Introduction to Programming – Basics

Sujit Chakrabarti

1 Introduction

Software engineering is the art and science for developing industrial strength software systems. What are industrial strength systems? For one, they are not classroom software systems. They are systems which real people use to get their real work done. Software systems on which people entrust their money and business (e.g. banking systems, power systems), often life (e.g. healthcare devices like MR, cars and aeroplanes), nations entrust their security (military systems), and the world entrust its safety (e.g. nuclear reactors).

Needless to say, the expectations from these software systems are sky-high. They shouldn't just be richly featured, these features must also come with certain promise of correctness and reliability. Such characteristics can be achieved only when software development is approached with certain degree of discipline and rigour.

In this programme, you will learn a variety of aspects of that discipline and rigour.

In the next two modules, I will be introducing you to the basic vocabulary or tools with which you have to work at the ground level when building a software system. We start with the basic elements of a programming language – features which you will find in all programming languages. Thereafter, we go to have a detailed discussion on a specific style of programming – called object oriented programming – which in today's world is pretty much the *de facto* style of programming. We conclude this module by discussing briefly on this programming style can be harnessed in the design and construction of larger software systems.

To keep our discussions grounded, we use Java as our programming language. That seems to be a good choice to us. For one, Java has been around for quite a while, and is still going quite strong. You pick any software system in use today, and the most likely language of implementation is Java. As the development ecosystem around Java is so widespread, there's been a lot of good code written in this language. Therefore, if you happen to take

up a new development project – whether it's GUI, web-based, Android, or embedded – there's a lot already out implemented in Java there to build upon. In short, Java is a very strong candidate as a language of choice for your next project.

However, just as a clarification, let me mention that much of what we discuss using Java smoothly carries over to most other modern programming languages like C++, C#, Python or PHP.

2 Computation

Computation is, in simple terms, *information processing*. Information is often hidden in data, and another alternative, and slightly restricted, term for computing is *data processing*. As evident from the name, this has two elements to it: *data* and *processing*.

As an example of data, let's consider the table shown in fig. 1. There's a lot of information in this table. Each piece of information is an answer to some potential question. For example: you may want to know the number of students in this table. Of course, 5. Pretty easy, right? What's the highest/lowest mark? Now, you may have to look a bit more closely. What's the average mark? Hm! This will take a while to figure out. What's the name of the top scorer? Oops! That information isn't there in this data. In other words, a piece of data may possess any amount of information. Some of it could be apparent. Some not so apparent. What's not so apparent may have to be revealed using some calculation. You could go ahead and do that calculation using a calculator, a pen or a paper. Or you could instruct a computer to do it using computer programming.

1	86.7
2	81.5625
3	85.85
4	89.2625
5	85.9625

Figure 1: Student marks

Learning to program is, to a large extent, all about learning methods of representing data in a computer, and learning methods of using a computer to processing them to give out answers to certain questions. Strictly speaking, computing is all about transforming data from one form to another so that certain information we are interested in gets revealed. However, in real life, a

computer program has to be able to do a couple of more things to be useful, namely, inputing data (from the keyboard, sensors, network, hard-disc etc.) outputing it (to the monitor, LED, actuators, printer, network, hard disc etc.).

As an example of a computer program, we can think of a program which takes the above data and gives us the answer to all the questions that we have asked above about it.

A program being a bunch of instructions given to the computer, we must use a language to give those instructions, and that's what a programming language is: a language to give instructions to the computer. A programming language would therefore have the primitives to allow you do the following things:

1. compute (i.e. process data)
2. input data
3. output data
4. store data

So, let's start learning some of these primitives:

3 Output

First let's start with a learning how you can output something from your program into the standard output. Fig. 3 shows the program that prints `"Hello world!"` on the standard output.

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

Figure 2: Java program to print `"Hello world!"` to the standard output

When we compile and execute this program, we get the output as expected:

```
Hello world!
```

There's a lot happening the the above little code. We will gradually get acquainted to each aspect. However, for the moment, I would suggest you to focus on only the line which prints the `"Hello world!"` message. It says: `System.out.println("Hello world!");`

The same instruction can be used to print various things, e.g. the value of a number. For example: `System.out.println(1234);` will print `1234` – instead of "Hello world!" – on the standard output.

Theme for Learning Java

With the above first lesson in programming in Java, let us set up – sort of – a theme that we will follow in learning to program in this language. Broadly, computing is about data processing, as mentioned earlier. A programming language is a set of features which allow us to process the data in a powerful way, and allows us to represent the data in a way that they represent the information in the 'best' possible way. These are called *data abstraction mechanisms*. It also allows us to manipulate them in the 'best' possible way. Let's call them *control abstraction mechanisms*. 'Best' in quotes simply because, it depends on specific problem in question, and is by and large the central topic to which the entire discipline of computer science is devoted.

4 Control Abstractions

Let's start with learning a few control abstraction mechanisms: mechanisms which let us compose bigger programs from atomic instructions. These mechanisms also allow us to decide which of the instructions in a program execute, and when. In other words, they let us decide how the control flows through the program we write.

4.1 Sequences of instructions

We could sequence as many print instructions as we wish, to have that many things printed in the same order. Like this:

```
System.out.println("Hello world!");
System.out.println(1234);
```

On running the above, we get the following output:

```
Hello world!
1234
```

Sequencing is the simplest of all control structures. It relies on the implicit semantics of a programming language which says that instructions would be executed in the same order as they appear in the program text. The program execution completes after the execution of the last instruction.

Even though this may appear obvious to most of you, it's good to remember that the programming languages – which are engineering artifacts themselves – have been designed to behave this way.

4.2 Decision by if-else blocks

The programmer would achieve significantly higher power if he/she could decide – based on certain conditions – which of the instructions in his program executed. This can be done using the if-else construct.

```
public class IfElse {
  public static void main(String[] arg) {
    if(1 == 1) {
      System.out.println("Hello world!");
    }
    else {
      System.out.println("Hello class!");
    }
  }
}
```

Figure 3: Java program to print `"Hello world!"` or `"Hello class!"` to the standard output based on the value of `c`.

The above program prints `"Hello world!"`. However, the expression `1 == 1` is changed to `1 == 2`, the output would be `"Hello class!"`.

Explanation. A program fragment `if` (E) B_1 `else` B_2 means that if E evaluates to true, then the block of instructions B_1 will be executed, otherwise the block B_2 will be executed. In the example we just saw, in the first case, the expression in place of E was `1 == 1` which is of course true. Therefore, `System.out.println("Hello world!");`, which is in the place of B_1, gets executed. In the second case, we change E to `1 == 2`, which is false, `System.out.println("Hello class!");`, which stands for B_2 in the example, gets executed.

5 Data Abstraction

5.1 Expressions

In the examples we have discussed so far, the only instructions to be executed are output instructions, which strictly don't even comprise of computation. The most primitive computational structures are expressions. Expressions are pieces of texts which the programming language can compute and assign a value as a result of that computation. We have already come across a number of expressions in the examples we have seen so far:

1. `1234` is a numerical constant, and broadly falls in the category of arithmetic expressions (all expressions which evaluate to numbers).

2. `"Hello world!"` is a string constant.

3. `1 == 1` is a relational expression

Calculation involves arithmetic expressions. Arithmetic expressions are composed by joining subexpressions – which themselves are arithmetic expressions – with arithmetic operators, e.g. + for addition, - for substraction, * for multiplication and / for division. Arithmetic expressions can be made arbitrarily large corresponding to full-fledged computations. We present an example here.

Consider three friends – Abhiraj, Leon and Vigyan – go for a weekend outing. They decide upfront that they will split the expenses equally. Abhiraj buys the movie tickets online (Rs. 990). Leon pays the bus tickets (Rs. 140) and also purchases the popcorns in the interval (Rs. 150). Vigyan pays for the meal after the movie (Rs. 1100). Write a Python program to work out (and print) the due amount on each friend (note that the due could either be positive or negative).

The amount due on any of the three friends is nothing but the average expenditure substracted by that friend's expenditure. For example, Abhiraj's due is Rs. $\frac{990+140+150+1100}{3} - 990$. This comes out to be approximately Rs. -196.67.

The program to calculate and print the dues of each of the above friends above is shown in fig. 4

```
public class Due {
  public static void main(String[] arg) {
    System.out.println("Abhiraj's due = " +
      ((990 + 140 + 150 + 1100)/3.0 - 990));
    System.out.println("Leon's due = " +
      ((990 + 140 + 150 + 1100)/3.0 - (140 + 150)));
    System.out.println("Vigyan's due = " +
      ((990 + 140 + 150 + 1100)/3.0 - 1100));

    System.out.println("Verifying the sum of all dues = " +
      (((990 + 140 + 150 + 1100)/3.0 - 990) +
      ((990 + 140 + 150 + 1100)/3.0 - (140 + 150)) +
      ((990 + 140 + 150 + 1100)/3.0 - 1100)));
  }
}
```

Figure 4: Java program to print the due of each friend in the given example.

An important observation is how Java allows us to use parenthesis – or round brackets `'('` and `')'` – to group sub-expressions for two reasons:

1. One is to control the associativity of the operation.
2. Second is to increase readability

5.2 Variables

Even though the code in fig. 4 works perfectly, there are a number of issues with the code in fig. 4.

1. **Unreadable code.** Even with parentheses, arithmetic expressions in the given example of fig. 4 tend to be too long to be very readable. There's hardly anything in our code which contain hints or helps to read the expressions. For example, the sub-expression `(990 + 140 + 150 + 1100)`, which appears in all the instructions above, has a special significance. It's the total expenditure. Nothing in the code tells us that, and it's left to the reader to figure that out.
2. **Efficiency.** Similary, the same sub-expression – `(990 + 140 + 150 + 1100)` – gets computed in every separate instruction, even though it's clear that everytime it would result in the same value. This is wasteful computation. Could we compute this once, and re-use it again and again?

3. **Computation and output.** This is the first program we have written so far which has some non-trivial computation. But it's all embedded inside these print statements. Could we do something to separate out the computation from the result output part?

The single answer to all the above questions lies in the idea of variables.

```
public class Due {
  public static void main(String[] arg) {
    double totalExpenditure   = 990.0 + 140.0 + 150.0 + 1100.0;
    double average            = totalExpenditure / 3.0;
    double abhirajExpenditure = 990.0;
    double leonExpenditure    = 140.0 + 150.0;
    double vigyanExpenditure  = 1100.0;
    double abhirajDue         = average - abhirajExpenditure;
    double leonDue            = average - leonExpenditure;
    double vigyanDue          = average - vigyanExpenditure;

    System.out.println("Abhiraj's due = " + abhirajDue);
    System.out.println("Leon's due = "    + leonDue);
    System.out.println("Vigyan's due = "  + vigyanDue);

    System.out.println("Verifying the sum of all dues = " +
      (abhirajDue + leonDue + vigyanDue));
  }
}
```

Figure 5: Java program to print the due of each friend in the given example using variables.

Take a look at the code in fig. 5. This is functionally identical to fig. 4. However, it's a good lot more readable – probably open to debate. But let's look at it as per the questions listed above. Here, we have given meaningful names to various sub-expressions. This makes it clearer as to what's going on in the program. Further, we compute `totalExpenditure` (standing for total expenditure) and `average` (standing for average expenditure per person) only once. And it gets used again three times where we were computing this value again and again in fig. 4. We presume that this makes fig. 5 a slight bit faster than fig. 5, though it mayn't be visible to the naked eyes. Finally, you can now mark out clearly the parts of the code which correspond to pure computation, and that which is outputing the result.

The above has been possible due to variables. Variables are names representing memory storages where you can temporarily *store* values generated out of expressions. These stored values can be *used* at a later point in the program. What can be used as a variable in Java? Here's a list:

- Single letters, e.g. `x`, `y`, `z` etc.
- Strings of letters e.g. `abc`, `XyZ` etc.
- Strings of letters and digits starting with a letter e.g. `abc1`, `z23y` etc.
- Above kinds of strings with underscores (`_`) at arbitrary positions: `abc_1`, `_abc1_` etc.

What can't be used in a variable name? Variable special characters e.g. `@`, `!`, etc are not allowed in variable names.

There are three ways a variable can appear in a program: *declaration*, *definition* and *use*.