- Reading 1: Static Checking
- Hailstone Sequence
- Computing Hailstones
- Types
- Static Typing
- Static Checking, Dynamic Checking, No Checking
- Surprise: Primitive Types Are Not True Numbers
- Arrays and Collections
- Iterating
- Methods
- Mutating Values vs. Reassigning Variables
- Documenting Assumptions
- Hacking vs. Engineering
- The Goal of 6.005
- Why we use Java in this course
- Summary

Reading 1: Static Checking

Objectives for Today's Class

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
2ⁿ, 2ⁿ⁻¹ , ... , 4, 2, 1
5, 16, 8, 4, 2, 1
7, 22, 11, 34, 17, 52, 6, 13, 40, ...? (where does this stop?)
```

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                       # Python
int n = 3;                    n = 3
while (n != 1) {              while n != 1:
    System.out.println(n);        print n
    if (n % 2 == 0) {             if n % 2 == 0:
        n = n / 2;                    n = n / 2
    } else {                      else:
        n = 3 * n + 1;                n = 3 * n + 1
    }
}
System.out.println(n);        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. The extra punctuation can be a pain, but it also gives you more freedom in how you organize your code – you can split a statement into multiple lines for more readability.
- Java requires parentheses around the conditions of the `if` and `while` .
- Java requires 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 ± 2^31, or roughly ± 2 billion)
- `long` (for larger integers up to ± 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 infix, prefix, or postfix operator.* For example, `a + b` invokes the operation `+ : int × int → int` .
- *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` s, 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.

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.
- wrong names, like `Math.sine(2)` . (The right name 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; otherwise it works. So in this expression, divide-by-zero is not a static error, but a dynamic error.
- unrepresentable return values, i.e., when the specific return value can't be represented in the type.
- out-of-range indexes, e.g., using a negative or too-large index on a string.
- calling a method on a `null` object reference (`null` is like Python `None`).

Static checking tends to be about types, errors that are independent of the specific value that a variable has. A type is a set of values. Static typing guarantees that a variable will have *some* value from that set, 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 `float` and `doubles`** . The `float` and `double` types have several special values that aren't real numbers: `NaN` (which stands for "Not a Number"), `POSITIVE_INFINITY` , and `NEGATIVE_INFINITY` . So operations that you'd expect to produce dynamic errors, like dividing by zero or taking the square root of a negative number, produce one of these special values instead. If you keep computing with it, you'll end up with a bad final answer.

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 T. 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 this is different syntax from `String.length()` – `a.length` is not a method call, so 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, bugs like these – overflowing a fixed-length array, which are commonly used in less-safe languages like C and C++ that don't do automatic runtime checking of array accesses – 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 `T`. 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()`

Note that `List` is an interface, a type that can't be constructed directly with new, but that instead specifies the operations that a List must provide. We'll talk about this notion in a future class on abstract data types. `ArrayList` is a class, a concrete type that provides implementations of those operations. `ArrayList` isn't the only implementation of the List type, though it's the most commonly used one. `LinkedList` is another. Check them out in the Java API documentation, which you can find by searching the web for "Java 8 API". Get to know the Java API docs, they're your friend. ("API" means "application programmer interface," and is commonly used as a synonym for "library.")

Note also that we wrote `List<Integer>` instead of `List<int>`. Unfortunately we can't write `List<int>` in direct analog to `int[]`. Lists only know how to deal with object types, not primitive types. In Java, each of the primitive types (which are written in lowercase and often abbreviated, like `int`) has an equivalent object type (which is capitalized, and fully spelled out, like `Integer`). Java requires us to use these object type equivalents when we parameterize a type with angle brackets. 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 hailstone 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 – search for "java 8 Math" on the web 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 sequence.
   * @param n  Starting number for sequence.  Assumes n > 0.
   * @return hailstone sequence starting with n and ending with 1.
   */
  public static List<Integer> hailstoneSequence(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;
  }
}
```

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 doesn't take a self parameter – which in Java is implicit anyway, you won't ever see it as a method parameter. Static methods can't be 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 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 and 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

The next reading will introduce *snapshot diagrams* to give us a way to visualize the distinction between changing a variable and changing a value. When you assign to a variable, you're changing where the variable's arrow points. You can point it to a different value.

When you assign to the con[illegible]s of a mutable value – such as an array or list – you're changing references inside that value.

Change is a necessary evil. Good programmers avoid things that change, because they may change unexpectedly.

Immutability (immunity from change) is a major design principle in this course. Immutable types are types whose values can never change once they have been created. (At least not in a way that's visible to the outside world – there are some subtleties there that we'll talk more about in a future class about immutability.) Which of the types we've discussed so far are immutable, and which are mutable?

Java also gives us immutable references: variables that are assigned once and never reassigned. To make a reference immutable, 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 immutable 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.

There are two variables in our hailstoneSequence method: can we declare them final, or not?

```
  public static List<Integer> hailstoneSequence(final int n) {
    final List<Integer> list = new ArrayList<Integer>();
```

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 change 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 class. 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.

The Goal of 6.005

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).
- **Easy to understand** . 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 6.005, 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.

Why we use Java in this course

Since you've had 6.01, 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 6.005?

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 can be used for web programming (both on the server and in the client), and native Android programming is done in Java. Although other programming languages are far better suited to teaching programming (Scheme and ML come to mind), 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). It has no interpreter like Python's, where you can learn by playing with small bits of code.

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.