2021/2022(1)
IF184301 Object Oriented Programming

Lecture #3c

Types, Conditionals & Loops

Misbakhul Munir IRFAN SUBAKTI
司馬伊凡
Мисбакхул Мунир Ирфан Субакти

Primitive number types

- There are two kinds of numbers that we need to work with: integers, and decimals.
- In Java, we have multiple primitive types to deal with each case.

Integers

- For integers, we have the `int` type, which we've all seen already.
- We also have three other integer types: `byte`, `short`, and `long`.
- Each has different limits on what they can store
  - `byte` can store numbers from -128 to 127
  - `short` from -32768 to 32767
  - `int` from -2147483648 to 2147483647
  - `long` from -9223372036854775808 to 9223372036854775807. Big numbers!
- The different types have different memory requirements, but for most cases, the difference isn't noticeable, so typically we'll use either `int` or `long` in our programs.
- Most Java methods will return one of these two types as well, so get used to them! The smaller types are more useful in cases where memory is at a real premium, for example, when writing code for a mobile phone.

Integer overflow

- Programmers need to be careful when using large numbers approaching the limits of their chosen type. It's important to note that these limits apply at every stage of the computation - for example, consider the following code.

```
int x = 100000; // 100,000
int y = 200000; // 200,000
int z = 5000;     // 5,000
System.out.println(x * y / z ); // 20,000,000,000 / 5,000 = 4,000,000 isn't it?
```

- The result of the calculation should be `4 million`, right? But we will find that it isn't the answer that gets printed. Put it into a main method and see what result you get!
  - `-294967`
- The issue is that the product of `x` and `y` is above the limits for `int`, and so even though none of the given variables overstep the bounds, it does do so midway through the calculation, causing what is known as an *integer overflow*.
- The solution to this problem is to be careful when approaching the bounds of a type, and declare variables as a different type (e.g. `long`) if necessary.

Floating point numbers

- Decimals are incredibly useful in almost all aspects of life, so we need a way to represent them to a computer.
- This is where floating point numbers come in - they are used in Java (and other languages!) to represent decimal values.
- We can find two types of floating point number in Java - `double` and `float`.
- `float` is a lower precision number, with lower memory usage, and `double` is twice the precision, and a higher memory requirement.
- Again, we'll generally find `double` more useful as most Java methods will return it, so unless we have a compelling reason to use `float`, `double` should be our first choice.

Computer cannot count! 

- Computers, by their nature, are generally limited to working with integers.
- In order to overcome this, floating point numbers use integers to approximate the decimal values they represent.
- Most of the time we won't notice a difference, but there are some kinks that may trip us up.
- For example, try the following to add 0.1 together ten times.

```
System.out.println(0.1f + 0.1f + 0.1f + 0.1f + 0.1f + 0.1f + 0.1f + 0.1f + 0.1f + 0.1f) ; // floats
System.out.println(0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1); // doubles
```

- What answer should we get? What answer do we actually get?
  - `1 & 1. 1.0000001 & 0.9999999999999999.`
- You should find that although the answer ought to be 1, neither statement prints it correctly.
- Instead, they both include a slight rounding error.
- This is due to the approximation at work - the main upshot of this is that we should never compare two floating point numbers directly, i.e. `if (a == b).`
- The approximations mean that although the numbers should be the same, they sometimes won't be.
- Instead, we need to check if `a - b` is less than, say, `0.00000001`, depending on the numbers involved.
- Note also that usual laws of arithmetic are not necessarily precisely true. For instance, check the results of `0.1 + 0.2 + 0.3` and `0.2 + 0.3 + 0.1`.

Fixing the integer division problem

- By now, many of us will have encountered the integer division problem.
- When dividing two integer types, the result is treated by Java as another integer type, which results in loss of the fractional part of the number, i.e. `1/16` becomes `0` instead of `0.0625`.
- The simple solution is to force one value to be a double by adding `.0` to the end of it, e.g. `1/16.0`. This will return the result as a `double` instead of an `integer`.
- For variables instead of constants, we can't do this however, so we need to use what is known as casting to convert a variable to a different type:

```
int x = 5;
int y = 6;
System.out.println((double) x / y); // 0.8333333333333334 without (double) = 0
```

- The `(double)` before the variable `x` (the brackets are necessary) tell Java to treat this variable as a `double` for the time being. We can also use this trick to avoid integer overflows, by casting one of our integers to a larger type such as `long`.

Other types

- Java provides two other primitive types: `char` & `boolean`.
- `char` represents a single character, be it letter, number, or punctuation.
- `boolean` stores either true or false, nothing else.
- `boolean` in particular is necessary to allow us to work with.

Conditionals & repetition

- Often when programming, we only want to do something when a certain condition is true.
- That is to say, “if something is true, then do this, else do that”.
- In Java we can use the `if` statement to do that.

if

```
int i=0;
int j=3;
if (i > 0) {
          System.out.println("i is greater than 0!");
          System.out.println("j is " + (j - 1));
} else {
          System.out.println("i is less than or equal to 0");
          System.out.println("j is " + j);
}
```

- What do you think will be printed in the example above? What would be printed if `i` was `1`? Note that we can continue adding `else`s indefinitely.

```
if (i > 5) {
. . .
} else if (i > 4) {
. . .
} else if(i > 3) {
. . .
} else {
. . .
}
```

switch

- If statements are useful for considering a few different cases, but they can get quite convoluted when we have a lot of different circumstances to consider. This is where the `switch` statement comes in. It allows us to use a variable of specific types (`byte`, `short`, `int`, and `char`) and react differently according to the value the variable takes.

```
char input = 'b';
switch (input) {
        case 'a': {
                System.out.println("Input was a");
        }
        case 'b': {
                System.out.println("Input was b");
        }
        case 'c': {
                System.out.println("Input was c");
        }
}
```

`switch` (continued)

- This code snippet should tell us what value input was, but if we run it, we'll find it prints the statements for both `b` and `c`. This is because we need to use the `break` keyword - this is important in switch statements as without it, it will continue to do everything following the case it matched. We add `break` to the end of each case, like so.

```
char input = 'b';
switch (input) {
        case 'a': {
                System.out.println("Input was a");
                break;
        }
        case 'b': {
                System.out.println("Input was b");
                break ;
        }
        case 'c': {
                System.out.println("Input was c");
                break ;
        }
}
```

`switch` (continued)

- This will print only one value - but what if the input is a different letter, like `z`? In the case above, it will simply do nothing. Here we can use the `default` keyword to give a catch-all command. For the above example, try adding the following before the final bracket in the example above:

```
default: {
        System.out.println("Unexpected letter");
}
```

- Then try any letter that isn't in the list above; we should find it prints the `Unexpected letter` statement. We don't need to include the `break` keyword in the `default` case as it is always the last to operate.
- Often, we'll not only want to test if something is true, but also to do something multiple times. There are a few ways to do that in Java.

`while` and `for` loops

- Sometimes we might want to do something several times.

```
System.out.println("Hello") ;
System.out.println("Hello") ;
System.out.println("Hello") ;
System.out.println("Hello") ;
```

- We can see why that's a bit annoying. It's better to be able to say "do this, n times". For that, we can use a loop. The two main types are `for` and `while`.

```
int i = 0;
while (i < 4) {
        System.out.println("Hello") ;
}
for (int j = 0; j < 4; j++) {
        System.out.println("Hello") ;
}
```

- Both of the above should print `Hello` four times. One of them isn't right—which? Why?

`while` and `for` loops

- The key difference between `for`- and `while`-loops is that with a `for` loop, it's clear exactly how many times we'll be doing something—everything is set up before the loop starts, and it's hard to go wrong. In a `while` loop, the content of the round brackets is a *boolean* condition (either `true` or `false`). If we don't do something to make that condition (eventually) `false`, the loop will never terminate.
- Exercise: How would we make a `for` loop that printed the numbers:
  - 1–10, in ascending order?
  - 1–10, in descending order?
  - 3, 6, 9, 12, 15?
  - 1, 4, 9, 16, 25, . . . , 100 (i.e., square numbers up to 10^2)?