Reading 1: Static Checking

Objectives

Today's class has two topics:

- static typing
- the big three properties of good software

Hailstone sequence

As a running example, we're going to explore the hailstone sequence, which is defined as follows. Starting with a number *n*, the next number in the sequence is *n/2* if *n* is even, or *3n+1* if *n* is odd. The sequence ends when it reaches 1. Here are some examples:

```
2, 1
3, 10, 5, 16, 8, 4, 2, 1
4, 2, 1
2^n, 2^(n-1) , … , 4, 2, 1
5, 16, 8, 4, 2, 1
7, 22, 11, 34, 17, 52, 26, 13, 40, .
```

Because of the odd-number rule, the sequence may bounce up and down before decreasing to 1. It's conjectured that all hailstones eventually fall to the ground – i.e., the hailstone sequence reaches 1 for all starting *n* – but that's still an open question. Why is it called a hailstone sequence? Because hailstones form in clouds by bouncing up and down, until they eventually build enough weight to fall to earth.

Computing hailstones

Here's some code for computing and printing the hailstone sequence for some starting *n*. We'll write Java and Python side by side for comparison:

```java
// Java
int n = 3;
while (n != 1) {
    System.out.pr
intln(n);
    if (n % 2 ==
0) {
        n = n /
2;
    } else {
        n = 3 * n
+ 1;
    }
}
System.out.printl
n(n);
```

```python
# Python
n = 3
while n != 1:
    print(n)
    if n % 2 =
= 0:
        n = n
/ 2
    else:
        n = 3
* n + 1


print(n)
```

A few things are worth noting here:

- The basic semantics of expressions and statements in Java are very similar to Python: `while` and `if` behave the same, for example.
- Java requires semicolons at the ends of statements.
- Java requires parentheses around the conditions of the `if` and `while`.
- Java uses curly braces around blocks, instead of indentation. You should always indent the block, even though Java won't

pay any attention to your extra spaces. Programming is a form of communication, and you're communicating not only to the compiler, but to human beings. Humans need that indentation. We'll come back to this later.

Types

The most important semantic difference between the Python and Java code above is the declaration of the variable `n`, which specifies its type: `int`.

A **type** is a set of values, along with operations that can be performed on those values.

Java has several **primitive types**, among them:

- `int` (for integers like 5 and -200, but limited to the range about ± 2^{31}, or roughly ± 2 billion)
- `long` (for larger integers up to about ± 2^{63})
- `boolean` (for true or false)
- `double` (for floating-point numbers, which represent a subset of the real numbers)
- `char` (for single characters like `'A'` and `'$'`)

Java also has **object types**, for example:

- `String` represents a sequence of characters, like a Python string.
- `BigInteger` represents an integer of arbitrary size, so it acts like a Python integer.

By Java convention, primitive types are lowercase, while object types start with a capital letter.

Operations are functions that take inputs and produce outputs (and sometimes change the values themselves). The syntax for operations varies, but we still think of them as functions no matter how they're written. Here are three different syntaxes for an operation in Python or Java:

- *As an operator.* For example, `a + b` invokes the operation `+ : int × int → int`.
  (In this notation: `+` is the name of the operation, `int × int` before the arrow describes the two inputs, and `int` after the arrow describes the output.)
- *As a method of an object.* For example, `bigint1.add(bigint2)` calls the operation `add: BigInteger × BigInteger → BigInteger`.
- *As a function.* For example, `Math.sin(theta)` calls the operation `sin: double → double`. Here, `Math` is not an object. It's the class that contains the `sin` function.

Contrast Java's `str.length()` with Python's `len(str)`. It's the same operation in both languages – a function that takes a string and returns its length – but it just uses different syntax.

Some operations are **overloaded** in the sense that the same operation name is used for different types. The arithmetic operators `+`, `-`, `*`, `/` are heavily overloaded for the numeric primitive types in Java. Methods can also be overloaded. Most programming languages have some degree of overloading.

Static typing

Java is a **statically-typed language**. The types of all variables are known at compile time (before the program runs), and the compiler can therefore deduce the types of all expressions as well. If `a` and `b` are declared as `int`, then the compiler concludes that `a+b` is also an `int`. The Eclipse environment does this while you're writing the code, in fact, so you find out about many errors while you're still typing.

In **dynamically-typed languages** like Python, this kind of checking is deferred until runtime (while the program is running).

Static typing is a particular kind of **static checking**, which means checking for bugs at compile time. Bugs are the bane of programming. Many of the ideas in this course are aimed at eliminating bugs from your code, and static checking is the first idea that we've seen for this. Static typing prevents a large class of bugs from infecting your program: to be precise, bugs caused by applying an operation to the wrong types of arguments. If you write a broken line of code like:

```
"5" * "6"
```

that tries to multiply two strings, then static typing will catch this error while you're still programming, rather than waiting until the line is reached during execution.

Support for static typing in dynamically-typed languages

Even though Python is dynamically-typed, Python 3.5 and later allow you to declare type hints in the code, e.g.:

```
# Python function declared with ty
pe hints
def hello(name:str)->str:
    return 'Hi, ' + name
```

The declared types can be used by a checker like Mypy to find type errors statically without having to run the code.

Other dynamically-typed languages have similar extensions. For example, JavaScript has been extended with static typing to create the language TypeScript, e.g.:

```
// TypeScript function
function hello(name:string):string
{
    return 'Hi, ' + name;
}
```

The addition of static types to these dynamically-typed languages reflects a widespread belief among software engineers that the use of static types is essential to building and maintaining a large software system. The rest of this reading, and in fact this entire course, will show reasons for this belief. Compared to a language like Java that is statically-typed from the outset, adding static types to a dynamically-typed language enables a programming approach called gradual typing, in which some parts of the code have static type declarations and other parts omit them. Gradual typing can provide a smoother path for a small experimental prototype to grow into a large, stable, maintainable system.

Static checking, dynamic checking, no checking

It's useful to think about three kinds of automatic checking that a language can provide:

- **Static checking**: the bug is found automatically before the program even runs.
- **Dynamic checking**: the bug is found automatically when the code is executed.
- **No checking**: the language doesn't help you find the error at all. You have to watch for it yourself, or end up with wrong answers.

Needless to say, catching a bug statically is better than catching it dynamically, and catching it dynamically is better than not catching it at all.

Here are some rules of thumb for what errors you can expect to be caught at each of these times.

Static checking can catch:

- syntax errors, like extra punctuation or spurious words. Even dynamically-typed languages like Python do this kind of static checking. If you have an indentation error in your Python program, you'll find out before the program starts running.
- misspelled names, like `Math.sine(2)`. (The correct spelling is `sin`.)
- wrong number of arguments, like `Math.sin(30, 20)`.

- wrong argument types, like `Math.sin("30")`.
- wrong return types, like `return "30";` from a function that's declared to return an `int`.

Dynamic checking can catch:

- illegal argument values. For example, the integer expression `x/y` is only erroneous when `y` is actually zero; for other values of `y`, its value is well-defined. So in this expression, divide-by-zero is not a static error, but a dynamic error.
- illegal conversions, i.e., when the specific value can't be converted to or represented in the target type. For example, `Integer.valueOf("hello")` is a dynamic error, because the string `"hello"` cannot be parsed as a decimal integer. So is `Integer.valueOf("8000000000")`, because 8 billion is outside the legal range of `int` values in Java.
- out-of-range indices, e.g., using a negative or too-large index on a string.
- calling a method on a `null` object reference (`null` is like Python's `None`).

Static checking can detect errors related to the *type* of a variable – the set of values it is allowed to have – but is generally unable to find errors related to a specific value from that type. Static typing guarantees that a variable will have *some* value from its type, but we don't know until runtime exactly which value it has. So if the error would be caused only by certain values, like divide-by-zero or index-out-of-range, then the compiler won't raise a static error about it.

Dynamic checking, by contrast, tends to be about errors caused by specific values.

Surprise: primitive types are not true numbers

One trap in Java – and many other programming languages – is that its primitive numeric types have corner cases that do not behave like the integers and real numbers we're used to. As a result, some errors that really should be dynamically checked are not checked at all. Here are the traps:

- **Integer division**. `5/2` does not return a fraction, it returns a truncated integer. So this is an example of where what we might have hoped would be a dynamic error (because a fraction isn't representable as an integer) frequently produces the wrong answer instead.
- **Integer overflow**. The `int` and `long` types are actually finite sets of integers, with maximum and minimum values. What happens when you do a computation whose answer is too positive or too negative to fit in that finite range? The computation quietly *overflows* (wraps around), and returns an integer from somewhere in the legal range but not the right answer.

- **Special values in floating-point types**. Floating-point types like `double` have several special values that aren't real numbers: `NaN` (which stands for "Not a Number"), `POSITIVE_INFINITY`, and `NEGATIVE_INFINITY`. So when you apply certain operations to a `double` that you'd expect to produce dynamic errors, like dividing by zero or taking the square root of a negative number, you will get one of these special values instead. If you keep computing with it, you'll end up with a bad final answer.

READING EXERCISES

Let's try some examples of buggy code and see how they behave in Java. Are these bugs caught statically, dynamically, or not at all?

1

```
int n = -5;
if (n) {
  System.out.println("n is a positive integer");
}
```

✓ [● static error] ✓
○ dynamic error
○ no error, wrong answer

This is a static type error in Java, because the `if` statement requires an expression of `boolean` type, but `n` has `int` type. Changing the code to `if (n > 0) ...` would fix both the type error and the bug.

CHECK EXPLAIN

2

```
int big = 200000; // 200,000
big = big * big;  // big shou
ld be 40 billion now
```

✓
- static error
- dynamic error
- [✓ no error, wrong answer] ✓

> This is an integer overflow, because an `int` value can't represent a number bigger than 2^{31} (about 2 billion). It isn't caught statically, but unfortunately in Java it isn't caught dynamically either. Integer overflows quietly produce the wrong answer.
>
> Some newer programming languages, like Swift, have rethought this design decision and produce a dynamic error instead. Others, like Python and Ruby, avoid the problem by automatically using unlimited-precision integers where necessary.

CHECK EXPLAIN

3

```
double probability = 1/5;
```

✓ static error
dynamic error
[✓ no error, wrong answer] ✓

> If the programmer's intent was to get 0.2, this is using the wrong operation. `/` is overloaded for both integer division and floating point division. But because it's being called with integers, `1` and `5`, Java chooses integer division, and it quietly truncates the fraction and produces the wrong answer: 0.

CHECK EXPLAIN

4

```
int sum = 0;
int n = 0;
int average = sum/n;
```

✓ static error
[✓ dynamic error] ✓
no error, wrong answer

> Division by zero can't produce an integer, so it produces a dynamic error instead.

CHECK EXPLAIN

5

```
double sum = 7;
double n = 0;
double average = sum/n;
```

✓ ○ static error
○ dynamic error
[✓ no error, wrong answer] ✓

Division by zero can't produce a real number either – but unlike real numbers, `double` has a special value for `POSITIVE_INFINITY`, so that's what it returns when you divide a positive integer by zero. If the code is trying to compute an average value, infinity is unlikely to be a correct or useful answer.

CHECK EXPLAIN

Note that these readings don't keep track of the exercises you've previously done. When you reload the page, all exercises reset themselves.

If you're a registered student, you can see which exercises you've already done by looking at **Omnivore**.

Arrays and collections

Let's change our hailstone computation so that it stores the sequence in a data structure, instead of just printing it out. Java has two kinds of list-like types that we could use: arrays and Lists.

Arrays are fixed-length sequences of another type, like integers. For example, here's how to declare an array variable and construct an array value to assign to it:

```
int[] a = new int[100];
```

The `int[]` array type includes all possible array values, but a particular array value, once created, can never change its length. Operations on array types include:

- indexing: `a[2]`
- assignment: `a[2] = 0`
- length: `a.length`

Note that `a.length` is different syntax from `String.length()`. Because `a.length` is an instance variable, rather than a method call, you don't put parentheses after it.

Here's a crack at the hailstone code using an array. We start by constructing the array, and then use an index variable `i` to step through the array, storing values of the sequence as we generate them.

```
int[] a = new int[100];  // <=====
DANGER, WILL ROBINSON!
int i = 0;
int n = 3;
while (n != 1) {
    a[i] = n;
    i++;  // very common shorthand
for i=i+1
    if (n % 2 == 0) {
        n = n / 2;
    } else {
        n = 3 * n + 1;
    }
}
a[i] = n;
i++;
```

Something should immediately smell wrong in this approach. What's that magic number 100? What would happen if we tried an `n` that turned out to have a very long hailstone sequence? It wouldn't fit in a length-100 array. We have a bug. Would Java catch the bug statically, dynamically, or not at all? Incidentally, this kind of bug is called a buffer overflow, because it is overflowing a fixed-length array. Fixed-length arrays are commonly used in less-safe languages like C and C++ that don't do automatic runtime checking of array accesses. Buffer overflows have been responsible for a large number of network security breaches and internet worms.

Instead of a fixed-length array, let's use the `List` type. Lists are variable-length sequences of another type. Here's how we can declare a

`List` variable and make a list value:

```
List<Integer> list = new ArrayList
<Integer>();
```

And here are some of its operations:

- indexing: `list.get(2)`
- assignment: `list.set(2, 0)`
- length: `list.size()`

Why `List` on the left but `ArrayList` on the right? `List` is an interface, a type that can't be constructed directly, but that instead just specifies the operations that a `List` must provide, like `get()` and `set()` and `size()`. `ArrayList` is a class, a concrete type that provides implementations of those operations. `ArrayList` isn't the only implementation of the `List` type, though; `LinkedList` is another. So we prefer the `List` type when declaring variable types or return types, because it allows the code to be more general and flexible, not caring which concrete kind of list is actually being used. We will revisit this idea of interfaces and implementation classes several times throughout the course, so it's okay if you don't understand it deeply yet.

You can see all the operations of `List` or the details of `ArrayList` or `LinkedList` in the Java API documentation: find it with a web search for "Java 15 API." Get to know the Java API docs, they're your friend. ("API" means "application programming interface," and here it refers to the methods and classes that Java provides to help you build Java applications.)

Why `List<Integer>` and not `List<int>`? Unfortunately, we can't write `List<int>`. Lists only know how to deal with object types, not primitive types. Each primitive type has an equivalent object type: e.g. `int` and `Integer`, `long` and `Long`, `float` and `Float`, `double` and `Double`. Java requires us to use these object type equivalents when we parameterize one type using another. But in other contexts, Java automatically converts between `int` and `Integer`, so we can write `Integer i = 5;` without any type error.

Here's the hailstone code written with lists:

```
List<Integer> list = new ArrayList
<Integer>();
int n = 3;
while (n != 1) {
    list.add(n);
    if (n % 2 == 0) {
        n = n / 2;
    } else {
        n = 3 * n + 1;
    }
}
list.add(n);
```

Not only simpler but safer too, because the list automatically enlarges itself to fit as many numbers as you add to it (until you run out of memory, of course).

Iterating

A `for` loop steps through the elements of an array or a `List`, just as in Python, though the syntax looks a little different. For example:

```
// find the maximum point of a hai
lstone sequence stored in list
int max = 0;
for (int x : list) {
    max = Math.max(x, max);
}
```

You can iterate through arrays as well as lists. The same code would work if the list were replaced by an array.

`Math.max()` is a handy function from the Java API. The `Math` class is full of useful functions like this – web search “Java 15 Math” to find its documentation.

Methods

In Java, statements generally have to be inside a method, and every method has to be in a class, so the simplest way to write our hailstone program looks like this:

```
public class Hailstone {
    /**
     * Compute a hailstone sequenc
e.
     * @param n  starting number f
or sequence; assumes n > 0.
     * @return hailstone sequence
starting with n and ending with 1.
     */
    public static List<Integer> ha
ilstoneSequence(int n) {
        List<Integer> list = new A
rrayList<Integer>();
        while (n != 1) {
            list.add(n);
            if (n % 2 == 0) {
                n = n / 2;
            } else {
                n = 3 * n + 1;
            }
        }
        list.add(n);
        return list;
    }
}
```

Let's explain a few of the new things here.

`public` means that any code, anywhere in your program, can refer to the class or method. Other access modifiers, like `private`, are used to get more safety in a program, and to guarantee immutability for immutable types. We'll talk more about them in an upcoming class.

`static` means that the method is a function that doesn’t take a `self` parameter (which in Java is called `this`, and is passed implicitly, so you won’t ever see it as an explicit method parameter). Static methods aren’t called on an object. Contrast that with the `List add()` method or the `String length()` method, for example, which require an object to come first. Instead, the right way to call a static method uses the class name instead of an object reference: `Hailstone.hailstoneSequence(83)`.

Take note also of the blue `/** ... */` comment before the method, because it’s very important. This comment is a specification of the method, describing the inputs and outputs of the operation. The specification should be concise, clear, and precise. The comment provides information that is not already clear from the method types. It doesn’t say, for example, that `n` is an integer, because the `int n` declaration just below already says that. But it does say that `n` must be positive, which is not captured by the type declaration but is very important for the caller to know.

We’ll have a lot more to say about how to write good specifications in a few classes, but you’ll have to start reading them and using them right away.

Mutating values vs. reassigning variables

Change is a necessary evil. But good programmers try to avoid things that change, because they may change unexpectedly. Immutability – intentionally forbidding certain things from changing at runtime – will be a major design principle in this course.

For example, an immutable type is a type whose values can never change once they have been created. The string type is immutable in both Python and Java.

Java also gives us immutable references: variables that are assigned once and never reassigned. To make a reference unreassignable, declare it with the keyword `final`:

```
final int n = 5;
```

If the Java compiler isn't convinced that your `final` variable will only be assigned once at runtime, then it will produce a compiler error. So `final` gives you static checking for unreassignable references.

It's good practice to use `final` for declaring the parameters of a method and as many local variables as possible. Like the type of the variable, these declarations are important documentation, useful to the reader of the code and statically checked by the compiler.

READING EXERCISES

final

Consider the variables in our `hailstoneSequence` method:

```
public static List<Integer> h
ailstoneSequence(int n) {
    List<Integer> list = new
ArrayList<Integer>();
    while (n != 1) {
        list.add(n);
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    list.add(n);
    return list;
}
```

Which variables can we declare `final`, because they are never reassigned in the code?

✓
- [ ] `int n`
- [✓ `List<Integer> list`] ✓

`n` is reassigned many times in the code, for example by `n = n / 2`. So we can't declare it `final`.

But `list` is only assigned once, to `new ArrayList<Integer>()`. Subsequently, even though we call `add()` on the list to add newly-discovered elements of the hailstone sequence, we never reassign the `list` *variable* to refer to a different list. So `list` can be declared `final`.

CHECK EXPLAIN

Documenting assumptions

Writing the type of a variable down documents an assumption about it: e.g., this variable will always refer to an integer. Java actually checks this assumption at compile time, and guarantees that there's no place in your program where you violated this assumption.

Declaring a variable `final` is also a form of documentation, a claim that the variable will never be reassigned after its initial assignment. Java checks that too, statically.

We documented another assumption that Java (unfortunately) doesn't check automatically: that `n` must be positive.

Why do we need to write down our assumptions? Because programming is full of them, and if we don't write them down, we won't remember them, and other people who need to read or change our programs later won't know them. They'll have to guess.

Programs have to be written with two goals in mind:

- communicating with the computer. First persuading the compiler that your program is sensible – syntactically correct and type-correct. Then getting the logic right so that it gives the right results at runtime.
- communicating with other people. Making the program easy to understand, so that when somebody has to fix it, improve it, or

adapt it in the future, they can do so.

Hacking vs. engineering

We’ve written some hacky code in this reading. Hacking is often marked by unbridled optimism:

- Bad: writing lots of code before testing any of it
- Bad: keeping all the details in your head, assuming you’ll remember them forever, instead of writing them down in your code
- Bad: assuming that bugs will be nonexistent or else easy to find and fix

But software engineering is not hacking. Engineers are pessimists:

- Good: write a little bit at a time, testing as you go. In a future class, we’ll talk about test-first programming.
- Good: document the assumptions that your code depends on
- Good: defend your code against stupidity – especially your own! Static checking helps with that.

READING EXERCISES

Consider the following simple Python function:

```
from math import sqrt
def funFactAbout(person):
  if sqrt(person.age) == int(sq
rt(person.age)):
    print("The age of " + perso
n.name + " is a perfect square:
" + str(person.age))
```

Assumptions

Which of the following are assumptions made by this code – i.e. they must be true in order for the code to run without errors?

✓
- [✓] `person` must be an object with `age` and `name` instance variables ✓
- [✓] `person` is not `None` ✓
- [✓] `person.age` must be a nonnegative number ✓
- [ ] `person.age` must be an integer
- [✓] `person.name` must be a string ✓

If `person` is not an object with the right instance variables, then the code will fail when it tries to refer to `person.age` or `person.name`.

If `person.age` is not a number, or if it's a negative number, then `sqrt()` will fail. But it doesn't necessarily need to be an integer, because `sqrt()` can handle numbers with fractional parts.

If `person.name` is not a string, then Python will complain of a type error when it tries to concatenate it with other strings. This is one difference between Python and Java

– Python insists that you use a conversion operation like `str()`, whereas Java will automatically convert any type into a `String` when you try to concatenate it with another `String`.

CHECK EXPLAIN

Checkable assumptions

If this code were written in Java instead, which of the following assumptions could be documented by type declarations and statically checked by the Java compiler? (Answer this question independently of the question above, i.e. include assumptions that the code doesn't actually depend on.)

✗
- [✓] `person` must be an object with `age` and `name` instance variables ✓
- [✓] `person` is not `null`
- [✓] `person.age` must be a nonnegative number
- [✓] `person.age` must be an integer ✓
- [✓] `person.name` must be a string ✓

› The `person` variable would be declared with some class type, perhaps called `Person`, and the

> definition of that class would have instance variables `name` and `age` declared with types `String` and `int` respectively.
>
> But we can't use a type declaration to forbid `person` from being `null`. Any object reference might be `null` in Java, just like any variable might be `None` in Python. Similarly, we can't forbid `age` from being negative using a type declaration. These assumptions would have to be documented in comments instead.

CHECK EXPLAIN

The goal of 6.031

Our primary goal in this course is learning how to produce software that is:

- **Safe from bugs**. Correctness (correct behavior right now) and defensiveness (correct behavior in the future) are required in any software we build.
- **Easy to understand**. The code has to communicate to future programmers who need to understand it and make changes in it (fixing bugs or adding new features). That future programmer might be you, months or years from now. You'll be surprised how much you forget if you don't write it down, and how much it helps your own future self to have a good design.
- **Ready for change**. Software always

changes. Some designs make it easy to make changes; others require throwing away and rewriting a lot of code.

There are other important properties of software (like performance, usability, security), and they may trade off against these three. But these are the Big Three that we care about in 6.031, and that software developers generally put foremost in the practice of building software. It's worth considering every language feature, every programming practice, every design pattern that we study in this course, and understanding how they relate to the Big Three.

READING EXERCISES

You are not logged in.

SFB

Which of the following ideas discussed in this reading help make your code more **safe from bugs**?

✓
- [✓] dynamic checking ✓
- [✓] final variables ✓
- [ ] integer overflow
- [✓] static typing ✓

Dynamic checking and static typing both catch bugs earlier than no checking at all. Final variables can prevent bugs associated with reassigning a variable that shouldn't have been reassigned. But integer overflow is unfortunately a *cause* of bugs.

CHECK EXPLAIN

ETU

Which of the following ideas discussed in this reading help make your code more **easy to understand**?

✓
- [✓] documented assumptions in comments ✓
- [✓] final variables ✓
- [ ] overloading
- [✓] static typing ✓

Documentation makes code easier to understand, whether it comes in the form of comments, final, or type declarations. But overloading (using the same operator with different meanings) can make code harder to understand.

CHECK EXPLAIN

RFC

Which of the following ideas discussed in this reading help make your code more **ready for change**?

✓
- [✓] documented assumptions in comments ✓
- [ ] fixed-length arrays
- [✓] methods ✓

> Documenting assumptions, and wrapping code into methods, make it easier for subsequent programmers to understand and change it. Fixed-length arrays, however, may mean that there are arbitrary limits in the code, which may make it harder to adapt.

CHECK EXPLAIN

Why we use Java in this course

Since you've had 6.009, we're assuming that you're comfortable with Python. So why aren't we using Python in this course? Why do we use Java in 6.031?

Safety is the first reason. Java has static checking (primarily type checking, but other kinds of static checks too, like that your code returns values from methods declared to do so). We're studying software engineering in this course, and safety from bugs is a key tenet of that approach. Java dials safety up to 11, which makes it a good language for learning about good software engineering practices. It's certainly possible to write safe code in dynamic languages like Python, but it's easier to understand what you need to do if you learn how in a safe, statically-checked language.

Ubiquity is another reason. Java is widely used in research, education, and industry. Java runs on many platforms, not just Windows/Mac/Linux. Java is popular for server-side web programming, and several other programming languages run on top of the Java virtual machine, among them Scala, Clojure, and Kotlin. Native Android programming is done in Java as well as Kotlin. Although some programming languages are better suited to teaching programming (notably Scheme and ML), regrettably these languages aren't as widespread in the real world. Java on your resume will be recognized as a marketable skill. But don't get us wrong: the real skills you'll get from this course are not Java-specific, but carry over to any language that you might program in. The most important lessons from this course will survive language fads: safety, clarity, abstraction, engineering instincts.

In any case, a good programmer must be **multilingual**. Programming languages are tools, and you have to use the right tool for the job. You will certainly have to pick up other programming languages before you even finish your MIT career (JavaScript, C/C++, Scheme or Ruby or ML or Haskell), so we're getting started now by learning a second one.

As a result of its ubiquity, Java has a wide array of interesting and useful **libraries** (both its enormous built-in library, and other libraries out on the net), and excellent free **tools** for development (IDEs like Eclipse, editors, compilers, test frameworks, profilers, code coverage, style checkers). Even Python is still behind Java in the richness of its ecosystem.

There are some reasons to regret using Java. It's wordy, which makes it hard to write examples on the board. It's large, having accumulated many features over the years. It’s internally inconsistent (e.g. the `final` keyword means different things in different contexts, and the `static` keyword in Java has nothing to do with static checking). It’s weighted with the baggage of older languages like C/C++ (the primitive types and the `switch` statement are good examples).

But on the whole, Java is a reasonable choice of language right now to learn how to write code that is safe from bugs, easy to understand, and ready for change. And that’s our goal.

Summary

The main idea we introduced today is **static checking**. Here’s how this idea relates to the goals of the course:

- **Safe from bugs.** Static checking helps with safety by catching type errors and other bugs before runtime.
- **Easy to understand.** It helps with understanding, because types are explicitly stated in the code.
- **Ready for change.** Static checking makes it easier to change your code by identifying other places that need to change in tandem. For example, when you change the name or type of a variable, the compiler immediately displays errors at all the places where that variable is used, reminding you to update them as well.

More practice

If you would like to get more practice with the concepts covered in this reading, you can visit the question bank. The questions in this bank were written in previous semesters by students and staff, and are provided for review purposes only – doing them will not affect your classwork grades.