Programming Design and Implementation

Lecture 2: Programming Basics

Updated: 21st February, 2020

Mark Upston
Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Outline

Program Documentation

Variables

Primitive Types

Reference Types

Arrays

“Zero” Marks

- A student who does any of the following in a submitted, assessable answer will receive heavy penalties, up to and including **zero** marks for that question:
  - Uses **`continue`**
  - Uses **`break`** in any other place than a **`switch`** statement
  - Uses **`goto`**
  - Has more than one **`return`** statement in a method
  - Has a **`return`** statement in a method anywhere but the last statement of the method
  - Uses **`System.exit()`** anywhere but the last statement of the **`main()`** method
  - Uses global variables for anything other than class fields
  - Uses a ternary operator
- Note: similar efforts in pseudo code will also receive zero marks

Program Documentation

- User Oriented:
  - Software requirements
  - Installation instructions
  - How to use it
  - Troubleshooting
- Programmer Oriented:
  - Detailed design
  - Philosophy of design
    - Overall justification
  - Class by class description of external interface to object
    - Public information
  - Class by class description of internal information
    - If programmer is modifying your class library

Online / Offline

- Variety of ways in which documentation can be stored electronically (html, pdf)
  - Very important to realise that electronic documentation should **always** be in a portable format (i.e., **not** MS Word)
- Storing help online is very useful but there should always be printed manuals for:
  - A deeper look
  - Emergencies
- User documentation can assume that the reader is familiar with the basics of operating the platform(s) upon which the software will operate
- User documentation cannot assume any knowledge of how the operating system functions or any knowledge of programming

Document What?

- Stating the obvious is straight forward and tedious
- Statign what is important can be extremely difficult to do in a way that the user can understand
- Deciding what is important is a non-trivial problem
- Must be careful not to make implicit assumptions about the user's knowledge or biases

"The manual said that the machine should have windows 7 or better, and so I installed Linux"

Intra Program Documentation

- In the form of Comments
- In the form of Naming
- In the form of Indentation

In the form of Comments

- Java provides two methods for comments
  - Line based comments:

```
// Anything in this line after the double slash is
// treated as a comment

int posTally; // Will keep a count of the positive
              // values > 0
```

  - Comment blocks

```
/* Anything up to and including the close comment
   is treated as a comment */

/**************************************************
 * Variables:                                     *
 *        posTally; Will keep a count of the      *
 *                  positive values > 0           *
 **************************************************/
```

Comments

- Comment blocks should be used:
  - For the program or class (later)
    - Include the authors name
    - Description of the purpose of the overall program/class
    - Dates modified
  - To describe all submodules
    - Method Contract
      `IMPORT`, **`EXPORT`** and **`ASSERTION`**
    - Purpose of the submodule - its job
    - Dates modified
- These have been omitted from the future slides to save room, and for clarity
- You are expected to do this for **<u>all</u>** of your programs/classes

Comment Blocks

```
/*************************************************
 * Author: Mark Upston                           *
 * Purpose: To do something with my App.         *
 * Date: 25/02/2020                              *
 *************************************************/
public class MyJavaApp
{
    public static void main(String [] args)
    {
        // Variable Declarations
        int a, b;
        double result;

        // Algorithm
        ... // Code
        result = myMethod(a, b);
    } // End Main

... // Continued on next slide
```

Comment Blocks (2)

```
    /*************************************************
     * Purpose: To divide two integers (as Reals)    *
     * Date: 25/02/2020                              *
     * Import: a (Integer)                           *
     *         b (Integer)                           *
     * Export: myVal (Real)                          *
     *************************************************/
     public static double myMethod(int a, int b)
     {
         // Variable Declarations
         double myVal;

         // Algorithm
         ... // Code
         return myVal;
     } // End myMethod
} // End Class
```

In the form of naming

- Programmer constantly has to invent names for classes, methods, variables, constants, etc.
    - We call these names identifiers
- You need to give considerable thought to every name you create - they are your bricks, glued together with the mortar of operators, ensuring that your code is readable
- Names should be:
    - Unique
    - Meaningful
    - Unambiguous
    - Consistent
    - Enhanced by case
- There are also reserved words (which are **always** in lower case): identifiers with special meaning, which must be used in predefined ways
    - **`class`**, **`public`**, etc.

Rules for Identifiers

- Consists only of letters, digits, _ or $
- Cannot start with a digit
- Cannot be a reserved word
- Case sensitive

```
StNo;
stno;
STNO;
Stno;
```

  - Are all different identifiers
- Can be any length

Guidelines for Identifiers

- Meaningful: give a name that truely reflects the nature of the value it holds
    - **studentNo** is different to **noOfStudents**
- Readable:
    - **studentNo** not **stdnbr**
- Consistent: be consistent in all aspects
    - Abbreviations, case, indentation, etc.
- Avoid verbosity:
    - Notwithstanding that the length is unlimited, avoid overly long names
    - Example:
        - Good: **StudentNo**
        - Bad: **Identification_Number_of_the_Student**
- Use underscore to good effect (Rarely)
- Use capitalisation to good effect (Always)

Java Identifier Naming Conventions

- Constants should be completely uppercase

  ```
  public static final int MAXSTUDENTS = 30000;
  ```

- Class names should be Capitalised

  ```
  public class ThisIsAClass
  ```

- Methods and variables should be internally capitalised

  ```
  public void thisIsAMethod()
  private double thisIsAVariable;
  ```

In the form of Indentation

- The indenting (and line spacing) of statements to support logic is a mandatory part of documentation
  - Indent code as you enter it - **don’t** leave it to be edited later
- The next slides show classes and methods
- We will cover indenting control structure statements in detail in lecture 3

Good

```
import java.util.*;
public class ExampleOne
{
    public static void main(String[] args)
    {
        int x, y;
        double avg;
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter 1st Number: ");
        x = sc.nextInt();
        System.out.print("Enter 2nd Number: ");
        y = sc.nextInt();
        avg = calculateMean(x, y);
        System.out.println("Mean of " + x + " & " + y + " = " + avg);
    } // End main

    public static double calculateMean(int a, int b)
    {
        return (double) (a + b) / 2.0;
    } // End calculateMean
} // End class
```

Bad

```java
import java.util.*;
public class ExampleOne {
public static void main(String[] args) {
int x, y;
double avg;
Scanner sc = new Scanner(System.in);
System.out.print("Enter 1st Number: ");
x = sc.nextInt();
System.out.print("Enter 2nd Number: ");
y = sc.nextInt();
avg = calculateMean(x, y);
System.out.println("Mean of " + x + " & " + y + " = " + avg);
}
public static double calculateMean(int a, int b) {
return (double) (a + b) / 2.0;
}
}
```

- No indentation, no line spacing and brackets not aligning make it very difficult to read and follow

Poor Documentation

- Stating the obvious is a waste of time. However it can be usually done with the brain in neutral
- Documentation can be extremely tedious
- Good rule is don't say what, always say **why**
  - Example of poor documentation:

    ```
    x++; // Add one to x
    ```
  - Better documentation:

    ```
    numPeople++;
    /* Above loop exits with numPeople being one less
    than the actual number of people input. The next loop
    relies on numPeople having the correct total. */
    ```

Rules of PDI

- Do intra program documentation as you go. Doing it afterwards is:
    - Is more time consuming
    - Means that you may have forgotten what needs to be documented
- Use people who are unfamiliar with your software to proof read your documentation
- Be honest. Clearly state what does not work
- Make sure the documentation is correct and up to date before it is released
- Be clear and a concise
- Organise information and index it in a sensible manner
- Electronic documentation should be easy to access, heavily cross linked and up to date

Assignment Statement

- In Mathematics, the equals sign is a statement of a fact
  - The left hand side of the equation is the same as the right hand side
- In programming the equals sign is performing the action of: place the value on the right hand side in the variable specified on the left hand side
  - Change the left hand side to be the same as the right hand side

Java: Good and Bad

- Good
  - Platform independant execution
  - Platform independant binary data (files etc)
  - Robust
  - Does not not allow operator overloading
  - Comes with a huge class library which allows:
    - File input/output
    - Graphics
    - Event trapping/handling
    - 3D modelling
- Bad
  - Syntax is adopted from C
    - This means that some control structures are primitive and unstructured
- As you work through the rest of this unit, the good and bad will make more sense to you.

Data Types

- As stated earlier, the manner of interpretation of the 1's and 0's varies for different data types stored.
- We need to specify the type for each piece of data held in memory
- Syntax of declarations is easy; the challenge is identifying the type
    - **triangleSideA** //This is obviously a Real
    - **yearOfBirth** //This is obviously an Integer
    - **studentName** //This is a string of Characters
      //AKA: String

Data Types (2)

- What about more complex variables:
  - **dateOfBirth** `//Single 6 (or 8) digit Integer` `//or 3 seperate Integers`
  - **age** `//Single 6 digit Integer` `//or 3 seperate Integers` `//or a single Real`
  - **phoneNo** `//Single Integer` `//or a String`
- In algebra we often assume what the type (and domain) is, but machines need the type to be unambiguously defined

Variables, Constants and Literal Values

- A variable is a piece of memory in which data can be stored and retrieved. It has a name associated with it and must be declared

```
int thisIsAnInteger;
```

- A constant is similar to a variable except that its value is set initially and can *never* be modified

```
public static final int MYCONST = 12;
```

- A literal value refers to the symbols which can be used to represent the possible values available for a particular data type

Integer	-12, 42, 0
Real	-10.2, 56.8, 0.0

Java's Primitive Data Types

- Java defines 8 primitive types:

Java Type	Memory Format	Range/Domain	Range/Domain
byte	8 bit integer	-2^{7} to $2^{7}-1$	-128 to 127
short	16 bit integer	-2^{15} to $2^{15}-1$	-32768 to 32767
int	32 bit integer	-2^{31} to $2^{31}-1$	-2147483648 to 2147483647
long	64 bit integer	-2^{63} to $2^{63}-1$	±9.22337E+18
float	32 bit floating point	±6 sig. digits (10^{-46},10^{38})	
double	64 bit floating point	±15 sig. digits (10^{-324},10^{308})	
char	16 bit character	All Characters	
boolean	boolean	**true**, **false**	

Integer Data Types

- Integer: positive or negative value that consists of a whole number
- The Java primitive types **`byte`**, **`short`**, **`int`** and **`long`** are abstractions of integers from the mathematical world
- The range of integers is determined by the amount of storage available (memory) for a particular data type
- The accuracy is guaranteed
    - Stored as the exact base2 (`Binary`) equivalent of the base10 (`Decimal`) integer

Range of Integers

- Determined by how many distinct base2 values can be stored in the given number of bits: every additional bit doubles the size of the range
- For N bits, you always need 1 bit for the sign and the remaining N-1 bits can represent 2^{N-1} different combinations that directly relate to their binary value
- Note that the lack of symmetry is because of the need to represent zero (0) as one of the 2^{N-1} values:
  - {2^{N-1} negative, 0, 2^{N-1}-1 positive} values
    - Negative values stored as the 2's compliment of the number
- When an attempt to store a number which is larger/smaller than the maximum/minimum value then Integer Overflow occurs

Real Numbers

- Positive or negative value that consists of a whole number plus a fractional part (expressed in floating point, or scientific notation)
- The Java types **`float`** and **`double`** are an abstraction of the real numbers that exist in the mathematical world
- The range and accuracy of real numbers are limited in any computing system
  - Why? How would you store $\frac{1}{3}$ or $\sqrt{2}$?

Range and Accuracy of Real Numbers

- Determined by number of bits and the split up of the **mantissa** and **exponent**
- There has to be a limit on the range, by definition, you need an infinite number of bits to represent infinity (∞)
- Accuracy is obviously limited
  - The number of significant digits is limited
    - There are an infinite number of real values between any two points on the number line
  - Irrational numbers
  - Recurring decimals
  - IEEE 754 form (binary conversion)

Real and Integer Expressions

- Real operands used with **+ - * /** produce Real results

Expression	Result
27.3 + 8.4	35.7
7.0 - 10.0	-3.0
3.0 * 5.0	15.0
11.0 / 4.0	2.75

- Integer operands used with **+ - * / %** produce Integer results

Expression	Result
27 + 8	35
7 - 10	-3
3 * 5	15
11 / 4	2
11 % 4	3
10 % 2	0

Integer Arithmetic

- The integer truncation feature of **/** (DIV) and the remainder operator **%** (MOD) are very useful and powerful tools
  - Think of long division:

```
DIV      2
      4 )11
MOD      3
```

  - Assume **year** holds 4 digit year e.g., 1998
    - `(year / 100) + 1   //Evaluates to Century`
  - Other examples:
    - `numPages = (numLines / linesPerPage) + 1   //Number of pages`
    - `hours    = (hhmm / 100)   //Hours from 24hr time`
    - `minutes  = (hhmm % 100)   //Minutes from 24hr time`

Mixed Mode Arithmetic

- Mixed mode arithmetic occurs when a numeric expression contains a mixture of integer and reals

```
y = 3 + 4.5;
z = 2 / 3.0;
```

- Programming languages always have a set of rules for evaluating mixed mode expressions, but:
  - It's not the same across different languages
  - Not always supported by the compiler
- Errors caused by mixed mode arithmetic in program code are *extremely* hard to find
- The rule is to **never** use mixed mode arithmetic

Type Casting

- To convert from one data type to another, a Type Cast is used
- The syntax is: `(NewDataType)(expression);`
- Examples:

```
int a, b, c;
double x, y, z, average;

... // Initialise variables

average = (double)(a + b + c + d) / 4.0;
    // a + b + c + d are added first, the value is converted to a double
    // then divided by 4.0, then assigned to average

z = (double)(a + b);
    // a and b are added, the value is converted to a double
    // then assigned to z

a = (int)y;
    // the value of y is truncated to an int, then assigned to a
    // (y is NOT changed)
```

Type Casting (2)

- Examples cont:

```
int a, b, c;
double x, y, z, average;

... // Initialise variables

x = (double)(a / b);
    // this is a div b, then converted to double and assigned to x
    // if a is 5 and b is 2, x is assigned 2.0
    // x = (double)(5/2);

y = (double)a / (double)b;
    // this is convert both the values of a and b to doubles
    // then normal division, same as y = 5.0/2.0;
```

- Note that conversion from a real data type to an integer data type involves truncating the real value (i.e., not rounding)

Operator Precedence

- Below are the Java operations listed from higher to lower precedence.
  Note: there are many more

Operator	Java	Associativity
Ops on References	`. []`	Left -> Right
Unary	`- ++ --`	Left -> Right
Multiplicative	`* / %`	Left -> Right
Additive	`+ -`	Left -> Right
Relational	`> >= < <=`	Left -> Right
Equality	`== !=`	Left -> Right
Logical AND	`&&`	Left -> Right
Logical OR	`||`	Left -> Right
Assignment	`=`	Right -> Left

- Remember: "= != =="
  - Assignment does **not** equal Equality

Operator Precedence Example

Expression	Result
7 + 23 * 6	= 7 + 138 = **145**
3 * 2 + 4 * 5	= 6 + 20 = **26**
-6 * 2	= **-12**
3 + 5 * 6 / 4 + 2	= 3 + 30 / 4 + 2 = 3 + 7 + 2 = **12**
3.0 + 5.0 * 6.0 / 4.0 + 2.0	= 3.0 + 30.0 / 4.0 + 2.0 = 3.0 + 7.5 + 2.0 = **12.5**
-6 * 2 + 3 / 4	= -12 + 0 = **-12**
2 * 5 % 2	= 10 % 2 = **0**

Expression Guidelines

- Never use mixed mode arithmetic
  - Use type casting to avoid mixed mode arithmetic
- Precedence rules are the same as in mathematics
- Use parentheses to simplify readability of complex expressions
- Use intermediate steps to split complex expressions into explicitly seperate steps
- Don't over-parenthesise simple expressions
- Beware or algebraic simplicity:

$$x = \frac{y - p}{z - q}$$

- This is written in Java as `x = (y - p) / (z - q)`

Assignment Operators

- Short hand way of modifying the contents of a variable

Traditional	Alternative
`x = x + 5;`	`x += 5;`
`x = x - 32;`	`x -= 32;`
`fred = fred * 2;`	`fred *= 2;`
`ralf = ralf / 6;`	`ralf /= 6;`

- Must be careful though:
  - `y *= x - 2;`
- Is the same as:
  - `y = y * (x - 2);`
- But **<u>not</u>** the same as:
  - `y = y * x - 2;` or `y = (y * x) - 2;`

The Increment/Decrement Operator

- Increment (++) / Decrement (--)
  - `x++;` is the same as `x = x + 1;`
  - `x--;` is the same as `x = x - 1;`
- Be careful though:
  - `x = x++;` is *nonsense*
- Also, `++x;` and `--x;`
  - These work differently in expressions
  - In this unit, do not use them in an expression

Character Data Types

- A **char** stores a single Character e.g., 'a', 'A', '6', '&', etc.
- Stored in a Unicode, a standard that arbitrarily designates a bit pattern to represent a particular character symbol
- If the character is a decimal digit e.g., '8' can't do arithmetic with it: `'8'+'6';` cant possibly be expected to be meaningful
- A character occupies 16 bits & is coded according to the Unicode standard, thus there are >32,000 different possible combinations to represent characters, more than enough
    - The lower (rightmost) 8 bits is identical to the ASCII system
- Order of the characters is determined by the codes:
    - 'A' < 'B' . . . < 'Z' < . . . < 'a' < 'b' < . . . < 'z'

The Java String Class

- Generally a string is a collection of 0 (empty) or more characters
- The Java String class provides us with the facility to handle strings
- String variables are objects but they can be used as if they were primitive variables:
    - `String unitName = "Programming";`
- Can also be treated like an object:
    - `String unitName = new String("Programming");`

The Java String Class (2)

- Must **never** forget **String**'s are objects
- Whats wrong with:
  - `(nameOne == nameTwo)`
- Strings can be concatenated using the plus (**+**) operator
- Example:

```
int x = 42;
String message;
message = ”x = ” + x;
```

- Java converts the value of **x** (42) to a **String** ("42") and joins it to the end of `”x = ”` to make `”x = 42”`

Java toString() method

- A convention is that **all** Java objects have a **toString()** method
  - This method is used to provide a **String** representation of the data stored in the object
    - Covered later

Arrays

- The variables we have seen so far represent a single item
  - e.g., `int numTimes;` is a single Integer number
- But we also often work with *sets* of similar data
  - e.g., the list of student marks in PDI. How would you handle this?
    - `double student1Mark, student2Mark, ..., studentXMark;`
  - Clumsy
    - Variables names defined at compile time - 'Hard Coding' means that the program can *never* change the number of students
    - Calculating the average involves a massive amount of typing
    - Can't convenientley pass the set of students around

Arrays (2)

- Arrays are a solution to this problem
- Simplest kind of data structure for storing sets of data
  - Arrays are built-in to *all* programming languages
  - Instead of just one element, an array is a variable that contains *many* elements
  - The array variable itself is a reference to the first element of the array
    - Java: the array variable also knows how large the array is
    - C: doesn't store the array length - you have to do it yourself!

Array Properties

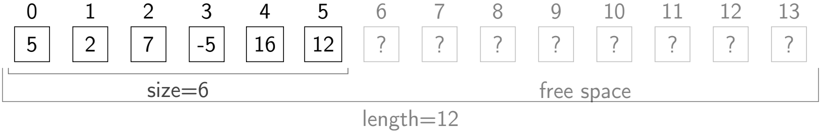
- Elements are located sequentially in memory
  - i.e., The array is a *contiguous* block of memory
- All elements must have same data type
  - e.g., **`double`**
- Arrays can be initialised to any size
  - Within memory limits
- However, once initialised they **cannot** be resized
  - Must create a new array and copy over the contents of an old array in order to 'resize' it

Array Properties

- Array **capacity** (length) vs **actually used** elements
  - Initialised an array to `length = 20` *doesn't* set 20 elements, it merely *reserves space* for 20 elements
    - Hence initialisation is also referred to as *allocation*
  - You therefore typically need to keep track of how many elements you have *actually used* in the array
    - i.e., the count of elements, as distinct from the array capacity
    - It is typical that you allocate more space than you initially need, since arrays have a fixed capacity and cannot be resized

Arrays - Accessing Elements

- Once you have allocated an array, you need to be able to work with the elements inside the array
- Elements are accessed via an *index* or *subscript*
    - The index is the element number in the array
        - `0, 1, 2, 3, 4, ..., N-1` where `N` is the allocated size (length)
        - The index is an *offset* from the first element



0	1	2	3	4	5	6	7	8	9	10	11	12	13
5	2	7	-5	16	12	?	?	?	?	?	?	?	?

size=6

free space

length=12

Arrays in Code (Java)

- Declaring: put '**[]**' on the end of the data type
  - `double[] theArray;`
  - Any data type can be used with arrays, including classes
- Allocating: use '**new**' keyword with special '**[]**' syntax
  - `theArray = new double[100];`
- Indexing: **theArray[index]**, '**index**' must be an **int**
  - Negative indexes or indexes that are past the end of the array (i.e., >= `length`) will cause an error during runtime
- Assignment: `sameArray = theArray;`
  - Assignment doesn't copy the array *contents*, it only makes the L.H.S variable point to the R.H.S
  - Same with passing an array as a parameter to a method

Primitive or Object

- All of the primitive datatypes have Object Equivalents:

Class	Primitive
Integer	byte, int, short, long
Float	float
Double	double
Character	char
Boolean	boolean

Initialising Variables

- What is stored in a variable when it is created?
- Java auto-initialises variables:
  - Primitive Variables:
    - Numeric: set to zero
    - char: set to blank
    - boolean: set to false
  - Object variables:
    - set to null
    - null represents an invalid memory address
- Not all programming languages auto-initialise so it is extremely poor programming style to rely on auto-initialisation
- You should **always** explicitly initialise variables

Next Week

- The next Lecture will address the following:
  - Testing
  - Boolean Operations
  - Control Structures (Selection)
    - The `If-Then-Else` Statement
    - The `Case` Statement