CS125 : Introduction to Computer Science

Lecture Notes #4
Type Checking, Input/Output, and Programming Style

Lecture 4 : Type Checking, Input/Output, and Programming Style

Type Checking

The various operations in your program (up to now, you've only learned about arithmetic operations, so you can imagine we are only talking about numerical calculations for now, but the same idea applies to other operations you will learn later) are performed by the machine when your program runs, NOT by the compiler when your program is compiled. Part of the reason for this is that, often, those operations would involve values inputted by the user, and the compiler would therefore not know those input values since the user would not enter values as input until the program is being run. However, as we mentioned earlier, the compiler does generate the machine code that will tell the processor to perform those calculations. In addition to that task, the compiler makes sure that the way in which we use our variables and values is legal – in part, by performing *type checking*, confirming that variable types match value types whenever they are supposed to.

We can do this type checking ourselves by replacing our variables, values, and expressions with their types, thus obtaining what we will call *type expressions* (as seen in the explanations that accompany the code in the next five examples). We can double-check that we are using types correctly by working through these type expressions to see what type of values our code expressions produce. The compiler does essentially the same thing when type-checking your code during compilation, so the next five examples below serve two purposes – one is to explain how you could reason through your own code to see what it does, and the second is to explain a bit about how the compiler does its job.

Example 1 – integer calculations

```
public class Example1
{
   public static void main(String[] args)
   {
      int weightedQuantity;
      int exam1, exam2;

      exam1 = 60;
      exam2 = 70;
      weightedQuantity = 20 * exam1 + 80 * exam2;

      // weighedQuantity is 6800 when program runs
   }
}
```

- `20 * exam1` is `int` * `int` which is an `int`
- `80 * exam2` is `int` * `int` which is an `int`
- The addition is `int` + `int` which is an `int`
- The assignment is `int = int`, which is okay.

Example 2 - a slight problem with division

```
public class Example2
{
   public static void main(String[] args)
   {
      int sum, average;
      int exam1, exam2;

      exam1 = 75;
      exam2 = 90;
      sum = exam1 + exam2;
      average = sum/2;

      // When program runs, average is 82, not 82.5
   }
}
```

- `exam1 + exam2` is `int + int` which is an `int`
- The assignment to `sum` is `int = int`, which is okay.
- The division is `int/int` which is an `int` – result is *truncated!!*
- The assignment is `int = int`, which is okay. Result: 82

Example 3 - change `average` to `double`

```
public class Example3
{
   public static void main(String[] args)
   {
      int sum;
      double average;
      int exam1, exam2;

      exam1 = 75;
      exam2 = 90;
      sum = exam1 + exam2;
      average = sum/2;

      // When program runs, average is 82.0, not 82.5.
   }
}
```

- `exam1 + exam2` is `int + int` which is an `int`
- The assignment to `sum` is `int = int`, which is okay.
- The division is `int/int` which is an `int` – result is *truncated!!*
- The assignment is `double = int`, which is okay. The int is automatically converted to a double...which is why we get 82.0 and not 82.

Example 4 – fix #1: change `sum` to `double`

```
public class Example4
{
   public static void main(String[] args)
   {
      double sum;
      double average;
      int exam1, exam2;

      exam1 = 75;
      exam2 = 90;
      sum = exam1 + exam2;
      average = sum/2;

      // average is 82.5 when program runs
   }
}
```

- `exam1 + exam2` is `int + int` which is an `int`
- The assignment to `sum` is `double = int`, which is okay. `sum` now holds not 165, but rather, 165.0.
- The division is `double/int` which is a `double` – finally, result is *not* truncated!!
- The assignment is `double = double`, which is okay. The result: 82.5

Example 5 – fix #2: change `int` literal to `double` literal

```
public class Example5
{
   public static void main(String[] args)
   {
      int sum;
      double average;
      int exam1, exam2;

      exam1 = 75;
      exam2 = 90;
      sum = exam1 + exam2;
      average = sum/2.0;

      // average is 82.5 when program runs
   }
}
```

- `exam1 + exam2` is `int + int` which is an `int`
- The assignment to `sum` is `int = int`, which is okay.
- The division is `int/double` which is a `double` – again, result is *not* truncated!!
- The assignment is `double = double`, which is okay. The result: 82.5

The rule is that, if both operands of an addition, subtraction, multiplication, or division, are of type `int`, then the result will be of type `int` – and as we have seen, for division, that means that the fractional portion is truncated. However, if one or both of the operands is a `double` instead of an `int`, then the result will be a `double` (and thus division would not be truncated).

Outputting text using `System.out.println`

Recall our `HelloWorld` program:

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

- The code `"Hello World!"` in the code above is a literal of type `String` – the first non-primitive type we have encountered. We won't talk about variables of type `String` for a while yet, but we *will* use literals of type `String`. Any text that appears in between double quotes is a literal of type `String`.
- The general form is `System.out.println(expr);` where `expr` is some expression. The expression is evaluated, and the result is then printed to the screen.
- If you have only double-quoted text inside the parenthesis, then since that is a `String` literal, the expression in parenthesis is of type `String`.

Output of variables

In the previous example, the item we were sending to the screen was literal text. However, it is also possible to print out the values of our variables.

```
public class Example6
{
   public static void main(String[] args)
   {
      int exam1, exam2, sum;

      exam1 = 60;
      exam2 = 70;
      sum = exam1 + exam2;

      System.out.println(sum);
   }
}
```

In the code above, the expression in parenthesis is of type `int`. The use of `System.out.println` activates some pre-written code already present in the virtual machine. There are many different pieces of pre-written code – one to print a `String`, one to print an `int`, one to print a `double`, and so on. The compiler picks the right one based on the type of the expression in the parenthesis.

`String` concatenation

- Concatenation is the merging of two `Strings` into one by placing them one after the other. The type of the expression is `String + String --> String`
- One example: `"Levia" + "than"` evaluates to the `String` value `"Leviathan"`
- If one of the operands is a variable of a primitive type instead of a `String`, that operand's value is automatically converted to a `String`, just as in our earlier type expression examples, the value of an `int` variable got automatically converted to a `double` value if we used an assignment statement to write an `int` value into a `double` variable.
- For example, in the expression:

  ```
  System.out.println("Sum is:  " + 20);
  ```

  we have a `String + int` operation, and that means that the `+` is a concatentation, and that means the value of the `int` should be converted to a `String` automatically and then the concatenation done.
- `"Sum is:  " + 20 -> "Sum is:  " + "20" -> "Sum is:  20"`
- So, again, the meaning of an operator depends on the type of the operand(s). The operator `+` in an `int + int` expression has a different meaning than the operator `+` in a `String + int` expression.

An example using concatenation

```
public class Example7
{
   public static void main(String[] args)
   {
      int exam1, exam2, sum;

      exam1 = 60;
      exam2 = 70;
      sum = exam1 + exam2;

      System.out.println(exam1);
      System.out.println(exam2);
      System.out.println();
      System.out.println("Sum is " + sum);
   }
}
```

- When there is nothing at all in the parenthesis, all that is printed is a blank line.
- In the last line, the type of the expression in parenthesis is `String`, and that expression evaluates to the `String` value `Sum is 130` (which is the `String` value that gets printed to the screen).

A bit more messing with types

Consider the following code:

```
public class Example8
{
   public static void main(String[] args)
   {
      int v1 = 5;
      int v2 = 7;
      System.out.println("Sum is " + v1 + v2);
      System.out.println(v1 + v2 + " is the sum");
      System.out.println("Sum is " + (v1 + v2));
   }
}
```

The `+` operator is read left-to-right, so in the first two output statements above, the leftmost `+` is done first. In the third output statement, the parenthesis take higher precedence and so the rightmost `+` is done first. Most operators are read left-to-right. A few, such as assignment, are read right-to-left, as with the statement `a = b = c = d = 2;`, where `d` is assigned the value `2` first, then `c` is assigned the value of `d`, and so on. A few other operators can't be used multiple times in a row to begin with. But most operators are read left-to-right.
So, the first output line:

```
   String + int + int   "Sum is " + 5 + 7
   ------------
          |
          |
        \./
       String   + int   "Sum is 5" + 7
       --------------
             |
             |
           \./
          String         "Sum is 57"
```

And the second output line:

```
   int + int + String   5 + 7 + " is the sum"
   ---------
       |
       |
     \./
      int  + String     12 + " is the sum"
      --------------
            |
            |
          \./
          String         "12 is the sum"
```

And the third output line:

```
   String + (int + int)  "Sum is " + (5 + 7)
           ------------
                 |
                 |
               \./
   String +     int      "Sum is " + 12
   ------------------
           |
           |
          \./
         String          "Sum is 12"
```

One last note...

Everything that is true for `System.out.println(...);` is also true for `System.out.print(...);` except that the latter doesn't start a new line after the printing, and the former does.

```
        System.out.println("Hello, ");
        System.out.println("world!");
will print: Hello,
            world!
            ^
```

and the next thing to be printed goes where the ^ is. (The ^ is not actually printed, it's just an indicator in our two examples on this page.)

```
         System.out.print("Hello, ");
         System.out.print("world!");
will print:  Hello, world!^
```

and the next thing to be printed goes where the ^ is.

This means that the statement `System.out.print();` is meaningless. With `System.out.println()`, there is at least a new line started even if no other text is printed. But if you don't start a new line *and* you don't print anything, then you did no work whatsoever. So there's no reason to have a `System.out.print();` command.

Input using `Keyboard.java`

So, now that we have seen how to output data to the screen, we also need to see how to input data from the keyboard. Java, oddly enough, does not have nice, easy-to-use facilities for reading input from the keyboard in a command-line application (at least as of this writing; there are rumors that such useful tools are being added to the language in a future revision). The only tools Java provides for reading input from the keyboard, are very low-level tools that a beginning student would not understand how to use. So the authors of one of your recommended textbooks have written a Java file, `Keyboard.java`, that builds the kind of nice facilities we need out of the more primitive facilities that Java provides us.

In order to use the input tools we describe below, you will need to put the `Keyboard.java` file in the same directory as your MP!!! Forgetting to do this is the single most common mistake made by students on the first MP. The compiler can't use the `Keyboard.java` file if the file isn't there! Your first MP will direct you to where to obtain a copy of this file, as part of an output-testing tutorial, so you'll know where to get a copy of the file when you need it.

If you happen to be curious, you can take a look at the `Keyboard.java` source code. But, you don't ever really have to do this. You can just type into your own code the kinds of expressions we'll discuss below and that will be enough to make input from the keyboard work for you. (As we learn more about Java in the first half of the course, the source code of the `Keyboard.java` file might start to make a little bit more sense to you than it would right now.)

The `Keyboard.java` file provides us with five tools for our use:

- `Keyboard.readInt()`
- `Keyboard.readChar()`
- `Keyboard.readDouble()`
- `Keyboard.readString()`
- `Keyboard.endOfFile()`

We won't be dealing with the last two right now – we'll only be using `Keyboard.readInt()`, `Keyboard.readChar()`, and `Keyboard.readDouble()`. Each of those three expressions evaluates to a particular type, namely, `int`, `char`, and `double`, respectively.

- `Keyboard.readInt()` – makes your program pause until the user types in an integer and hits return; then, the value of the expression is that integer that the user typed in.
- `Keyboard.readChar()` – makes your program pause until the user types in an character and hits return; then, the value of the expression is that character that the user typed in.
- `Keyboard.readDouble()` – makes your program pause until the user types in an floating-point value and hits return; then, the value of the expression is that floating-point value that the user typed in.

Input example

Read in an integer and print it out again.

```
public class Example9
{
   public static void main(String[] args)
   {
      int myValue;
      System.out.println("Enter new value:");
      myValue = Keyboard.readInt();
      System.out.println("Your value was: "
                                  + myValue);
   }
}
```

- `Keyboard.readInt()`, like any other expression, is on the right of the assignment operator.
- The first output line is called a *prompt*; you should always print a prompt before you ask for input, in order to let the user know what they are supposed to input. Otherwise the program seems to just halt suddenly.
- Note that the last line is broken into two lines so as not to run off the end of the slide. Likewise you can use that technique to avoid running past 70 or 80 characters on a line in your code. Just don't put a split in the middle of a double-quoted `String` literal; if you start a value with double-quotes, the closing double-quotes must be on the same line.
- When you are writing your first MP, try running the following two lines:

  ```
  javac Keyboard.java
  java Keyboard
  ```

  The first line will work fine, and will produce a file `Keyboard.class` which – as we described earlier – will contain the machine code the compiler produced from the source code in the `Keyboard.java` file. However, the second line will NOT work – you will get an error message telling you in some way that your file (`Keyboard.class`) is missing information about `main()`. Remember the word `main` that was part of our "program skeleton"? Well, it turns out that every program you ever run in Java, needs a `main` somewhere. If your file has no `main`, then your file is not a program you can run. And, indeed, `Keyboard.java` does not contain the code for a program you can run. What it contains are additional utilities which can be *used* by a program such as the ones you are going to write. That "program skeleton" we talked about, is what the virtual machine looks for when you start the virtual machine to run your program, and that is why files like `Keyboard.java`, which do not have that "program skeleton", can be *used* by other programs, but are not complete programs in and of themselves.

Programming Style

While certainly it is most important to write a program that actually works, it is also important to write your code using good programming style. Doing this makes your code more readable and understandable, which is very important.

Imagine if our "Hello World" program was stored in a file named `Foo.java`, and was written as follows:

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

That is legal! The compiler can understand this! But you cannot.

Think of the text of a program as being a bunch of characters one after another in a pipe. The compiler reads these characters in one by one, and all spaces, blank lines, and tab characters (collectively known as "whitespace") are discarded by the compiler. Likewise, as we have stated, all the names you use get traded in by the compiler for actual memory locations.

So, you can use all the whitespace you want, and make names as long as you want. Those things only serve to help *you* (and others) read your program.

Style component #1: Indentation

Indenting your source code properly is very important, and becomes even more important as your programs get larger. There are various "indentation standards" which vary slightly, but the general rule of thumb they are all based on is that stuff inside curly braces should be moved to the right a few spaces (three to five is a good choice).

Wrong:

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

The first thing you need to do is realize that it would be easier to tell what code was inside the class if that code was indented three spaces. (This will be even more important when we starting adding other methods besides `main` to a class.)

Better, but still not ideal:

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

The second thing to realize is that, likewise, it will be easier to read the code inside `main` if *that* code is indented three spaces from the curly braces that begin and end `main`. Again, the more code you have, the more important this becomes.

Correct:

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

Now, it is very easy to visually pick out with one glance where the class begins and ends, what code is inside it, and where `main` begins and ends, and what code is inside it.

A slight variation on this is that some people like to put the open brace on the same line as the Java syntax tool it is starting:

Also correct:

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

Since the word `public` lines up with the closing brace, you still have the same kind of visual structure. It's a *bit* harder to see it, since the code is crushed together a bit more, but on the other hand, you can fit more code in the viewing window at once. Some people think this tradeoff is worth it and others (myself among them) don't. You can make your own choice.

That is what was meant by "indentation standards" – there are slightly different ways of doing things. To give another example, some people indent three spaces, some prefer five. You should indent at least three but you don't want to indent so many spaces that your code runs off the end of the page all the time.

You can use whichever variation you prefer – just be consistent and keep your line lengths at about 80 characters or so.

Style component #2: Descriptive class, method, and variable names

As we have already discussed, it is a lot easier to quickly figure out what a given chunk of code is doing if the variables in that code have been given names that describe what data they hold.

Unclear:

```
c = a + b;
```

Much better:

```
totalScore = exam1 + exam2;
```

With relatively few exceptions, using single letters for variable names tends to be a bad idea. Even a short 4- or 5-character abbreviation of a longer word is more descriptive than `a` or `G` or `q`. Likewise, as you start to write other methods besides `main`, and as you start to write other classes, you should choose descriptive names for those methods and classes as well. Any names you choose should in some way describe the purpose of what is being named.

Style component #3: Commenting

Indentation makes it easy to read your code; descriptive variable names make it easy to tell what those variables are for. However, not every collection of statements has an immediately-decipherable purpose even if you know what each of the variables holds. In addition, it often helps to explain the general purpose of a particular file, or the individual major divisions of code within that file.

So, for the purposes of documentation, Java provides syntax for *commenting* our code. Comments are basically just text descriptions of whatever it is you want to describe or explain – the purpose of the file, a quick summary of the intent behind a particular section of code, or whatever else. We use a special Java syntax to notate these remarks as comments; as a result, the compiler ignores them just as it ignores extra whitespace. Thus, we can add as many comments as we want without affecting the resultant machine code at all. The comments are there for the code reader's benefit only.

One syntax for commenting

Most of the time, the commenting syntax you will want to use is the use of the “double slash”: `//`. You can place that two-character symbol anywhere in your code, and anything from that point to the end of the line of text will be ignored by the compiler.

Example:

```
// Class: HelloWorld
//    - in control of greeting generations
//    - Written August 1999 by Jason Zych
public class HelloWorld
{

   // Method: main
   //    - controls main execution of program
   public static void main(String[] args)
   {
      // line prints out a greeting
      System.out.println("Hello World!");
   }

}
```

Another commenting syntax

If you want to quickly comment out a large chunk of text or code, you can also surround the area as follows:

```
/*    <---- slash-asterisk to open comment

   whatever you want goes in here

*/    <---- asterisk-slash to close comment
```

Example 2:

```
/*
   Class: HelloWorld
      - in control of greeting generations
      - Written August 1999 by Jason Zych
*/
public class HelloWorld
{
   public static void main(String[] args)
   {
      System.out.println("Hello World!");
   }
}
```

Example mixing the two kinds of comments:

```
/*
   Class: HelloWorld
      - in control of greeting generations
      - Written August 1999 by Jason Zych
*/
public class HelloWorld
{

   // Method: main
   //    - controls main execution of program
   public static void main(String[] args)
   {
      // line prints out a greeting
      System.out.println("Hello World!");
   }

}
```

New programmers sometimes take this to an extreme and place a comment on every single line, even the lines whose purpose is obvious. For example:

```
// adds 1 to the number of CDs
numCDs = numCDs + 1;
```

Comments like this are generally a bad idea. If you put comments even at lines where the purpose of the code is clear, then the comments start to clutter up the code. Usually, the purpose of a comment is just to remark on a line whose purpose isn't immediately obvious, or to remark on the general purpose of an entire section of code. With experience, you will gain a sense of the right balance between "not enough commenting" and "too much commenting".

Remember – if you choose descriptive variable names, then often your code becomes mostly "self-documenting" – i.e. fewer comments are necessary because, due to your choice of variable names, the code's purpose is mostly clear at first glance. Any time a variable name helps to document the variable's purpose, that's another comment you might not have to bother writing.

So, when you write your MPs – and when you write other code as well – it will be important to keep in mind the elements of good coding style:

1. Indentation
2. Descriptive variable names
3. Comments where needed

Part of your grade will depend on writing your code in a good style.