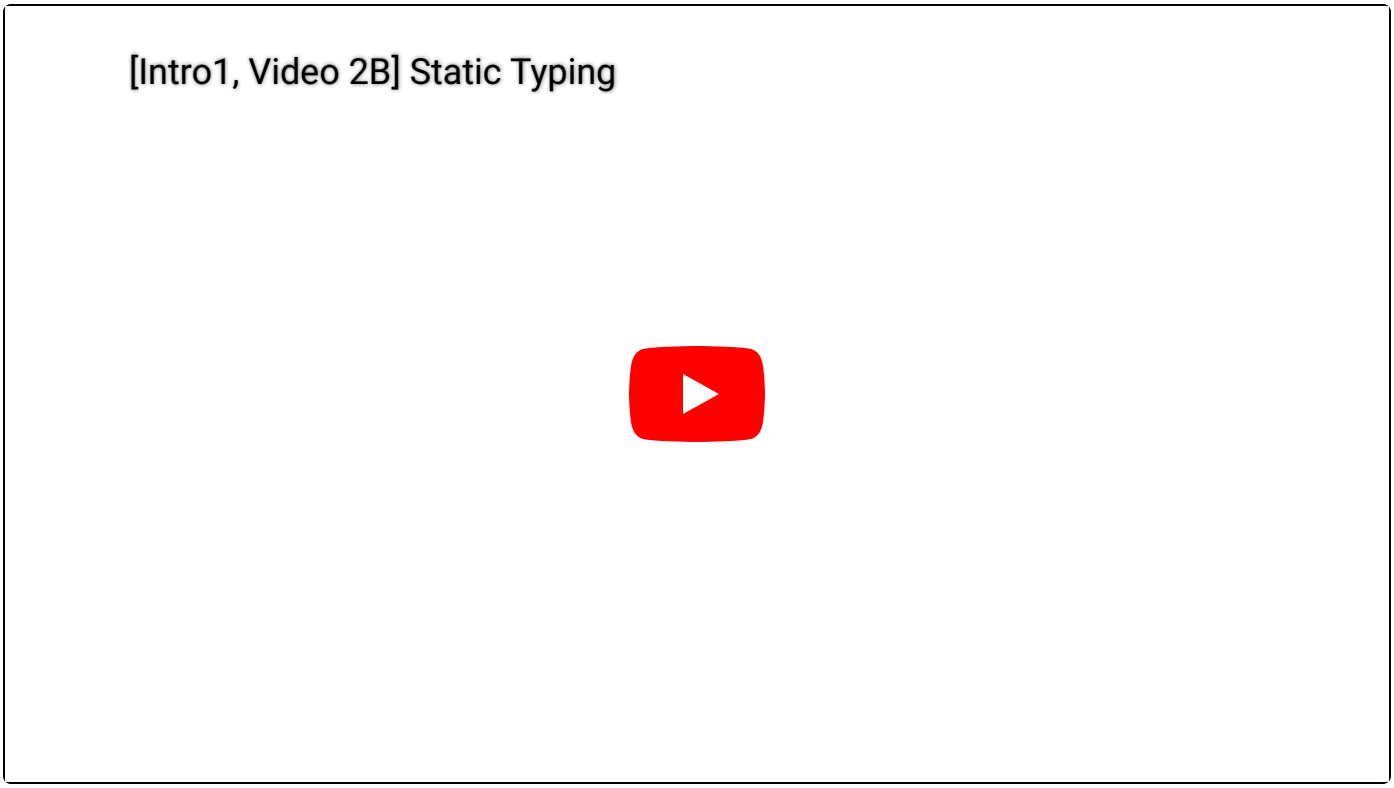
1.3 Basic Java Features

Variables and Loops



The program below will print out the integers from 0 through 9.

```
public class HelloNumbers {
    public static void main(String[] args) {
        int x = 0;
        while (x < 10) {
            System.out.print(x + " ");
            x = x + 1;
        }
    }
}
```

When we run this program, we see:

```
$ javac HelloNumbers.java
$ java HelloNumbers
$ 0 1 2 3 4 5 6 7 8 9
```

Some interesting features of this program that might jump out at you:

- Our variable x must be declared before it is used, *and it must be given a type!*
- Our loop definition is contained inside of curly braces, and the boolean expression that is tested is contained inside of parentheses.
- Our print statement is just `System.out.print` instead of `System.out.println`. This means we should not include a newline (a return).
- Our print statement adds a number to a space. This makes sure the numbers don't run into each other. Try removing the space to see what happens.
- When we run it, our prompt ends up on the same line as the numbers (which you can fix in the following exercise if you'd like).

Of these features the most important one is the fact that variables have a declared type. We'll come back to this in a bit, but first, an exercise.

Exercise 1.1.2. Modify `HelloNumbers` so that it prints out the cumulative sum of the integers from 0 to 9. For example, your output should start with 0 1 3 6 10... and should end with 45.

Also, if you've got an aesthetic itch, modify the program so that it prints out a new line at the end.

The program below will print out the integers from 0 through 9.

```
public class HelloNumbers {
    public static void main(String[] args) {
        int x = 0;
        while (x < 10) {
            System.out.print(x + " ");
            x = x + 1;
        }
    }
}
```

When we run this program, we see:

```
$ javac HelloNumbers.java
$ java HelloNumbers
$ 0 1 2 3 4 5 6 7 8 9
```

Some interesting features of this program that might jump out at you:

- Our variable x must be declared before it is used, *and it must be given a type!*
- Our loop definition is contained inside of curly braces, and the boolean expression that is tested is contained inside of parentheses.
- Our print statement is just `System.out.print` instead of `System.out.println`. This means we should not include a newline (a return).
- Our print statement adds a number to a space. This makes sure the numbers don't run into each other. Try removing the space to see what happens.
- When we run it, our prompt ends up on the same line as the numbers (which you can fix in the following exercise if you'd like).

Of these features the most important one is the fact that variables have a declared type. We'll come back to this in a bit, but first, an exercise.

Exercise 1.1.2. Modify `HelloNumbers` so that it prints out the cumulative sum of the integers from 0 to 9. For example, your output should start with 0 1 3 6 10... and should end with 45.

Also, if you've got an aesthetic itch, modify the program so that it prints out a new line at the end.

Static Typing

Java is a **statically typed language**, which means that all variables, parameters, and methods must have a declared type. After declaration, *the type can never change*. Expressions also have an implicit type; for example, the expression `3 + 5` has type `int`.

Because all types are declared statically, the compiler checks that types are compatible before the program even runs. This means that expressions with an incompatible type will fail to compile instead of crashing the program at runtime.

The advantages of static typing include:

-

- catching type errors earlier in the coding process, reducing the debugging burden on the programmer.
- avoiding type errors for end users.
- making it easier to read and reason about code.
- avoiding expensive runtime type checks, making code more efficient.

However, static typing also has several disadvantages; namely:

- more verbose code.
- less generalizable code.

One of the most important features of Java is that all variables and expressions have a so-called `static type`. Java variables can contain values of that type, and only that type. Furthermore, the type of a variable can never change.

One of the key features of the Java compiler is that it performs a static type check. For example, suppose we have the program below:

```
public class HelloNumbers {
    public static void main(String[] args) {
        int x = 0;
        while (x < 10) {
            System.out.print(x + " ");
            x = x + 1;
        }
        x = "horse";
    }
}
```

Compiling this program, we see:

```
$ javac HelloNumbers.java
HelloNumbers.java:9: error: incompatible types: String cannot be converted to in
        x = "horse";
                ^
1 error
```

The compiler rejects this program out of hand before it even runs. This is a big deal, because it means that there's no chance that somebody running this program out in the world will ever run into a type error!

This is in contrast to dynamically typed languages like Python, where users can run into type errors during execution!

In addition to providing additional error checking, static types also let the programmer know exactly what sort of object he or she is working with. We'll see just how important this is in the coming weeks. This is one of my personal favorite Java features.

To summarize, static typing has the following advantages:

- The compiler ensures that all types are compatible, making it easier for the programmer to debug their code.
- Since the code is guaranteed to be free of type errors, users of your compiled programs will never run into type errors. For example, Android apps are written in Java, and are typically distributed only as .class files, i.e. in a compiled format. As a result, such applications should never crash due to a type error since they have already been checked by the compiler.
- Every variable, parameter, and function has a declared type, making it easier for a programmer to understand and reason about code.

However, static typing also has several disadvantages, which will be discussed further in later chapters. To name a few:

- More verbose code.
- Less generalizable code.

Extra Thought Exercise

In Java, we can say `System.out.println(5 + " ");`. But in Python, we can't say `print(5 + "horse")`, like we saw above. Why is that so?

Consider these two Java statements:

```
String h = 5 + "horse";
```

and

```
int h = 5 + "horse";
```

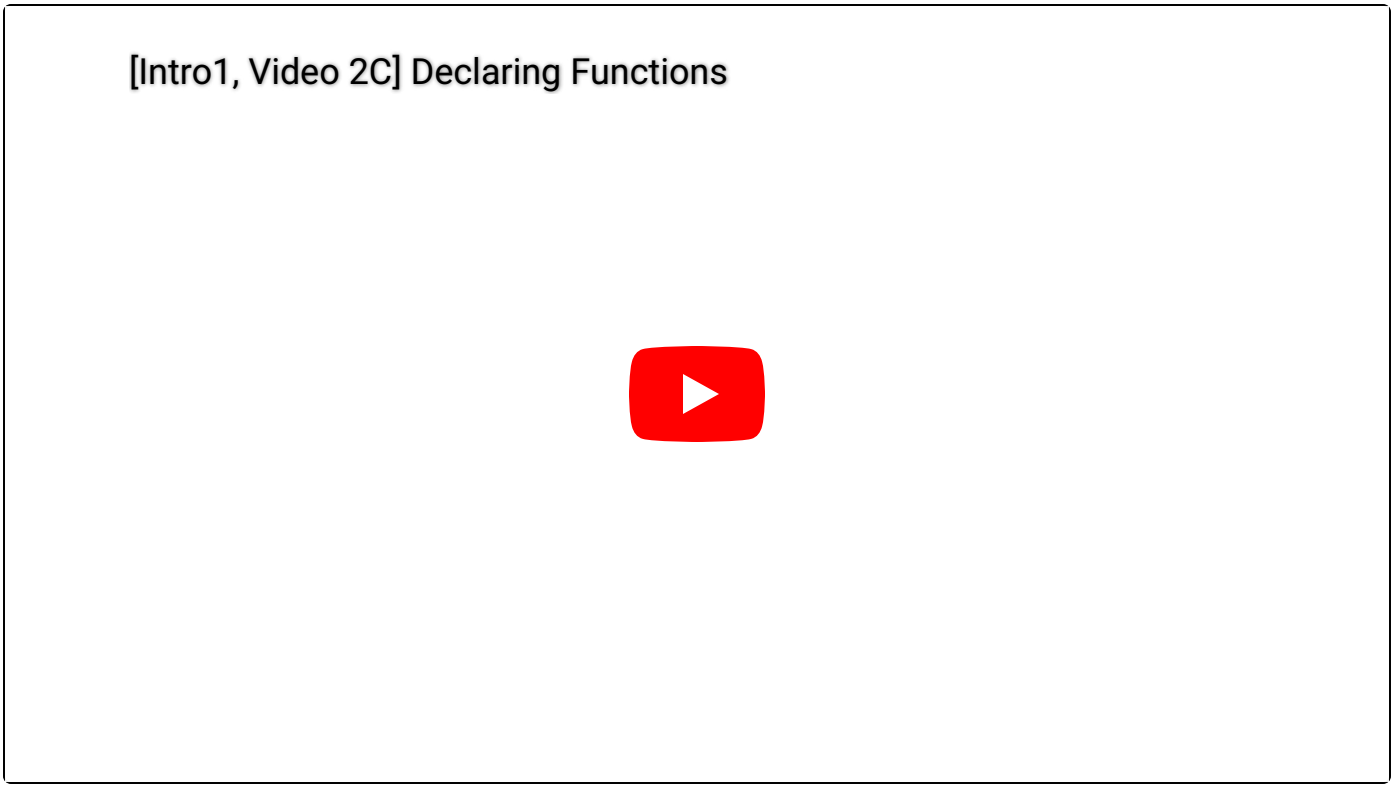
The first one of these will succeed; the second will give a compiler error. Since Java is strongly typed, if you tell it `h` is a string, it can concatenate the elements and give you a string. But when `h` is an `int`, it can't concatenate a number and a string and give you a number.

Python doesn't constrain the type, and it can't make an assumption for what type you want. Is `x = 5 + "horse"` supposed to be a number? A string? Python doesn't know. So it errors.

In this case, `System.out.println(5 + "horse");`, Java interprets the arguments as a string concatentation, and prints out "5horse" as your result. Or, more usefully, `System.out.println(5 + " ");` will print a space after your "5".

What does `System.out.println(5 + "10");` print? 510, or 15? How about `System.out.println(5 + 10);`?

Defining Functions in Java



In languages like Python, functions can be declared anywhere, even outside of functions. For example, the code below declares a function that returns the larger of two arguments, and then uses this function to compute and print the larger of the numbers 8 and 10:

```
def larger(x, y):
    if x > y:
        return x
    return y

print(larger(8, 10))
```

Since all Java code is part of a class, we must define functions so that they belong to some class. Functions that are part of a class are commonly called "methods". We will use the terms interchangably throughout the course. The equivalent Java program to the code above is as follows:

```
public class LargerDemo {
    public static int larger(int x, int y) {
        if (x > y) {
            return x;
        }
        return y;
    }

    public static void main(String[] args) {
        System.out.println(larger(8, 10));
    }
}
```

The new piece of syntax here is that we declared our method using the keywords `public static`, which is a very rough analog of Python's `def` keyword. We will see alternate ways to declare methods in the next chapter.

The Java code given here certainly seems much more verbose! You might think that this sort of programming language will slow you down, and indeed it will, in the short term. Think of all of this stuff as safety equipment that we don't yet understand. When we're building small programs, it all seems superfluous. However, when we get to building large programs, we'll grow to appreciate all of the added complexity.

As an analogy, programming in Python can be a bit like Dan Osman free-soloing Lover's Leap. It can be very fast, but dangerous. Java, by contrast is more like using ropes, helmets, etc. as in this video.

Code Style, Comments, Javadoc

Code can be beautiful in many ways. It can be concise. It can be clever. It can be efficient. One of the least appreciated aspects of code by novices is code style. When you program as a novice, you are often single mindedly intent on getting it to work, without regard to ever looking at it again or having to maintain it over a long period of time.

In this course, we'll work hard to try to keep our code readable. Some of the most important features of good coding style are:

- Consistent style (spacing, variable naming, brace style, etc)
- Size (lines that are not too wide, source files that are not too long)
- Descriptive naming (variables, functions, classes), e.g. variables or functions with names like `year` or `getUserName` instead of `x` or `f`.
- Avoidance of repetitive code: You should almost never have two significant blocks of code that are nearly identical except for a few changes.
- Comments where appropriate. Line comments in Java use the `//` delimiter. Block (a.k.a. multi-line comments) comments use `/*` and `*/`.

The golden rule is this: Write your code so that it is easy for a stranger to understand.

Here is the course's official style guide. It's worth taking a look!

Often, we are willing to incur slight performance penalties, just so that our code is simpler to grok. We will highlight examples in later chapters.

Comments

We encourage you to write code that is self-documenting, i.e. by picking variable names and function names that make it easy to know exactly what's going on. However, this is not always enough. For example, if you are implementing a complex algorithm, you may need to add comments to describe your code. Your use of comments should be judicious. Through experience and exposure to others' code, you will get a feeling for when comments are most appropriate.

One special note is that all of your methods and almost all of your classes should be described in a comment using the so-called Javadoc format. In a Javadoc comment, the block comment starts with an extra asterisk, e.g. `/**`, and the comment often (but not always) contains descriptive tags. We won't discuss these tags in this textbook, but see the link above for a description of how they work.

As an example without tags:

```
public class LargerDemo {
    /** Returns the larger of x and y. */
    public static int larger(int x, int y) {
        if (x > y) {
            return x;
        }
        return y;
    }

    public static void main(String[] args) {
        System.out.println(larger(8, 10));
    }
}
```

The widely used javadoc tool can be used to generate HTML descriptions of your code. We'll see examples in a later chapter.

Previous
1.2 Java Workflow

Next
1.4 Exercises

Last updated 7 months ago