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 JavaScript and Python versions of the code side by side for comparison:

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

print(n)
```

```
// JavaScript (or TypeScript, this code works in both languages)
let n = 3;
while (n !== 1) {
    console.log(n);
    if (n % 2 === 0) {
        n = n / 2;
    } else {
        n = 3 * n + 1;
    }
}
console.log(n);
```

The basic semantics of expressions and statements in JavaScript are very similar to Python. The `while` and `if` keywords behave the same, for example.

A few points of syntax are worth noting here:

- JavaScript requires parentheses around the conditions of the `if` and `while`.
- A JavaScript statement ends with a semicolon. This is technically optional, because JavaScript has rules for automatically inserting a semicolon at the end of a line. But these rules have obscure pitfalls that can lead to unexpected behavior and bugs, so always using semicolons is a good practice.
- JavaScript uses curly braces around blocks instead of indentation. But you should still always indent the block, even though JavaScript 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.

You may also notice that the JavaScript version uses `===` and `!==` where the Python uses `==` and `!=`. That's because `==` and `!=` in JavaScript do a variety of automatic type conversions to try to make the values on the lefthand side and righthand side comparable to each other. For example, `0 == ""` is true in JavaScript, which is surprising and confusing. The triple-equals versions of these operators are much safer and more predictable: `0 === ""` is false. Good JavaScript programmers will only use `===` and `!==`. This is a potential pitfall when moving from Python.

Types

But we are not using JavaScript in this class, strictly speaking. We are using *TypeScript*, which extends JavaScript with the ability to declare types in the program. In this case, we can specify that the variable `n` has type `number`:

```
let n: number = 3;
```

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

TypeScript has several built-in types, including:

- `number`, which represents both integers and floating-point numbers
- `boolean`, which represents true or false
- `string`, which represents a sequence of characters

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

- *As an operator.* For example, `a + b` invokes the operation `+ : number × number → number`. (In this notation: `+` is the name of the operation, `number × number` before the arrow gives the types of the two inputs, and `number` after the arrow gives the type of the output.)

- *As a function.* For example, `Math.sin(theta)` calls the operation `sin: number → number`. Here, `Math` is not an object. It's the class that contains the `sin` function.

- *As a method of an object.* For example, `str1.concat(str2)` calls the operation `concat: string × string → string`.
- *As a property of an object.* For example, `str.length` calls the operation `length: string → number`. Note the lack of parentheses after `str.length`.

Contrast TypeScript's `str.length` with Python's `len(str)`. It's the same operation in both languages – taking a string and returning its length – but it just uses different syntax.

Some operations are **overloaded** in the sense that the same operation name is used for functions that take different types of arguments. The operator `+` is overloaded in TypeScript. With numbers, 5 + 3 naturally produces 8. But when `+` is used on strings, it does string concatenation instead, so "5" + "3" produces "53". Overloading is not limited to operators like `+`; methods and functions can also be overloaded. Most programming languages have some degree of overloading.

Static typing

TypeScript is a **statically-typed language**. Variables can be assigned a type at **compile time** (before the program runs), and the compiler can therefore deduce the types of expressions using those variables. If `a` and `b` are declared as `number`, then the compiler concludes that `a+b` is also `number`. In fact, the VS Code environment does this while you’re writing code, so you can see all the relevant errors while you’re still typing.

JavaScript and Python, by contrast, are **dynamically-typed languages**, because 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 types at compile time, dynamic types at runtime

Static type declarations are used at compile time to perform static checking. Adding static types doesn’t change the fact that the behavior of our program at runtime is driven by actual values. In the hailstone program, if we declare `let n: number = 3;` to specify the static type of `n`, the initial value at runtime is still `3`, not any other `number`. We'll need to keep this distinction in mind when our static type encompasses more than one possible runtime type: a variable of static type `Animal` (let's imagine) might really be a `Dog` or a `Fish` or a `SpottedOwl` at runtime.

In fact, TypeScript – like many statically-typed languages – *throws away* the static type information after compilation. What does the TypeScript compiler generate? JavaScript! Here is a bit of code in well-typed TypeScript, and the generated JavaScript:

```
// TypeScript to compile
function hello(name: string): string {
    return 'Hi, ' + name;
}
let greeting: string = hello('types');
```

```
// JavaScript generated
function hello(name) {
    return 'Hi, ' + name;
}
let greeting = hello('types');
```

Support for static typing in dynamically-typed languages

Just like TypeScript adds static typing to the dynamically-typed JavaScript language, Python also has extensions that support static typing. 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.

The addition of static types to dynamically-typed languages like Python and JavaScript 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.

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 a `number`.

In general, static checking can detect errors related to the *type* of a variable – the set of values it is allowed to have, which is known at compile time in statically-typed languages like TypeScript – but is generally unable to find errors related to a specific value from that type.

Dynamic checking can catch, for example:

- specific illegal argument values. For example, the expression `x/y` is erroneous when `y` is zero, but well-defined for other values of `y`. So divide-by-zero is not a static error, because you can't know until runtime whether `y` is actually zero or not. But divide-by-zero can be caught as a dynamic error; Python throws `ZeroDivisionError` when it happens.

- illegal conversions, i.e., when the specific value can't be converted to or represented in the target type. For example, in Python, `int("hello")` throws `ValueError`, because the string `"hello"` cannot be parsed as a decimal integer.
- out-of-range indices or missing keys, e.g., using a too-large index on a string or array. In Python, `"hello"[13]` throws `IndexError`, and `{'a': 1}['b']` throws `KeyError`.

But you may have noticed that the dynamic-checking examples above all came from Python. What about dynamic checking in TypeScript? TypeScript does the static checking, but its runtime behavior is entirely provided by *JavaScript*, and JavaScript's designers decided to do **no checking** for many of these cases. So, for example, when a string or array index is out of bounds, or a key is not found in a map, JavaScript returns the special value `undefined`, rather than throwing an error as Python would. When dividing by zero, JavaScript returns a special value representing infinity rather than throwing an error. No checking makes bugs harder to find than they might otherwise be, because the special values can propagate through further computations until a failure finally occurs much farther away from the original mistake in the code.

Surprise: `number` is not a true number

Another trap in TypeScript – which is shared by many other programming languages – is that its numeric type has 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 some of the traps hiding in TypeScript:

- **Limited precision for integers**. All numbers in TypeScript are floating-point numbers, which means that large-magnitude integers can only be represented approximately. Integers between -2^{53} and 2^{53} (exclusive) can be represented exactly, but beyond that range, the floating-point representation preserves only the most-significant binary digits of the number. What does that mean if you have, say, 2^{60} and try to increment it? You get the same number back again: `2**60 + 1 === 2**60` in TypeScript. So this is an example of where what we might have hoped would be a dynamic error (because the computation we wrote can't be represented correctly) produces the wrong answer instead.

  These limits on representable integers are available as built-in constants:

  - `Number.MAX_SAFE_INTEGER` (roughly 2^{53} or 10^{36}) is the largest precisely-representable integer

  - `Number.MIN_SAFE_INTEGER` (roughly -2^{53} or -10^{36}) is the smallest precisely-representable integer

  Arithmetic on integers within these safe bounds is guaranteed to remain precise.

- **Special values**. The `number` type has several special values that aren't real numbers:
  - `Number.NaN` (which stands for "Not a Number")
  - `Number.POSITIVE_INFINITY` (which prints as just `"Infinity"`)
  - `Number.NEGATIVE_INFINITY` (which prints as `"-Infinity"`)

  When you apply certain operations to a `number` 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.

- **Overflow and underflow**. TypeScript is also unable to represent numbers that are too large (far from zero) or small (close to zero):
  - `Number.MAX_VALUE` (roughly 10^{308}) is the largest number that can be safely represented
  - `Number.MIN_VALUE` (roughly 10^{-324}) is the smallest positive number that can be safely represented

> What happens when you do a computation whose answer is too large or too small to fit in that finite range? The computation quietly *overflows* (becoming `POSITIVE_INFINITY` or `NEGATIVE_INFINITY`) or *underflows* (becoming zero).

These special constant values start with a capitalized `Number`, which is a class wrapper for `number`. Don't use `Number` as a type – only use it to access these constants. When you are declaring a type, always use the lowercase `number` instead.

READING EXERCISES

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

1

```
let n: number = 5;
n = n + ".0";
```

- ○ static error
- ○ dynamic error
- ○ no error, wrong answer

CHECK

2

```
let odd: number = 2**30 - 1;
// an odd number
let oddSquared: number = odd
* odd;  // the square of an o
dd number should be odd
```

- static error
- dynamic error
- no error, wrong answer

CHECK

3

```
let sum: number = 7;
let n: number = 0;
let average: number = sum/n;
```

- static error
- dynamic error
- no error, wrong answer

CHECK

4

```
let x: number = 0;
x += 0.1; x += 0.1;
x += 0.1; x += 0.1;
x += 0.1; x += 0.1;
x += 0.1; x += 0.1;
x += 0.1; x += 0.1;
// x should be 1.0
```

- static error
- dynamic error
- no error, wrong answer

CHECK

5

```
let n: number = 2**53 - 1000;
while (n !== 2**53 + 1000) {
    n = n + 1;
}
```

- static error
- dynamic error
- no error, infinite loop

CHECK

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

Let's change our hailstone computation so that it stores the sequence in a data structure instead of just printing it out.

For that, we can use an array type. Arrays are variable-length sequences of another type, similar to Python lists. Here's how we can declare an array of numbers and make an empty array value:

```
let array: Array<number> = [];
```

Here's the hailstone code written with arrays:

```
let array: Array<number> = [];
let n: number = 3;
while (n !== 1) {
    array.push(n);
    if (n % 2 === 0) {
        n = n / 2;
    } else {
        n = 3 * n + 1;
    }
}
array.push(n);
```

Functions

If we want to wrap this code into a reusable module, we can write a function:

```
/**
 * Compute a hailstone sequence.
 * @param n  starting number for sequence.  Assumes n > 0.
 * @returns hailstone sequence starting with n and ending with 1.
 */
function hailstoneSequence(n: number): Array<number> {
    let array: Array<number> = [];
    while (n !== 1) {
        array.push(n);
        if (n % 2 === 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    array.push(n);
    return array;
}
```

Take note of the `/** ... */` comment before the function, because it's very important. This comment is a specification of the function, 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 function types. It doesn't say, for example, that the return value is an array of numbers, because the `Array<number>` return type declaration just below already says that. But it does say that the sequence starts with `n` and ends with 1, which is not captured by the type declaration but is 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.

READING EXERCISES

Static checking of function return

In the code we wrote above, notice that the return type of the function is explicitly declared to be an array of numbers:

```
function hailstoneSequence(n:
number): Array<number> { ...
}
```

This means that TypeScript will statically check whether you are returning the right kind of value from the function. How would the TypeScript compiler judge each of the following `return` statements, if it appeared at the end of the body of `hailstoneSequence()` ?

`return array;`

- ○ static error
- ○ legal

`return [];`

- ○ static error
- ○ legal

`return ["8", "4", "2", "1"];`

- ○ static error
- ○ legal

`return Array<number>;`

- ○ static error
- ○ legal

`return new Array<number>();`

- ○ static error
- ○ legal

`return;`

- ○ static error
- ○ legal

no return statement at all at the end of the body of `hailstoneSequence()`

- ○ static error
- ○ legal

CHECK

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.

There are two aspects to immutability: whether a value is mutable or immutable, and whether a reference (such as a variable) can be reassigned or not.

Mutation refers to changing the content of a value. 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 TypeScript. Once a string is created, you can't make it shorter, or longer, or replace any of its characters. You would have to create a new string containing the changes you want.

By contrast, the array type is *mutable* in TypeScript, just like lists are mutable in Python. You can change the contents of an array or list – adding elements, removing elements, or replacing elements.

Python also has immutable sequences, *tuples*, which cannot be changed once created. TypeScript doesn't have immutable tuples, but it does have `ReadonlyArray`, which is an array type that omits the mutator operations. When you use `ReadonlyArray` as a type in your code, you can't call `push()` to add elements, or `splice()` to delete elements, or reassign elements like `arr[i] = ...`. TypeScript will report a compiler error, because it can't find those operations in the `ReadonlyArray` type.

Reassignment means changing where a variable points. Python allows any variable to be reassigned, but TypeScript allows us to declare *constants*, which are unreassignable: names that are assigned to a value once, and can never be reassigned. To make a reference unreassignable, declare it with the keyword `const` instead of `let`:

```
const n: number = 5;
```

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

It's good practice to use `const` for as many 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

const

Consider the variables in our `hailstoneSequence` function:

```
function hailstoneSequence(n:
number): Array<number> {
    let array: Array<number>
= [];
    while (n !== 1) {
        array.push(n);
        if (n % 2 === 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    array.push(n);
    return array;
}
```

Which variables might we declare `const`, because they are never reassigned in the code?

- ☐ `n: number`
- ☐ `array: Array<number>`

CHECK

ReadonlyArray

Suppose we change all the uses of `Array` in the function into `ReadonlyArray` instead:

```
function hailstoneSequence(n:
number): ReadonlyArray<number
> {
    let array: ReadonlyArray<
number> = [];
    while (n !== 1) {
        array.push(n);
        if (n % 2 === 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    array.push(n);
    return array;
}
```

Which lines will have static errors?

- ☐ `let array: ReadonlyArray<number> = [];`
- ☐ `while (n !== 1) {`
- ☐ `array.push(n)`
- ☐ `return array`

CHECK

Documenting assumptions

Writing the type of a variable down documents an assumption about it: e.g., the variable `n` will always refer to a number. TypeScript 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 `const` is also a form of documentation: a claim that the variable will never be reassigned after its initial assignment. TypeScript checks that too, statically.

We documented another assumption that TypeScript (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 (possibly you) 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 (and everyone using your code) will 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 our own! Static checking helps with that.

READING EXERCISES

Consider the following simple Python function:

```
from math import sqrt
def fun_fact_about(person):
    if sqrt(person.age) == int(
sqrt(person.age)):
        print("The age of " + p
erson.name + " is a perfect squ
are: " + 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.age` must be a nonnegative number
- ☐ `person.age` must be an integer
- ☐ `person.name` must be a string

CHECK

Checkable assumptions

If this code were written in TypeScript instead, which of the following assumptions could be documented by type declarations and statically checked by the TypeScript compiler?

- ☐ `person` must be an object with `age` and `name` instance variables

- [ ] `person.age` must be a nonnegative number
- [ ] `person.age` must be a number
- [ ] `person.name` must be a string

CHECK

The goals of 6.102

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 6.102, 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

SFB

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

- ☐ dynamic checking
- ☐ const variables
- ☐ overflow
- ☐ static typing

CHECK

ETU

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

- ☐ documented assumptions in comments
- ☐ const variables
- ☐ overloading
- ☐ static typing

CHECK

RFC

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

- ☐ documented assumptions in comments
- ☐ functions
- ☐ static typing

CHECK

Why we use TypeScript in this course

Since you've had 6.101 [formerly 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 TypeScript/JavaScript in 6.102?

Safety is the first reason. TypeScript 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. TypeScript dials safety up to a high level, 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 or JavaScript, 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. TypeScript compiles into pure JavaScript, which is widely used in research, education, and industry. JavaScript runs on many platforms, not just in the web browser where it originally became widespread, but now also in web servers and even in desktop applications for Windows/Mac/Linux.

Compared to Java, TypeScript has a richer type system, needs less boilerplate code to write a program, and is a better choice for creating modern user interfaces and web apps.

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 very likely have to pick up other programming languages (such as Java, C/C++, Scheme, Ruby, or Haskell) for classes, internships, or UROPs before you even finish your MIT career, so we're getting you started now by learning a second one.

As a result of its ubiquity, JavaScript has a wide array of interesting and useful **libraries** and excellent free **tools** for development (IDEs like VS Code, editors, compilers, test frameworks, profilers, code coverage, style checkers). JavaScript and Python are close competitors in the richness of their ecosystems, and TypeScript can take full advantage of JavaScript libraries.

There are some reasons to regret using JavaScript. It's large, having accumulated many features in the decades since it was originally designed. It's weighed down by the baggage of older languages like C/C++ — the `switch` statement is a good example of an antiquated and unsafe language construct inherited from C. And JavaScript itself started with some poor design decisions, including type conversion rules that make `0 == ""` return true and `[] +`

`[]` somehow produce the empty string, as well as less dynamic checking than other languages. TypeScript fixes some of these warts in JavaScript, but not all. There are many parts of JavaScript that are now deprecated and avoided by programmers because they're risky to use.

But on the whole, TypeScript 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.

The real skills you'll get from this course are not language-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.

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.