frame call the getContentPane method, which returns a reference to a content pane object. On that object call the method add passing it as input a reference to a newly constructed MySupportPanel object.

Try This

Try the same two level description of the following statements from the support class of Example 1:

add (label);

push.addActionListener (**new** MyListener());

label.setText("Pushes: " + count);

COMP160 Lecture 18 Second Notes

Example 1

```
/*  Version of LDC 6.1 PushCounter.java – class names changed */

import javax.swing.JFrame;

public class MyAppClass {

   public static void main(String[ ] args) {
      JFrame frame = new JFrame ("Push Counter");
      frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
      frame.getContentPane().add( new MySupportPanel() );
      frame.pack();
      frame.setVisible(true);
   }

}
```

Application class is the same as Lec 17 Slide 4:

Changes in support class compared to Lec 17 Slide 5:

Imports classes for event handling

```
/*  Version of LDC 6.1 PushCounterPanel.java – class names changed */

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

public class MySupportPanel extends JPanel {
   private int count;
   private JButton push;
   private JLabel label;

   public MySupportPanel ()   {
      count = 0;
      push = new JButton ("Push Me!");  // Slide 3 Step (1)
      push.addActionListener (new MyListener());  // Slide 3 Steps (2a), (3)
      label = new JLabel ("Pushes: " + count);
      add (push);
      add (label);
      setPreferredSize (new Dimension(300, 40));
      setBackground (Color.cyan);
   }

   // declare inner class, an instance is registered as a listener for the JButton
   private class MyListener implements ActionListener { // Slide 3 Step (2b)

      public void actionPerformed (ActionEvent event) {  // Slide 3 Step (4)
         count++;
         label.setText("Pushes: " + count);
      }

   } // inner class

} // outer class
```

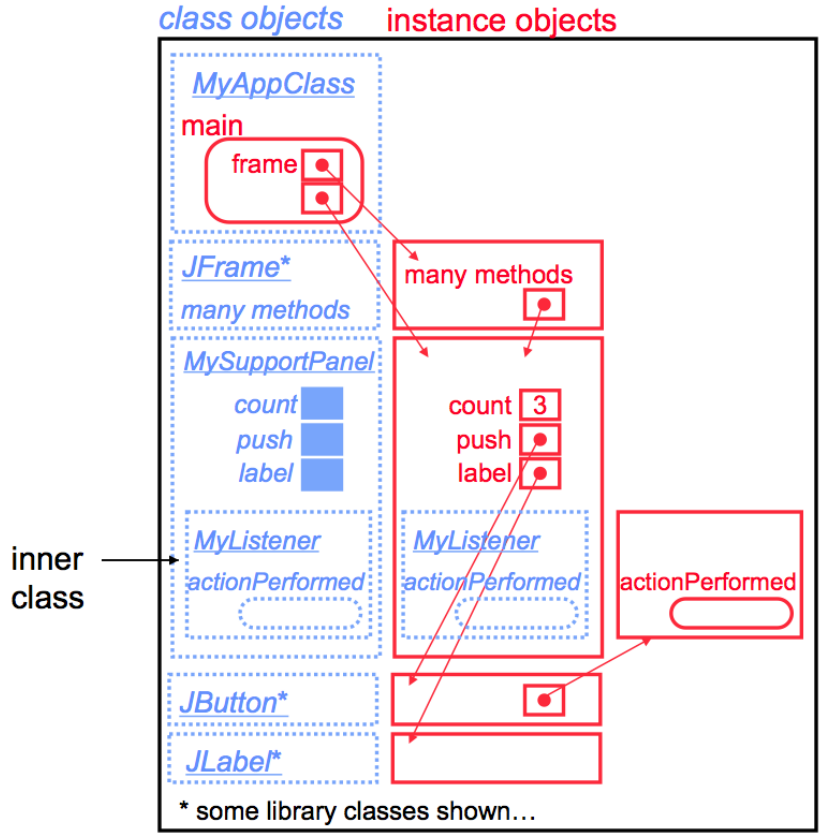
Variables must be data fields to allow access by inner class.

count declared and initialised, used to count button presses.

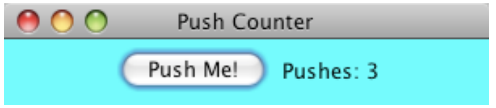
Creates and registers listener object for button

Sets background color of the panel.

Inner class added to make listener object for the button (JButton).



Output after 3 clicks on button:

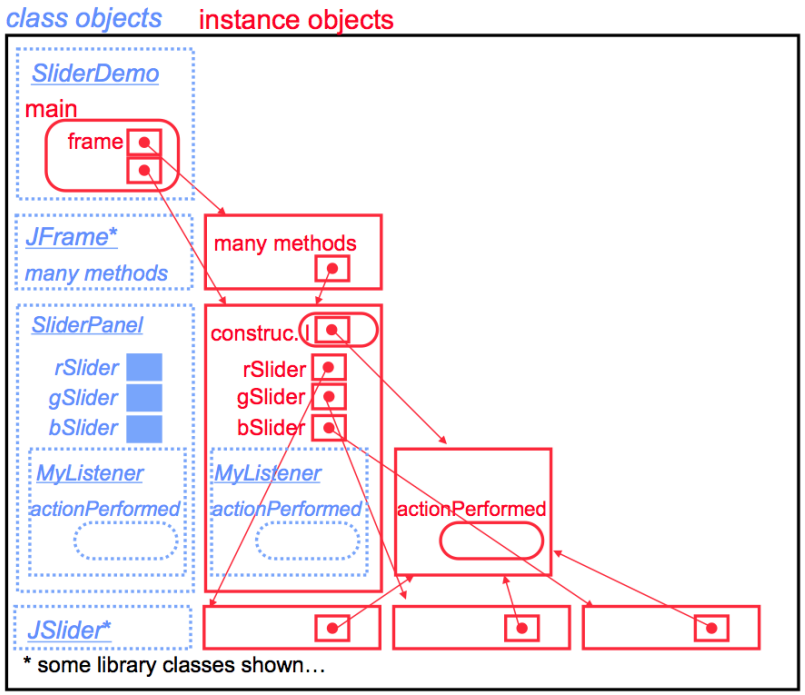


Example 2

```
/* Anthony & Michael Albert, August 2006, JDK 1.5
   Demonstrate events using sliders to set background colour */

import javax.swing.JFrame;

public class SliderDemo {

   public static void main (String[ ] args) {
      JFrame frame = new JFrame ("Slider Demo");
      frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
      frame.getContentPane().add( new SliderPanel() );
      frame.pack();
      frame.setVisible(true);
   }

}
```

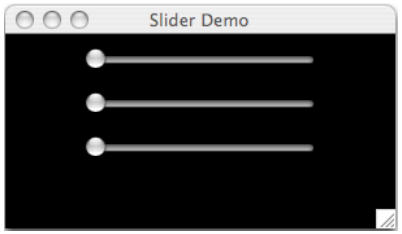
creates a temporary reference to a new object that is not stored in a variable

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

public class SliderPanel extends JPanel {
   // use three sliders to generate values representing red, green, blue
   private JSlider rSlider = new JSlider(JSlider.HORIZONTAL, 0, 250, 0); // Slide 3 Step (1)
   private JSlider gSlider = new JSlider(JSlider.HORIZONTAL, 0, 250, 0);
   private JSlider bSlider = new JSlider(JSlider.HORIZONTAL, 0, 250, 0);

   public SliderPanel () {
      setPreferredSize (new Dimension(300, 150));
      setBackground(new Color(rSlider.getValue(), gSlider.getValue(), bSlider.getValue()));
      add(rSlider);
      add(gSlider);
      add(bSlider);
      MyListener l = new MyListener();  // Slide 3 Step (2a)
      rSlider.addChangeListener(l);      // Slide 3 Step (3)
      bSlider.addChangeListener(l);
      gSlider.addChangeListener(l);
   }

    // declare inner class, an instance is registered as listener for JSlider events
   private class MyListener implements ChangeListener{  // Slide 3 Step (2b)
     public void stateChanged(ChangeEvent e) {          // Slide 3 Step (4)
       // print out the values returned by the sliders (optional!)
       System.out.println("Red: " + rSlider.getValue() + "  Green: "  + gSlider.getValue()
                          + "  Blue: " + bSlider.getValue()     );
       // use the values from the sliders to set the background of the SliderPanel
       // (inner classs accessing setBackground method in the outer class)
       setBackground(new Color(rSlider.getValue(), gSlider.getValue(), bSlider.getValue()));
     }
   } // MyListener

} // SliderPanel
```



Output:

Adjusting the sliders changes the background colour.



Graphics 3 examples

COMP160 Lecture 19
Anthony Robins

- Introduction
- Pizza example
- Graphics and animation
- An alternative design

See second notes document for this lecture.

Reading: LDC: 6.7, Appendix F (6.4 – 6.6 optional / not examined)

LDC 3e to (2e) p277 (p293)

Introduction

More detailed GUI, graphics and animation examples. Let's get on with it!...

Note: "Event model" steps in the comments are from Lec 18 Slide 3.

Pizza example

See the Pizza example program in the second notes. It highlights several topics:

- A complete interactive GUI – selected options create two output strings.
- The use of arrays of (refs to) objects (JCheckBoxes and JRadioButtons) processed by for-each loops to reduce duplication / repetition.
- Event handling for two different kinds of generator components (JCheckBoxes and JRadioButtons).

The program uses five sub panels added to the overall PizzaOptionsPanel. The middle three panels contain check box and radio button components.

Note: JRadioButtons must be assigned to a group so that only the one most recently clicked button in the group can be selected at any time. JCheckBoxes are independent, any combination can be selected or not.

Arrays of components

The program uses arrays of components and for-each loops. The JCheckBoxes, for example. Declaring:

```
private JCheckBox [ ] extras = { new JCheckBox("Mushroom"), new JCheckBox("Onion"),
                                 new JCheckBox("Capsicum"), new JCheckBox("Olives") };
```

Setting up event handling:

```
CheckListener cl = new CheckListener();
for (JCheckBox e : extras) {
   e.addItemListener(cl);    // register listener for check box  (Event model Step 3)
   extrasPanel.add(e);       // add check box to extrasPanel
}
```

Processing events:

```
See next slide.
```

Without arrays / for-each loops
each step (statement) would need to be written n times for n checkboxes.
This array based version is shorter, easier to read, and much easier to maintain…

Event handling

There are similarities and differences between event handling for the radio buttons and for the check boxes…

```
/* this inner class is the listener for all JCheckBoxes   (Event model Step 2b) */
private class CheckListener implements ItemListener {

  public void itemStateChanged (ItemEvent event) {  // Event model Step 4
    extrasString = "";   // must start from an empty string and check each box
    // check boxes have a useful boolean isSelected method!
    for(JCheckBox e : extras) {
      // if this box is selected, get its text and add to the output string
      if (e.isSelected() ) extrasString = extrasString + e.getText() + " ";
    }
    // now set label text
    extrasLabel.setText("Extras: " + extrasString);
  } // end of method

} // end of ButtonListener class
```



```
/* this inner class is the listener for all JRadioButtons (Event model Step 2b)*/
private class ButtonListener implements ActionListener {

  public void actionPerformed (ActionEvent event) { // Event model Step 4
    Object source = event.getSource();  // record source of event (which radio button)
    for(JRadioButton s : size) {              // check all size buttons
      if (s == source) sizeString = s.getText();  // get text from source button and set sizeString
    }
    for(JRadioButton b : base) {              // check all base buttons
      if (b == source) baseString = b.getText(); // get text from source button and set baseString
    }
    // now set label text
    orderLabel.setText("Order: " + sizeString + " " + baseString);
  } // end of method

} // end of ButtonListener class
```


Casting from Object

Consider the following from the previous slide:

```
Object source = event.getSource();   // record source of event (which radio button)
for(JRadioButton s : size) {              // check all size buttons
    if (s == source) sizeString = s.getText();  // get text from source button and set sizeString
}
```

Here we are comparing every reference **s** with **source** to see if they are the same (similar to Lec 18 Slides 12, 13). If we find a match we use **s** to reference the object (and call its getText method). However, if we use the reference **source** directly we don't need the loop at all! Replace the above with:

```
Object source = event.getSource();   // record source of event (which radio button)
sizeString = ( (JRadioButton) source ).getText();  // get text from source button and set sizeString
```

The getSource method returns a reference which is of type Object.

This is a cast operator (Lec 4). Here it casts from the generic type Object into type JRadioButton.

Call the getText method on the object referred to.

Try This

For each statement describe (like Lec 18 Slide 14) *(1) what it specifies in the GUI*, and (2) what it specifies in terms of the processes of the program.

```
headerPanel.setBackground( Color.red );
```

```
add( headerPanel, BorderLayout.NORTH );
```



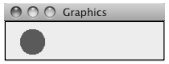
```
if (e.isSelected() ) extrasString = extrasString + e.getText() + " ";
```

Graphics and animation

Graphics

By now you should know how to draw basic graphics in a panel.

Given a usual application class which uses an instance of this panel, the following output is created…




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

public class GraphicsPanel extends JPanel {

   public GraphicsPanel() {
      setPreferredSize( new Dimension(200,50) );
   }

   public void paintComponent(Graphics g) {
      g.setColor(Color.red);
      g.fillOval(20, 10, 30, 30);
   }

}
```


Timers

LDC p277 describe a class which is useful for creating animations (or any other rapid sequence of timed method calls).

Here we show a simple example of text output. Every 1000 msec (1 sec) the timer generates an event and calls the registered listener / method.

Output (new line every 1 sec):

```
Timer called this method.
Timer called this method.
Timer called this method.
```

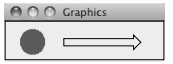
```
/* Anthony, Sept 2010, JDK 1.6, Demo Timer */
import java.awt.event.*;
import javax.swing.Timer;

public class TickTock {

   public static void main (String [ ] args) {
      // make timer which generates action every 1 sec
      Timer t = new Timer(1000, new MyListen() );
      t.start();    // start timer
      // t.stop();  // could be used to stop timer
   }

   private static class MyListen implements ActionListener{

      public void actionPerformed(ActionEvent ae) {
         System.out.println("Timer called this method.");
      }
   }
}
```

Note that this program will keep running until reset / interrupted!

Animation

Combining basic graphics with a timer we can make a simple animation!

The oval is drawn at a position controlled by the data field x. Every 10 msec the timer calls the listener which increases x (moves the circle to the right). If x is near the edge of the panel we reset it to its starting value (20).




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

public class AnimationPanel extends JPanel {
   int x = 20;  // initial x position of oval at left of panel

   public AnimationPanel() {
      setPreferredSize( new Dimension(200,50) );
      Timer t = new Timer(10, new MyListen() );
      t.start();
   }

   public void paintComponent(Graphics g) {
      super.paintComponent(g);  // clears all graphics - later!
      g.setColor(Color.red);
      g.fillOval(x, 10, 30, 30); // draw oval at specified x value
   }

   private class MyListen implements ActionListener {
      public void actionPerformed(ActionEvent ae) {
         x += 1;  // move x to the right
         if (x > 150) x = 20;  // reset x to start if close to right
         repaint();  // automatically calls paintComponent
      }
   }
}
```


Graphical objects

See the example Splat program LDC Appendix F.

This is good preparation for coming labs!

So far we have done drawing / graphics with all drawing code within the paint method (of a JFrame) or paintComponent method (of a JPanel). It is more flexible if we can represent the things to be painted as their own objects.

In the Splat example every circle drawn is represented by its own object (instances of class Circle). These are referred to by data fields in SplatPanel.

This is a very OO approach. Each circle is an object that manages itself, and will draw itself in whatever graphics context you pass it (in this example "page").

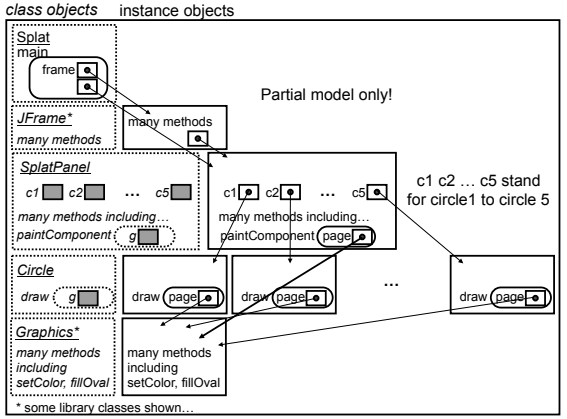
In class SplatPanel the method paintComponent doesn't do any drawing directly. Instead, it calls a method on each Circle object and passes it a reference to the Graphics object for the panel.

```
public void paintComponent(Graphics page) {
   super.paintComponent(page);
   circle1.draw(page);
   circle2.draw(page);
   circle3.draw(page);
   circle4.draw(page);
   circle5.draw(page);
}
```

In class Circle a method has been written to take (a ref to) a Graphics object as input, and to draw the circle / oval in that graphics context. Here: color, x y and diameter are all data fields of Circle, all initialised by the Circle constructor for each instance to create the different circles…

```
public void draw(Graphics page) {
   page.setColor (color);
   page.fillOval (x, y, diameter, diameter);
}
```

Common design:
paintComponent calls a draw method on every object to be drawn and passes it the relevant graphics object / context.



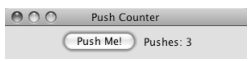
An alternative design

In all our GUI examples so far we have used / assumed the graphics design from Lec 17, where the application class makes a frame and adds a support class panel (where all the real work happens).

That's a good design, but many many others can achieve the same result!

The program on the next slide is a complete working alternative version of the "push counter" program from Lec 18 (Example 1). It:

- has only an application class (which implements the listener interface)
- makes one instance **p** of the application class (used as the listener object and registered with **this** reference to itself – needs the listener method but no separate inner class)
- sets up both a JFrame and JPanel in the constructor for the application class



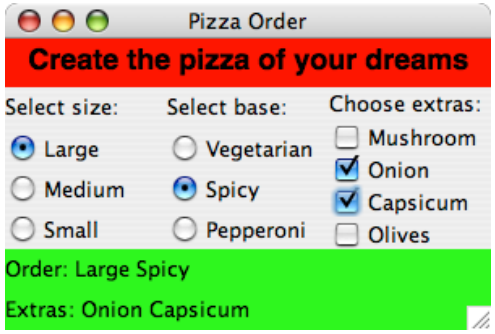

```
import java.awt.*; import javax.swing.*;  import java.awt.event.*;
public class PushAppVariant implements ActionListener {  // Event model Step (2b)
   private int count;  private JButton push;   private JLabel label;

   public static void main (String[ ] args) {          // Makes a single instance of itself
      PushAppVariant p = new PushAppVariant();         // Event model 3 Step (2a)
   }

   public PushAppVariant() { // constructor makes the required JFrame and JPanel, and
      count = 0;             // registers this instance as a listener for a JButton
      // set up a panel
      JPanel pan = new JPanel();
      push = new JButton ("Push Me!");                 // Event model Step (1)
      push.addActionListener ( this );                 // Event model Step (3)
      label = new JLabel ("Pushes: " + count);
      pan.add ( push );
      pan.add( label );
      pan.setPreferredSize ( new Dimension(300, 40) );
      pan.setBackground ( Color.cyan );
      // set up a frame
      JFrame frame = new JFrame ("Push Counter");
      frame.setDefaultCloseOperation ( JFrame.EXIT_ON_CLOSE );
      frame.getContentPane().add( pan );
      frame.pack();   frame.setVisible( true );
   }

   public void actionPerformed ( ActionEvent event ) { // Event model Step (4)
      count++;
      label.setText("Pushes: " + count);
   }
}
```

Sorry it's so cramped!

COMP160 Lecture 19 Second Notes

File PizzaApp.java

```
/* Anthony, Sept 2012, JDK1.6.  Example GUI handling events from checkboxes and
radiobuttons.  Features arrays of (references to) these generator objects. */

import javax.swing.JFrame;

public class PizzaApp {

  public static void main (String[ ] args) {
    JFrame frame = new JFrame ( "Pizza Order" );
    frame.getContentPane().add( new PizzaOptionsPanel() );
    frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    frame.pack();
    frame.setVisible( true );
  }
}
```



file PizzaOptionsPanel.java

```
/* See comments in application class  */

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

public class PizzaOptionsPanel extends JPanel {
  // the data fields are accessible by the constructor and by the inner classes (listeners)
  // declare the event generators (Event model Step 1)
  private JRadioButton [ ] size =  {  // create array of (refs to) JRadioButton objects
    new JRadioButton("Large"), new JRadioButton("Medium"), new JRadioButton("Small") };
  private JRadioButton [ ] base =  {  // create array of (refs to) JRadioButton objects
    new JRadioButton("Vegetarian"), new JRadioButton("Spicy"),
    new JRadioButton("Pepperoni") };
  private JCheckBox [ ] extras = { // create array of (refs to) JCheckBox objects
    new JCheckBox("Mushroom"), new JCheckBox("Onion"), new JCheckBox("Capsicum"),
    new JCheckBox("Olives") };
  // declare the output labels and the strings displayed in them
  private JLabel orderLabel = new JLabel("Order:  None");   // displays baseString,SizeString
  private JLabel extrasLabel = new JLabel("Extras:  None"); // displays extrasString
  private String baseString = "", sizeString = "", extrasString = "";  // strings to display

  /* constructor - creates the panels used for the GUI*/
  public PizzaOptionsPanel() {
    // this panel has layout positions NORTH WEST CENTER EAST SOUTH
    setLayout(new BorderLayout());
    // declare inner panels, one for each position
    JPanel headerPanel = new JPanel();  // to add at NORTH
    JPanel sizePanel = new JPanel();    // to add at WEST
    JPanel basePanel = new JPanel();    // to add at CENTER
    JPanel extrasPanel = new JPanel();  // to add at EAST
    JPanel orderPanel = new JPanel();   // to add at SOUTH

    // set up header panel - to add at position NORTH
    JLabel header = new JLabel();
    header.setFont(new Font("Helvetica", Font.BOLD, 18));
    header.setText("Create the pizza of your dreams");
    headerPanel.setPreferredSize( new Dimension(300,30) );
    headerPanel.setBackground( Color.red );
    headerPanel.add( header );
```

```
// set up size buttons and size panel - to add at position WEST
sizePanel.setPreferredSize( new Dimension(100,100) );
sizePanel.setLayout( new GridLayout(4,1) );      // holds 4 components (4 rows 1 column)
sizePanel.add( new JLabel("Select size:") );      // add label
ButtonListener bl = new ButtonListener();         // make listener (Event model Step 2a)
ButtonGroup sizeGroup = new ButtonGroup(); // buttons in a group are mutually exclusive
for (JRadioButton s : size) {
  s.addActionListener( bl );  // register listener for radio button  (Event model Step 3)
  sizeGroup.add( s );         // add button to group of mutually exclusive buttons
  sizePanel.add( s );         // add button to sizePanel
}

// set up base buttons and base panel - to add at position CENTER
basePanel.setPreferredSize( new Dimension(100,100) );
basePanel.setLayout( new GridLayout(4,1) );     // holds 4 components (4 rows 1 column)
basePanel.add( new JLabel("Select base:") );    // add label
ButtonGroup baseGroup = new ButtonGroup(); // buttons in a group are mutually exclusive
for (JRadioButton b : base) {
  b.addActionListener( bl ); // register listener for radio button   (Event model Step 3)
  baseGroup.add( b );       // add button to group of mutually exclusive buttons
  basePanel.add( b );       // add button to basePanel
}

// set up extras check boxes and extras panel - to add at position EAST
extrasPanel.setPreferredSize( new Dimension(100,100) );
extrasPanel.setLayout( new GridLayout(5,1) );   // holds 5 components (5 rows 1 column)
extrasPanel.add(new JLabel("Choose extras:")); // add label
CheckListener cl = new CheckListener();           // make listener (Event model Step 2a)
for (JCheckBox e : extras) {
  e.addItemListener( cl );   // register listener for check box  (Event model Step 3)
  extrasPanel.add( e );      // add check box to extrasPanel
}

// set up order panel for text output in labels - to add at position SOUTH
orderPanel.setPreferredSize( new Dimension(300,50) );
orderPanel.setBackground( Color.green );
orderPanel.setLayout( new GridLayout(2,1) );   // holds 2 components (2 rows 1 column)
orderPanel.add( orderLabel );                  // add each label
orderPanel.add( extrasLabel );

// add inner panels to this panel at specified positions
add(headerPanel, BorderLayout.NORTH);
add(sizePanel, BorderLayout.WEST);
```



```
  add(basePanel, BorderLayout.CENTER);
  add(extrasPanel, BorderLayout.EAST);
  add(orderPanel, BorderLayout.SOUTH);
 } // end of constructor


 /* this inner class is the listener for all JRadioButtons (Event model Step 2b)*/
 private class ButtonListener implements ActionListener {
  public void actionPerformed (ActionEvent event) { // Event model Step 4
    Object source = event.getSource();   // record source of event (which radio button)
    for(JRadioButton s : size) {              // check all size buttons
      if (s == source) sizeString = s.getText();  // get text from source button and set sizeString
    }
    for(JRadioButton b : base) {              // check all base buttons
      if (b == source) baseString = b.getText(); // get text from source button and set baseString
    }
    // now set label text
    orderLabel.setText("Order: " + sizeString + " " + baseString);
  } // end of method
 } // end of ButtonListener class


/* this inner class is the listener for all JCheckBoxes   (Event model Step 2b) */
 private class CheckListener implements ItemListener {
  public void itemStateChanged (ItemEvent event) {   // Event model Step 4
    extrasString = "";   // must start from an empty string and check each box
    // check boxes have a useful boolean isSelected method!
    for(JCheckBox e : extras) {
      // if this box is selected, get its text and add to the output string
      if (e.isSelected() ) extrasString = extrasString + e.getText() + " ";
    }
    // now set label text
    extrasLabel.setText("Extras: " + extrasString);
  }  // end of method
 } // end of ButtonListener class

} // end of PizzaOptionsPanel class
```

Files
input output, sorting

COMP160 Lecture 20
Anthony Robins

- Introduction
- Streams & tokens
- try..catch
- Files and Input Output (I/O)
- Sorting

Reading: LDC: 2.6, 4.6, 10.1, 10.2, 10.3, 10.6. (optional – look at 13.1, 13.2).
LabBook: "Debugging code", "Writing safe programs", "Java Input and Output", "Locating support files".

Introduction

This lecture takes material from various places in LDC.

We do this to establish the background that we need for labs, and for 200 level.
We deal with two main topics (files, sorting), and relate both to arrays.

Recall that an **array** is a named collection of data (of the same type, of a fixed size).

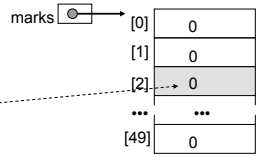
A declaration:

```
int [ ] marks = new int [50];
```

will create an array "object" such as this. ⇨

Individual elements are referred to as e.g.:

```
marks[2]
```



Streams & Tokens

A **stream** is a source or destination of a sequence of data (a standard programming concept). Streams are used as an abstraction for dealing uniformly with files and devices (e.g. the keyboard, console, memory, printers and so on). See the Lab Book reading "Java Input and Output".

Java 1.5 introduced the Scanner class, which (compared to earlier versions of Java) simplifies many aspects of reading streams (hides the details). Scanner is set up to deal with **tokens** (another standard programming concept):

Tokens are text items separated by "white space" - blanks or tabs
(usually, though other separating characters - "delimiters" - can be used)

For example, the tokens in this string are: " This 1 thatone %%%$# bye3. "

The next slide shows examples of a Scanner used to read tokens from a String…


```
Scanner sc = new Scanner("Axe Bag Cat");
System.out.println( sc.next() );
System.out.println( sc.next() );
System.out.println( sc.next() );
```

The next method returns the next token from the scanner as a String.

```
Scanner sc = new Scanner("Axe Bag Cat");
while ( sc.hasNext() ) {
   System.out.println( sc.next() );
}
```

In this example a loop is used to process each token from the scanner (until the hasNext method returns false).

In both cases above the output is:

```
Axe
Bag
Cat
```

Scanner also has many other methods, e.g. nextInt (returns the next token as an int), nextDouble (returns a double), and nextLine (return all the remaining text on the line as a single String). See LDC 2.6, 4.6.

try..catch

Java was designed to be robust – deal with **exceptions** (errors) well. The main mechanism for this is **exception handling** (LDC Ch 10). In many cases there are three Options for dealing with exceptions:

(1) do nothing (if exceptions occur the program will crash)
(2) handle them where they occur with **try..catch** (examples in this lecture)
(3) "throw" them to be handled elsewhere in the program (not in COMP160)

In some cases (e.g. code working with files, a common source of errors) the compiler forces us to deal with possible exceptions (Option 1 is not allowed, the program will not compile). In COMP160 we use Option 2, **try..catch**.

```
try {
   // code that might cause an exception
} catch (ExceptionClass variable) {
   // warn the user or try to handle the exception
}
```

The catch block gets as input a (reference to an) object of some exception class / type. Call methods on the object to find out more about the exception.

For example, the statement "**int** i = 10 / 0;" will cause a program to crash, unless it occurs in a **try..catch**.

```
try {
     int i = 10 / 0;    // deliberate error
} catch (ArithmeticException z) {
     System.out.println("There was a problem with your maths!");
     System.out.println( z );   // calls toString on z
}
```

```
There was a problem with your maths!
java.lang.ArithmeticException: / by zero
```

Instead of crashing, it prints out a message, and the program carries on.

try..catch can be used to make programs safer. See the two versions of a readInt method discussed in the Lab book reading "Writing safe programs".

Files

Programs need to be able to store information in (read from / write to) **files**. Files can save data (persistence) and hold lots of it! Reading data from files into arrays is a very common task.

We will deal only with **text files**, which can be thought of as storing a sequence of strings (one string per line).

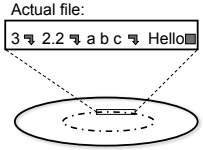
We will use a **Scanner** object to read data from a file (one kind of stream). We have seen similar examples, using a Scanner to read from System.in or a String.

Scanner is an **iterator** (it **implements** the Iterator **interface**, Lec 11). Like other iterators (LDC 4.6) it has methods for dealing with collections of data (in this case sequences of tokens).

Conceptually:

```
"3"
"2.2"
"a b c"
"Hello"
```

Actual file:



Reading from a file (LDC 4.6)

Create a **Scanner** for the file. This can fail (e.g. if the file can't be found) and generate an IOException, so the compiler forces us to handle this in some way (Slide 5).

The example in LDC 4.6 deals with this weakly, by throwing the IOException (Option 3) from the main method (same effect as not handling it, exception = crash).

The following examples use **try..catch** (Option 2) for exceptions. Data files must be in the same directory as the program (see Lab Book reading "Locating support files").

If the scanner is created successfully we can use its methods to read from the file / stream. Example 1 reads individual tokens. Example 2 reads each complete line as a String, then extracts a certain token (an int value) from each line / string, then saves it (in an array of int).

Example 1

Creates a **Scanner** for the file. Any problems will generate an exception and pass control to the first **catch** block.

Then calls methods to read in a particular sequence of data values (double int int int). Any other sequence will generate an exception and pass control to the second **catch** block.

File "data1.txt".

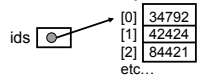
```
2.3
4 7
9
```

```
// this code could occur in a method of a class which
// imports java.util.* (includes Scanner) and java.io.*
double d;  int i1, i2, i3;
try {
   Scanner sc = new Scanner(new File("data1.txt") );
   d = sc.nextDouble();  // read a double (2.3)
   i1 = sc.nextInt();       // read an int (4)
   i2 = sc.nextInt();       // read an int (7)
   i3 = sc.nextInt();       // read an int (9)
} catch (IOException ioex) {
    // some problem with the file (not found?)
    System.out.println("File problem!: " + ioex );
} catch (InputMismatchException imex) {
     // a data value was not as expected
     System.out.println("Unexpected data!: " +  imex );
}
```

Example 2

Assume a file where each line is a name, id number, and lab marks (max of 50 lines). This example will process every line, reading the id numbers into an array.

- create the array (max of 50)
- creates a Scanner for the file
- checks to see if the file has a next token, and if it does…
- creates a Scanner for the line of the file (treated as one string)
- throws away first token (name)
- reads the second token as an int and stores it in the array.



File "data2.txt".

```
Anne 34792 1 1 1 1 0
Barry 42424 1 1 0 0 1
Craig 84421 0 0 0 1 1
etc…
```

```
// imports as for Example 1
int [ ] ids = new int [50];  // declare array
try {
   Scanner sc = new Scanner(new File("data2.txt") );
   int i = -1;  // initialise index for array
   while ( sc.hasNext() ) {
      Scanner line = new Scanner( sc.nextLine() );
      i++;  // increment array index (first time sets to 0)
      line.next();  // read / discard first token
      ids[ i ] = line.nextInt(); // read next token as int
   }
} catch ( IOException ioex ) {
   // some problem with the file (not found?)
   System.out.println("File problem!: " + ioex );
} catch ( InputMismatchException imex ) {
   // a data value was not as expected
   System.out.println("Unexpected data!: " + imex );
}
```


Try this

Assume a file data3.txt holding an unknown number of integer values. Write a **try** block which, for every integer in the file, will read the integer, save it in a variable, and print out the variable. (Assume **catch** blocks as for the previous examples).

Writing to a file (LDC 10.6)

We won't be writing to files in COMP160, but you should be aware of how it's done. See an example, LDC 10.6. **Scanner** cannot be used to write to files, so this example uses more basic **Writer** objects. As above, this LDC example throws the IOException from main, using a **try..catch** would have been safer!

Always close a file when you are finished writing – in this example "outfile.close();" – or you may lose data (or in rare cases damage your file system).

Sorting

At the heart of Computer Science are two (related) topics:

Searching
Sorting

Why are they related?

Imagine you had a huge list of unsorted data, and were looking for a particular item. What's the **worst** number of items you would have to check? What's the **average** number of items?

Now imagine the data is sorted. What's the **worst** number of items you have to check now? (Think telephone book...)

Other reasons to sort data: finding medians, finding successors and predecessors, displaying data (e.g. in library systems), lots more...

So sorting is not a "Java" topic it's a computer science topic – the kind you meet at 200 level. (And it's great for illustrating the use of arrays!).

There are many algorithms for sorting a collection of data into order. They have different strengths and weaknesses. Examples include:

bubble sort
keep swapping smaller neighbours with bigger ones

selection sort
keep putting the smallest thing you can find at the start

merge sort
split into two sub-lists, sort them, then merge the two sorted lists

insertion sort
take the next thing and put it into order with things you already sorted.

Different sorting algorithms have different properties. Compare for example **selection sort**, n^2 comparisons, and **merge sort**, **n(log n)** comparisons.

items	selection sort	mergesort
1000	1 second	1 second
2000	4 seconds	2.2 seconds
3000	9 seconds	3.5 seconds
4000	16 seconds	4.8 seconds

Imagine what happens at 10000 or 100000 items!

Is merge sort better? Maybe not - it has its own costs, it requires **an extra array** for temporary storage.

Selection sort

To sort a sequence of anything (e.g. an array of integers):

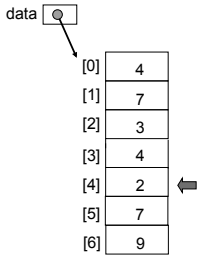
1. Find the smallest value (Slide 16) and swap it with the "top" vaue (Slide 17)
2. Move where you think the "top" is one step down
3. Repeat until there's nothing left at the "top" (reached end of the list)

Selection sort is simple; but it has weaknesses…

Selection sort has to work hard. It has to do lots of comparisons to find the smallest item each time. For **n** items, it does about n^2 comparisons.

Try this

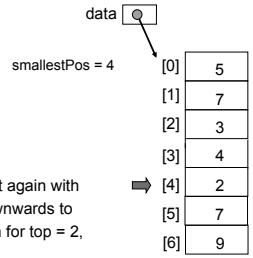
The core of selection sort is to keep finding the smallest value. Given an array **data**, write code that finds the **position** of the smallest item in the array (in this e.g. array position 4) and stores it in a variable. You need a **for** loop, and an **if**…



Try this

Now that you've found the smallest item, write code that swaps that value with the value stored in position 0 the top (the start - Slide 15).

int top = 0;



To sort the whole array (selection sort), start again with top = 1 and repeat this process (search downwards to find the smallest, swap with top), then again for top = 2, and so on…

Notes

- Selection sort has the same characteristics regardless of which language it is written in, or on what machine. Its properties are a result of the nature of the algorithm.
- Is selection sort useless since it doesn't scale up well? No! It is faster than "better" sorting methods on **small enough** arrays, which can be useful.
- This kind of topic is covered in **COSC242**, the data structures and algorithms paper.
- Sorting is a common exercise in computer science, but in practice there is usually no need to reinvent the wheel - Java supplies methods for sorting collections (Lec 24). E.g. for an array called data: Arrays.sort(data)

Hierarchies, Inheritance.

COMP160 Lecture 21
Anthony Robins

- Overview
- Hierarchies
- Extends
- Java class hierarchy
- this
- super

Reading: LDC: 8.1, 8.3.
LabBook: Object-Oriented Design

Overview

In Chapter 8 we move back to the overall design and organisation of a Java program.

In broad design terms these topics relate to **abstraction** – modelling a task / problem by representing the vital information and its relationships in our program structure.

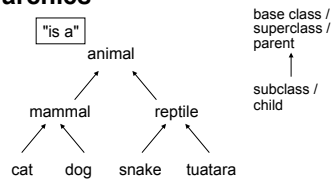
We start with a basic tool for organising information, the hierarchy. Hierarchies can prevent unnecessary work / duplication (one aspect of the "software reuse" that is a supposed advantage of OO languages)…

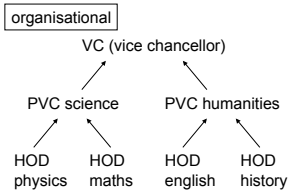
(The lecture contains lots of versions of classes Person and Student, the different versions illustrate different points).

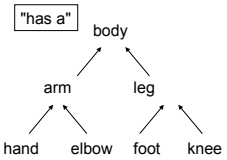
Hierarchies

Here are examples of three kinds of hierarchy:

Hierarchies are often called "trees" and the top element the "root".







Extends

As we have already seen, a class can extend one (only one) other, e.g.

MyFrame **extends** JFrame

In the past I have said MyFrame "is a version of" JFrame. Now in more detail…

We can **extend** almost any existing class (the base class, superclass or parent) to create / define a new class (the subclass or child). A child class has its own members (data fields & methods), and also the (non **private**) members of its parent class that it **inherits**.

If a class does not explicitly extend another class, then by default it extends the class called Object. In other words, the following class definitions are equivalent:

```
public MyClass {
    // class body
}
```

```
public MyClass extends Object {
    // class body
}
```

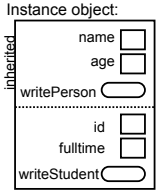

Extends

Examples in this lecture use public visibility, which is not ideal. See more on visibility in Lec 22…

The superclass Person has a subclass Student.

This is defined by the use of the keyword **extends** in the subclass header.

Instance object:




```
public class Person {

    public String name;  // private data fields would
    public int age;      // not be inherited.

    public void writePerson( ) {
        System.out.println("A person " + name + " " + age);
    }
}
```

```
public class Student extends Person {

    private int id;
    private boolean fulltime;

    public void writeStudent( ) {
        System.out.println("A Student "  + name + " " + age + "\n"
                           + id + " full time: " + fulltime);
    }
}
```

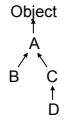

Try this

Write a class Staff that extends Person. Staff should have data fields int phone and String department, and a method writeStaff. Sketch a model of a typical instance object showing all members (data fields, methods).

Design

Even this small example shows how this kind of organisation / inheritance can save work. To change the details of all examples of Person (e.g. add a data field) I only have to change one class in the program (Person). All the subclasses (e.g. Student, Staff, Farmer, Technician, Politician…) automatically reflect the change.

Java class hierarchy



Extends creates a hierarchical relationship between classes.
For example, B and C might extend A, D might extend C, and so on.

Every class is part of the "Java class hierarchy" somewhere!

> Every class extends Object, or extends some other class that (extends another class that) extends Object. Hence Object is the "root" of the Java class hierarchy.

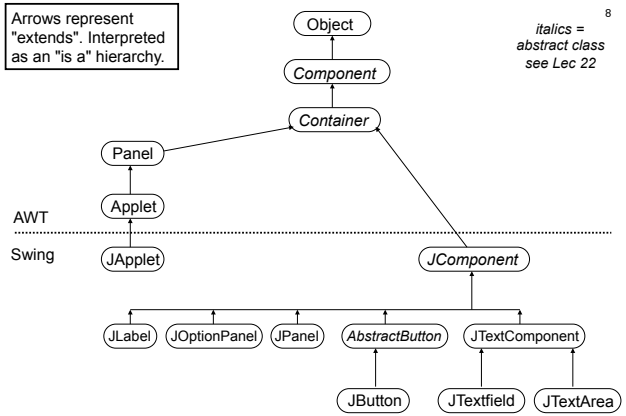
A class inherits from its parent, and its parent's parent (and so on). So D inherits members from C and A (and Object). This is how objects get methods, such as toString, that aren't declared in their own class body.

The next slide shows a small part of the class hierarchy – some of the classes in the graphics libraries (the older AWT library, and the more recent Swing, see Lec 5).

(Do not confuse class hierarchies and containment hierarchies (Lec 17)!)

Arrows represent "extends". Interpreted as an "is a" hierarchy.

italics = abstract class see Lec 22



Subclasses are also subtypes

An object is a legal instance of its own class / type, and its parent's. In the student example (Slide 5) a Student object is of types Student, Person and Object.

```
Student s1 = new Student( );   // as usual
Person p1 = new Student( );    // this is new!
```

In fact, an object is also a legal instance of its parent's parent, and so on up the hierarchy to Object. (Every instance object is a legal example of type Object).

Try this

Using the hierarchy on the previous slide, what are the types of an object constructed from the class Applet ?

this

Recall (Lecture 14): **this** is a reference to the current (instance) object.

Within a class, we can optionally refer to members of the class using **this**.

A common use is to allow the passing of parameters with the same name as a data field.

```
public class Person {
   public String name;
   public int age;

   // constructor
   public Person (String name, int age) {
      // this.name is data field, name is local variable
      this.name = name;
      // this.age is data field, age is local variable
      this.age = age;
   }

   // methods in class could call this method with
   // writePerson() or this.writePerson()
   public void writePerson( ) {
      // no local variables, name and age ref. to data fields
      System.out.println("A person " + name + " " + age);
   }
}
```


this can also be used to call other constructors for an object, as shown here.

The constructor called will be the one with matching parameters (same number, type and order, the parameter names are not relevant).

We save work / duplication:

```
this.name = name;
this.age = age;
```

only a bit in this example! Also ensures that setting name and age is always done in one place / the same way.

```
public class Person {
   public String name;
   public int age;
   public int height;  // in cms

   // constructors
   public Person ( ) { }  // replaces default, Lecture 9

   public Person (String name, int age) {
      this.name = name;
      this.age = age;
   }

   public Person (String name, int age, int height) {
      this(name, age);
      this.height = height;
   }

   // other method(s)
   public void writePerson( ) {
      System.out.println(name + " " + age + " " + height);
   }
}
```

calls matching

such a call must be the **first statement**

With multiple constructors we can make objects with whatever information we have, which may not be complete information.

For Person on the previous slide:

```
Person fred = new Person("Fred", 21);          // partial information
Person mary = new Person("Mary", 23, 172); // complete information
fred.writePerson( );
mary.writePerson( );
```

Output:

height was not initialised so default value

```
Fred 21 0
Mary 23 172
```

Try this

Given Person on Slide 11, assume further data fields, int weight, and boolean married. Write a new constructor for the class to deal with initialising all five data fields, e.g.

```
Person boris = new Person("Boris", 29, 178, 87, true);
```

super

super is a keyword (similar to **this**) referring to members of the current class / object inherited from the parent / superclass. For example, Slide 5, within the writeStudent method: - name and super.name are equivalent,
- writePerson() and super.writePerson() are equivalent.

One common use of **super** is to call constructors (similar to **this**). Same reasons:

Don't: duplicate the work of setting data fields. It is harder to maintain (changing the way a data field is set means changing every duplicate copy).

Do: use **super** and **this** so that in general the statement setting a data field is in one constructor, and called by others if necessary. Keep initialisations in the class where the data is declared (cohesion).

General design principle - don't duplicate!
Use the method, constructor or data field where the work is already done. It's easier and it's safer.


```
public class Person {

   private int someData;
   public String name;
   public int age;

   // constructors
   public Person ( ) { }   // replaces default, Lecture 6

   public Person (String name, int age) {
      this.name = name;
      this.age = age;
   }

   // other method(s)
   public void writePerson( ) {
      System.out.println("A person " + name + " " + age);
   }

}
```

Such replacements may be needed for constructor chaining (Slide 17). If a sub-class tries to call the default automatically, and this replacement doesn't exist, error: cannot resolve symbol : constructor Person ()


```
public class Student extends Person {

   private int id;
   private boolean fulltime;

   public Student (String name, int age, int id, boolean fulltime) {
      super(name, age);
      this.id = id;
      this.fulltime = fulltime;
   }

   public void writeStudent() {
      writePerson();  // or super.writePerson( )
      System.out.println("A Student " + id + " full time: " + fulltime);
   }

}
```

Better design than:

```
this.name = name;
this.age = age;
```

Constructor chaining

A call to **super()** must be the first statement in a constructor. Same for a call to **this()**. So you can have one or the other, but not both!

When a constructor is executed:
- the first statement may call a super constructor (as on the previous slide), or
- there is an automatic call to the default constructor of its parent – **super()**.

So every object is constructed by calling its own constructor, which calls (possibly automatically) a parent constructor, which calls a parent etc. up the hierarchy to the Object constructor – so objects are properly initialised. This is **constructor chaining.**

See the simple example below, and note on Slide 15.

```
public class Hello {
   public static void main(String [ ] args) {
      B b = new B();
   }
}
```

A here
B here

```
public class A {
   public A () { System.out.println("A here"); }
}

public class B extends A {
   public B () { System.out.println("B here"); }
}
```


A case similar to constructor chaining occurs where classes call methods in their parent class to set some state correctly, for example calling the parent's paint / paintComponent method to get a clear drawing component (Lec 19 slides 12, 14).

Try this

Given the classes on Slides 15 & 16, we may sometimes need to construct a Student object with partial information, e.g. name, age and id. Write a new constructor for this case. Sketch a model showing the data fields and methods (but not constructors) of a Student object.

Visibility. Overriding.

COMP160 Lecture 22
Anthony Robins

- Building safe programs
- Packages
- Visibility

- Revising this and super
- Method overloading
- Method overriding

Reading: LDC: 8.2, 8.4, 3.3, 5.8, Appendix E.

Building safe programs

Java has mechanisms at many levels to keep programs well organised and data safe from accidental access.

Large programs / teams	**packages**	Group related classes together (in the libraries / APIs – Lecture 3).
	classes / visibility	Set visibility of data fields. Abstraction. Encapsulation. Information hiding.
Small programs / individuals	**methods / scope**	Organise processing. Variables only within a method (scope, Lecture 10).
	blocks { } / scope	Organise processing. Can define local variables in blocks within methods (scope).

Packages

A **package** is a related group of classes. Every class is part of a package, such as java.lang or javax.swing. Classes that you write and don't put in a named package (all ours so far) are put in the **default package**. Packages:

- help support some OO concepts in Java, because classes in the same package have special access to each other
- involve some practical details about managing classes and files
- define a "namespace" for the classes they contain.

A library consists of one or more packages. Packages in the standard Java libraries / APIs include:

java.lang	Core classes like System, Math, String. ← automatically imported
java.io	Classes for handling data input and output.
java.util	General "utility classes" like Arrays, Random.
java.net	Classes for networking and programming for the internet!
javax.crypto	Classes for encrypting and decrypting data.

Practical details

The package (if not default) to which a class belongs is stated at the top of the file.

package java.lang; *// standard packages are already named*

package com.sun.name;*// new packages name with "reverse" internet address*

The classes in a package must be stored in a single directory / folder which **should** have the same name as the package.

A program can use a package in any directory that is included in the **classpath** (tells the compiler which directories to search, see Lab Book reading "Locating Support Files"). The Java libraries and the current directory (with your source files) are automatically in the classpath, so...

> The easiest way to use a package (not already in the libraries) is to put the directory containing it in the same directory as your source files.

Namespace

There are two ways for code in one class to refer to another class:

- using its **fully qualified** name – e.g. java.util.Random
- using its **simple** name - e.g. Random

You can use the simple name if:

- the other class is in the same package, or
- your source file imports the other class (e.g. java.util.* or java.util.Random), or
- the other class is in java.lang (which is imported automatically).

> Importing a package just saves us the bother of typing out the fully qualified name every time we want to use a class in the package.

Visibility

Visibility rules describe which members the users of a class / object can access. See LDC 8.4 and Appendix E. Here is a summary:

Visibility	Applied to members of a class
private	accessible only within this class
package nothing specified means the default, package / friendly.	accessible only to classes in this package
protected	accessible to classes in this package and any class **extending** this one (last lec.)
public	accessible to all other classes (the class must also be public).

E.g. if a data field is declared: **int** i; *// has package visibility*

Classes also have visibility: **public** classes are accessible to all other classes, **package** classes are accessible to other classes in the package.

Package is the default because classes in a package are assumed to be well designed / tested to work together. If you start to use package visibility it is time to explicitly define and use packages.

> **Most users of a class should have access only to public members, usually public methods** (with access to data fields via methods).

In this way classes can only be used the way they were designed (and tested), not directly accessed in unexpected ways and broken (see example Lecture 6 Slide 11). This is one of the benefits of the OO approach! It promotes "software reuse".

These practical issues relate to a couple of general OO design concepts…

Abstract data type (ADT)

A general OO term for a data type defined solely through operations for creating and manipulating values of that type. All internal structure is hidden from the user. In Java terms we can use classes as abstract data types by allowing access only via public methods.

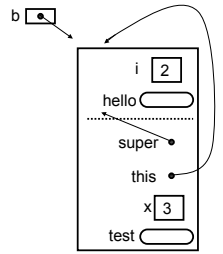
Interface

A general OO term for the public methods of a class. Only the interface needs to be understood / used / explained. Everything else is hidden / safe (encapsulation, information hiding, ADT). (Note: Java also uses the term "interface" in a specific way, see Lecture 17).

Revising this and super

Try this

Given an object b of type B, what is printed out when we call the b.test() method?




```
public class A {
   int i = 2;  // default "package" visibility

   public void hello( ) {
      System.out.println("A hello");
   }
}

public class B extends A {
  int x = 3; // default "package" visibility

  public void test( ) {
    System.out.println( x );
    System.out.println( this.x );
    System.out.println( i );
    System.out.println( this.i );
    System.out.println( super.i );
    hello();
    this.hello();
    super.hello();
  }
}
```

Method overloading

A method's **signature** is just a term for its name and formal parameter specification (number type and order of parameters).

```
public void aMethod( int i, double d ) { };   // signature "aMethod int double"
```

Methods are identified by their **signature**.

In other words, a class can have multiple methods with the same **name** as long as they have different parameter specifications (different **signatures**). This is called **method overloading** (LDC 5.8). See e.g. Lab 13, multiple Box constructors.

In design terms method overloading lets us make the structure of a program clear, by giving different methods that do the same task the same name.

In practical terms, when we call an overloaded method, the particular method that we execute is the version with the matching signature. (If there is no matching signature we get a compile time error).

Try this

The method go is overloaded. Given:

```
Stuff s = new Stuff();
s.go(1, 2.5);
s.go(3.0, 5);
s.go("20");
```

What is the output?

```
public class Stuff {

    public void go(int i) {
        System.out.println("Sum is input " + i);
    }

    public void go(int i, double d) {
        System.out.println("Sum is " + (i + d) );
    }

    public void go(double d, int i) {
        System.out.println("Sum equals " + (i + d) );
    }

    public void go(String s) {
        System.out.println("Number " + s );
    }

    // can't have duplicate signature
    // public void go(String s1) { …
}
```


Method overriding

Method overriding is when methods with the same **signature** exist in a **parent class** and a **child class** (child has access to the parent class members! – last lecture).

```
class A {
    public void fred(int i) { /* stuff */ }  // overridden method
    public void jane( ) { /* stuff */ }
}
```

```
class B extends A {
    public void fred(int t) { /* stuff */ }  // overriding method
    public void jane(char c) { /* stuff */ }
}
```

Method jane has the same name, but different signatures, in the parent and the child class. This is neither overloading nor overriding, just weak polymorphism (Lecture 24).

In an object of the child class (B) we now have two methods with the same "fred int" signature!

But we can still tell them apart. Within any other method in B:

```
fred(5);  // this class
```

and…

```
super.fred(5) // inherited
```


Try this

Identify the examples of overloading and overriding in these classes:

```
class Person {
  public Person ( ) { }   // constructor
  public Person (String name) { }   // constructor
  public void walk( ) { }
  public void printInfo( ) { }
}
```

```
class Adult extends Person {
  public Adult (String name, String address) { }
  public void walk (String destination) { }
  public void walk (String destination, int speed ) { }
  public void printInfo( ) { }
}
```

Overloading (within a class):

Person in Person

Overriding (child and parent):

Accessing overridden methods

"Internally" (from a method in the class) we can access the overriding method and the overridden method (using super). If the test method is called:

```
Greetings
A hello
```

"Externally" (e.g. from a main method that creates an instance b of type B) we can only access the overriding method.

```
b.hello();
```

writes out:

```
Greetings
```

```
public class A {
    public void hello( ) {  // overridden
        System.out.println("A hello");
    }
}
```

```
public class B extends A {
    public void hello( ) {  // overriding
        System.out.println("Greetings");
    }
    public void test( ) {
        hello();  // calls overriding
        super.hello(); // calls overridden
    }
}
```


Accessing shadowed data fields

When data fields have the same name, it is called **shadowing**. The consequences are very similar to overriding.

"Internally" (from a method in the class) we can access the shadowing data field and the shadowed field (using super). If the test method is called:

```
3
2
```

"Externally" (e.g. from a main method that creates an instance b of type B) we can only access the shadowing field.

```
System.out.println( b.i );
```

writes out:

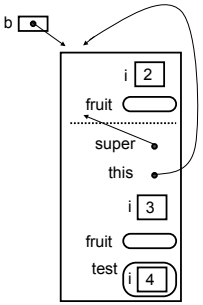
```
3
```

```
public class A {
    int i = 2;   // shadowed
}
```

```
public class B extends A {
    public int i = 3;   // shadowing
    public void test( ) {
        System.out.println( i );
        System.out.println( super.i );
    }
}
```


Try this

Given an object b of type B, what is printed out when we call the b.test() method?




```
public class A {
  int i = 2;  // shadowed data field

  public void fruit( ) {  //overridden
    System.out.println("Apple");
  }
}
```

```
public class B extends A {
  int i = 3; // shadowing data field

  public void fruit( ) {  //overriding
    System.out.println("Banana");
  }

  public void test( ) {
    int i = 4;
    System.out.println( i );
    System.out.println( this.i );
    System.out.println( super.i );
    super.fruit();
    fruit();
  }
}
```

compare with i on Slide 9

Why bother?

What use is overriding / shadowing and the use of **super** and **this**? Why not just avoid the complication by giving everything different names in child and parent?

A similar argument to the use of multiple constructors calling each other (Lec 21) or the use of method overloading (Slide 10). **Because these features help us design well structured programs.** If methods do the same job they should have the same name. Methods should be able to easily call related methods. Data and the methods that operate on it should be kept together in one class (cohesion).

Hierarchies. Abstract classes.

COMP160 Lecture 23
Anthony Robins

- Hierarchies
- Polymorphism
- Abstract classes
- Multiple inheritance
- final

See second notes document for this lecture.

Reading: LDC: 8.3, 8.5, Appendix E.

Hierarchies

Recall (Lec 21) that subclasses are also subtypes (so class hierarchies are also type hierarchies). For example, given:

class Shape
class Square **extends** Shape

every Square object is also a legal Shape object…

Shape s = **new** Square( ); *// an object of type Square is also of type Shape*

instanceof

This is a useful operator which returns true if an object is an instance of a class (or implements an interface). The following are all true:

s **instanceof** Square s **instanceof** Shape s **instanceof** Object

An example use:

```
if (s instanceof Shape)
    // process s…
```


Matching signatures in a hierarchy

Recall that a method's signature is its name, and the number, type and order of its parameters. Methods with the same name can exist in:

- the same class with different signatures (**overloading**)
- a class and its parent or other superclass with the same signature (**overriding**)
- a class and its parent with different signatures (weak polymorphism, Slide 7).

> When a method is called the specific version of the method which is executed is the version with a signature that matches the signature of the call.

If a method with a matching signature cannot be found in the declarations of a class, the compiler looks in the parent class, and so on up the hierarchy, to see if it can find a matching signature. This mechanism is similar to the handling of overriding and shadowing (last lecture).

Try this

```
class A {
    public void polly(int i, int j, int k) {
        System.out.println("polly A 1 here");
    }
    public void polly(int i, double d) {
        System.out.println("polly A 2 here");
    }
} // A
```

```
class B extends A {
    public void polly(int i, double d) {
        System.out.println("polly B 1 here");
    }
} // B
```

```
class C extends B {
    public void polly(int i, int j) {
        System.out.println("polly C 1 here");
    }
} // C
```

Given this class hierarchy, what is printed out / happens for the following code in main:

```
C c = new C();
c.polly(1, 5);
c.polly(3, 4, 6);
c.polly(1, 5.4);
c.polly("Hello", 1);
```

Note: sometimes automatic casts (e.g. int to double) can make this even more complicated.

Class/static members in a hierarchy

Recall (Lec 13) that members are instance members (part of instance objects) unless the modifier **static** specifies that they are class members (part of class objects).

Class data fields are **shared** by / have the same value for all instances of a class (useful in some aspects of design).

Class methods can only access class data.

Class members **already exist** as part of any class object when a program starts running. They are independent of any instance objects.

You should refer to class members with ClassName.name .

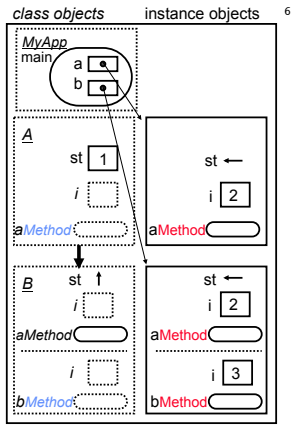
In a hierarchy class members are shared by all instances, includes all instances of subclasses…


```
public class A {
    static int st = 1;  // class member
    int i = 2;

    public void aMethod( ) {
        System.out.println("a: " + st + " " + i);
    }
}
```

all data fields have package visibility

```
public class B extends A {
    int i = 3;

    public void bMethod ( ) {
        System.out.println ("b: " + st + " " + i);
    }
}
```

In e.g. main method:	Output:
A a = **new** A( ); B b = **new** B( );	
a.aMethod();	a: 1 2
b.bMethod ();	b: 1 3
b.aMethod ();	a: 1 2
System.out.println(A.st);	1
System.out.println(B.st);	1
System.out.println(a.st);	1
System.out.println(b.st);	1



Polymorphism

Polymorphism ("many shapes") is a central concept in OO theory. There is general agreement that the polymorphism refers to:

> "the ability of objects belonging to different types to respond to method calls of methods of the same name, each one according to an appropriate type-specific behaviour. The programmer (and the program) does not have to know the exact type of the object in advance, so this behavior can be implemented at run time (this is called late binding or dynamic binding)." (Wikipedia)

See for example LDC 9.1, 9.2 (optional readings!).

There is less agreement about the use of more specific terms to describe specific kinds of polymorphism. We have looked at two examples, **overloading** and **overriding**. Other cases (e.g. where methods have the same names but different signatures in a parent and a child class) are sometimes called weak polymorphism or polymorphism across classes. But there is some confusion in the literature.

Abstract classes

Abstract classes are ordinary class in most ways. They are used when you want to create instances of subclasses, but **not** the abstract parent / superclass class itself. For example, an abstract class Shape where we only create instances of subclasses like Circle, Square, Triangle…

Abstract classes are similar to interfaces (Lec 17) in that declaring an abstract class Fred (or an interface Fred) lets you:

- treat its subclasses as all being of type Fred
- ensure that every object of type Fred will have certain methods
- ensure that every object of type Fred will have certain data fields

Abstract classes different from interfaces in that:

- you **extend** (only) one abstract class, but can **implement** many interfaces
- an abstract class can have its own normal methods (and data fields do not have to be public static final).


```
public abstract class Shape {
    public int dim = 2; // ordinary data field inherited by subclasses

    // ordinary method inherited by subclasses
    public void info ( ) {
        System.out.println("A " + dim + " dimensional shape.");
    }

    // abstract method, must be implemented in all subclasses
    public abstract void showArea();    // Note! ; but no body { }

}
```

Example of a possible abstract class Shape, and possible subclasses (next slide).

Note that:

- you cannot create instances / objects of an abstract class (only subclasses)
- an abstract class can specify **abstract methods** that subclasses must define, but it does not define them itself (static / class methods can't be abstract)
- by specifying accessors and modifiers that must exist an abstract class can indirectly ensure that subclasses should have certain data fields too

```
public class Circle extends Shape {
    private double radius = 3.0;     // radius of the circle

    public void showArea ( ) {      // must have this method
        System.out.println("Circle area: " + (Math.PI * radius * radius));
    }

}
```

```
public class Square extends Shape {
    private double length = 2.0;    // length of a side

    public void showArea ( ) {      // must have this method
        System.out.println("Square area: " + (length * length) );
    }

}
```


Collections of related objects

To deal with a number of Square and Circle (Triangle etc.) objects, we can use the fact that they are also of type Shape to store them in an array of Shape, e.g.:

```
Shape[ ] shapes = new Shape[4];    // or use an initialiser list { }
shapes[0] = new Circle();
shapes[1] = new Square();
shapes[2] = new Circle();
shapes[3] = new Triangle();
```

Shape is an **abstract class** so it can specify abstract methods that any subclass must have, thus we can perform an operation (call versions of the same method) on all the objects in the array. It only works if a matching (e.g. abstract) method in the superclass exists, in general subclass members are not available via superclass.

shapes[0].showArea( ); *// own version of abstract method* shapes[0].info( ); *// method inherited from Shape* shapes[1].showArea( ); *// own version of abstract method* shapes[1].info( ); *// method inherited from Shape*	Circle area: 28.274333 A 2 dimensional shape. Square area: 4.0 A 2 dimensional shape.

The program in the second notes shows a longer worked example of an array of objects that are subclasses of an abstract parent. You will be working with the same kind of design (using an ArrayList) in Lab 24!

Try this

Write an abstract class Fruit which has:

- an ordinary data field, String status set to "is healthy".
- an ordinary method (accessor), getStatus, which returns status
- an abstract method, getCost with a single input **int** weight, returning **double**

Try this

Write two classes Apple and Plum that extend Fruit.

Apple should have a data field pricePerKg initialised to 3.1 and a toString method that returns "Apple".

Plum should have a data field pricePerKg initialised to 5.4 and a toString method that returns "Plum".

In each case complete the required method. The cost of an Apple is "weight * pricePerKg", the cost of a Plum is "weight * pricePerKg * 0.8" (plums are 20% off today!).

```
public class Apple extends Fruit {
   double pricePerKg = 3.1;

   public String toString() {
      return "Apple";
   }




}
```


Try this

Declare an array of Fruit and initialise with an initialiser list (not element by element as the example on Slide 11) containing just one Apple and one Plum.

Write a for-each loop which prints a line for every Fruit in the array by calling the toString, getStatus and getCost(2) (so weight = 2) methods to create the output shown:

```
Apple   is healthy      $6.2
Plum    is healthy      $8.64
```

Apple and Plum are both guaranteed to have a getCost method, but the details are different in each class. A richer example would have many subclasses of fruit and use constructors etc. to create examples with different weights and costs.

Multiple inheritance

In Java a parent class can have many children, but a child class can have only one parent. A situation like:

```
class Toy
class Elephant
class ToyElephant extends Toy, Elephant
```

is called **multiple inheritance**, and it is not allowed in Java (it is in C++).

Conceptually some times it would be nice to have multiple inheritance. A ToyElephant class has Toy properties (soft, safe, fun to play with) and Elephant properties (a kind of elephant, four legs, a trunk).

But multiple inheritance gets very messy when inherited values are inconsistent. For example, the toy elephant might inherit isLiving = true from Elephant, and isLiving = false from Toy. What do you do? There are no easy answers.

Java avoids the issue by not allowing full multiple inheritance, but the use of interfaces allows a restricted form.

Recall (Lec 17) that interfaces are very similar to classes, in fact they are like "even more abstract abstract classes". They may contain **only**:

- abstract methods
- static final data fields

A class can implement any number of interfaces, which allows a restricted form of multiple inheritance, but the restrictions above mean that the worst of the possible complications are avoided.

See the excellent discussion on "designing for inheritance" LDC Section 8.5.

Extending and implementing

For example, assume interfaces Stats (specifying methods to compute facts about a shape) and Drawable (specifying methods to display a shape). We could define a class Square in many ways:

```
public class Square { /* class body */ }

public class Square implements Stats { /* class body */ }

public class Square implements Stats, Drawable { /* class body */ }

public class Square extends Shape implements Stats, Drawable { /* body */ }
```

The last version says that Square extends Shape (inherits data fields and methods) and implements two interfaces (inherits data fields and must itself define the methods specified in the interfaces).

final

final is another special keyword which can be used to modify classes and class members.

A final class may not be extended.

A final method may not be overridden.

A final data field must be initialised when it is declared, and can never be changed (no other value can be assigned).

A **static final** data field is the same for all objects of the class and can never change. Such data fields are called **constants** (naming convention CAPITALS e.g. Math.PI). See Lec 2 Slide 3; LDC 2.2, 8.5, Appendix E.

COMP160 Lecture 23 Second Notes

```
/* Anthony, September 2012, JDK 1.6
   Demonstrate the use of a collection (Array) of related objects.  Classes Staff
   and Student extend the abstract class Person.

   Staff and Students have slightly different information represented in their
   data fields, and have different ways of determining their email addresses -
   see getEmail() method.

   Imagine a university database where each course is represented as a
   collection of Person objects, e.g. comp160 is the collection of people
   involved as staff or students...
*/

public class Demo {

  public static void main(String [ ] args) {

    // array comp160 holds various objects of type Staff and Student, both of
    // which are subclasses of the abstract class Person.
    Person [ ] comp160 = {
      new Staff("Sandy", 99987, "computer-science"),
      new Staff("Anthony", 99981, "computer-science"),
      new Student("Anne", 34623, "BSc"),
      new Student("Bob", 99143, "BSc"),
      new Student("Chris", 57574, "BA"),
      new Student("Danny", 12622, "BSc"),
    };

    for(Person p: comp160) {
       p.printInfo();
    }
    // this version would also work...
    // for (int i = 0; i < comp160.length; i++) {
    //    comp160[i].printInfo();
    // }

  } // main

} // Demo
```

```
public abstract class Person {
  String name;
  int id;

  // constructor
  public Person(String name, int id) {
    this.name = name;
    this.id = id;
  }

  // normal method
  public void printInfo() {
    System.out.println(name + "  id: " + id );
  }

  // abstract method
  // subclasses must declare their own version
  public abstract String getEmail();

} // Person
```

OUTPUT:

```
Sandy  id: 99987
Department: computer-science
Email: Sandy@computer-science.otago.ac.nz

Anthony  id: 99981
Department: computer-science
Email: Anthony@computer-science.otago.ac.nz

Anne  id: 34623
Degree: BSc
Email: Anne@student.otago.ac.nz

Bob  id: 99143
Degree: BSc
Email: Bob@student.otago.ac.nz

Chris  id: 57574
Degree: BA
Email: Chris@student.otago.ac.nz

Danny  id: 12622
Degree: BSc
Email: Danny@student.otago.ac.nz
```

```
public class Student extends Person {
  String degree;

  // constructor, which calls parent constructor
  public Student(String name, int id, String degree) {
    super(name, id);
    this.degree = degree;
  }

  // own version of abstract method in Person
  public String getEmail() {
    return name + "@student.otago.ac.nz";
  }

  // normal method which also calls related method in parent
  public void printInfo() {
    super.printInfo();
    System.out.println("Degree: " + degree);
    System.out.println("Email: " + getEmail() + "\n" );
  }

} // Student
```

```
public class Staff extends Person {
  String department;

  // constructor, which calls parent constructor
  public Staff(String name, int id, String department) {
    super(name, id);
    this.department = department;
  }

  // own version of abstract method in Person
  public String getEmail() {
    return name + "@" + department + ".otago.ac.nz";
  }

  // normal method which also calls related method in parent
  public void printInfo() {
    super.printInfo();
    System.out.println("Department: " + department);
    System.out.println("Email: " + getEmail() + "\n" );
  }

} // Staff
```

Collections, ArrayList, Applets

COMP160 Lecture 24
Anthony Robins

- Enumerated types
- Generics
- Collections
- ArrayList
- Applets

Reading: LDC: 3.7, 12.1 (or 14.1 in 2e), Appendix G.

LDC 3e to (2e) p156 (p177)

Enumerated Types

Enumerated types (LDC 3.7) are a feature in many languages. They were introduced in Java only recently (JDK 1.5).

Java implements these types as classes, but (because they are set up to "look" similar to the way they look in other languages) they don't work quite the same way as usual classes.

Why are they useful? So that a program can use the "natural values" of the task. If I'm working with a task involving fruit I might want to use the values: apple, orange, banana, cherry, grape. See other examples in LDC.

Enumerated types are a way of creating a new type by enumerating (listing) every possible value…


```
/* Demonstrate an enumerated type */
public class EnumDemo {
  enum Fruit {apple, orange, banana, cherry, grape}

  public static void main (String [ ] args) {
     Fruit f = Fruit.apple;
     System.out.println( f );
     System.out.println( f.ordinal() );
     System.out.println( f.name() );

     for ( Fruit fru : Fruit.values() ) {
        System.out.println( "A nice " + fru );
     }
  }
}
```

```
apple
0
apple
A nice apple
A nice orange
A nice banana
A nice cherry
A nice grape
```

Lists the values of the type (declares a class Fruit with data fields of type Fruit each referring to an instance / object of Fruit).

Declares variable f initialised to the value apple (f refers to same object as public static data field Fruit.apple).

prints f.toString()
prints ordinal position of f (from 0)
prints the value / name of f

for-each value of Fruit repeat loop with with fru set to current value (the method Fruit.values() returns an array of type Fruit [ ]).

Although they look non standard, enums work in many ways like other classes. As well as having built in methods, we can add our own data fields, constructors, or methods. Here is one simple example:

```
enum Fruit {apple, orange, banana, cherry, grape;

    public String message() {
       return "is good for you";
    }

}
```

Enums let you organise the program in the "natural language of the task"

```
for ( Fruit fru : Fruit.values() ) {
   System.out.println( fru + " " + fru.message() );
}
```

```
apple is good for you
orange is good for you
banana is good for you
cherry is good for you
grape is good for you
```


Generics

Generic programming is a conceptual framework where programs can be written in terms of "to be specified later" types that are then **instantiated** when needed for specific types (provided as parameters).

Java (JDK 1.5) implemented generics in the form of classes (or interfaces) designed so that the exact type is specified by a <parameter> when declaring or instantiating.

Before generics, collections (next slide) are of type Object, and explicit casts are needed when accessing objects.

With generics a collection can be defined to be of a specific type with a <parameter>.

```
ArrayList a = new ArrayList();
a.add( "Hello" ); // add a String at position 0
System.out.println( ((String)a.get(0)).length() );

ArrayList<String> b = new ArrayList<String>();
b.add( "Hello" ); // add a String at position 0
System.out.println( b.get(0).length() );
```

```
5
5
```


Collections

As well as arrays, Java has many other ways of representing and processing collections of **objects**, using classes and interfaces in the java.util package. These are called the **collection classes**, **collection framework**, or **collection API** (Application Programming Interface). See LDC 12.1 (or 14.1 in 2e). The collection API includes:

- ArrayList
- Hashtable
- List
- Vector
- HashMap
- Map
- Set
- and more...

Collections have useful methods such as:

- copy
- fill
- min
- max
- reverse
- shuffle
- sort
- contains

Collection classes often implement interfaces, such as **Iterator** (e.g. hasNext(), next() etc.) and **Iterable** (e.g. for-each loop support) – Lec 11, LDC p 156.

Collections and generics go hand in hand! We look at just one example…

ArrayList

An **ArrayList** is a data structure which can store varying numbers of (references to) objects in a flexible (growing or shrinking) list.

The ArrayList class is defined in the package java.util so we must:

import java.util.ArrayList; *// at the start of the file*

An ArrayList consists of a sequence of elements. They:

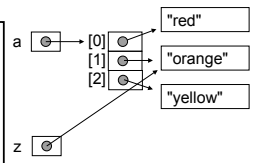
- can hold objects (instance objects, strings, arrays, **not** primitive types)
- you don't need to declare the size
- because an ArrayList can grow and shrink as required
- there are methods for useful operations such as adding and removing objects

ArrayLists (and other collections) support generics, so the type (kind of object that the ArrayList manages) can be established when a variable is declared:

```
ArrayList<String> a1 = new ArrayList<String>();        // a1 is of type ArrayList<String>
ArrayList<MyClass> a2 = new ArrayList<MyClass>(); // a2 is of type ArrayList<MyClass>
```


The following code illustrates some of the basics, for other useful methods see online documentation (Lec 3).

```
ArrayList<String> a = new ArrayList<String>();
a.add("red");     // add elements
a.add("orange");
a.add("yellow");
// print items in the list
System.out.println("---------");
for (String s : a ) {
   System.out.println( s );
}
System.out.println("---------");
// access element at position 1
String z = a.get(1);
System.out.println( z );
// find first occurrence of an element (-1 if not found)
System.out.println( a.indexOf("yellow") );
```

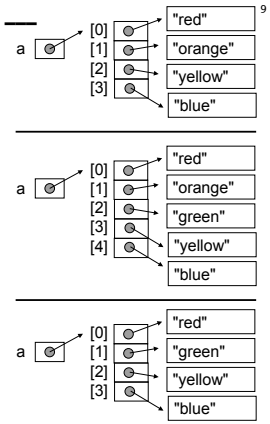



```
---------
red
orange
yellow
---------
orange
2
```


Adding and removing elements

The ArrayList grows and shrinks as required.

```
ArrayList<String> a = new ArrayList<String>();
a.add("red");
a.add("orange");
a.add("yellow");
a.add("blue");

a.add(2, "green"); // add at specified position

a.remove(1);  // delete element at position
```



Try this

Assume a class MyClass with a data field d set by a constructor, and a toString method that returns ">> d" (where d is the value of the data field). Sketch MyClass objects briefly as shown:

MyClass d = 3

Sketch a model of the ArrayList after the following operations:

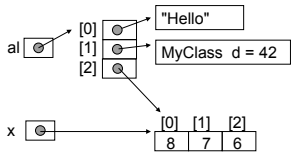
```
ArrayList<MyClass> mc = new ArrayList<MyClass>();
mc.add( new MyClass(7) );
mc.add( new MyClass(42) );
mc.add( new MyClass(9) );
mc.add( 0, new MyClass(17) );
mc.add( 0, new MyClass(2) );
mc.remove(2);
```

Write a foreach loop that will process each element in turn, printing output from its toString method. Show the output.

ArrayList of Object

If no <type> is specified the ArrayList stores objects as type Object (Slide 5). It can mix object types (String, MyClass, Array etc. are all also of type Object).

```
int [ ] x = {8, 7, 6};
// no <type> means Object
ArrayList al = new ArrayList();
al.add( "Hello" );
al.add( new MyClass(42) );
al.add( x );
```



Useful, but risky! There is no type checking. Depending on your compiler / IDE settings you may get a warning: Name.java uses unchecked or unsafe operations.

To use objects in such a list we must cast them to the right type, e.g.:

String s = (String) al.get(0); *// create an alias to element0 (cast Object to String)*

((MyClass) al.get(1)).myMethod() *// call myMethod on elmt1 (Object to MyClass)*

If you mix object types (as in the previous slide) there is a lot of work to remember what kind of object is at each location!

You can't store primitive values in an ArrayList unless you turn them into objects using wrapper classes (Lec 8, LDL 3.8).

Try this

Write a for-each loop which will print out every int in the array of int stored in the ArrayList al at index 2 (previous slide) (don't use the alias x). You will need to cast!

Applets

Until now COMP160 has used application programs. The other main kind of Java program is the applet (Lec 3).

Applets are slightly different in the way they are set up and run, but otherwise all the details of Java remain the same.

Applets are not run directly like applications, but by and within another program called an **applet context** (which uses the interpreter / JVM). Typical examples are **web browsers** and **applet viewers.**

From now on I will just say "browser" as short for "applet context".

The browser imposes restrictions on applets for security reasons, e.g. they cannot access files on the machine. This restricted security environment is called the "sandbox".

Running an applet

We run an applet by loading an **html** file into a browser (html is the basic language for describing the structure of a web page). The html file holds information about the applet. At a minimum this includes the name of the applet class file (and its directory / location of the class file(s) if not the same as the html file), and the default size of the area that it runs in.

A minimal html file for running an applet based on the class PizzaApplet.

```
<html>
<head><title>Pizza Options Applet </title></head>
<body>
<applet code = "PizzaApplet.class" width = "300" height = "180" >
</applet>
</body>
</html>
```

file: MyPizza.html

The browser calls the init method (next slide) to initialise the applet. It calls other methods (e.g. start and stop, inherited from JApplet) to control the applet as the user e.g. leaves or returns to the web page (see LDC Appendix G).

The applet class

The **applet class** is the special class that defines where the program "starts" (in the same way that the application class / main method is the start of an application).
It:

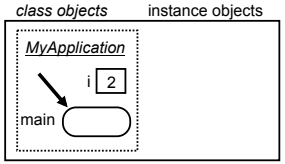
- must **import** javax.swing.JApplet;
- must **extend** JApplet (so inherits methods including a default init method)
- must have an init method (short for "initialise", usually write your own overriding the default) which is where the applet starts (there is no main method).

An application class is created as a class object. The interpreter calls the main method on the class object. Its members have to be static to be used.

An applet class is created as a class object (all classes are!) **and** an instance is also created automatically. The browser calls the init method on **the instance object**. Its members do not have to be static.

Application

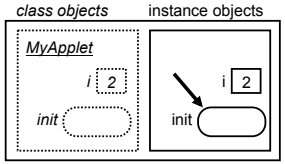
```
public class MyApplication {
  private static int i = 2;
  public static void main(String [ ] args) {
    // program starts running here
    System.out.println("i is: " + i);
    // output written to Java console
  }
}
```



Applet

```
import javax.swing.JApplet;
public class MyApplet extends JApplet {
  private int i = 2;
  public void init( ) {
    // program starts running here
    System.out.println("i is: " + i);
    // output written to browser log file
  }
}
```

log is browser dependent - text output is not usual for an applet!



Converting an application into an applet

The LDC graphics design (Lec 17 Slide 4) makes it easy to convert applications to applets! The primary panel contains all of the detail / functionality.

For example, the application class for the PizzaApp program in Lec 19 is as follows:

```
import javax.swing.JFrame;
public class PizzaApp {
  public static void main (String[ ] args) {
    JFrame frame = new JFrame ("Pizza Order");
    frame.getContentPane().add( new PizzaOptionsPanel() );
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);
  }
}
```

Display in a window (JFrame)

Add the primary panel to the JFrame.

All other statements in main specify the appearance and behaviour of the window.

All we need to do to convert the PizzaApp application program into an applet is replace the application class on the previous slide with the applet class below (and compile of course).

Put the html file from Slide 14 in the same directory, and use a browser to load the html file. Easy!

```
import javax.swing.JApplet;
public class PizzaApplet extends JApplet {
  public void init() {
    getContentPane().add( new PizzaOptionsPanel() );
  }
}
```

Display in a browser.

Add the primary panel to the JApplet.

The appearance and behaviour of the applet are handled by the browser.

Simulation. Programming.

COMP160 Lecture 25
Anthony Robins

- Design choices
- Design Examples
- Simulation
- Programming

See second notes document for this lecture.

Reading: Start revising!

Design choices

There many ways to write a program, from major styles of programming (imperative vs. OO, event driven), to different programming languages, to different designs for a specific task.

Recall the simple graphics design (Lec 17 Slide 4, also Applet design Lec 24) has alternatives (e.g. Lec 19 Slide 16).

Java provides lots of mechanisms to help good design:
packages, visibility, scope, inner classes, type checking, array bounds checking, references instead of pointers, garbage collection instead of memory allocation and deallocation…

Design Examples

The following examples look at the way objects can be set up to access a data field in another object.

All the examples (except slight variation in 4):

- have the same main method (below) that makes an instance p of Primary,
- have a Primary class that makes two instances x and y of Other,
- create the same output (below).

In each example we look at how a method in x and y (the method happens to be a constructor, but the same principles apply to any method) can access the data field in Primary.

```
public class Example {
  public static void main(String [ ] args) {
    Primary p = new Primary();
  }
}
```

Output:

```
2
2
```


Example 1 - pass a copy

The method in Other (the constructor) is passed a separate copy of the value of the data field i in Primary.

In general, when the variable passed is:

a primitive type x and y have their own separate copies of the (primitive) value

a reference type x and y have their own separate copies of the reference value (and the copies refer to the same actual object as the original data field in Primary).

See Lectures 3, 13, 14.

```
public class Primary {
  private int i = 2;

  public Primary() {
    Other x = new Other( i );
    Other y = new Other( i );
  }

} // Primary
```

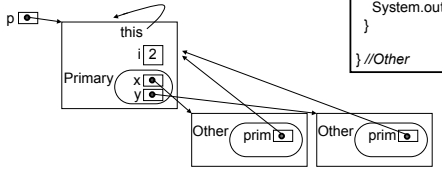
```
public class Other {

  public Other(int i) {
    System.out.println( i );
  }

} // Other
```


Example 2 - pass a reference

The method in Other (constructor) is passed a reference to the instance of Primary ("this" object).

x and y save the reference in their own data field prim. They can use prim any time to refer to the instance of Primary, and access any (non private) data field or method.

See Lecture 14 (passing references, this).




```
public class Primary {
  int i = 2;  // package visibility

  public Primary() {
    Other x = new Other( this );
    Other y = new Other( this );
  }

} // Primary
```

```
public class Other {

  public Other(Primary prim) {
    System.out.println( prim.i );
  }

} //Other
```


Example 3 - class member

The method in Other (constructor) accesses the data field i, which is **static** (a class member, part of the Primary class object).

x and y can access the class object Primary, and hence access any visible static data field or method.

See Lectures 7, 13.

```
public class Primary {
  public static int i = 2;

  public Primary() {
    Other x = new Other();
    Other y = new Other();
  }

} // Primary
```

```
public class Other {

  public Other() {
    System.out.println( Primary.i );
  }

} // Other
```


Example 4 - inheritance

The class Other extends Primary.

x and y (instances of Other) get their own separate copy of any (non private) data field or method in the parent class (Primary).

See Lecture 21.

In this particular example the main method must make x and y:

```
Other x = new Other();
Other y = new Other();
```

It can't be done in the Primary constructor because the Other constructor automatically calls the Primary constrcutor, and this would set up an infinite loop…

```
public class Primary {
  protected int i = 2;

  public Primary() {
    // main method makes x and y
  }

} // Primary
```

```
public class Other extends Primary {

  public Other() {
    System.out.println( i );
  }

} // Other
```


Example 5 - inner class

Other is an inner class of Primary.

x and y (instances of an inner class), can access all members of the outer class (even private members).

See Lecture 17.

```
public class Primary {
  private int i = 2;

  public Primary () {
    Other x = new Other();
    Other y = new Other();
  }

  class Other {  // inner class

    public Other() {
      System.out.println( i );
    }

  }  // Other

} // Primary
```


Discussion

In all of the above examples instances x and y (of type Other) work with a value from the data field i, which is in an instance of Primary (or in the Primary class object for Example 3).

The examples all use different mechanisms to access i (or get a copy of its value).

Each of the mechanisms has its own consequences, strengths and weaknesses. Designing programs is largely about knowing about the range of alternatives you have available, evaluating them, and choosing the right one for the task!

The example "Life" program discussed next makes a lot of use of inner classes (Example 5).

Simulation

One of the things that I love about programming is that it lets you create worlds! A program can create a simulation of any world or system that you can imagine (and describe in code). See for example many highly complex computer game worlds.

> A computer simulation is an attempt to model a real-life situation on a computer so that it can be studied to see how the system works. By changing variables, predictions may be made about the behaviour of the system.
>
> See: http://en.wikipedia.org/wiki/Simulation

We can do simulations to study situations that it would be impossible to study in any other way.

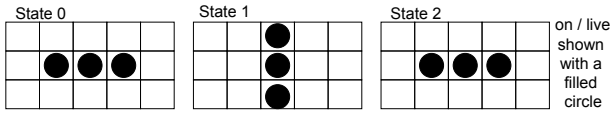
Some simulations show how apparently very complicated behaviour can arise from very simple underlying rules. A classic example is Conway's "Game of Life"…

Life

Life is a simulation (specifically a "cellular automata") proposed by British mathematician John Conway in 1970.

The world of Life is a grid. Each cell of the grid can be either on (alive) or off (dead). Given some state of the world, very simple rules determine how to calculate the next state:

1. Any live cell with fewer than two neighbours dies (loneliness).
2. Any live cell with more than three neighbours dies (overcrowding).
3. Any live cell with two or three neighbours lives (unchanged).
4. Any dead cell with exactly three neighbours comes to life (birth).



That's all there is to it! Those simple rules create a huge range of fascinating behaviours.

Background:
http://en.wikipedia.org/wiki/Conway's_Game_of_Life

Text "images" of interesting life states, can be used as inputs for this program:
http://www.argentum.freeserve.co.uk/lex.htm

Animations of interesting life states:
http://www.ericweisstein.com/encyclopedias/life/animations.html

The Second notes show an implementation of Life. It is available in the COMP160 Labs (lab files). You can copy interesting states from the second of the websites listed above to load as start states for the program (see the README file with the code in the lab).

Programming

In a course like COMP160 it is so easy to drown in the details. The really important lessons in COMP160 are the general principles for good programming:

0) Solve the problem. You can't write a program if you don't understand the problem and how to solve it. Ten minutes thinking might save you ten hours hacking a badly designed program…

1) Organise your program well. In Java use classes to represent entities with state and behaviour. Methods to implement behaviours, and to organise sensible "chunks" of processing within a class. Don't duplicate work / code (use methods, super, this). Choose clear names but don't reuse them where it could be confusing.

2) Keep your data safe! Encapsulate. Keep variables as local as you can. In Java use private data fields with accessor and modifier methods.

3) Write a big program in small pieces. Build it up slowly, testing as you go.

4) A working program is not necessarily a good program. Test your programs, especially "at the edges" (maximum, minimum, and null data / inputs). Test your inputs (who knows what the user will do!)

5) Anticipate errors. Plan for them before they happen. (Java helps with try catch).

6) Don't keep trying to patch a badly designed program or piece of code. Start again, and write it properly.

7) Clear code is almost always better than clever code.

8) Good formatting and commenting will save you (and everyone else) time.

9) Read the "friendly" manual (RTFM).

Sadly, good advice makes little difference:

> "Good programming cannot be taught by preaching generalities.
> The way to learn to program well is by seeing, over and over, how real programs can be improved by the application of a few principles of good practice, and a little common sense."
>
> – Kernighan & Plauger, *The Elements of Programming Style* (1974).

So: Write lots of programs. Doing it properly will save you endless frustration.

When you've finished COMP160

You will have learned a lot about Java, but also about related languages:

C not OO (a program is like one app. class with all static data fields/methods)
C++ similar to Java but more complex (more control of memory allocation)
C# "C sharp" is Microsoft's Java-like language

You know a powerful and flexible language that will be used as the foundation for further studies in Computer Science and Information Science at Otago.

You have had experience with various major "styles" of programming, particularly OO and event driven.

You have all the tools you need to write:

- simple text based programs
- GUI / event based programs

} as application programs for Macs, PCs, Android etc…

- applets for web pages / simple web based programming

What you have learned is very powerful and useful.

As you become a more experienced programmer, you are not limited to the programs that you write yourself. There is a world of "free" or "open source" program code out there:

Operating systems:	Linux, Free BSD, Darwin
Tools:	GNU (Free Software Foundation)
Office applications:	OpenOffice
Web browsers:	Firefox
Web servers:	Apache
Game engines:	Quake 1, 2, 3 and others

See: http://en.wikipedia.org/wiki/Open_source

Get the code. Do what you like with it. Take control of your computer at any level! Write the next sensational program, get employed, found a company, contribute to open source software, or just have fun...

Yes, there is a lecture on Thursday.

An introduction to our advisor of studies,
a short presentation on local industry and
interesting topics in computer science,
a chance to ask any questions…

File: Life.java

```
/*
 Anthony, October 2012, JDK 1.6, Conway's game of Life simulator.
 This and related files can be found in the COMP160 Lab (LabFiles).
 See also Lecture 25 for more details.  Note hardcoded grid cell size is 10.
 Background:
     http://en.wikipedia.org/wiki/Conway's_Game_of_Life
 Text "images" of interesting life states, can be used as inputs for this program:
     http://www.argentum.freeserve.co.uk/lex.htm
 Animations of interesting life states:
     http://www.ericweisstein.com/encyclopedias/life/animations.html
*/

import javax.swing.JFrame;

public class Life {
  public static final int SIZE = 75;        // x and y dimension of the grid
  public static final int SPEED = 50;       // speed of timer events (msec)

  public static void main (String[ ] args) {
    JFrame frame = new JFrame ("Conway's Game of Life");;
    frame.getContentPane().add( new PrimaryPanel() );
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);
  }

} // Life
```

File: PrimaryPanel.java

```
import javax.swing.*;  import java.awt.*;  import java.util.Scanner;
import java.awt.event.*;  import java.io.*;

public class PrimaryPanel extends JPanel {
  //data fields refer to a button panel and a grid panel (both are inner classes)
  private ButtonPanel buttonPanel = new ButtonPanel();
  private GridPanel gridPanel = new GridPanel();
 // constructor simply adds these panels to the PrimaryPanel
  public PrimaryPanel() {
    add( buttonPanel );
    add( gridPanel );
  }
  // inner classes follow (ButtonPanel on page 2, GridPanel on pages 3 and 4)
```



```
// panel to hold buttons (inner class of PrimaryPanel)
class ButtonPanel extends JPanel {
  private ButtonListener buttonListener = new ButtonListener();
  private JButton start = new JButton("Start");
  private JButton stop = new JButton("Stop");
  private JButton step = new JButton("Step");
  private JButton rand = new JButton("Random");
  private JButton load = new JButton("Load");
  private JButton [ ] buttons = {start, stop, step, rand, load};

  // constructor sets up the button panel
  public ButtonPanel() {
    setPreferredSize( new Dimension( 100, Life.SIZE * 10 ) );
    setBackground (Color.yellow);
    for (int i = 0; i < buttons.length; i++) {  // set up buttons
      buttons[ i ].addActionListener(buttonListener);
      buttons[ i ].setBackground(Color.yellow);
    }
    add(new JLabel("Control:"));
    add(start);
    add(stop);
    add(step);
    add(new JLabel("Initialise:"));
    add(rand);
    add(load);
  }

  // listener to handle button presses (inner class of inner class ButtonPanel)
  class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
      Object source = ae.getSource();
      if      ( source == start)  gridPanel.timer.start();
      else if ( source == stop )  gridPanel.timer.stop();
      else if ( source == step ) {gridPanel.timer.stop(); gridPanel.step();}
      else if ( source == rand ) {gridPanel.timer.stop(); gridPanel.initRandom();}
      else if ( source == load ) {gridPanel.timer.stop(); gridPanel.initLoadFile();}
    }
  } // ButtonListener

} // ButtonPanel  (end of first inner class)
```



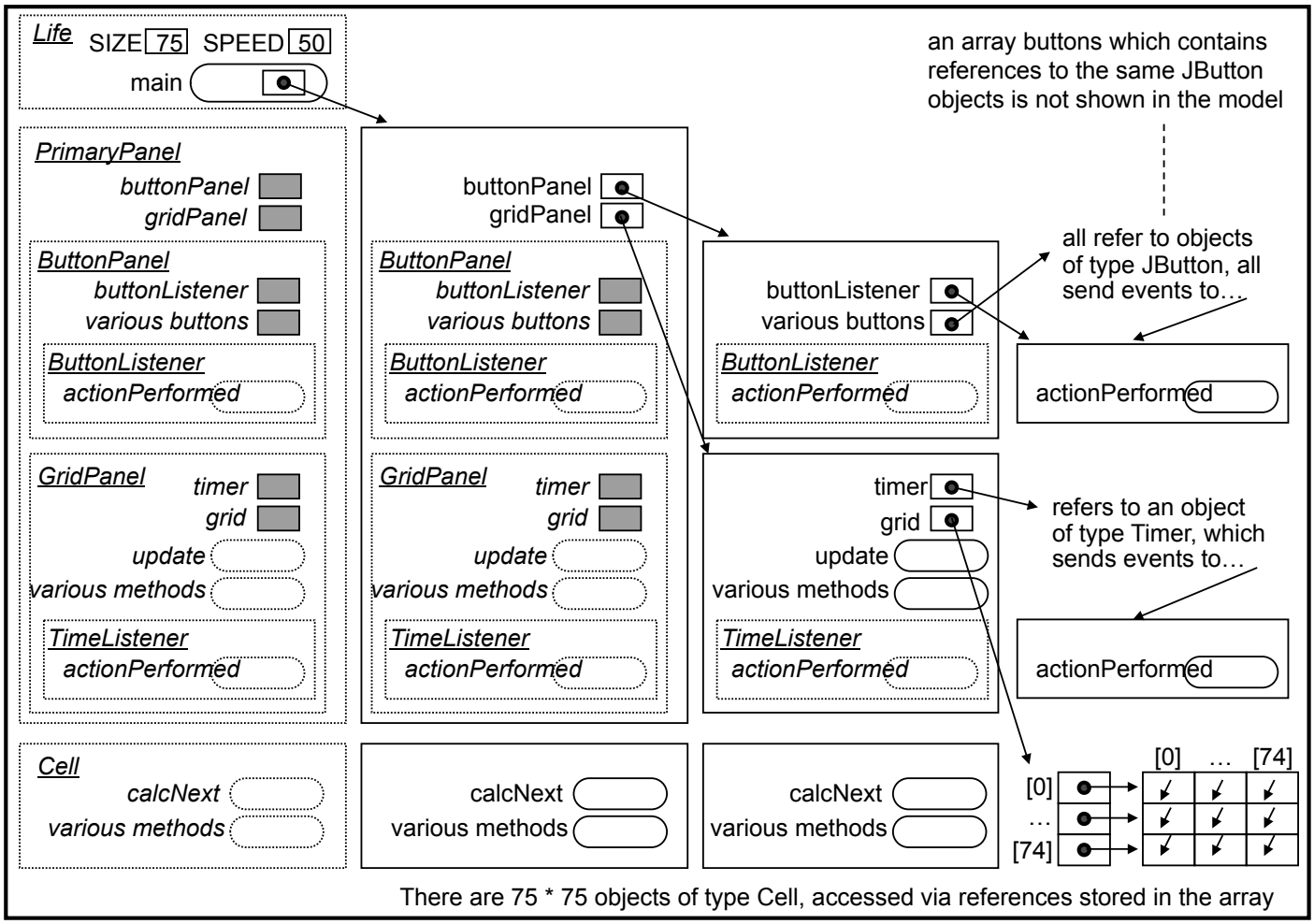
```
// panel to define life grid (inner class of PrimaryPanel)
class GridPanel extends JPanel {
  private Timer timer = new Timer( Life.SPEED,  new TimeListener() );
  private Cell[ ][ ] grid = new Cell[ Life.SIZE ][ Life.SIZE ];

  // constructor to set up the grid panel
  public GridPanel() {
    setPreferredSize( new Dimension( Life.SIZE * 10, Life.SIZE * 10) );
    // create Cell object for each place in the grid, all set to off
    for(int y = 0; y < Life.SIZE; y++) {
      for(int x = 0; x < Life.SIZE; x++) {
        grid[x][y] = new Cell(x, y, false);
      }
    }
  }

  // initialise the grid to a random start state
  public void initRandom() {
    boolean on;
    for(int y = 0; y < Life.SIZE; y++) {
      for(int x = 0; x < Life.SIZE; x++) {
        if (Math.random() > 0.5) on = true; else on = false;
        grid[x][y].setOn(on);
      }
    }
    repaint();
  }

  // initialise the grid to a state read from an input file
  public void initLoadFile() {
      // for simplicity the code of this method is not included in this handout,
      // see full code available in the lab…
  }

  // get each cell to calculate what its next state will be, then repaint grid
  public void step() {
    for(int y = 0; y < Life.SIZE; y++) {
      for(int x = 0; x < Life.SIZE; x++) {
        grid[x][y].calcNext(grid);
      }
    }
    repaint();
  }
```



```
  // called by repaint, calls a method on each cell to set next state and draw
  public void paint(Graphics g) {
    g.clearRect(0, 0, Life.SIZE * 10, Life.SIZE * 10);
    for(int y = 0; y < Life.SIZE; y++) {
      for(int x = 0; x < Life.SIZE; x++) {
        grid[x][y].setAndDrawNext( g );  // passes the Cell the Graphics object
      }
    }
  }

  // listener triggered by timer events (inner class of inner class GridPanel)
  class TimeListener implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
      step();  // every timer event triggers a step of the grid
    }
  } // TimeListener

} // GridPanel  (end of second inner class)


} // PrimaryPanel  (end of PrimaryPanel)
```

File: Cell.java

```
/* This class is used to construct the objects representing the
   cells of the grid.  Cell objects represent their current state (on) and can
   calculate their next state (next).  They can update to their next state and draw
   themselves.
*/

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

public class Cell {
  int x, y;
  boolean on, next;

  public Cell(int x, int y, boolean on) {
    this.x = x;
    this.y = y;
    this.next = this.on = on;
  }

  public boolean getOn() {
    return on;
  }

  public void setOn(boolean on) {
    this.next = this.on = on;
  }

  // update to next state and draw in the Graphics object (passed by GridPanel)
  public void setAndDrawNext(Graphics g) {
    on = next;
    // draw circle size 8x8 in grid cell size 10x10
    if( on ) g.fillOval(x * 10 + 1, y * 10 + 1, 8, 8);
  }

  // calculate the state this cell will have in the next update -
  // takes as input a reference to the grid so it can examine its neighbours
  public void calcNext( Cell[ ][ ] grid ) {
    int sum = 0, mx = x - 1, my = y - 1, px = x + 1, py = y + 1;
    // neighbours "wrap around" at the edge of the grid
    if( mx < 0 ) mx = Life.SIZE - 1;
    if( my < 0 ) my = Life.SIZE - 1;
    if( px >= Life.SIZE ) px = 0;
    if( py >= Life.SIZE ) py = 0;
    // test 8 neighbours, 3 above, 3 below, 1 left, 1 right
    if( grid[ mx ][ my ].getOn() ) sum++;  // check 3 above (my is y minus 1)
    if( grid[  x ][ my ].getOn() ) sum++;
    if( grid[ px ][ my ].getOn() ) sum++;
    if( grid[ mx ][  y ].getOn() ) sum++;  // check 1 left and 1 right
    if( grid[ px ][  y ].getOn() ) sum++;
    if( grid[ mx ][ py ].getOn() ) sum++;  // check 3 below (py is y plus 1)
    if( grid[  x ][ py ].getOn() ) sum++;
    if( grid[ px ][ py ].getOn() ) sum++;
    // this test implements the central update rule of the game
    if ( (on && sum == 2) || sum == 3 ) next = true;
    else next = false;
  }

} // Cell
```



Attendance Record for Terms Requirement

You must attend 11 of your first 13 scheduled lab sessions.

Date range of lab streams for this lab	Lab number	For your record
July 9th - 11th	1	
July 14th - 16th	2	
July 16th - 18th	3	
July 21st - 23rd	4	
July 23rd -25th	5	
July 28th - 30th	6	
July 30st – Aug 1st	7	
August 4th - 6th	8	
August 6th- 8th	9	
August 11th - 13th	10	
August 13th - 15th	11	
August 18th - 20th	12	
August 20th - 22nd	13	

You must attend 10 of your next 12 scheduled lab sessions.

Date range of lab streams for this lab	Lab timetabled	For your record
September 1st - 3rd	14	
September 3rd - 5th	15	
September 8th - 10th	16	
September 10th - 12th	17	
September 15th -17th	18	
September 17th -19th	19	
September 22nd - 24th	20	
September 24th - 26th	21	
September 29th - October 1st	22	
October 1st - 3rd	23	
October 6th - 8th	24	
October 8th - 10th	25	

Attendance is defined as **either** having completed the lab timetabled for that lab session before the end of your scheduled lab session **or** being in the lab for the duration of your scheduled lab session.

This information is verified by electronic means such as lab submission data and machine log records. This page is for you to fill in as your copy of the information for your own reference.