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".

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, Otago 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 language** was originally called "Oak" (developed for embedded processors - small and robust). The **Java Platform** is a set of tools and resources (from Sun / Oracle) that support the language. One form of the platform is the **The Java Developer Kit** (JDK), which includes:

- the **compiler** (needed for creating programs) and the **interpreter** (runs programs using the **Java Virtual Machine** (JVM))
- the standard **class libraries** (classes - parts of Java programs - that 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 program (in the Java language) consists of **code** specifying one or more **classes**. 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.

The classes from the package java.lang in the **libraries** are automatically part of every Java program.

To use a Java program the source code must be **compiled** and then **run**.

A Java program usually works by creating and using **objects** (instances of classes).

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 documents, there are many tools for writing programs. We use an Integrated Development Environment (IDE), DrJava (drjava.org).

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".

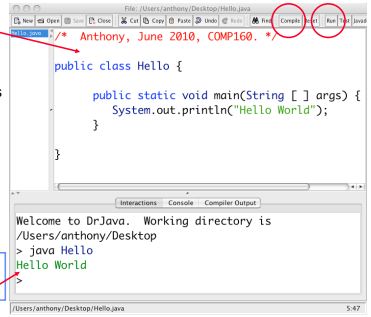
compile compiler "javac" + library classes

Class file "Name.class" contains object / byte code (portable - different machines).

run interpreter "java" + JVM

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:

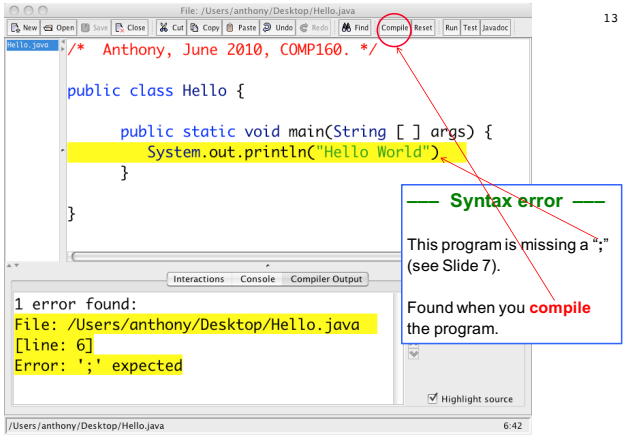
The code is not "legal". It breaks the strict rules of how Java can be written. Picked up when you **compile** the program (error messages).

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:

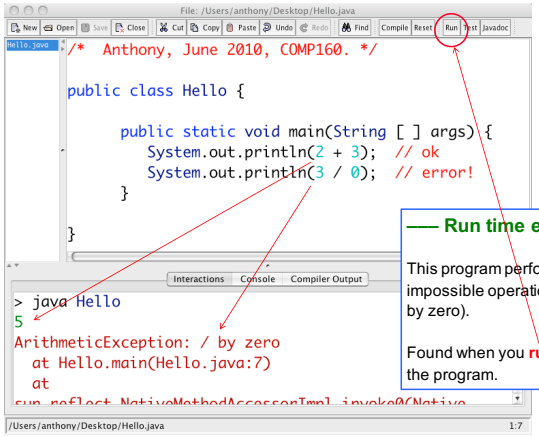
The code works, it compiles and runs. But it doesn't do what you wanted it to do. No error messages! This kind of error can only be noticed by a human.



Syntax error

This program is missing a ";" (see Slide 7).

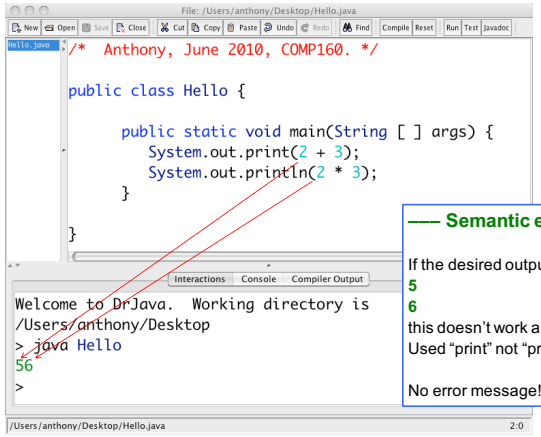
Found when you **compile** the program.



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).

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 (sequences 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

See second notes document for this lecture.

Readings LDC: Chapter 1, and Sections 2.1, 2.2, 2.3, 2.6
LabBook: "Where Do You Begin?" and "Object-Oriented Design"

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 ±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 equal, etc. Result is boolean e.g. 3 < 5 is **true**
multiplication	*		
division	/ or %		

Scientific notation

Java writes large and small values in **scientific notation** For example:

number	scientific notation	Java scientific notation
49800000.0	4.98 X 10^{7}	4.98e7 or 4.98e+7
0.000038	3.8 X 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
int  sum, total;
double  range;
boolean  test;
char  letter;

statements
sum = 5;
total = sum * 3;
range = 47.248573;
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;
```

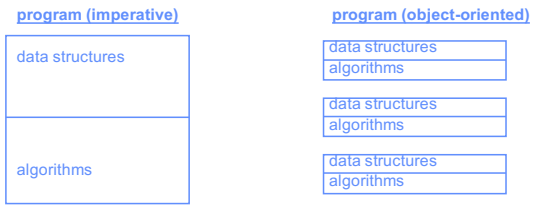

Object-oriented programming

"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 designed structures called **objects**.

Each cook in their own small kitchen.



Objects

Objects are ways to represent data that is more complex than a primitive value, and possibly also related actions. In design terms an object should be a useful "chunk" of the task or provide a useful service (like a Scanner). 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.

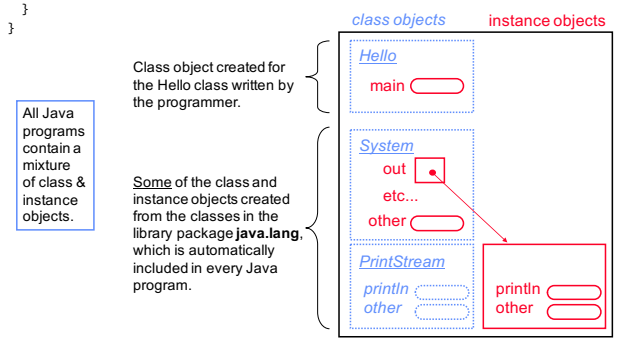
Most of object-oriented programming
is about the creation and use of **instance objects**.

For our early programs we write just one class / **class object**. We don't focus on multiple classes and making **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!



All Java programs contain a mixture of class & instance objects.

Class object created for the Hello class written by the programmer.

Some of the class and instance objects created from the classes in the library package **java.lang**, which is automatically included in every Java program.

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**).

int x = 5;

x 5

String day = **new** String("Tuesday");

day → "Tuesday"

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

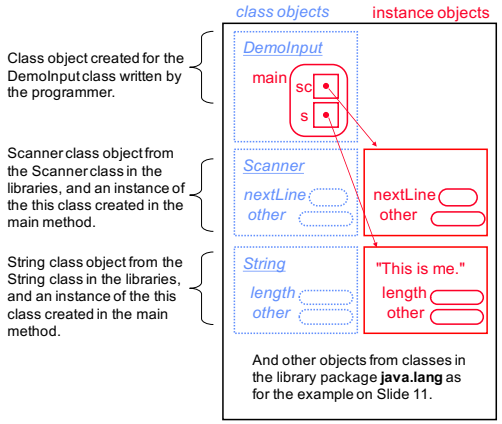
For simple text input from the keyboard make a Scanner object set to System.in as shown. (We're making and using instance objects from library classes already!).

```
/*  Anthony, July 2015, 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.



Class object created for the DemoInput class written by the programmer.

Scanner class object from the Scanner class in the libraries, and an instance of the this class created in the main method.

String class object from the String class in the libraries, and an instance of the this class created in the main method.

And other objects from classes in the library package **java.lang** as for the example on Slide 11.

COMP160 Lecture 2 Second notes

These are optional notes. There won't be exam questions on this stuff, but it will help you to understand how programming languages represent data / information.

Computer memory and representing data (where and what)

Computers represent all information (numbers, characters, strings/text, objects and other complex data structures) using an internal "language" of just 0 and 1 (in hardware these are either electrical or magnetic states "off" or "on"). The smallest element of memory is a **bit** (binary digit, 0 or 1, "bin" means two). Sequences of bits, depending on size and context, have other names like byte, word or nibble. A byte is 8 bits, for example 00100111.

Computer memory consists of numerically addressed locations that contain sequences of bits. In order to represent information the computer processor has to be told what memory location to look in, and how to interpret the sequence of bits that it finds there (e.g. as a number, as a character, as a reference to another location in memory, and so on). In other words, the processor need to know **where** data is and **what** it is.

Programming languages let us specify **where** data is in various ways. One of them (which is highly automated and convenient) is to create named variables, e.g. `int x = 1;` The name / identifier x is now associated with a location in memory, and we can assign a value to it, or use that value in other operations (values are just sequences of bits).

Programming languages let us specify **what** data is in various ways. Some of them (like Java) are statically / strongly "typed". E.g. `int x = 1;` the value held in the variable (memory location) x will always be interpreted as an integer number, currently 1.

Binary (how)

How do we interpret sequences of bits as data of various kinds? By reading them as numbers, then treating the numbers as representing whatever we want to.

How do we represent numbers in binary - how do we read sequences of bits as numbers?

The numbers we are used to are base-10, i.e. **decimals**. There are ten digits (0 – 9) and the "places" of a multi-digit numbers are multiplied by ten as we read right to left. For example:

place:	1000	100	10	1
digit	0	5	9	1

The decimal number above is: (0 * 1000) + (5 * 100) + (9 * 10) + (1 * 1), which is the value which we call (in our decimal English language) five hundred and ninety one.

But we can represent numbers in any base! Base-8 (octal) and base-16 (hexadecimal) are common alternatives.

COMP160 Lecture 2 Second notes

The numbers used by computers are base-2, i.e. **binary**. There are two digits (0 – 1) and the "places" of multi-digit numbers are multiplied by two as we read right to left. For example:

place:	128	64	32	16	8	4	2	1
digit	0	1	1	0	0	0	0	1

The binary number above is: (0 * 128) + (1 * 64) + (1 * 32) + (0 * 16) + (0 * 8) + (0 * 4) + (0 * 2) + (1 * 1), or in simpler terms 64 + 32 + 1, which is the value which we call (in our decimal English language) ninety seven.

A binary number with 1 place can represent 2^1 = 2 values (the decimal range 0 – 1)
A binary number with 2 places can represent 2^2 = 4 values (the decimal range 0 – 3)
A binary number with 8 places can represent 2^8 = 256 values (decimal range 0 – 255)
A binary number with 12 places can represent 2^{16} = 4096 values (decimal range 0 – 4095)
The more places / bits, the larger range of values can be represented.

Putting it all together

If my Java program has a variable / location in memory called y and it holds the binary number 01100001 (as above) the binary can be interpreted in various ways.

If y is type int then the binary will be interpreted as the integer 97, e.g. y may have been declared: `int y = 97;`

If y is type char then the binary is interpreted as the character 'a', e.g. y may have been declared: `char y = 'a';`
(In the Unicode character set the number 97 represents 'a', 98 represents 'b', and so on…)

If y is some other type the binary is interpreted as (a number representing) a value of that type. Remember, for computers and programming languages to represent information we need to know **where** it is (e.g. a variable name like y) and **what** it is (its type – how to interpret the binary number at that location).

Further reading

http://www.mathsisfun.com/binary-number-system.html
http://en.wikipedia.org/wiki/Binary_number
http://en.wikipedia.org/wiki/ASCII
http://en.wikipedia.org/wiki/Unicode

Final note: The computer program code that we write in a programming language is itself also stored as information in memory (this was a big breakthrough in the history of programming, see http://en.wikipedia.org/wiki/Stored-program_computer). The computer uses other programs (e.g. compilers, interpreters, loaders) to run the programs that we write. Programs are just another kind of data.

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!).

The main kind of Java program is the **Application**. These are typical "stand alone" programs that can run directly on a machine (i.e. using the Java interpreter).

Historically **Applet** programs were significant. These programs run within browsers like Firefox or Chrome, or within Applet Viewers.

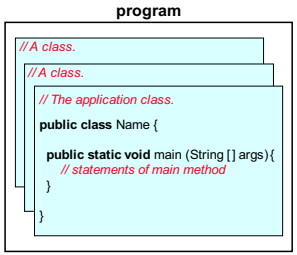
Program structure

An application 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 one per line) specify every step of the program's execution.

```
/* Anthony Robins, July 2015, JDK 1.7
   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
```

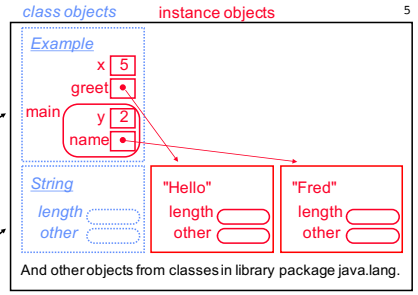
```
7
Hello Fred
```
program output

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 use programs where we write just an application class (**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).

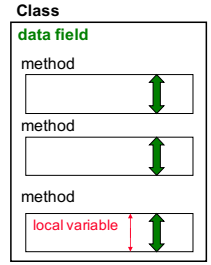
Variables again

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 2015, 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!)

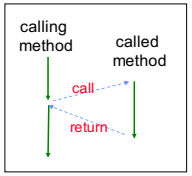
```
One fish
Two fish
A poem by
Dr Seuss
Red fish
Blue fish
```

```
/* Anthony Robins, March 2015, JDK 1.7 */

// 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 it 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/8/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.

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
- Example program
- Einstein example
- A brief history of programming languages

Reading: LDC: Appendix F.

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[1].

```
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 use the device's OS / windowing system (heavyweight) which requires customisation of a version of the Java platform for that device.

We mostly do graphics towards the end of COMP160, but today we introduce the basics of simple drawing in a JPanel within a JFrame (a kind of window).

[1] Java 8 included JavaFX - intended to eventually replace Swing – but it's more complicated and Swing suits our needs.

Drawing

We draw in panels (see next lecture).

All graphics "drawing" code should be in (or called by) the panel's **paintComponent** method.

This is called automatically when needed, e.g. as windows get made, resized, moved, redrawn etc. It is passed a reference to a Graphics object.

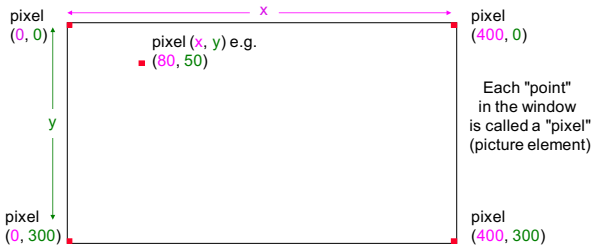
```
public void paintComponent (Graphics g) {
    g.drawLine (0, 0, 80, 50);   // draw a line between two points / pixels
    g.setColor(Color.red);       // set color for drawing
    g.fillRect(50, 50, 20, 30);  // draw a solid rectangle at given points
}
```

The graphics object (in this case referred to by the local variable g) has various useful methods.

Color (American spelling!) is black unless it is changed. Other colors - see documentation…

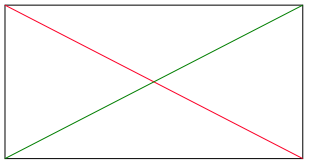
Coordinate system

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 paintComponent method below (similar to Slide 4) to draw in some window of size 400 by 300 (e.g. Slide 5) the following two lines:

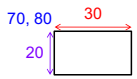



```
public void paintComponent (Graphics g) {



}
```

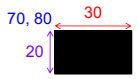

Drawing methods

Graphics objects have useful methods for drawing shapes etc, for example:

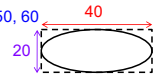
```
drawRect(int x, int y, int width, int height);
drawRect(70, 80, 30, 20);
```




```
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);
```

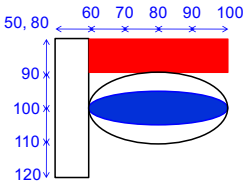



```
drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)
drawArc(10, 10, 60, 30, 20, 90)
```

See LDC p968 – p969!

Try this

Write a paintComponent method that produces the following (without numbers & arrows). The filled box is red, the filled elipse is blue.



Example program

The following slide shows a complete working program that creates a drawing (the colored cross from Slide 6). It is in two separate files, Cross.java (the **application** class) and DrawIt.java (the **support** class). Programs that you use in Lab 5 have this structure.

For today we are mostly interested in what's going on in paintComponent. This method manages the drawing, which happens in a 400x300 panel (JPanel). We look at the OO details in the next lecture.

Optional preview of the next lecture: The program makes a visible window (a JFrame object) and adds a panel (JPanel object) to it. Drawing happens in the panel. The panel is an instance of my class DrawIt which **extends** the Java library class JPanel. Extending a library class lets you configure or add to what that library class can do.

```
/* Anthony Robins, first working graphics demo */
import javax.swing.*; import java.awt.*;  // all graphics
public class Cross {
   public static void main( String[ ] args ) {
      JFrame f = new JFrame();
      DrawIt d = new DrawIt();
      f.getContentPane().add(d);
      f.pack();
      f.setVisible(true);
   }
}
```

File: Cross.java

```
import javax.swing.*; import java.awt.*;  // all graphics
public class DrawIt extends JPanel {
   DrawIt() {
      setPreferredSize(new Dimension(400,300));
   }
   public void paintComponent ( Graphics g ) {
      g.setColor(Color.red);          // set color for drawing
      g.drawLine (0, 0, 400, 300);  // draw red line  \
      g.setColor(Color.green);        // set color for drawing
      g.drawLine (400, 0, 0, 300);  // draw green line /
   }
}
```

File: DrawIt.java

More on this program next lecture.

Today we focus on the paintComponent method which does the actual drawing.

It is called automatically when the window is drawn, and passed a Graphics object (referred to by local variable g).

We call methods on the Graphics object to draw.

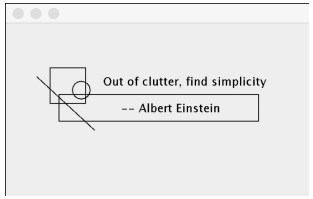
Try this

At the moment the size of the DrawIt panel (JPanel) and drawing depend on hard coded values 400 and 300, which are repeated. This is poor design, what if we wanted to draw a different size? Write a version of the DrawIt class that uses data fields instead of repeating hard coded values.

Einstein example

Replace the method paintComponent in the previous program with this one and you get more interesting output.

```
public void paintComponent(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	Counting to 10 in assembly language

23fc 0000 0001 0000 0040
0cb9 0000 000a 0000 0040
6e0c
06b9 0000 0001 0000 0040
60e8
	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:

- 4000+ programming languages...
- Impact of the internet and the Web - exciting times! HTML, XML, network portable applications and "applets", **Java**.
- Better integration with databases.

2000+:

- Languages for distributed and parallel / concurrent computing.
- Impact of Artificial Intelligence / Neural Networks.
- Quantum computing?

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 – artificial intelligence techniques, inference languages
- 6GL – neural networks, machine learning

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 –

Objects 1 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

A program can be just **an application class** (with main), represented (when the program runs) by one **class object** (data fields and methods must be **static**)

But usual OO design whould include **support classes** that we make **instance objects** from (data fields and methods don't need to be **static** – later!)

Recall that one **class object** is created automatically for every class (including the application class, suppport classes and library classes)

Various i**nstance objects** may be created from the class objects (for some library classes this happens automatically)

See the reading: 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**).

Constructing an instance object

Instances are usually created with the keyword **new**. In Example 1 the relevant statement is:

```
JFrame obj1 = new JFrame();
```

This declares a local variable obj1 of type JFrame (JFrame is a class in the libraries).

It initialises the variable with a reference to a **new** instance object created using a constructor (Slide 13) for the JFrame class.

See Example 1 (second notes for this lecture) for a model of the objects in the program.

Example 1 (see Second notes)

Constructs one instance of type / class JFrame (from the libraries), referred to by obj1.

It calls methods on the instance to draw a representation of it on the screen (a window / frame).

JFrame is one of the many classes in java.awt and javax.swing (part of the libraries) that can be drawn as Graphical User Interface (GUI) elements – later!

Usually we can't "see" the objects in a program, but graphical examples are a good way to introduce the concept of instance objects…

Example 2 (see Second notes)

Constructs two instances of JFrame (and make them look different on screen by giving them different background colors).

We can make as many intances of a class (instance objects from a class object) as we like.

Example 3 (see Second notes)

Constructs one instance of my own support class Book (not a library class).

This the first time we focus on
writing our own 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.

There is no graphics / visible reresentation of the Book instance object on screen, but the program makes some standard text output.

Example 4 (see Second notes)

Constructs various instance objects. This is the Cross program from Lecture 5 (with some extra comments added). It draws some graphics within a panel (within a frame / window). Now we're looking more at the OO detail.

It involves my **application class** Cross, my **support class** DrawIt, various support classes from the libraries (including JFrame, JPanel and Graphics), and instances of (at least) three of those classes

Every class is represented by a **class object**. The main method makes an **instance** of JFrame referred to by local variable f, and an **instance** of DrawIt referred to by d. (DrawIt extends the library class JPanel, extending a class lets you configure or add to what that class can do, later!)

The paintComponent method uses an **instance** of the library class Graphics referred to by local variable g. (This instance object is created and passed to the method automatically, the method is also run automatically, Java graphics is complicated.)

Try this

Consider Example 3. Write a new main method 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

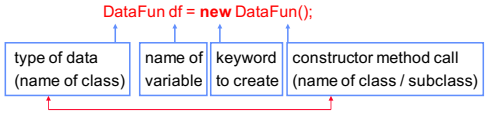
Accessors and mutators are perfectly ordinary methods. Constructors are not! We've already used them (Example 3, Book), time to look at what they are...

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

The example on Slide 9 used a default constructor:



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 DataFun, as follows:

```
public DataFun() { }
```


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 2015, 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.getInfo() );
      System.out.println( point2.getInfo() );
      System.out.println( point3.getInfo() );
   }

} // PointApp
```

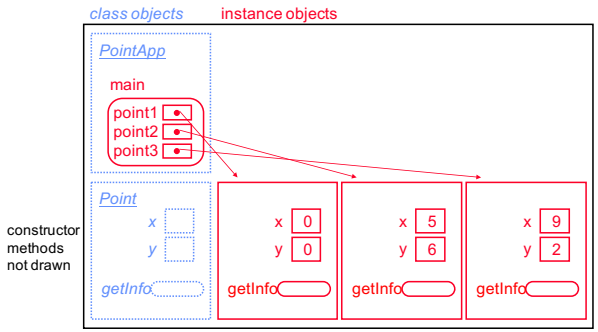
Output:

```
A point 0 0
A point 5 6
A point 9 2
```

```
/* Anthony, Aug 2015, 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 getInfo() {
      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 7).

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.

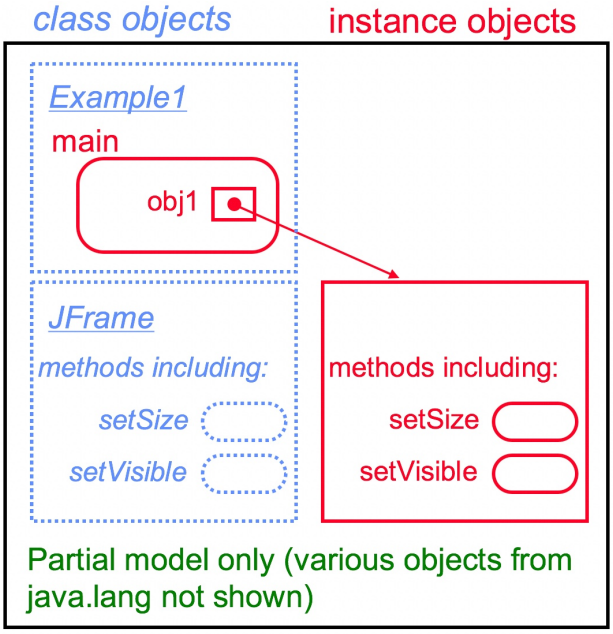
Write a constructor that takes one input and sets all three data fields to that value, and a constructor which takes three different inputs and assigns them to the three data fields.

COMP160 Lecture 6 Second notes

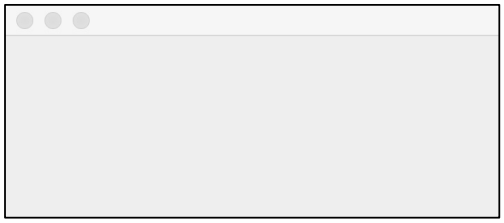
Example 1 Construct and display a single frame (library class JFrame instance object)

```
/* Anthony, May 2017, JDK 1.8,  Construct a JFrame object    */

import javax.swing.JFrame;  // import the JFrame class

public class Example1 {

   public static void main(String [ ] args) {
      // construct a JFrame instance object then set it up
      JFrame obj1 = new JFrame();
      obj1.setSize(350, 150);          // call a method to set the size
      obj1.setVisible(true);           // call a method to make it visible
   }

}
```

File Example1.java



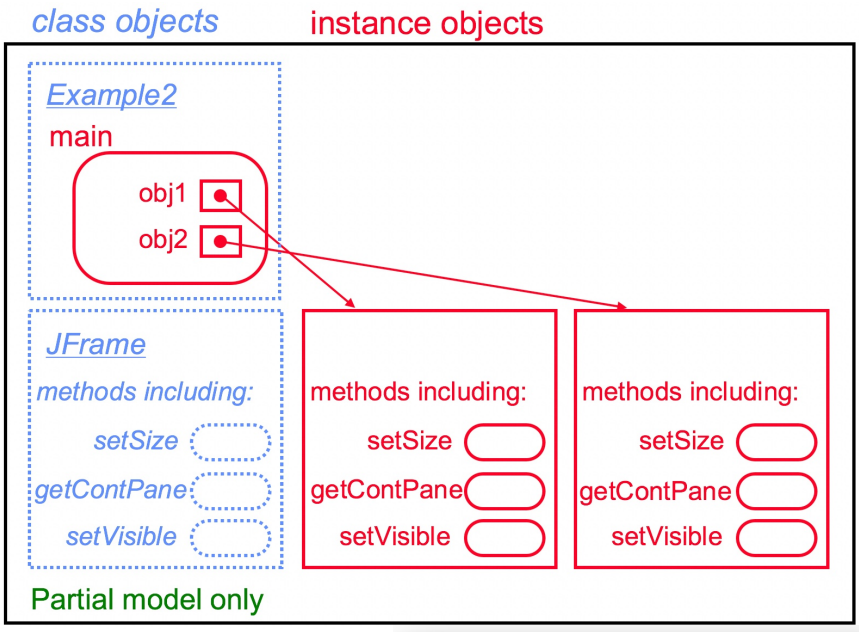
Output: The frame appears as a standard window on the device.



Example 2 Construct and display a two frames (library class JFrame instance objects)

```
/* Anthony, May 2017, JDK 1.8,  Construct two JFrame objects  */
import javax.swing.JFrame;  // import the JFrame class
import java.awt.Color;          // import the Color class
public class Example2 {
  public static void main(String [ ] args) {
    // construct a JFrame instance object then set it up
    JFrame obj1 = new JFrame("Object 1"); // create the instance
    obj1.setSize(350, 150);                    // call a method to set the size
    obj1.getContentPane().setBackground(Color.blue);  // color the background
    obj1.setVisible(true);                     // call a method to make it visible
    // do it again for a second JFrame instance
    JFrame obj2 = new JFrame("Object 2");
    obj2.setSize(350, 150);
    obj2.getContentPane().setBackground(Color.red);
    obj2.setVisible(true);  // on screen it appears on top of the first one!
  }
}
```

File Example2.java



Output: The frames appear in the same location, so the top one has to be moved aside to see the one underneath. Can place in specified locations with e.g. obj2.setLocation(200, 200);

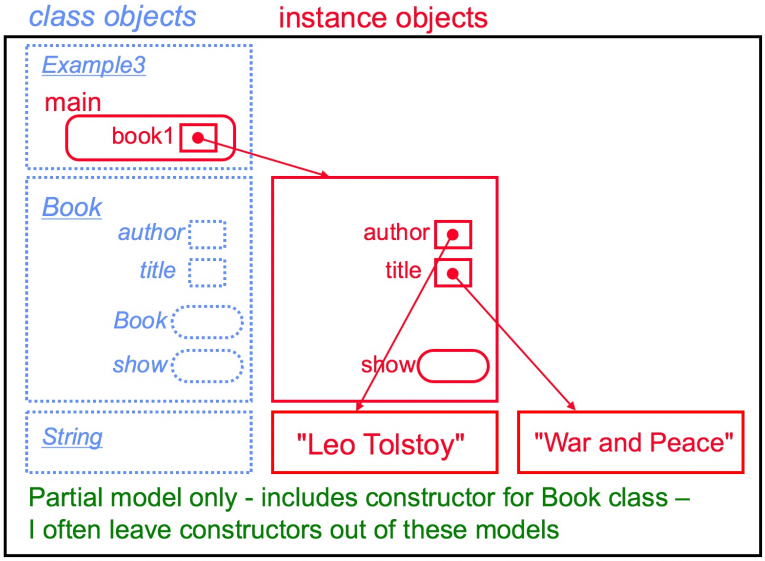
Example 3 Construct and use a Book (my class Book instance object)

```
/* Anthony, May 2017, JDK 1.8, Construct a Book object */
public class Example3 {  // application class
  public static void main(String [ ] args) {
    // construct a Book instance object set up with constructor
    Book book1 = new Book("War and Peace", "Leo Tolstoy");
    // use the object
    book1.show();
  }
}
```

File Example3.java

```
/*  Anthony, May 2017, JDK 1.8, Represent books by title and author  */
public class Book { // support class for Example 3
  private String title;
  private String author;
  Book (String t, String a) {  // constructor – see Slide 13
    title = t;
    author = a;
  }
  public void show() {
    System.out.println( title + " by " + author );
  }
}
```

File Book.java



Standard text output (no graphics):

War and Peace by Leo Tolstoy

Example 4 Construct various objects (this is the example program from Lecture 5)

```
/* Anthony Robins, May 2017, JDK 1.8 , From Lecture 5 Slide 10
 * Construct and use various objects for graphics */

import javax.swing.*;  // import all graphics classes
import java.awt.*;

public class Cross {  // application class
   public static void main( String[ ] args ) {
      // construct instance objects
      JFrame f = new JFrame();   //construct JFrame instance
      DrawIt d = new DrawIt();   //construct DrawIt instance (a JPanel)
      // set up the frame (appears a window on device)
      f.getContentPane().add(d);  // add panel to frame
      f.pack();       // size the frame around its contents
      f.setVisible(true);    // make the frame visible
   }
}
```

File Cross.java

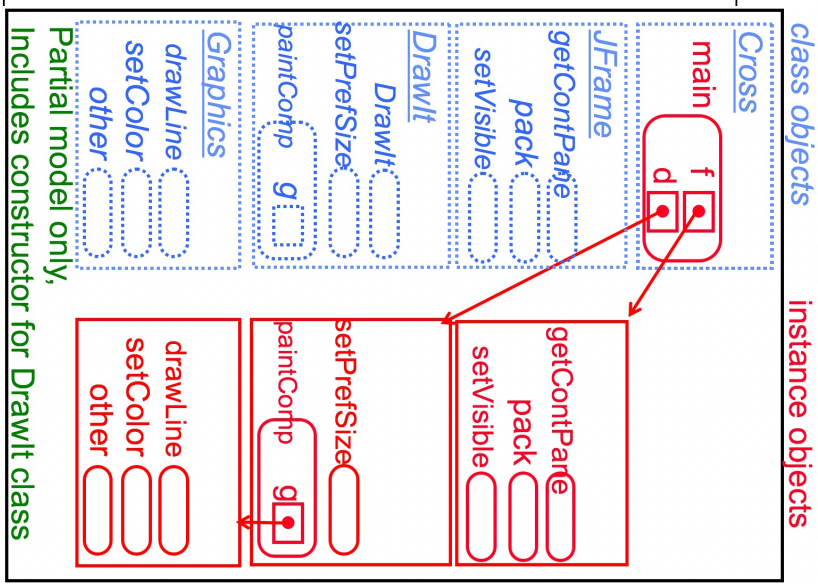
```
/* Anthony, May 2017, JDK 1.8, Simple graphics in a panel */

public class DrawIt extends JPanel { // support class for Cross

   DrawIt() { // constructor - see Slide 13
      setPreferredSize(new Dimension(400,300));  // set size of panel
   }

   public void paintComponent ( Graphics g ) {
      g.setColor(Color.red);           // set color for drawing
      g.drawLine (0, 0, 400, 300);  // draw red line  \
      g.setColor(Color.green);         // set color for drawing
      g.drawLine (400, 0, 0, 300);  // draw green line /
   }
}
```

File DrawIt.java

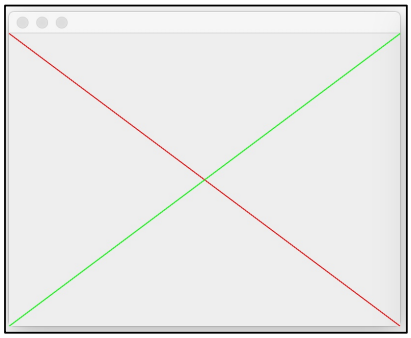
The main method makes an instance of the library class JFrame (referred to by local variable f) and an instance of my support class DrawIt (referred to by d).

DrawIt extends the library class JPanel, so it is a JPanel. Extending a class lets you configure or add to what that class can do. That is how DrawIt has a setPreferredSize method from the library that it uses. More on this Later!

The JFrame instance is seen as a window on the device and it contains the JPanel (which shows the cross drawing).



Output:



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.

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**).

Data fields and methods are called **members** of a class. A member can be declared with **static** (it is part of the **class** object, a **class member**), or without static (it is part of any **instance** objects from the class, an **instance member**).

the terminology is a bit confusing

Class members (static) 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 class members (static), support classes usually not.

The program can use **static** / class members and the instance members of instance objects.

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

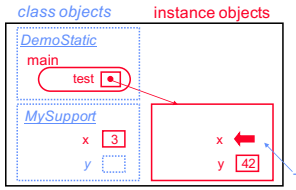
(4) We can seem to use class members in instance objects: variableName.memberName but we are really using the member in the underlying class object instead.

See Cases 1 - 4 on the next slide in an example using data fields…


```
/* Anthony, April 2015, JDK 1.7, 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:

"dot notation"

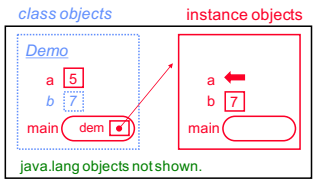
Math.PI	*a data field in the class object Math (Lec 5)*
Math.sqrt()	*a method in the class object Math (Lec 5)*
MySupport.x	*a data field in the class object MySupport (Slide 6)*
MySupport.hello()	*a method in the class object MySupport (Slide 6 - if class MySupport had a static method called hello)*

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 printlns will fail, and why?

An application class making an instance of itself is unusual design, but fine.

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() );
```

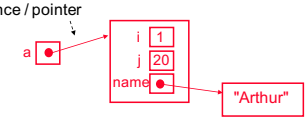
Output:

```
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).

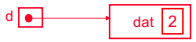
An object is distinct and separate from a reference to the object.

(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

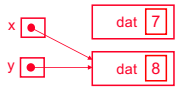
Key concept: a and b are aliases

```
a.dat = 5;  // dat must be public
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!

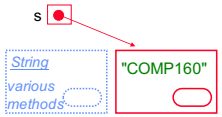
Strings

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");

Usual partial model



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.

+ 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) or string:

hi.indexOf('e') is 1 hi.indexOf('x') is –1 "wibble".indexOf('w') is 0

not found

hi.indexOf("lo") is 3

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("ne");

String s = dun.substring(1, 5);

String t = dun.substring(0, dun.indexOf('e') + 1);

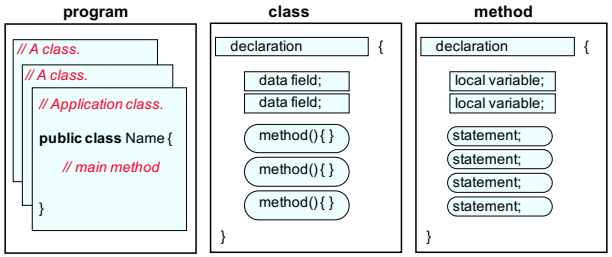
Structured programming, more maths

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 2015, JDK 1.7
   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 2015, JDK 1.7 */

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-"

```
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);
  }
```

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);
    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

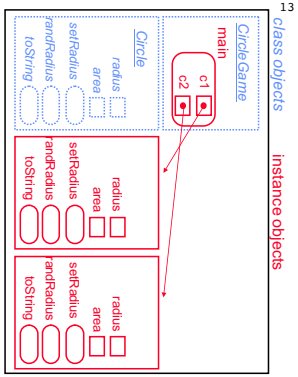
LDC 3.5 discusses the class Math (in java.lang so automatically imported). Math defines useful mathematical:

- 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 2015, JDK 1.7,
    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
```

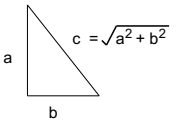


```
/* A class to represent circles, with specified radius or random radius.  Anthony, JDK 1.7 */

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 readDouble method (like readInt Lec 4 – it returns a double entered by the user). Complete the method calcHyp so that it uses readDouble 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 ( ) {
    double a = readDouble("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/cstdio/printf/

Wrapper classes

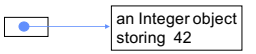
In some languages "everything is an object". In Java we occasionally want to treat primitive values like objects (e.g. to store them in collections of objects - later).

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.

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 of 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) {
    one or more 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 2

COMP160 Lecture 10
Anthony Robins

- Introduction
- if..else continued
- Nested and multiple if..else
- Switch
- Scope and design
- Short-circuit evaluation

Reading: LDC: 4.2, 4.4

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.

Read a character and count
- any 'a'
- any vowels ('a' is also a vowel)
- consonants (everything else)

```
ch = readChar("Enter lowercase letter: ");
switch( ch ) {
    case 'a':
        aCount++;  // no break!
    case 'e': case 'i': case 'o': case 'u':
        vowelCount++;
        break;
    default:
        consonantCount++;
} // end of switch
System.out.println("Results are..." );
```


Scope and design

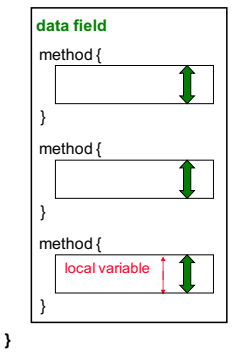
Where a variable can be "accessed" is called the **scope** of the variable.

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).



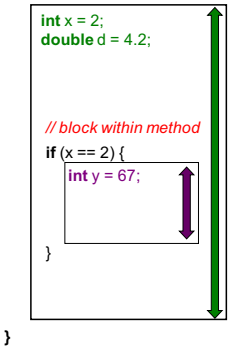
Scope within a method

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



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.

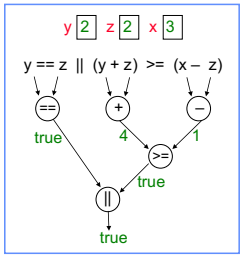
Short–circuit evaluation

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!).

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.

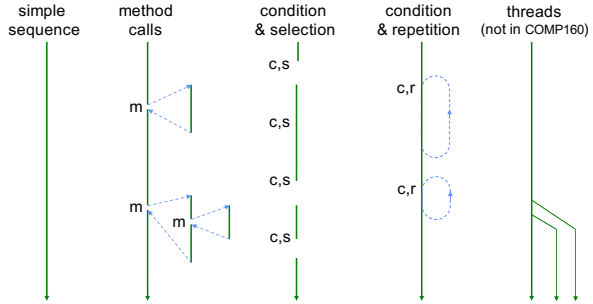
Introduction

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:
6
Incrementing to 10
6
7
8
9
10
all done
```

```
Enter an Integer:
12
Incrementing to 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:
6
Incrementing to 10
6
7
8
9
10
all done
```

```
Enter an Integer:
12
Incrementing to 10
12
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 sum with an unknown starting value. As long as sum is less than 100, we want to repeat the following process: read an integer value, if the value is greater than 10 print "big", otherwise print "small", then (in either case) add input to sum.

(4) Read in an integer value. If it is even, 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
```



```
int i = 1;
int sum = 0;
do {
    System.out.println("Hello " + i);
    sum += i;
    i++;
} while(i <= 5);
System.out.println("Sum " + sum);
```

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;
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 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
```

Slide 12

```
while (sum < 100) {
    int input = readInt("Enter an integer:");
    if (input > 10) {
        System.out.println("big");
    } else {
        System.out.println("small");
    } // end if
    sum += input;
}
```

```
int input = readInt("Enter an integer:");
if (input % 2 == 0) {
    for (int i = 1; i <= input; i++) {
        System.out.println( i );
    }
} // 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"

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 2015
   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…

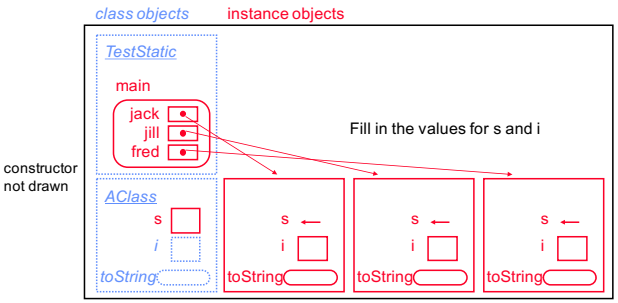
Try this

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".

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;
```

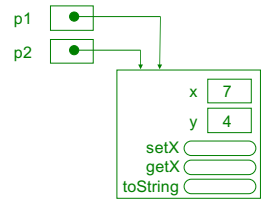
a 3
b 3

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):

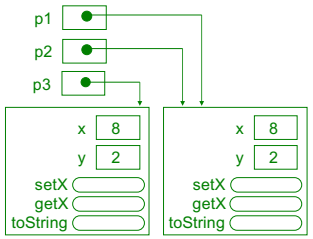
```
int a = 3, b = 3;
// a == b is true
```

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
```

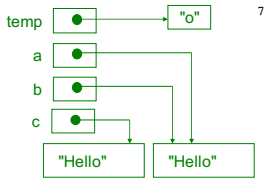
Objects with the same state (values for their data fields) are **not** the same object!



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.

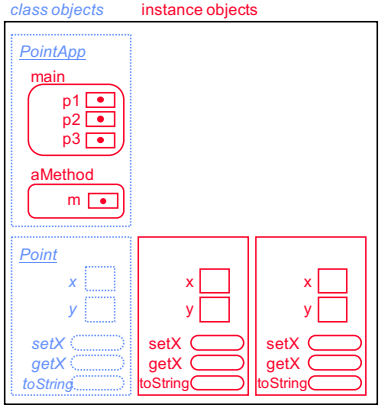
Try this

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 2015, JDK 1.7 */
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
```

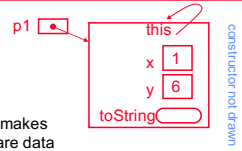

class objects instance objects



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 **parameters (and other local variables) can have the same name as data fields** (do this for clarity e.g. in a constructor or mutator that sets the data field).

```
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 are local variables
this.x and this.y are the data fields.

Within the toString method:
x and y are read as / could be written as this.x and this.y the data fields.
(Because there are no local variables of those names).

this.name	always refers to a data field in the current class
this.name()	always refers to a method in the current class
name	refers to a local variable (if it exists) or is read as this.name
name()	is read as this.name()

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.

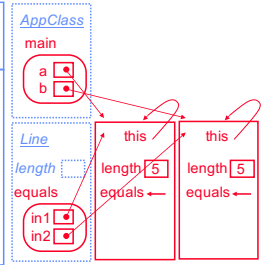
Example 2 shows the use of a class data field to represent information about the whole class / type (all the instances). The imagined task has the simplified structure of some graphics application where there are three individual "button" objects. The underlying class object has a data field which is 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!

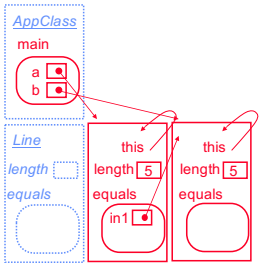
constructors not drawn

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

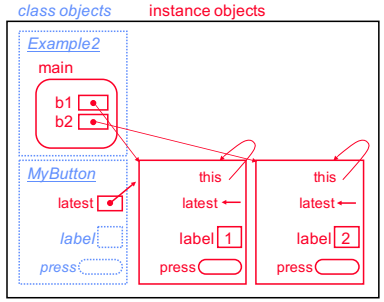
```
/*  Anthony, April 2019, this and class */

public class Example2 {

  public static void main(String [ ] args) {

    MyButton b1 = new MyButton(1);
    MyButton b2 = new MyButton(2);

    b2.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
Latest 2
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

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

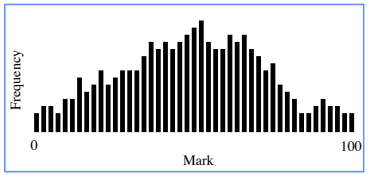
LDC 7.3 – next lecture

LDC 7.4, 7.5 – your own reading (7.4 is relevant to Lab 17)

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?



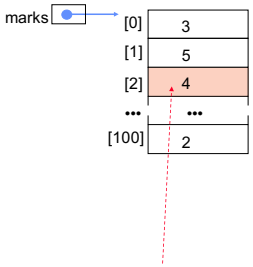
We could use 101 variables, mark0, mark1, mark2 and so on – but that's not pretty!

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] );
```

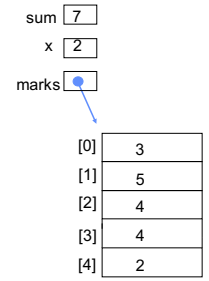
If you try to use an index that is **out of range** the interpreter will generate an error:

```
marks[200] = 5;      // run time error, message in console window
```


Try this

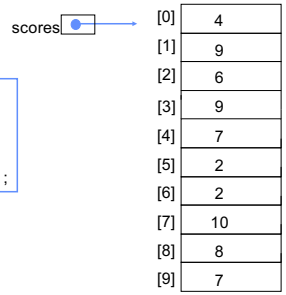
What is the effect of the statements below, given the array and the int variables with initial values shown on the right:

```
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:



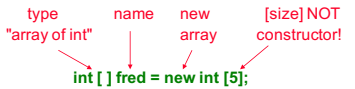

```
double sum = 0.0;
for (int i = 0;  i < 10;  i++) {
    sum += scores[ i ];
}
System.out.println("Average: " + (sum / 10) ) ;
```

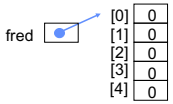

Array types

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 a ref. to array
```

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 a ref. to an array of 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!

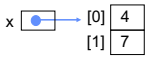
For example, the following definitions of x are equivalent:

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
```

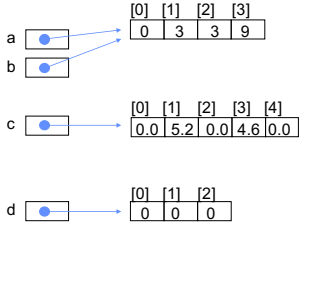
```
int [ ] x;  // 1
x = new int[2];  // 2
x[0] = 4;  // 3
x[1] = 7;  // 3
```



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.

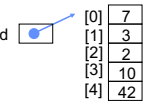
Here's an example use, a method which takes an array of integers as input and returns the average of the values stored in the array:

```
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

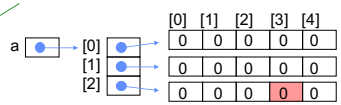
The arrays we have seen so far have all been "**one dimensional**" (1D), a single list of data (e.g. previous slide).

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]

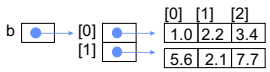
The length of the array, a.length, is the number of rows, 3.
The length of a row, e.g. a[0].length, is the number of columns, 5.

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

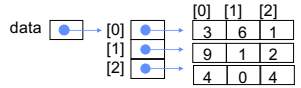
```
// print a 2D array called b
for (int row = 0;  row < b.length;  row++) {        // for each row
    for (int col = 0; col < b[row].length; col++) { // for each column in row
        System.out.print( b[row] [col ] + " " );    // print element
    }
    System.out.println();                           // 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

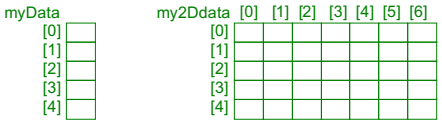
COMP160 Lecture 16
Anthony Robins

- Arrays are flexible
- Arrays are objects
- Arrays of (references to) objects
- More on using arrays
- Main explained

Reading: LDC: 7.3
LabBook: Reference Types

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.

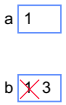
Their internal structure (references to arrays within arrays)
lets us access and modify them (and parts of them) in a variety of ways.

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
```



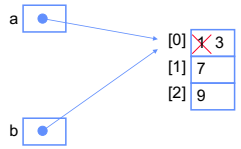
a and b are separate values (ints).

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
```



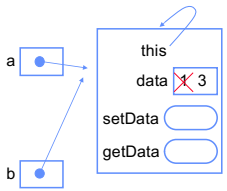
a and b are separate values (references to an array, in this case aliases for the same array - Lec 7).

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).

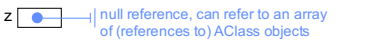
myMethod does nothing to the value of b, but it changes the value of b.data (changes the state of the object b refers to).

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.

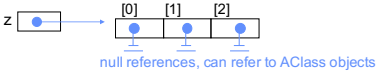
(1) Declaring:

```
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.
```

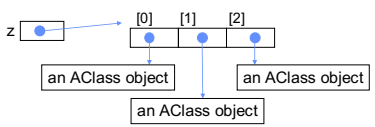


As usual we can combine these steps in one statement:

```
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).

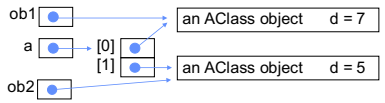
```
AClass [ ] z = { new AClass(), new AClass(), new AClass() };   // same model
                                                               // as above
```


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:

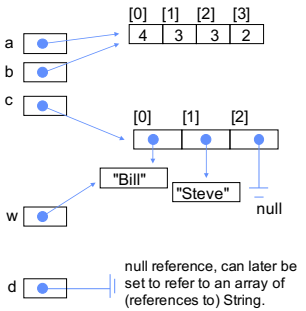
```
AClass ob1 = new AClass(7); // ob1 refers to a new object with d = 7
AClass [ ] a = new AClass[2]; // create array of (null) refs to 2 AClass objects
a[0] = ob1;                 // a[0] refers to the same object as ob1
a[1] = new AClass(5);       // a[1] refers to a new object with d = 5
AClass ob2 = 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

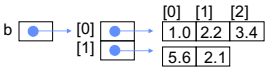
After the declarations and statements on the left have executed, show the arrays on the right. Assume that AClass is a class defined in the program.

```
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

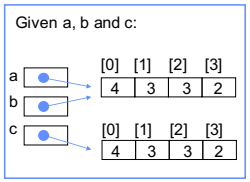
See Lec 14, and the Lab book reading **Reference Types** for background.

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.

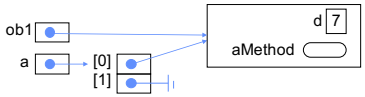
The variable can be an indexed variable (cell of an array).

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).

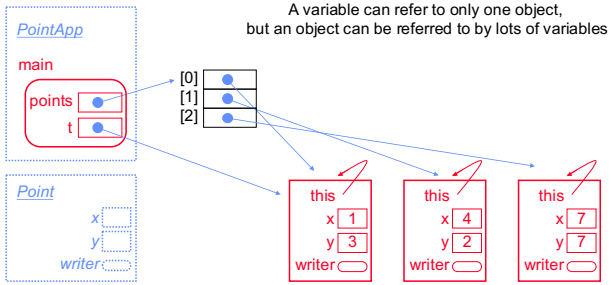
Write a for-each loop (Slide 12) that does the same task.

```
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

Our graphics examples (e.g. Slide 4) use WindowConstants.EXIT_ON_CLOSE
The textbook uses the older JFrame.EXIT_ON_CLOSE
which still works but it's being replaced

Reading: LDC: 6.1 – 6.3 (will be same for next lecture too)

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 panel added to a frame (Lec 5, Lab 5, 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:

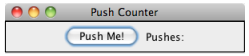
```
import javax.swing.JFrame; import javax.swing.WindowConstants;  // required resources

public class MyAppClass {

  public static void main(String [ ] args) {                    // Steps:
    JFrame frame = new JFrame("Push Counter");                   // 1 make frame / window
    frame.setDefaultCloseOperation(WindowConstants.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.

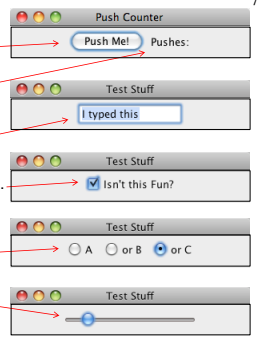
Label (JLabel): displays a line of text.

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.

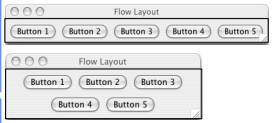
The first example has no explicit layout manager, so the default flow is used. The second example uses border layout. For more see LDC 6.3.


```
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);
}
```

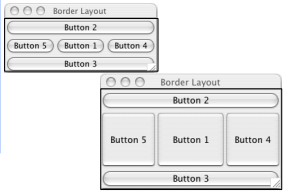
```
public MySupportPanel () { // constructor
  setLayout( new BorderLayout() );
  JButton b1 = new JButton("Button 1");
  JButton b2 = new JButton("Button 2");
  JButton b3 = new JButton("Button 3");
  JButton b4 = new JButton("Button 4");
  JButton b5 = new JButton("Button 5");
  add(b1, BorderLayout.CENTER);
  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)



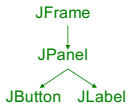
Containment hierarchies

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:

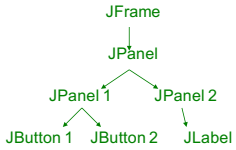
1: Create components
2: Add components to panels
3: Add inner panels (if any) to the main / outer panel
4: Add the outer panel to the (content pane of the) frame (or other window)

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.

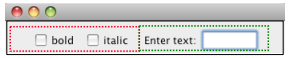
I prefer the previous version, make all components then add them.

```
// 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. Sketch the containment hierarchy.



Inner classes

To prepare for events (next lecture).

Inner classes are a feature for safety / encapsulation.

An inner class is declared inside another class (it is a member of the outer):

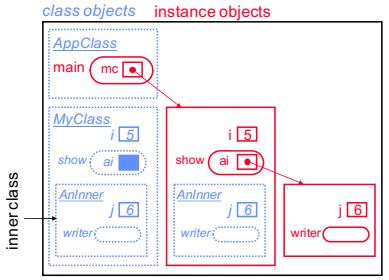
(1) Only the outer can see / create instances of the inner.
(2) Members (e.g. data fields) of outer are in scope for inner (here outer data field i is used by AnInner).

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 {  // outer class
   private int i = 5;

   public void show () {
      AnInner ai = new AnInner();
      ai.writer();
   }

   // this is an inner class – make private
   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).

An interface is like a restricted form of a class. They may contain only:

- **static final** data fields: static = a class member, final = cannot change value,
- **abstract** methods: a method with a declaration / header but no body

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. Named in all CAPS, e.g. Math.PI

In an interface the modifiers abstract, public, static and final are **all optional** because they are assumed by default if they are omitted.

Other classes may **implement** one or more interfaces. Such a class has access to (inherits) the interface's 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.

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 interfaces:
WindowListener - to define all methods required when the window is opened, closed, selected etc.
ActionListener - to define all methods when a button in the window is clicked.

Other example interfaces:
Comparable - specifies e.g. that a class must have a compareTo method (implemented by e.g. String).
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 2018, 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("ASquare 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

- Introduction
- Event model
- Finding an event source
- Reading the code

See second notes document for this lecture.

Reading: LDC: 6.1 – 6.3

Introduction

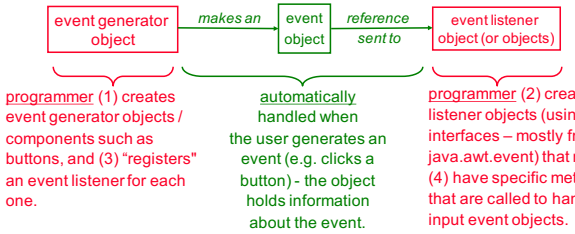
Last lecture we looked at the components of a GUI, how to group and organise them (into a containment hierarchy), how to lay them out, and a design for graphics programs (application class creates a frame with the content pane set to a panel, and the work done in the constructor for the panel object).

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!



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)

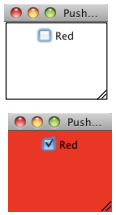
This example builds a GUI with three sliders that are used to set the background colour of the panel displayed in the JFrame. (Java follows a standard convention of representing a colour as a mixture of three primary colours, red, green and blue, with intensity values in the range 0 to 255). The key 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.

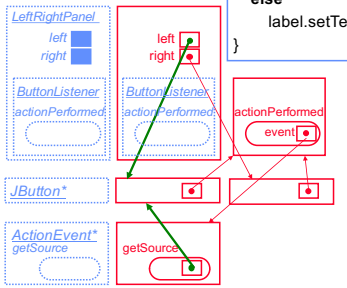
In your reading, try and identify the four steps of setting up event handling (Slide 3).

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");
}
```



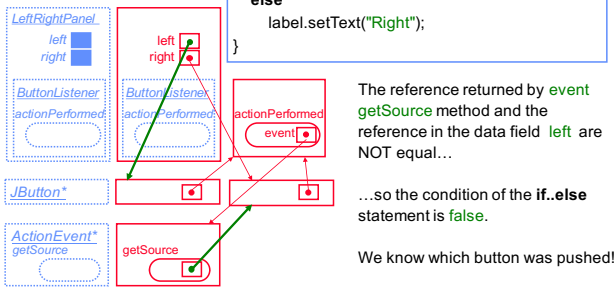
The reference returned by event getSource method and the reference in the data field left are equal (refer to same object)…

…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(WindowConstants.EXIT_ON_CLOSE);  // 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 WindowConstants 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;
import javax.swing.WindowConstants;

public class MyAppClass {

   public static void main(String[ ] args) {
      JFrame frame = new JFrame ("Push Counter");
      frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
      frame.getContentPane().add( new MySupportPanel() );
      frame.pack();
      frame.setVisible(true);
   }

}
```

Application class is the same as Lec 17 Slide 4:

creates a temporary reference to a new object that is not stored in a variable

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
```

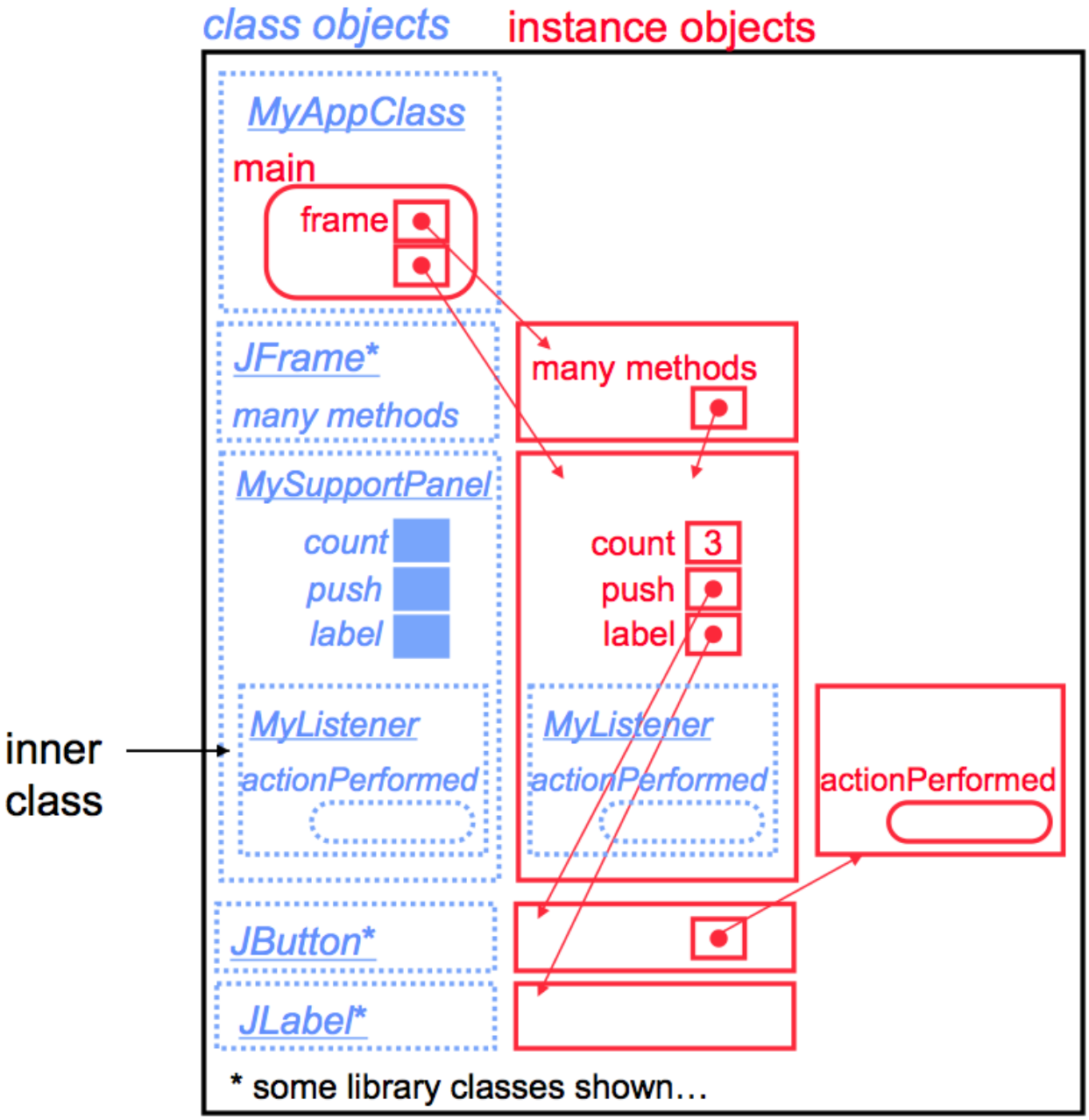
Variables must be data fields to allow access by inner class.

count declared and initialised, used to count button presses.

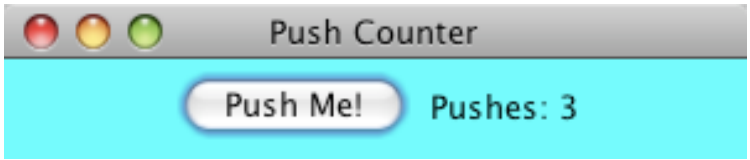
Creates and registers listener object for button

Sets background color of the panel.

Inner class added to make listener object for the button (JButton).



Output after 3 clicks on button:

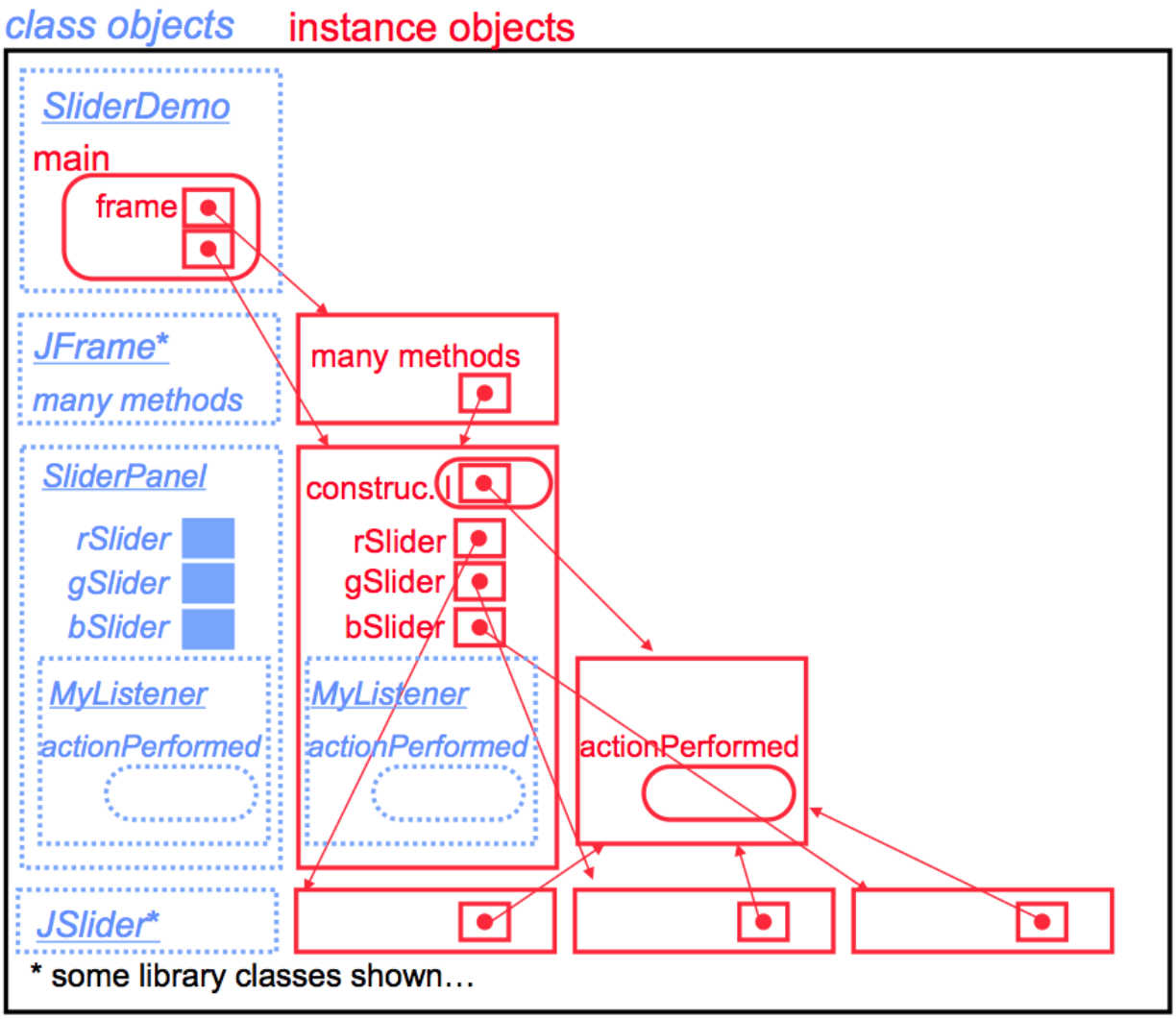


Example 2

```
/* Anthony & Michael Albert, May 2017, JDK 1.8
   Demonstrate events using sliders to set background colour */

import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class SliderDemo {

   public static void main (String[ ] args) {
      JFrame frame = new JFrame ("Slider Demo");
      frame.setDefaultCloseOperation( WindowConstants.EXIT_ON_CLOSE);
      frame.getContentPane().add( new SliderPanel() );
      frame.pack();
      frame.setVisible(true);
   }

}
```

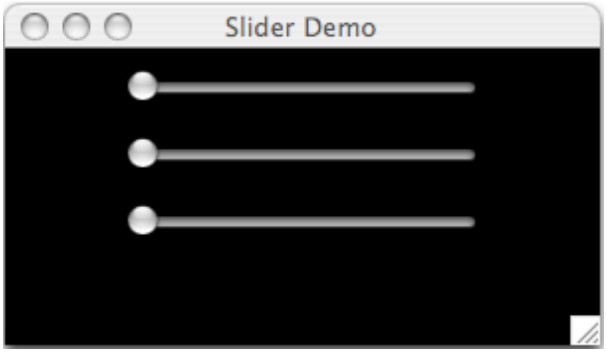
creates a temporary reference to a new object that is not stored in a variable

```
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)

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 );
```



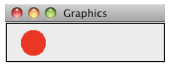
```
if (e.isSelected() ) extrasString = extrasString + e.getText() + " ";
```

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.
```

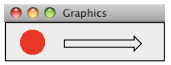
```
/* Anthony, Sept 2015, JDK 1.7, Demo Timer */
import java.awt.event.*;
import javax.swing.Timer;

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);  // initialise (parent - 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 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").

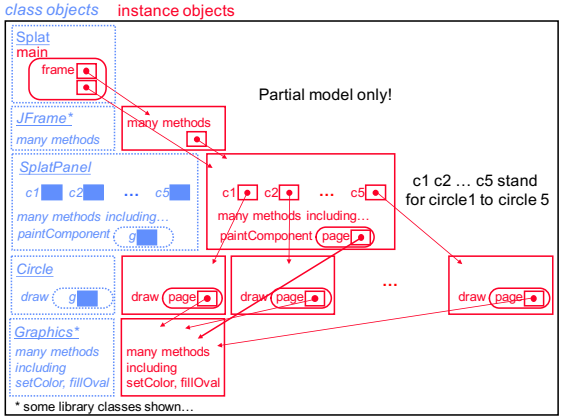
In class SplatPanel the method paintComponent doesn't do any drawing directly. Instead, it calls a method on each Circle object and passes it a reference to the Graphics object for the panel.

```
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 (WindowConstants.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, May 2017, JDK1.8.  Example GUI handling events from checkboxes and
radiobuttons.  Features arrays of (references to) these generator objects. */

import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class PizzaApp {

  public static void main (String[ ] args) {
    JFrame frame = new JFrame ( "Pizza Order" );
    frame.getContentPane().add( new PizzaOptionsPanel() );
    frame.setDefaultCloseOperation( WindowConstants.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
- Exceptions - try..catch
- Files and Input Output (I/O)
- Sorting

Reading: LDC: 2.6, 4.6, 10.1, 10.2, 10.3, 10.6.
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).

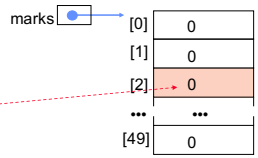
A declaration:

```
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.

Exceptions - try..catch

Java was designed to be robust – deal with **exceptions** (errors) well. This is called **exception handling** (LDC Ch 10). There are three exception handling options:

(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) Option 1 is not allowed, the compiler forces us to deal with possible exceptions.

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).

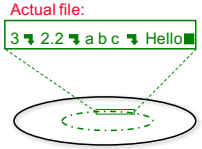
We will use a **Scanner** object to read data from a file (one kind of stream). We have seen similar examples, using a Scanner to read from System.in or a String.

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 = 0;  // initialise index for array
   while ( sc.hasNext() ) {
      Scanner line = new Scanner( sc.nextLine() );
      line.next();  // read / discard first token
      ids[ i ] = line.nextInt(); // read next token as int
      i++;  // increment array index
   }
} 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 );
}
```

ids

[0]	34792
[1]	42424
[2]	84421

etc…

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?

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):

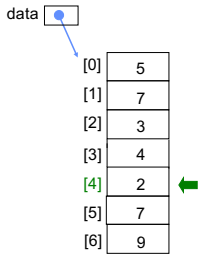
1. Find the smallest value (Slide 16) and swap it with the "top" vaue (Slide 17)
2. Move where you think the "top" is one step down
3. Repeat until there's nothing left at the "top" (reached end of the list)

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

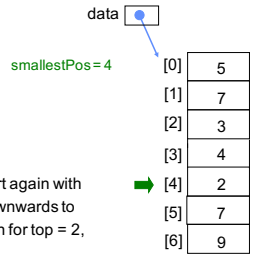
The core of selection sort is to keep finding the smallest value. Given an array **data**, write code that finds the **position** of the smallest item in the array (in this e.g. array position 4) and stores it in a variable. You need a **for** loop, and an **if**…



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.

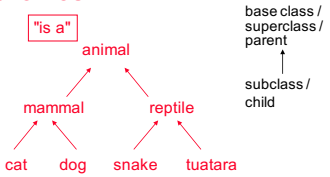
We start with a basic tool for organising information, the hierarchy. Hierarchies can prevent unnecessary work / duplication (one aspect of the "software reuse" that is a supposed advantage of OO languages)…

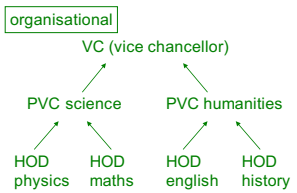
(The lecture contains lots of versions of classes Person and Student, the different versions illustrate different points).

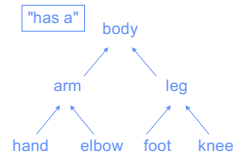
Hierarchies

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 class MyClass {
    // class body
}
```

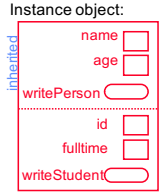
```
public class MyClass extends Object {
    // class body
}
```


Extends

The superclass Person has a subclass Student.

This is defined by the use of the keyword **extends** in the subclass header.

Instance object:



Examples in this lecture use public visibility, which is not ideal. See more on visibility in Lec 22…

```
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);
    }
}
```

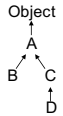

Try this

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.

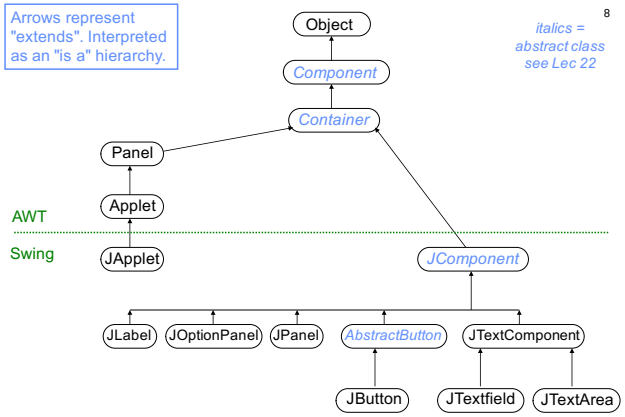
A class inherits from its parent, and its parent's parent (and so on). So D inherits members from C and A (and Object). This is how objects get methods, such as toString, that aren't declared in their own class body.

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. More in Lecture 24.

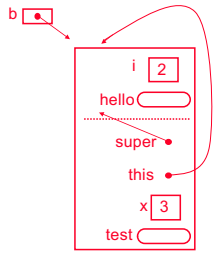
Interface

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 7, 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 );
```

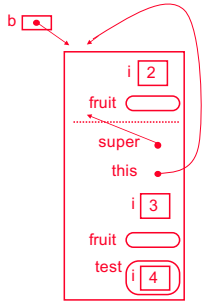
writes out:

```
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 .

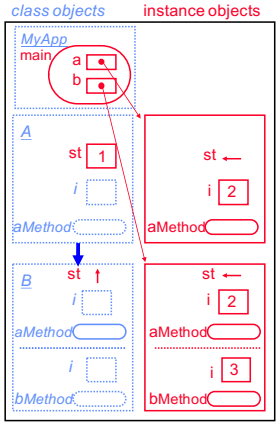
In a hierarchy class members are shared by all instances, includes all instances of subclasses…


```
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, and more

COMP160 Lecture 24
Anthony Robins

- Enumerated types
- Generics
- Collections
- ArrayList
- Stack – an example ADT
- Applets

Reading: LDC: 3.7, 12.1, 12.2, 12.3, 12.6, 12.7

Enumerated Types

If I'm working with a task involving fruit I might want to use the values: apple, orange, banana, cherry, grape.

Enumerated types (LDC 3.7) lets a program use the "natural values" or "natural language" of the task. We create a new type by enumerating (listing) every possible value.

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 look quite the same as usual classes.


```
/* 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 most ways like other classes.

As well as having built in methods, we can add our own data fields, constructors, or methods as usual.

```
enum Fruit {apple, orange, banana, cherry, grape;

    public String message() {
      return "is good for you";
    }

}
```

```
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

Generics are a conceptual framework where programs can be written in terms of "to be specified later" types that are **instantiated** when needed as specific types.

Java implements 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

Java has many ways (including arrays) of representing and processing collections of **objects**, using classes and interfaces in java.util. These are called the **collection classes**, **collection framework**, or **collection API** (Application Programming Interface). The collection API (LDC 12.1) 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:

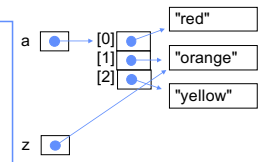
- can hold (refs to) objects (instance objects, strings, arrays, **not** primitive types)
- you don't need to declare the size
- because an ArrayList can grow and shrink as required
- there are methods for useful operations such as adding and removing objects

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") );
```

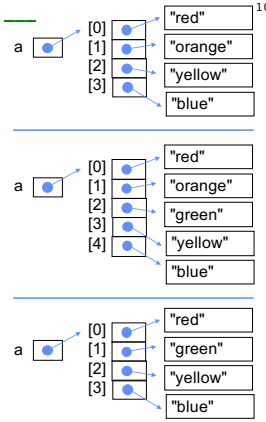



```
---------
red
orange
yellow
---------
orange
2
```

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:

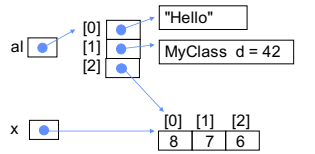
```
ArrayList<MyClass> mc = new ArrayList<MyClass>();
mc.add( new MyClass(7) );
mc.add( new MyClass(42) );
mc.add( new MyClass(9) );
mc.add( 0, new MyClass(17) );
mc.add( 0, new MyClass(2) );
mc.remove(2);
```

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!

Stack - an example ADT

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. Typical example ADTs are:

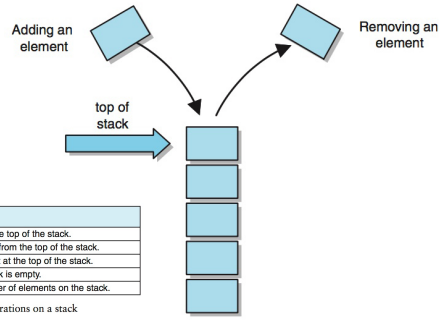
Stack: We can place items on the top of the stack, and remove them from the top of the stack - Last In First Out (LIFO)

Queue: We can place items on the back of the queue and remove them from the front of the queue - First In First Out (FIFO)

We can treat Java collections as ADTs and use them via the interface / public methods without understanding the internal details. (Some of them implement classic examples, there are stack and queue collections.)

Or we can write our own ADTs, the best way is to use classes allowing access to data only via the interface / public methods.

LDC 12.2, 12.6 discusses the stack as an example ADT…



Operation	Description
push	Adds an element to the top of the stack.
pop	Removes an element from the top of the stack.
peek	Examines the element at the top of the stack.
isEmpty	Determines if the stack is empty.
size	Determines the number of elements on the stack.

FIGURE 12.4 The operations on a stack

FIGURE 12.3 A conceptual view of a stack

Implementing a stack

A given ADT can usually be implemented with various actual data types in a language.

LDC 12.7 describes implementing a stack using an Array (actual data type). In design terms this is a clumsy choice because (1) it imposes a maximum size on the stack, (2) you need to manually keep track of an index variables that represents the curent top of the stack (the largest indexed cell of the Array that is currently in use), and (3) you need to manually add and remove items from the stack (manipulate the Array using the index).

It would be a useful exercise to look at this section, and consider how much easier it would be to implement a stack using an ArrayList (at the cost of slower performance). Identify ArrayList methods that correspond to the stack operations on the previous slide.

Applets

A standard Java program is an "application program". Historically **Applet** programs were significant. These run within browsers like Firefox, Chrome or Safari, and have some restrictions for security reasons (e.g. no access to files).

To convert a GUI program you need to write an Applet class instead of an application class, and to create a simple html file that locates the class file.

For the example Pizza program (Lecture 19) the following Applet class, with the unchanged support JPanel class, and with the html file…

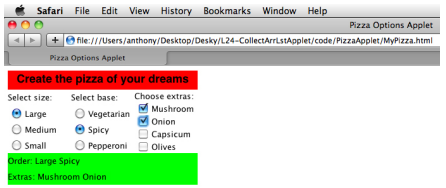
```
import javax.swing.JApplet;  // Applet class
public class PizzaApplet extends JApplet {

  public void init() {  // has init method not main
    getContentPane().add( new PizzaOptionsPanel() );
  }
}
```

All the graphics work is done by the support class which doesn't change at all.


```
<html>
<head><title>Pizza Options Applet </title></head>
<body>
<applet code = "PizzaApplet.class" width = "300" height =
"180" >
</applet>
</body>
</html>
```

file: MyPizza.html

Put this file in the same directory as code, and open with a browser…



Applets are optional (not examinable) but see LDC Appendix G if interested.

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 our graphics design (Lec 17 Slide 4) 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.

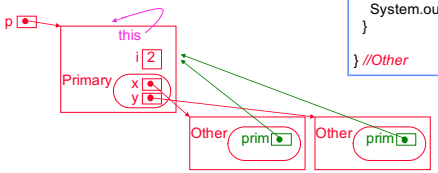
```
public class Primary {
  private int i = 2;

  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 – direct access

If x and y have a reference to p they can use direct access (dot notation) to a visible data field. Here x and y get the reference to p as the value of a method (constructor) parameter. The instance p sends a reference to itself (this).

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();
  }

  private class Other {  // inner

    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.

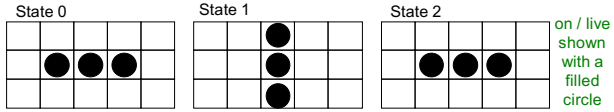
Some simulations show how apparently very complicated behaviour can arise from very simple underlying rules. A classic example is Conway's "Game of Life"…

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
- applets for web pages / simple web based programming

} as application programs for Macs, PCs, Android etc…

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 3, Godot, Ogre3D 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.

A comment on course advice, plus
short presentations on the local IT industry and
interesting topics in the computer science department…

File: Life.java

```
/*
 Anthony, October 2006, JDK 1.5, 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
```