



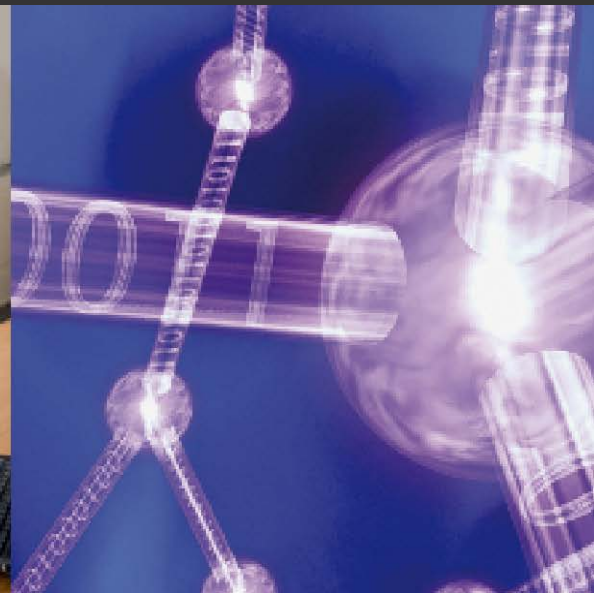
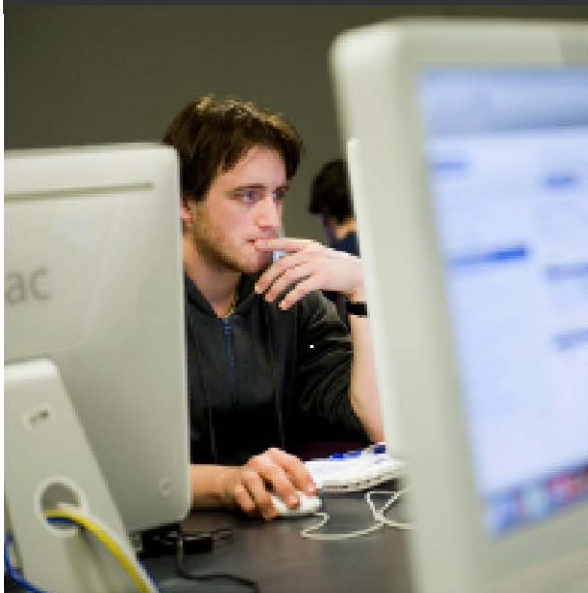
MONASH University

Information Technology

Module 1: Introduction to Java

FIT2034 Computer Programming 2

Faculty of Information Technology



Our First Java Program

Our First Java Program

- As we analyse our first Java program it's a good time to develop one of the most important skills of a programmer:
 - Learning what to focus on and what to ignore (for the time being)
 - This prevents you becoming overwhelmed with information
 - e.g. in our initial analysis of a Java program we will not be too concerned with syntax details but more with identifying the main components of the program
 - Sometimes details are called “plumbing” to imply it disappears into the background allowing more attention to be concentrated on the things they are currently trying to understand

Our First Java Program

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

■ Program Output

- It displays: `Hello World` to the *Console*
- Java programs always begin by executing the program statements in the method called “main”

■ Java is Case Sensitive

- e.g. `system` is not the same as `System`

Classes

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

- Classes are the basic unit of a Java program
- Classes have a name
 - We supply the name (black not purple font colour)
 - In this case it's HelloWorld
 - We could call it anything we like (within some syntax rules)
- All of a Class's code is contained within its open and close braces: { ... }

Methods

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

- Classes usually contain one or more methods
- Methods have a name
 - We supply the name (black not purple font colour)
 - In this case it's main
 - main has a special meaning in Java: “execute my code first”
 - Similar to the green flag in Scribble
- All of a Method's code is contained within braces: { ... }

Program Statements

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

- Methods usually contain one or more program statements
- It's these statements that are executed
 - Generally they have an effect
 - In this case they cause some text to be displayed on screen
- Note the syntax pattern. Do you think the following will work:

```
System.out.println("Hey, Java is Easy");
```

Practicalities

- A Class definition should be placed:
 - In a text file of the same name with the extension .java
 - In this case the definition of the class HelloWorld should be placed in the text file HelloWorld.java
- We will always follow this convention in this unit
 - One Class per file
 - Class name and text file name identical
 - Do not forget case sensitivity

Java Comments

Comments are not even compiled to Bytecode so a CPU never sees them

■ A Comment in Java

- Is a message from one human being to another
 - They document code e.g.
 - Describe the purpose of a Class or Method
 - Explain the purpose of a fragment of code that is not immediately obvious

■ Java has three types of comments:

- In-line `// comment text`
 - Start with `//` and ends at end of text line (so always one line)
- Block `/* comment text */`
 - Starts with `/*`, end with `*/` (so can extend over multiple lines)
- JavaDoc (special purpose) `/** comment text */`
 - Start with `/**`, end with `*/` (so can extend over multiple lines)

Java Comments

■ Examples

```
public class HelloWorldCommented {  
  
    /* This is a common block style which uses asterisks at the start  
     * of each interior line although these are not required.  
     * This style is common to document the purpose of a method and  
     * appears directly before the method  
     */  
    public static void main(String[] args) {  
        //This is an in-line comment  
        System.out.println("Welcome to FIT2034");  
  
        /* This is a block comment.  
         It can extend over several text lines.  
         */  
  
        /**  
         * This is a Javadoc comment  
         */  
    }  
}
```

Errors

- Programming in a textual language is error-prone
 - Scribble's blocks reduced the chance of making errors but did not entirely eliminate them
- **Syntax Errors** can arise:
 - when you mistype something (e.g. system with no capital)
 - you are missing something that was expected
 - you provide something that was not expected
- Compiler will report syntax errors to you for you to fix.
 - and it will refuse to generate the .class file

Errors

- **Run-time Errors** arise when the program is executing
 - They are sometimes caused by sloppy coding
 - Example: trying to divide by zero
 - They are sometimes the result of special circumstances at run time.
 - A data file is deleted while your program was reading from it.
 - Will cause the program to cease executing and display an exception trace
 - Indicates the line of code where the problem occurred.
- The compiler may warn you of the possibility, but usually does not

Errors

- Logic Errors

- Are the result of incorrect coding
- E.g. going beyond the end of a list in a loop
- E.g. miscalculations
- Incorrect algorithm

- Compiler will not notify you of the possibility

- It assumes you meant what you wrote

Part 2

Values and Data Types

Objectives – Part 2

- Understand primitive data types
 - And know when to use which type
- Be able to declare and use variables, constants and literals of primitive types
- Understand and construct expressions
- Understand operator precedence
- Understand data type conversion

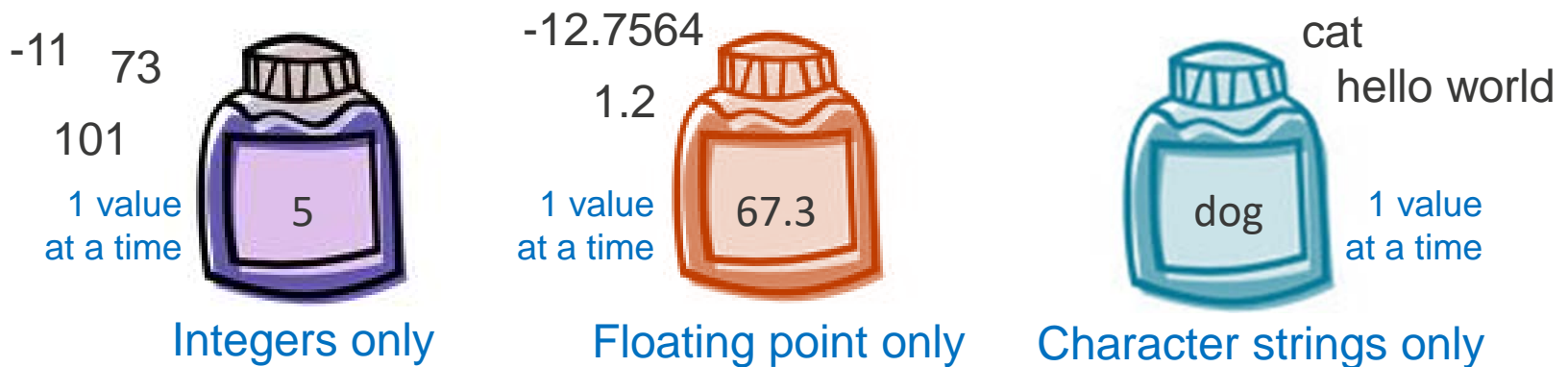
Data - Values

- Can be numerical or alphabetical
 - e.g. 23, 5834345.327, hello
 - Hello is a string of 5 characters regarded as a single unit of data
 - There are other types of data
 - Discussed later
- When a data value appears in program code it is called a **literal**
- Data values can represent
 - quantities, amounts, measurements, ...

Data - Variables

■ Variables

- Are named containers for values
 - i.e. places to store values
- Can only contain 1 value at any one time
- Can only contain values of a specified type
 - In this sense Variables are said to have a data type, i.e. the data type of the values they are permitted to contain



Java Variables

Compiler translates variable names (human-friendly) to variable memory locations (CPU-friendly)

- A Variable is actually a named address in the computer's memory
 - In Java variables must be declared before they can be used
- Declaring a variable
 - Names the variable and associates that name with the variable's address in memory
 - Specifies what type of data values the variable can hold
 - Optionally sets the initial value of the variable

Variable data type

A variable called total that can hold integer values is declared

```
int total;  
int count, temp, result;
```

3 integer variables are declared

Variable name

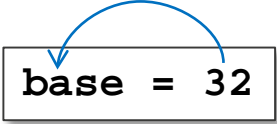
```
int sum = 0;  
int base = 32, max = 149;
```

It's possible to set an initial value for variables as they are declared

Assignment

■ The Assignment Operator (=)

- Is used to store a specified value in a variable
- Any value currently stored in the variable is replaced (overwritten) and is lost

■ e.g. 

- First the expression to the right of the operator is evaluated
 - Easy in this case – it's just the literal integer value 32
- Second this value is stored in the variable named on the left hand side of the operator (i.e. it's stored at the memory location associated with the variable name)
 - The variable's data type should be the same as the value being assigned to it

Constants

- A Constant is similar to a variable except:
 - It must have an initial values assigned to it at declaration AND
 - That value cannot be overwritten
 - Any attempt to assign a new value to a constant will be picked up by the Java Compiler and Bytecode will not be created for the program it occurs in
- Constant Declaration
 - e.g. `final int MIN_HEIGHT = 100;`
 - “final” is a modifier, in this case it modifies a variable declaration into a constant declaration
 - Note the style of the Constant’s name: all upper case with words separated by underscores

Constants – Why?

- No Magic Numbers

- e.g. which is more clear in code?
39 or `NUMBER_OF_DEPARTMENTS`

- Constants facilitate program maintenance

- e.g. due to a company reorganisation the number of departments changes
- If the literal value, 39, has been used in many code locations it must now be carefully updated consistently in all those locations
- If on the other hand `NUMBER_OF_DEPARTMENTS` has been declared just once and used in each location where 39 is required there is just one update point: the constant's declaration

Syntax Rules for Java Identifiers

- Identifiers

- Names of variables and constants are chosen by programmers as are method and class names
- They are all called identifiers

- Variable and Constant names must:

- Consist of letters, digits, underscores (_) and dollar signs (\$)
- Must begin with a letter, underscore (_) or dollar sign (\$)
- Note: No Spaces, and can't begin with a digit

- Java is case sensitive

- Upper and lower cases are considered different
- e.g. A variable called “Number” is different to a variable called “number” or “NUMBER”

Syntax Rules for Java Identifiers

- **Keywords (also known as Reserved Words)**
 - Every language has a number of keywords that have a special meaning for the language's compilers
 - e.g. `class`, `final`, `int`
 - In code examples in the early lecture slides they generally appear in a purple font
- **Keywords cannot be used as identifiers**
 - The compiler would understandably become very confused
 - e.g. after encountering `int` a compiler expects to find an identifier with possible value assignment, possibly followed by more identifiers and optional assignments finally terminated by a semi-colon
 - Keywords can be embedded in identifiers
 - e.g. `int` could not be used as an identifier but `myInt` could

Style of Java Identifiers

■ Syntax Rules vs. Style Rules (Conventions)

- Compilers only care about Syntax rules
- Styles (conventions) are optional rules imposed upon correct syntax to facilitate the creation and maintenance of programs by programmers through consistency and readability

■ In general

- Variables and constants identifiers (names) should be meaningful
- Some sort of consistent format should be used
- Abbreviations should not be used as they can be cryptic and confusing since many abbreviations may exist for the same word
- Within each programming environment (or company) there will be a set of style rules that must be followed
 - These vary from environment to environment

Style of Java Identifiers

- The format convention we will follow for variable identifiers is:
 - Variable names should begin with a lowercase letter subsequently a capital letter should begin each new word
 - This is called Camel Case (for obvious reasons) and is a very common convention
 - It's used in the prescribed text
 - e.g.
 - firstName
 - accountBalance
 - num1 (is this a good name?)
 - Average
 - All capital letters to represent constant
 - E.g. MAX_HEIGHT

Other Coding Conventions

- There are also a number of other coding conventions we will expect you to use which include:
 - Code layout
 - Commenting and Documentation conventions
 - Placement of braces
 - Indentation of code lines
 - Variable initialisation
 - Naming of other program elements

Java Primitive Data Types

- Java has 8 Primitive Data Types

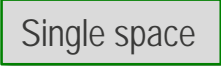
Java Data Type	Values	Examples
byte	integers	19 -174 2121654765
short		
int		
long		
float	floating point numbers	17.95 -102.3333335
double		
char	single Characters (as we will see shortly this type is really another integer type)	A z # ?
boolean	Boolean	true false

Numeric Primitive Data Types

- Integer Types (values stored accurately)
 - Have a trade off between storage size and range
- Floating Point Types (values stored approximately)
 - Have a trade off between storage size and range + significant digits (precision)

Type	Storage Size (bits)	Min Value	Max Value
byte	8	-128	127
short	16	-32,768	32,767
int	32	-2,147,483,648	2,147,483,647
long	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
float	32	~ -3.4E+38 (7 significant digits)	~ 3.4E+38 (7 significant digits)
double	64	~ -1.7E+308 (15 significant digits)	~ 1.7E+308 (15 significant digits)

char Primitive Data Types

- Values of the char primitive data type Are single characters
 - Including lower and upper case letters, digits and punctuation characters and the space character
- Literal Character Values Are enclosed in single quotes
 - e.g. 'a', '7', '%', ' ' 
 - So they are not mistaken for variable or constant names
- Do not confuse the char type with the String type
 - The String type has not been explained yet
 - It's not a primitive type
 - Values of String type are strings of zero or more characters
 - String literals are enclosed in double quotes
 - Therefore 'a' and "a" are entirely different things
 - "hello" is a String literal
 - "" (an empty string) is valid but ' ' (2 single quotes, no space) is not

Unicode Character Set

- A character set is an ordered list of characters, with each character corresponding to a unique integer
- Valid values of the char data type are:
 - Any characters from the Unicode character set
 - Stored as their corresponding Unicode character code integer in the range 0 – 65,536
- The Unicode character set
 - Can accommodate 65,536 characters which allows for characters from most world languages
 - This requires 16 bits to represent each character

ASCII Character Set

- The original standard character set
 - 128 characters which requires 7 bits to store
 - From the table displayed to the right it can be seen with respect to integer codes:
 - Digits < upper case letters < lower case letters
 - Digits and letters occur in blocks in the expected order
 - Punctuation and special characters are distributed between these blocks
 - There are also some non-printable control characters such as carriage return (13) and tab (9) (not shown in the table)

002	003	004	005	006	007
SP	0	@	P	`	p
!	1	A	Q	a	q
"	2	B	R	b	r
#	3	C	S	c	s
\$	4	D	T	d	t
%	5	E	U	e	u
&	6	F	V	f	v
'	7	G	W	g	w
(8	H	X	h	x
)	9	I	Y	i	y
*	:	J	Z	j	z
+	;	K	[k	{
,	<	L	\	l	
-	=	M]	m	}
.	>	N	^	n	~
/	?	O	_	o	DEL

boolean Primitive Data Type

■ Values

- The Java reserved words (keywords) true and false are the only valid boolean values

■ Example declaration and initialisation

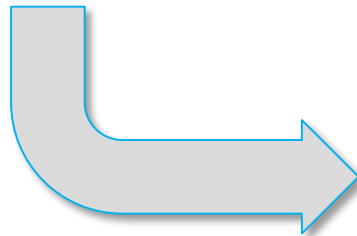
- e.g. `boolean done = false;`

■ Use boolean variables whenever a variable can only have 2 values

- e.g.
 - done/not done
 - on/off
 - high/low

Expressions

- An Expression is any syntactically valid combination of one or more operators and operands
- Arithmetic Expressions
 - Use numeric values and variables as operands and numeric operators
 - Evaluate to a numeric value
- Java Arithmetic Operators



Addition	+
Subtraction	-
Multiplication	*
Division	/
Remainder after Division	%

Arithmetic Operations

■ Multiplication and Division Operators

- If at least one of the operands of these operations is floating point then the result is floating point

■ Division Operator

- If both operands are integer then the operation performed is Integer Division
 - i.e. any fractional part of the division is discarded
 - e.g. $14 / 3$ results in 4 (remainder 2 discarded)
 - e.g. $8 / 12$ results in 0 (remainder 8 discarded)

■ Remainder Operator

- The discarded remainder from integer division can be retrieved using the remainder operator
 - e.g. $14 \% 3$ results in 2
 - e.g. $8 \% 12$ results in 8

Operator Precedence

- When an expression includes more than one operator (possibly sharing operands)
 - The question arises as to the order in which operators perform their operation
 - i.e. which operator takes precedence at any point during expression evaluation
 - e.g. does $3 + 7 * 2$ result in:
 - 17 (* operated first) or
 - 20 (+ operates first)
- Java has a strict order of operator precedence
 - For Java arithmetic expressions it's the same as the one used to evaluate mathematical expressions

Operator Precedence

- Multiplication, Division, Remainder
 - Have equal precedence
- Addition and Subtraction have equal precedence
 - It's lower than Multiplication, Division and Remainder
- If two operators are of equal precedence the leftmost gets highest precedence
- Parentheses (round brackets)
 - Override precedence rules demanding what they enclose is evaluated first
 - Parentheses can be nested
 - Innermost has the highest precedence

Operator Precedence	
	()
	*, /, %
↓	+, -

Increment and Decrement Operators

- The increment operator (`++`)
 - adds one to its operand
 - e.g. `count++` results in 7 if count has the value 6
- The decrement operator (`--`)
 - subtracts one from its operand
 - e.g. `count--` results in 5 if count has the value 6
- These are examples of *unary operators*
 - They have just one operand
- Beware: `++` and `--` are complex
 - e.g. `count++` and `++count` mean quite different things when embedded in expressions so only use as a statement

Data Type Conversion

- Conversions between primitive value types is permitted in some circumstances
- Widening Conversion:
 - Usually no information is lost
 - Typically from a type with a smaller value range to a type with a larger value range
 - e.g. int (23) → double (23.0)
 - Although there is never a range problem in widening conversions, there may be a loss of accuracy (remember integers are stored accurately, floating points only approximately)

Data Type Conversion

■ Narrowing Conversion

- The possibility of information loss exists
- Typically converting from a type with a larger value range (or higher precision) to a type with a smaller value range (or lower precision)
- e.g. `double (1.2) → int (?)`
- e.g. `int (500) → byte (?)`
- e.g. `double (1.23456789) → float(?)`

Data Type Conversion

- In Java, data conversions occur in three contexts
 - Assignment conversion
 - A value of one type is assigned to a variable of another type
 - Promotion
 - An operator has operands of mixed types
 - Operators can only operate on operands of the same type (this is a CPU limitation)
 - Casting
 - This is explicit conversion by a programmer

Assignment Conversion

- Java compilers allows this if the conversion is widening but report an error if the conversion is narrowing and no Bytecode is produced
 - e.g. the following assignment involves a conversion
 - A COPY of the value stored in **dollars** is automatically converted to the float type and assigned to the variable **money**
 - The value stored in the variable **dollars** is not altered in any way

```
float money;  
  
int dollars = 3;  
  
//widening conversion OK  
money = dollars;
```

Promotion

- Arithmetic operators automatically perform these data type conversions
 - If their operands are mixed
- e.g. evaluation of the RHS of the following assignment involves a conversion
 - A COPY of the value of `count` is automatically converted to double before the division proceeds
 - The value stored in `count` is not altered in any way
 - In this case this promotion prevents integer division occurring

```
double sum, result;  
int count;  
  
:  
result = sum / count;
```

Promotion rules are specified exactly in the *Java Language Specification*

Casting

- Programmers can explicitly Cast (type convert) a value
- e.g. The explicit cast `((int))` causes a conversion in the following code
 - This forcing of a narrowing conversion comes at a cost, namely truncation (not rounding) of the floating point value
 - A COPY of the value stored in the variable `money` is converted to an integer by truncating its fractional part ($5.67 \rightarrow 5$) this value is then stored in the integer variable `dollars`
 - The value stored in `money` is not altered in any way

```
double money = 5.67;
int dollars = 3;

dollars = money; //type mismatch compiler error reported
dollars = (int) money; //cast forces conversion with truncation
```

- e.g. here a cast is used to avoid an unwanted integer division

```
double average;
int total, count;
:
average = (double) total / count;
```