Inf1B
Getting Started

Perdita Stevens
adapting earlier versions by Ewan Klein, Volker Seeker, et al.

School of Informatics
Where have you left off last semester?

Haskell
- functional

Java
- imperative
- object oriented
Pancake Recipe

- Take a bowl
- Add flour
- Add eggs
- Add milk
- While not yet smooth
  - Whisk the batter
- Fry in a pan

- statements are used which are processed step by step
- programs carry state which in OO is expressed in objects
What is object orientation?

It means: your program is structured like the domain (real world). Objects (organised into classes of similar objects) typically represent things (organised into types of similar things). Objects have

- state: they can store data
- behaviour: they can do things, in response to messages
- identity: two objects with the same state can still be different objects.

Any of state, behaviour, identity can be trivial for a particular object, though.

In Java, all behaviour is associated with a class. However, it can be static – that is, not associated with any particular object of the class.
A First Example

HelloWorld.java

/*************************************/
/* Prints "Hello, World!"*/
*************************************/

public class HelloWorld {
    public static void main (String[] args) {
        System.out.println("Hello, World!");
    }
}
}
Creating a New Class

1. All Java code sits inside a class.
2. By important convention, class names are capitalised and in ‘CamelCase’.
3. Each class goes into a file of its own (usually; and always in this course).
4. So, use a text editor (e.g., gedit) to create a file called HelloWorld.java.
5. The name of the file has to be the same as the name of the class, and suffixed with .java.

At the terminal

gedit HelloWorld.java
A First Example

Declare a class

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

- Basic form of a class definition.
- Class definition enclosed by curly braces.
A First Example

Declare the `main()` method

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

- We need a `main()` method to actually get our program started.
- All our other code is invoked from inside `main()`.
- `void` means the method doesn’t return a value.
- The argument of the method is an array of `Strings`; this array is called `args`.
- Definition of a method enclosed by curly braces.
public class HelloWorld {
    public static void main (String[] args) {
        System.out.println("Hello World!");
    }
}

- System.out is an object (a rather special one).
- println("Hello World!") is a message being sent to that object: println is the method name, "Hello World!" is the argument.
- The whole line is a statement: must be terminated with a semi-colon (;).
- Strings must be demarcated by double quotes.
- Strings cannot be broken across a line in the file.
The program needs to be compiled before it can be executed.

Use the `javac` command in a terminal.

At the terminal

```
javac HelloWorld.java
```

If there’s a problem, the compiler will complain.

If not, compiler creates a Java bytecode file called `HelloWorld.class`.
Running the Program

- Now that we have compiled code, we can run it.
- Use the `java` command in a terminal.

**At the terminal**

```
java HelloWorld
Hello World!
```
Running the Program

▶ Now that we have compiled code, we can run it.
▶ Use the `java` command in a terminal.

At the terminal

```
java HelloWorld
Hello World!
```

▶ Note that we omit the `.class` suffix in the run command. The `java` command wants a classname as argument, not a filename.
Edit-Compile-Run Cycle

Type in the program using an editor and save the program to a file. Use the name of the main class and the suffix .java for the file. This is called a source file.

The process of compiling a source file generates the bytecode file. The byte code will have a .class suffix; the prefix will be the same.

A java interpreter will read the bytecode file and execute the instructions in it. If an error occurs while running, the interpreter will stop its execution.
Edit-Compile-Run Cycle

Type in the program using an editor and save the program to a file. Use the name of the main class and the suffix `.java` for the file. This is called a source file.

The process of compiling a source file generates the bytecode file. The bytecode will have a `.class` suffix; the prefix will be the same.

A Java interpreter will read the bytecode file and execute the instructions in it. If an error occurs while running, the interpreter will stop its execution.
Edit-Compile-Run Cycle

**Edit**
Type in the program using an editor and save the program to a file. Use the name of the main class and the suffix `.java` for the file. This is called a source file.

**Compile**
The process of compiling a source file generates the bytecode file. The byte code will have a `.class` suffix; the prefix will be the same.

**Run**
A java interpreter will read the bytecode file and execute the instructions in it. If an error occurs while running, the interpreter will stop its execution.
Edit-Compile-Run Cycle

- **Edit**: Type in the program using an editor and save the program to a file. Use the name of the main class and the suffix `.java` for the file. This is called a source file.

- **Compile**: The process of compiling a source file generates the bytecode file. The byte code will have a `.class` suffix; the prefix will be the same.

- **Run**: A java interpreter will read the bytecode file and execute the instructions in it. If an error occurs while running, the interpreter will stop its execution.
Edit-Compile-Run Cycle

- The program needs to be compiled before it can be executed.
- If you edit a program, you need to compile it again before running the new version.
- However, if you use an integrated development environment, this may compile your code automatically.
Development Best Practices

Golden Rules of Programming

1. Compile often
2. Save regularly
Golden Rules of Programming

1. Compile often
2. Save regularly

Why? Detect errors early!
- Compiler checks syntactical correctness
- Running checks (some) semantic correctness
- Unit tests check (more) semantic correctness
Basic Functionality
public class Calc {

    public static void main(String[] args) {
        System.out.print("The sum of 6 and 2 is ");
        System.out.println(6 + 2);

        System.out.print("The quotient of 6 and 2 is ");
        System.out.println(6 / 2);
    }
}

Output

The sum of 6 and 2 is 8
The quotient of 6 and 2 is 3
public class Calc {

    public static void main(String[] args) {
        System.out.print("The sum of 6 and 2 is ");
        System.out.println(6 + 2);

        System.out.print("The quotient of 6 and 2 is ");
        System.out.println(6 / 2);
    }
}

Output

The sum of 6 and 2 is 8
The quotient of 6 and 2 is 3
public class Concat {

    public static void main(String[] args) {
        System.out.println("The name is " + "Bond, "
            + "James Bond");
    }
}

Output

The name is Bond, James Bond
public class Concat {
    public static void main(String[] args) {
        System.out.println("Is that you, 007");
    }
}

Output

Is that you, 007?
Assignment: Basic Definitions

Variable: A name that refers to a value

Assignment Statement: Associates a value with a variable

```
int a, b;
a = 1234 ;
b = 99;
```

Important: = is the operator in an imperative statement, not a logical assertion.
Assignment: Combining Declaration and Initialisation

Variables that have been declared, but not assigned to, are a potential source of error. (Exercise for the keen: understand what happens to them in Java.)

It’s often best to declare a variable and *initialise* it at the same time.

```java
int a, b;
a = 1234;
b = 99;
int c = a + b;
```

combined declaration and assignment statement
public class HelloWorld {

    public static void main ( String [] args ) {
        String msg = "Hello World!";
        System.out.println( msg );
    }
}
# Built-in Data Types

<table>
<thead>
<tr>
<th>type</th>
<th>value set</th>
<th>literal values</th>
<th>operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>characters</td>
<td>'A', '$'</td>
<td>compare</td>
</tr>
<tr>
<td>String</td>
<td>sequences of characters</td>
<td>&quot;Hello World!&quot;, &quot;Java is fun&quot;</td>
<td>concatenate</td>
</tr>
<tr>
<td>int</td>
<td>integers</td>
<td>17, 1234</td>
<td>add, subtract, multiply, divide</td>
</tr>
<tr>
<td>double</td>
<td>floating-point numbers</td>
<td>3.1415, 6.022e23</td>
<td>add, subtract, multiply, divide</td>
</tr>
<tr>
<td>boolean</td>
<td>truth values</td>
<td>true, false</td>
<td>and, or, not</td>
</tr>
</tbody>
</table>
## Integer operations

<table>
<thead>
<tr>
<th>expression</th>
<th>value</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 + 3</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5 - 3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5 * 3</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

- **5 / 2** has precedence over **2**.  
- **5 % 2** results in a remainder of 1.  
- **3 * 5 - 2** has precedence over **13 ***.  
- **3 + 5 / 2** has precedence over **5 /**.  
- **3 - 5 - 2** results in -4.  
- **(3 - 5) - 2** results in -4, which is a better style than **3 - (5 - 2)**, which results in 0.  
- **1 / 0** results in a run-time error.  
- **3 - 5 - 2** is left associative.
## Integer operations

<table>
<thead>
<tr>
<th>expression</th>
<th>value</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 + 3</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5 - 3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5 * 3</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>5 / 2</td>
<td>2</td>
<td>no fractional part</td>
</tr>
</tbody>
</table>
# Integer operations

<table>
<thead>
<tr>
<th>expression</th>
<th>value</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 + 3</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5 - 3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5 * 3</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>5 / 2</td>
<td>2</td>
<td>no fractional part</td>
</tr>
<tr>
<td>5 % 2</td>
<td>1</td>
<td>remainder</td>
</tr>
</tbody>
</table>
## Integer operations

<table>
<thead>
<tr>
<th>expression</th>
<th>value</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 + 3</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5 - 3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5 * 3</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>5 / 2</td>
<td>2</td>
<td>no fractional part</td>
</tr>
<tr>
<td>5 % 2</td>
<td>1</td>
<td>remainder</td>
</tr>
<tr>
<td>1 / 0</td>
<td></td>
<td>run-time error</td>
</tr>
</tbody>
</table>
## Integer operations

<table>
<thead>
<tr>
<th>expression</th>
<th>value</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 + 3</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5 - 3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5 * 3</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>5 / 2</td>
<td>2</td>
<td>no fractional part</td>
</tr>
<tr>
<td>5 % 2</td>
<td>1</td>
<td>remainder</td>
</tr>
<tr>
<td>1 / 0</td>
<td></td>
<td>run-time error</td>
</tr>
<tr>
<td>3 * 5 - 2</td>
<td>13</td>
<td>* has precedence</td>
</tr>
</tbody>
</table>
## Integer operations

<table>
<thead>
<tr>
<th>expression</th>
<th>value</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 + 3</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5 - 3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5 * 3</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>5 / 2</td>
<td>2</td>
<td>no fractional part</td>
</tr>
<tr>
<td>5 % 2</td>
<td>1</td>
<td>remainder</td>
</tr>
<tr>
<td>1 / 0</td>
<td></td>
<td>run-time error</td>
</tr>
<tr>
<td>3 * 5 - 2</td>
<td>13</td>
<td>* has precedence</td>
</tr>
<tr>
<td>3 + 5 / 2</td>
<td>5</td>
<td>/ has precedence</td>
</tr>
</tbody>
</table>
# Integer operations

<table>
<thead>
<tr>
<th>expression</th>
<th>value</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 + 3</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5 − 3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5 * 3</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>5 / 2</td>
<td>2</td>
<td>no fractional part</td>
</tr>
<tr>
<td>5 % 2</td>
<td>1</td>
<td>remainder</td>
</tr>
<tr>
<td>1 / 0</td>
<td></td>
<td>run-time error</td>
</tr>
<tr>
<td>3 * 5 − 2</td>
<td>13</td>
<td>* has precedence</td>
</tr>
<tr>
<td>3 + 5 / 2</td>
<td>5</td>
<td>/ has precedence</td>
</tr>
<tr>
<td>3 − 5 − 2</td>
<td>−4</td>
<td>left associative</td>
</tr>
<tr>
<td>(3 − 5) − 2</td>
<td>−4</td>
<td>better style</td>
</tr>
<tr>
<td>3 − (5 − 2)</td>
<td>0</td>
<td>unambiguous</td>
</tr>
</tbody>
</table>
Floating-Point Numbers

The default floating-point type in Java is `double`. 
## Floating-Point Operations

<table>
<thead>
<tr>
<th>expression</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.141 + .03</td>
<td>3.171</td>
</tr>
<tr>
<td>3.141 - .03</td>
<td>3.111</td>
</tr>
<tr>
<td>6.02e23 / 2.0</td>
<td>3.01e23</td>
</tr>
<tr>
<td>5.0 / 3.0</td>
<td>1.6666666666666667</td>
</tr>
<tr>
<td>10.0 % 3.141</td>
<td>0.577</td>
</tr>
<tr>
<td>1.0 / 0.0</td>
<td>Infinity</td>
</tr>
<tr>
<td>Math.sqrt(2.0)</td>
<td>1.4142135623730951</td>
</tr>
<tr>
<td>Math.sqrt(-1.0)</td>
<td>NaN</td>
</tr>
</tbody>
</table>
## Type Conversion

Sometimes we can **convert** one type to another.

- **Automatic**: OK if no loss of precision, or converts to string
- **Explicit**: use a **cast** or method like `parseInt()`

<table>
<thead>
<tr>
<th>expression</th>
<th>result type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;1234&quot; + 99</td>
<td>String</td>
<td>&quot;123499&quot;</td>
</tr>
<tr>
<td><code>Integer.parseInt(&quot;123&quot;)</code></td>
<td>int</td>
<td>123</td>
</tr>
<tr>
<td><code>(int) 2.71828</code></td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td><code>Math.round(2.71828)</code></td>
<td>long</td>
<td>3</td>
</tr>
<tr>
<td><code>(int) Math.round(2.71828)</code></td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td><code>(int) Math.round(3.14159)</code></td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td><code>11 * 0.3</code></td>
<td>double</td>
<td>3.3</td>
</tr>
<tr>
<td><code>(int) 11 * 0.3</code></td>
<td>double</td>
<td>3.3</td>
</tr>
<tr>
<td><code>11 * (int) 0.3</code></td>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td><code>(int) (11 * 0.3)</code></td>
<td>int</td>
<td>3</td>
</tr>
</tbody>
</table>
Let’s practise that

TOP HAT
Type Conversion

Moral:
If you want a floating-point result from division, make at least one of the operands a double
Command-line Arguments

**Unix commands**

`mkdir` **MyJavaCode**

`mkdir` is a command and **MyJavaCode** is an argument
Command-line Arguments

Unix commands

mkdir MyJavaCode

mkdir is a command and MyJavaCode is an argument

Using Java to carry out commands

% java Add 3 6
9

3 and 6 are command-line arguments for the program Add
public class Add {
    public static void main(String[] args) {
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);
        System.out.println(a + b);
    }
}
public class Add {
    public static void main(String[] args) {
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);
        System.out.println(a + b);
    }
}

int a = Integer.parseInt(args[0]);

- This reads in a string (e.g., "3") from the command line,
- parses it as an int, and
- assigns this as the value of variable a.
Command-line Arguments

**Missing an argument**

% java Add 3
java.lang.ArrayIndexOutOfBoundsException: 1

This a run-time error — we didn’t provide anything as a value for `args[1]`:

```java
int b = Integer.parseInt(args[1]);
```
Summary

- Java is an object-oriented, imperative programming language.
  - Statements are executed step by step.
  - Objects carry state and have behaviour.
- Java is a compiled language (Edit-Compile-Run).
- The entry point into every Java program is the `main` function.
- Variables carry values of different types (`int`, `char`, `float`, `boolean`, `String`, ...).
- A range of arithmetic operations can be used.
- **Casting** is one way to convert between types.
- Programs can receive user input at start time using **command line arguments**.
Reading

Java Tutorial
pp1-68, i.e. Chapters 1 Getting Started, 2 Object-Oriented Programming Concepts, and Chapter 3 Language Basics, up to Expressions, Statements and Blocks
– except note:
  ► We use IntelliJ, not NetBeans as our IDE.
  ► We’ll come to the Chapter 2 material later.
  ► We’ll talk about Arrays later.
I suggest skimming Ch 2 and the Arrays section, and rereading them later.

Objects First
Appendix B.1 - B.2, Appendix C.1, Appendix E.1 and E.3
This book has a different order of topics but is generally great for beginners and has some excellent summaries of basics.