List of Slides

Title

Java Just in Time

John Latham

September 27, 2018

Chapter 2

Sequential execution and program errors

Chapter aims

- Introduce some very basic Java concepts.
  - Especially **sequential execution**.
- Look at kinds of errors we can have in programs.
  - Because you will make errors!
  - You don't need to be afraid of them
    - they are part of the programming experience!

Section 2

Example: Hello world

Aim

AIM: To introduce some very basic Java concepts, including the **main method** and `System.out.println()`.

Class: programs are divided into classes

- Program source text separated into pieces called **class**es.
- Each piece (usually) stored in separate **file**.
- File name is name of class, with `.java` appended.
  - E.g. `HelloWorld` in `HelloWorld.java`.
- One reason for dividing – makes management easier
  - program maybe thousands of lines.
- Another reason: make sharing between programs easier
  - **software reuse** helps productivity.
- Every program has at least one class.
- Its name reflects intention of the program.
- Convention: class names start with upper case letter.

Class: public class

- A **class** declared **public** can be accessed from anywhere in the running Java environment;
  - in particular the **virtual machine** can access it.
- Source text starts with **reserved word** **`public`**.
- A reserved word is part of the Java language
  - e.g. cannot have a program called **`public`**.

Class: definition

- After **`public`** we write
  - **reserved word `class`**,
  - then name,
  - then left brace ({),
  - body of text
  - and finally closing right brace (}).

```
public class MyFabulousProgram
{
  ... Lots of stuff here.
}
```

Hello world

- The heading for our `HelloWorld` class.

  ```
  public class HelloWorld
  ```

  Then the opening bracket.

  ```
  {
  ```

Method: main method: programs contain a main method

- All Java programs contain a section of code called `main`.
- This is where the computer will start to **execute** the program.
- Sections of code are called **method**s
  - contain instructions how to do something.
- The **main method** always starts with following heading.

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

Method: main method: is public

- The **main method** starts with **reserved word** **`public`**
  - so **virtual machine** has access to it.
- **`public`**

Method: main method: is static

- The **main method** has **reserved word `static`**.
- Thus is allowed to be used in the **static context**.
  - A context is an allocation of computer memory for the program and data, etc..
- The **virtual machine** creates the static context when program is loaded.
  - A **dynamic context** is a kind of allocation of memory made during **run** of the program.
- Main method must be able to run in the static context
  - else program could not be started!
- `public static`

Method: main method: is void

- A **method** might calculate and **return** some result
  - if so we state this in its heading.
  - E.g. method might calculate square root of a number, and return the answer as a number.
- If it does not we write **reserved word `void`**.
  - Void means `without contents'.
- The **main method** does not return a value.
- `public static void`

Method: main method: is the program starting point

- The program starting part – **main method** – is always called `main`
  - it is main part of program.
- **`public static void`** `main`

Command line arguments: program arguments are passed to main

- Programs can be given **command line arguments**.
  - So can Java programs.
- Program arguments are **list** of text strings.
- In Java, `String[]` means 'list of strings'.
- Must give a name for this list, usually `args`
  - so we can refer to given data from within program if needed.
- `public static void main(String[] args)`

Method: main method: always has the same heading

- Java program **main method**s always have this heading:

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

  - Even if we do not intend to use **command line argument**s.

- Typical single **class** program looks like:

```
public class MyFabulousProgram
{
  public static void main(String[] args)
  {
    ... Stuff here to perform the task.
  }
}
```

Hello world

- Back to HelloWorld....

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

Type: `String`: literal

- A **string literal** is a fixed piece of text to be used as **data**.
- We enclose text in double quotes:

```
"This is a fixed piece of text data -- a string literal"
```

- Might be used as a message to the user.

Standard API: `System`: `out.println()`

- Simplest way to print a message on **standard output**:

  ```
  System.out.println("This text will appear on standard output");
  ```

- `System` is a **class** in Java's **application programming interface** (**API**).
- Inside `System` there is a thing called `out`. This has a **method** called `println`.
- Overall is called `System.out.println`.
- It takes a string in its brackets
  - displays it on the standard output.

Hello world

- Back to `HelloWorld`. . . .

```
005:     System.out.println("Hello world!");
```

- Observe semi-colon. . . .

Statement

- A command that makes computer perform a task is a **statement**.
- E.g. `System.out.println("I will output whatever I am told to")`

Statement: simple statements are ended with a semi-colon

- All simple Java **statement**s must end with semi-colon.
  - a rule of the Java language **syntax**.

Coffee time: Can you think of a reason why Java insists on the programmer putting a semi-colon at the end of statements?

Hello world

- Back to HelloWorld....

```
  }
}
```

The full HelloWorld code

```java
public class HelloWorld
{
  public static void main(String[] args)
  {
    System.out.println("Hello world!");
  }
}
```

Trying it

- We create **source code** and **compile** it.

Console Input / Output

```
$ ls -l HelloWorld.java
-rw-------   1 jtl jtl 117 Jul 01 19:12 HelloWorld.java
$ javac HelloWorld.java
$ ls -l HelloWorld.*
-rw-------   1 jtl jtl 426 Jul 01 19:12 HelloWorld.class
-rw-------   1 jtl jtl 117 Jul 01 19:12 HelloWorld.java
$ _
```

Run

- We run program to get message on **standard output**.

Console Input / Output

```
$ java HelloWorld
Hello world!
$ _
```

Run

Coursework: `HelloWorld` in French

(Summary only)

Write a program to greet the whole world, in French!

Section 3

Example:
Hello world with a syntactic error

Aim

AIM: To introduce the principle of program errors, in particular **syntactic error**s. We also see that a **string literal** must be ended on the same line its starts on.

Error

- To err is Human. . . .
  - when you write **source code** you will get some things wrong.
- Lots of rules of Java to obey for a valid program.
  - Being new to it you will break these rules.
  - Even seasoned Java programmers make errors.

Error: syntactic error

- When we break **syntax** rules of Java we have a **syntactic error**.
  - E.g. omitting closing bracket, semi-colon. . . .
- Similar to grammatical error in natural language.
  - E.g. sign strapped to back of a poodle. . .

Error: syntactic error

My other dog an Alsatian.

Error: syntactic error

- The **compiler** gives error messages for syntactic errors.
  - Watch out: compiler can get confused. . . .

Hello world with a syntactic error

```java
public class HelloWorld
{
  public static void main(String[] args)
  {
    System.out.println("Hello world!);
  }
}
```

Coffee time: Can you spot the **syntactic error**?

Trying it

Console Input / Output

```
$ javac HelloWorld.java
HelloWorld.java:5: unclosed string literal
    System.out.println("Hello world!);
                       ^
HelloWorld.java:5: ';' expected
    System.out.println("Hello world!);
                                      ^
HelloWorld.java:7: reached end of file while parsing
}
 ^
3 errors
$ _
```

Run

- Error messages from **compiler** can look very scary.
- Read carefully – observe the parts. . . .

Type: `String`: literal: must be ended on the same line

- In Java **string literal**s must end on same line they start on.

Trying it

Coffee time: What has caused the other error message(s)?

Coursework: Fortune syntactic errors

(Summary only)

Take a given program that has **syntactic errors** in it, and get it working.

Section 4

Example:
Hello world with a semantic error

Aim

AIM: To introduce **semantic errors** and note that these and **syntactic errors** are **compile time errors**.

Error: semantic error

- A **semantic error**
  - we obey **syntax** rules
  - but write something with no meaning (semantics).
- E.g. another sign, another poodle. . .

Error: semantic error

My other dog is a Porsche.

Error: compile time error

- Java **syntactic errors** and **semantic errors**
  - are detected by **compiler**.
  - Collectively called **compile time errors**.

Hello world with a semantic error

```
public class HelloWorld
{
  public static void main(Text[] args)
  {
    System.out.println("Hello world!");
  }
}
```

Coffee time: Can you spot the **semantic error**?

Trying it

Console Input / Output

```
$ javac HelloWorld.java
HelloWorld.java:3: cannot find symbol
symbol  : class Text
location: class HelloWorld
  public static void main(Text[] args)
                          ^

1 error
$ _
```

Run

- A little cryptic?
  - Read carefully.
  - You'll get used to it.

Coursework: `ManchesterWeather` semantic errors

(Summary only)

Take a given program that has **semantic errors** in it, and get it working.

Section 5

Example: Hello solar system

Aim

AIM: To introduce the principle of **sequential execution**.

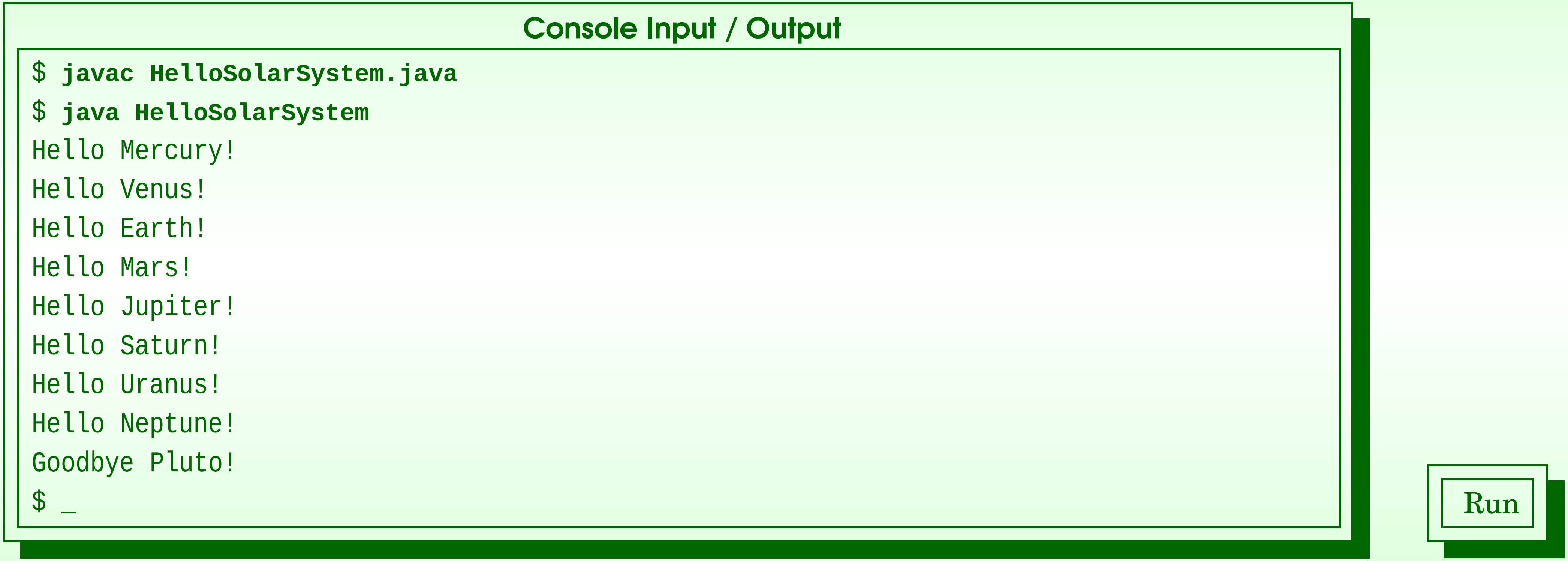
Execution: sequential execution

- Programs have many **statement**s in a list.
- Usually placed on separate lines
  - enhance human readability.
  - Java doesn't care about layout – *we should*.
- Statements in a list are **execute**d one after the other.
  - Actually **compiler** turns each into **byte code**s.
  - The **virtual machine** executes each collection of byte codes in turn.
- Known as **sequential execution**.

Hello solar system

```java
public class HelloSolarSystem
{
  public static void main(String[] args)
  {
    System.out.println("Hello Mercury!");
    System.out.println("Hello Venus!");
    System.out.println("Hello Earth!");
    System.out.println("Hello Mars!");
    System.out.println("Hello Jupiter!");
    System.out.println("Hello Saturn!");
    System.out.println("Hello Uranus!");
    System.out.println("Hello Neptune!");
    System.out.println("Goodbye Pluto!");
  }
}
```

Trying it



```
$ javac HelloSolarSystem.java
$ java HelloSolarSystem
Hello Mercury!
Hello Venus!
Hello Earth!
Hello Mars!
Hello Jupiter!
Hello Saturn!
Hello Uranus!
Hello Neptune!
Goodbye Pluto!
$ _
```

Coursework: `HelloFamily`

(Summary only)

Write a program to greet some of your family.

Section 6

Example:
Hello solar system with a run time error

Aim

AIM: To introduce the principle of **run time error**s.

Error: run time error

- Errors detected when the program is **run** are **run time error**s.
- Java calls them **exception**s.
- Messages can look very cryptic?
  - Read carefully, get used to them.
- E.g.

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

- Best clue: look either side of the colon (:).

Hello solar system with a run time error

```
public class HelloSolarSystem
{
  public static void Main(String[] args)
  {
    System.out.println("Hello Mercury!");
    System.out.println("Hello Venus!");
    System.out.println("Hello Earth!");
    System.out.println("Hello Mars!");
    System.out.println("Hello Jupiter!");
    System.out.println("Hello Saturn!");
    System.out.println("Hello Uranus!");
    System.out.println("Hello Neptune!");
    System.out.println("Goodbye Pluto!");
  }
}
```

Coffee time: What will cause a **run time error**?

Trying it

- It compiles okay.

Console Input / Output

```
$ javac HelloSolarSystem.java
$ _
```

Run

- But when we **run** it. . . .

Console Input / Output

```
$ java HelloSolarSystem
Exception in thread "main" java.lang.NoSuchMethodError: main
$ _
```

Run

- The **virtual machine** says our program has no **main method**.
  - Called it `Main` instead of `main`!

Trying it

- Another example **run time error**.

Console Input / Output

```
$ java HelloMum
Exception in thread "main" java.lang.NoClassDefFoundError: HelloMum
$ _
```

Run

Trying it

Coffee time: Imagine a version of `HelloSolarSystem`, called `HelloSolarSystemNoArgs`, with a lower case `m` on `main`, but `String[] args` has been omitted.

Explain the following.

Console Input / Output

```
$ javac HelloSolarSystemNoArgs.java
$ java HelloSolarSystemNoArgs
Exception in thread "main" java.lang.NoSuchMethodError: main
$ _
```

Run

Coursework: `Quote` run time errors

(Summary only)

Take a given program that has **run time errors** in it, and get it working.

Section 7

Example:
Hello anyone

Aim

AIM: To introduce the principle of making Java programs perform a variation of their task based on **command line arguments**, which can be accessed via an **index**. We also meet string **concatenation**.

Command line arguments: program arguments are accessed by index

- The **command line argument**s given to **main method** – a **list** of strings
  - from the **command line**.
- Each has **integer index**, starting from zero.
- To access one, use its index in square brackets.
  - E.g. `args[0]` is first command line argument.

Type: `String`: concatenation

- The + **operator** gives **concatenation** of two strings.
  - E.g. `"Hello " + "world"` has same value as `"Hello world"`.
    - (Note where space came from.)
- Most useful with one or more **variable** values.
  - E.g. `"Hello " + args[0]`
- E.g. `System.out.println("Hello " + args[0])`

Coffee time: When might we concatenate two **string literal**s?

Hello anyone

```
public class HelloAnyone
{
  public static void main(String[] args)
  {
    System.out.println("Hello " + args[0]);
  }
}
```

Trying it

Console Input / Output

```
$ javac HelloAnyone.java
$ java HelloAnyone John
Hello John
$ java HelloAnyone Lizzy
Hello Lizzy
$ _
```

Run

- What if no argument?

Console Input / Output

```
$ java HelloAnyone
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at HelloAnyone.main(HelloAnyone.java:5)
$ _
```

Run

- Observe source name and line number.

Trying it

- What if name contains space?

Console Input / Output

```
$ java HelloAnyone "John Latham"
Hello John Latham
$ java HelloAnyone John Latham
Hello John
$ _
```

Run

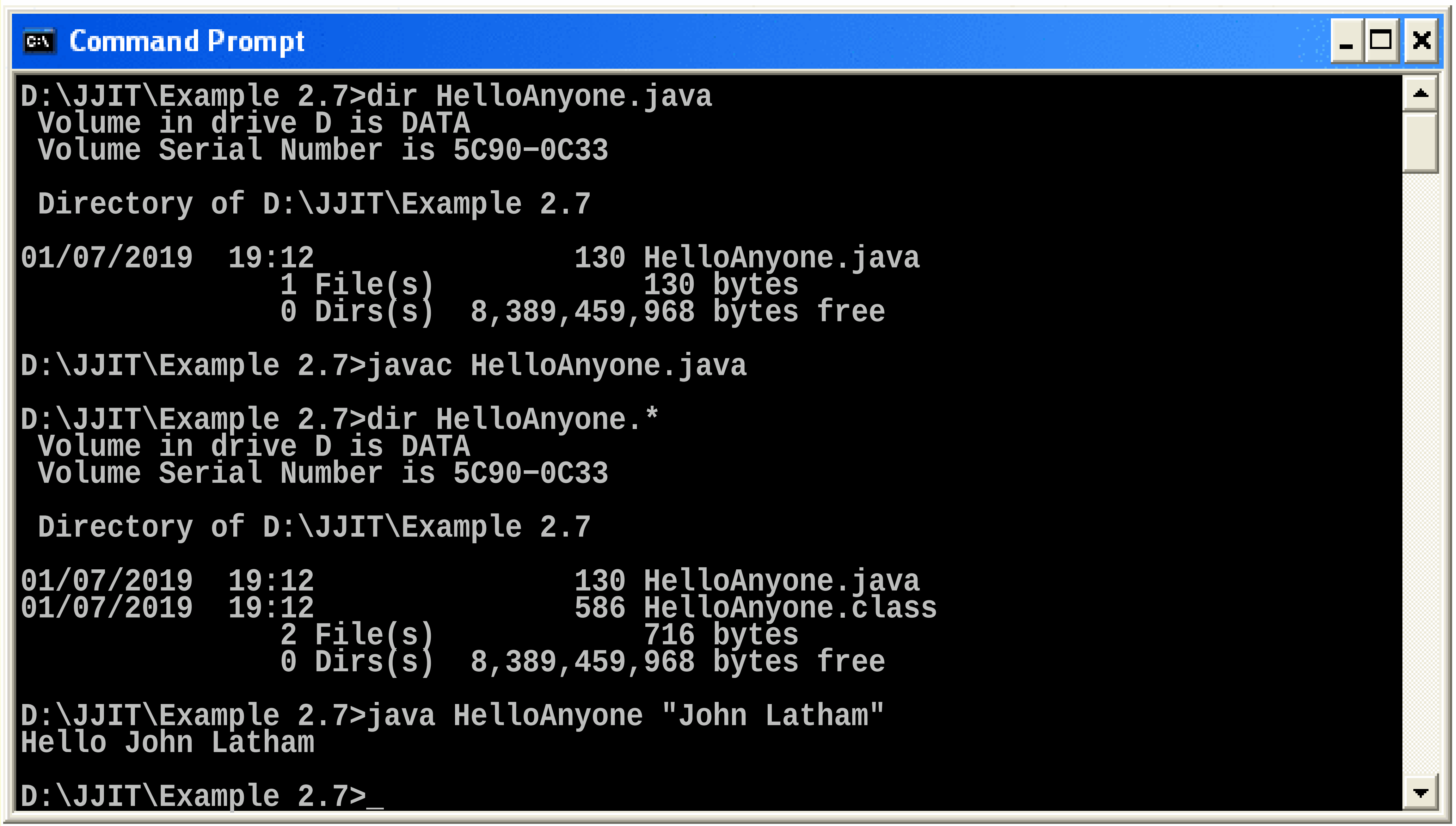
- Empty string?

Console Input / Output

```
$ java HelloAnyone ""
Hello
$ _
```

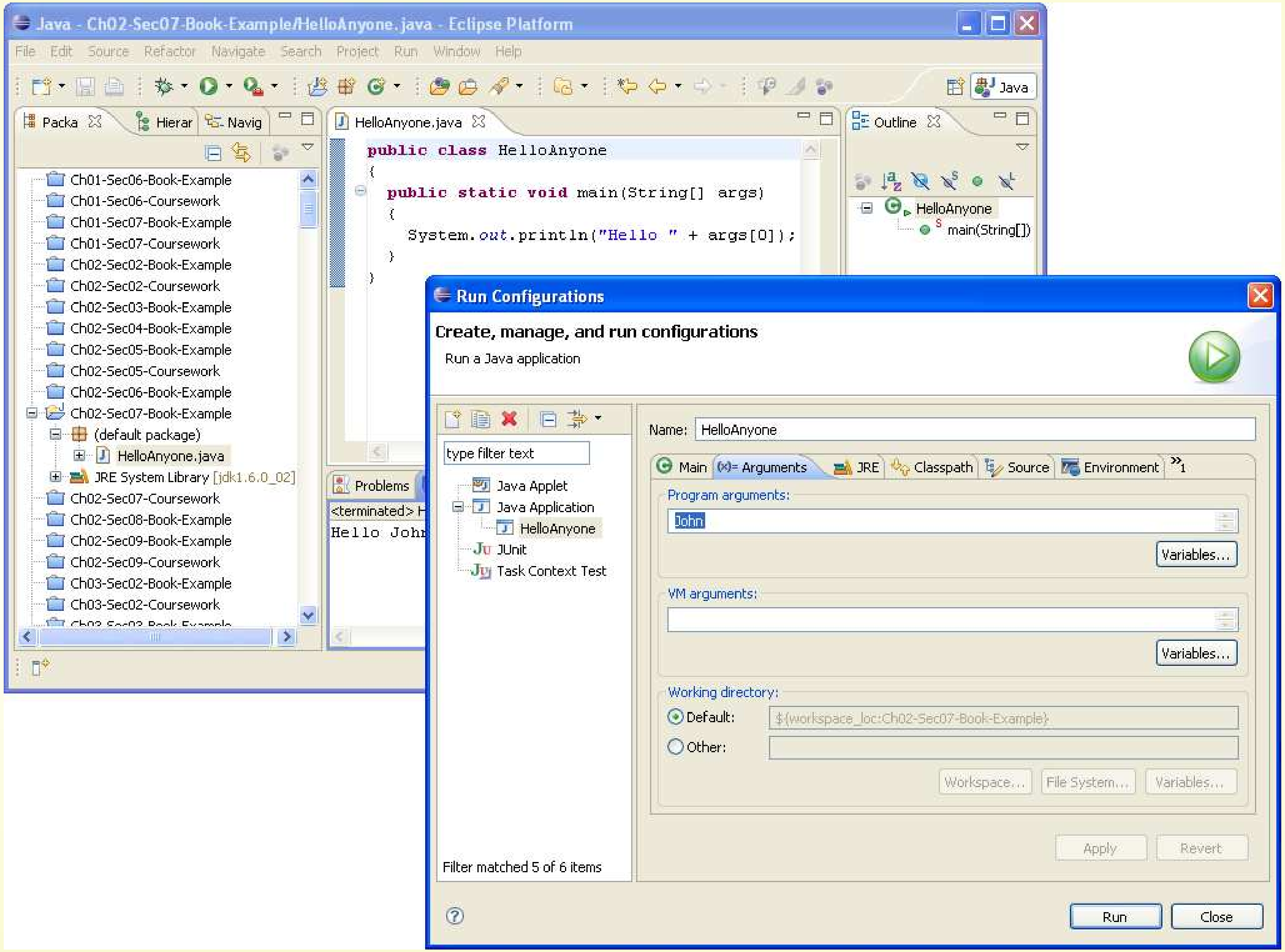
Run

Trying it



```
D:\JJIT\Example 2.7>dir HelloAnyone.java
 Volume in drive D is DATA
 Volume Serial Number is 5C90-0C33

 Directory of D:\JJIT\Example 2.7

01/07/2019  19:12               130 HelloAnyone.java
               1 File(s)            130 bytes
               0 Dirs(s)  8,389,459,968 bytes free

D:\JJIT\Example 2.7>javac HelloAnyone.java

D:\JJIT\Example 2.7>dir HelloAnyone.*
 Volume in drive D is DATA
 Volume Serial Number is 5C90-0C33

 Directory of D:\JJIT\Example 2.7

01/07/2019  19:12               130 HelloAnyone.java
01/07/2019  19:12               586 HelloAnyone.class
               2 File(s)            716 bytes
               0 Dirs(s)  8,389,459,968 bytes free

D:\JJIT\Example 2.7>java HelloAnyone "John Latham"
Hello John Latham

D:\JJIT\Example 2.7>_
```

Trying it



Java - Ch02-Sec07-Book-Example/HelloAnyone.java - Eclipse Platform

HelloAnyone.java

```
public class HelloAnyone
{
  public static void main(String[] args)
  {
    System.out.println("Hello " + args[0]);
  }
}
```

```
<terminated> H
Hello John
```

Run Configurations

Create, manage, and run configurations

Run a Java application

Name: HelloAnyone

Program arguments: John

VM arguments:

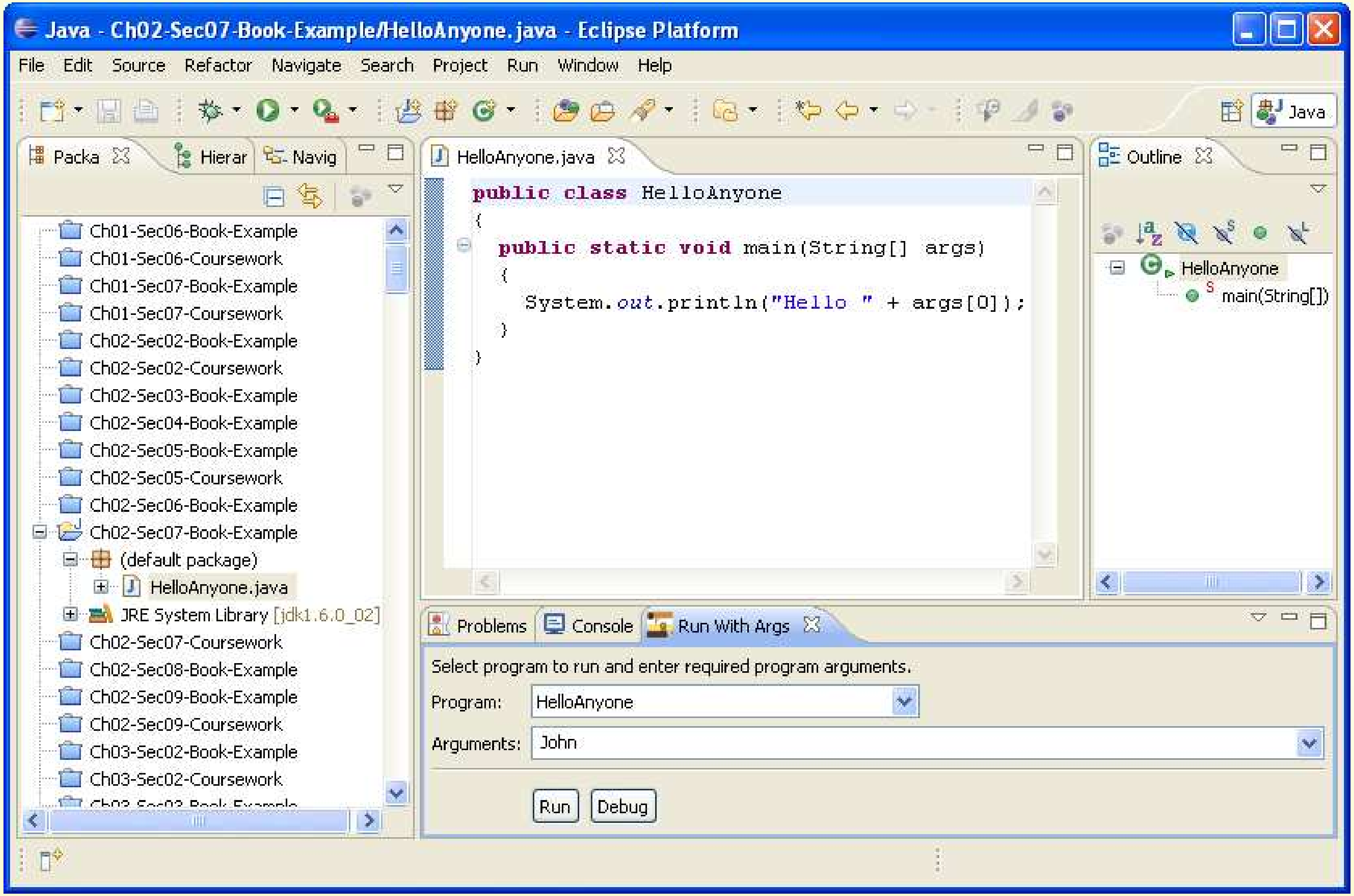
Working directory:

● Default: ${workspace_loc:Ch02-Sec07-Book-Example}

○ Other:

Filter matched 5 of 6 items

Trying it



Java - Ch02-Sec07-Book-Example/HelloAnyone.java - Eclipse Platform

HelloAnyone.java

```
public class HelloAnyone
{
  public static void main(String[] args)
  {
    System.out.println("Hello " + args[0]);
  }
}
```

Select program to run and enter required program arguments.

Program: HelloAnyone

Arguments: John

Coursework: `FlatterMe`

(Summary only)

Write a program to say how wonderful the user is.

Section 8

Example:
Hello anyone with a logical error

Aim

AIM: To introduce the principle of **logical errors**.

Error: logical error

- Most tricky kind of error – **logical error**.
- No help from **compiler**, nor **virtual machine**.
  - Code is meaningful to Java.
- But program does not do what we want!
  - Java is ‘too stupid’ to know that.
- Subtle ones slip through our testing.
  - i.e. **bug**s.

Hello anyone with a logical error

```java
public class HelloAnyone
{
  public static void main(String[] args)
  {
    System.out.println("Hello + args[0]");
  }
}
```

Coffee time: Can you spot the **logical error**?

Trying it

- Compiles and runs without error.

Console Input / Output

```
$ javac HelloAnyone.java
$ java HelloAnyone John
Hello + args[0]
$ _
```

Run

Coursework: `Birthday` logical errors

(Summary only)

Take a given program that has **logical errors** in it, and get it working.

Section 9

Hello solar system, looking at the layout

Aim

AIM: To begin to explore the decisions behind the way we lay out the **source code** for a program.

Code clarity: layout

- Java doesn’t care about layout – **white space** must separate symbols that would be one symbol otherwise.
  - E.g. `public void` would be `publicvoid`.
- Could put program on one line, minimum space.

```
public class HelloSolarSystem{public static void main(String[]args){System.out.println("Hello Mercu
```

- Or split just to fit on page.

```
public class HelloSolarSystem{public static void main(String[]args){
System.out.println("Hello Mercury!");System.out.println(
"Hello Venus!");System.out.println("Hello Earth!");System.out.println
("Hello Mars!");System.out.println("Hello Jupiter!");System.out.
println("Hello Saturn!");System.out.println("Hello Uranus!");System.
out.println("Hello Neptune!");System.out.println("Goodbye Pluto!");}}
```

THE UNIVERSITY
of MANCHESTER

Code clarity: layout

- Layout important for human reader.
  - Take pride in making your work most readable.
- Split lines in good places.
- Use **indentation** to show structure.

Hello solar system, looking at the layout

```
public class HelloSolarSystem
{
  public static void main(String[] args)
  {
    System.out.println("Hello Mercury!");
    System.out.println("Hello Venus!");
    System.out.println("Hello Earth!");
    System.out.println("Hello Mars!");
    System.out.println("Hello Jupiter!");
    System.out.println("Hello Saturn!");
    System.out.println("Hello Uranus!");
    System.out.println("Hello Neptune!");
    System.out.println("Goodbye Pluto!");
  }
}
```

New line after **class** heading.

New line plus indentation – 2 or 3 spaces – for **main method**.

New line, same indentation.

More indentation,
each statement on own line.

Line up with opening braces.

Code clarity: layout: indentation

- A **class** contains **nested** structures:
  - class has heading and body
    - body has **main method**
    - main method has heading and body
    - body has statements.
- Use **indentation** to show structure
  - the more nested, the more space.
- Be consistent: always same number of spaces per nesting
  - two or three is good.
  - don't use tabs!
- Opening and closing braces have same indentation.

Code clarity: layout: indentation

- Some people prefer this style – subjectively less clear?

```
public class HelloWorld {

  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

Coursework: `Limerick` layout

(Summary only)

Take a given program and lay it out properly.

Concepts covered in this chapter

- Each book chapter ends with a list of concepts covered in it.
- Each concept has with it
  - a self-test question,
  - and a page reference to where it was covered.
- Please use these to check your understanding before we start the next chapter.