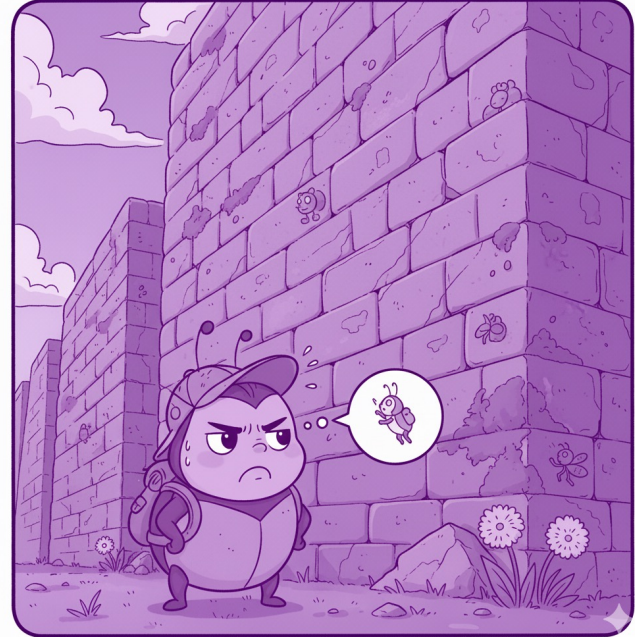


CSE 331

Making Bugs Impossible

James Wilcox and Kevin Zatloukal



Making Bugs Impossible

- Goal to make bugs impossible via
 - clever design and/or
 - type system
- This means no need for tests or reasoning here
 - same is not be true for runtime checks (e.g. `checkRep`)
- Worthwhile for common, **awful** bugs...

Bug 1: Method Call Order

```
public interface A {  
    void foo();  
    int bar(); // ONLY call this after foo!!  
}
```

- Bug if *some* path calls `bar` without `foo` first
 - type checker does not catch this
 - need for this does arise occasionally
 - can only happen with mutators
- Not good design
 - someone will eventually make a mistake here

Bug 2: Argument Order

```
void foo(int a, String s);  
void bar(int a, int b);
```

```
foo("b", 1);    // wrong order: compiler error  
bar(2, 1);      // wrong order: no error
```

- Bug if arguments are swapped
 - type checker will not catch if the types are the same
- Hard to remember order of long argument list
 - easy mistake to make

Bug 2: Argument Order

```
// @modifies A
// @effects A[j] = val for 0 <= j < len
void memset(int[] A, int val, int len);
```

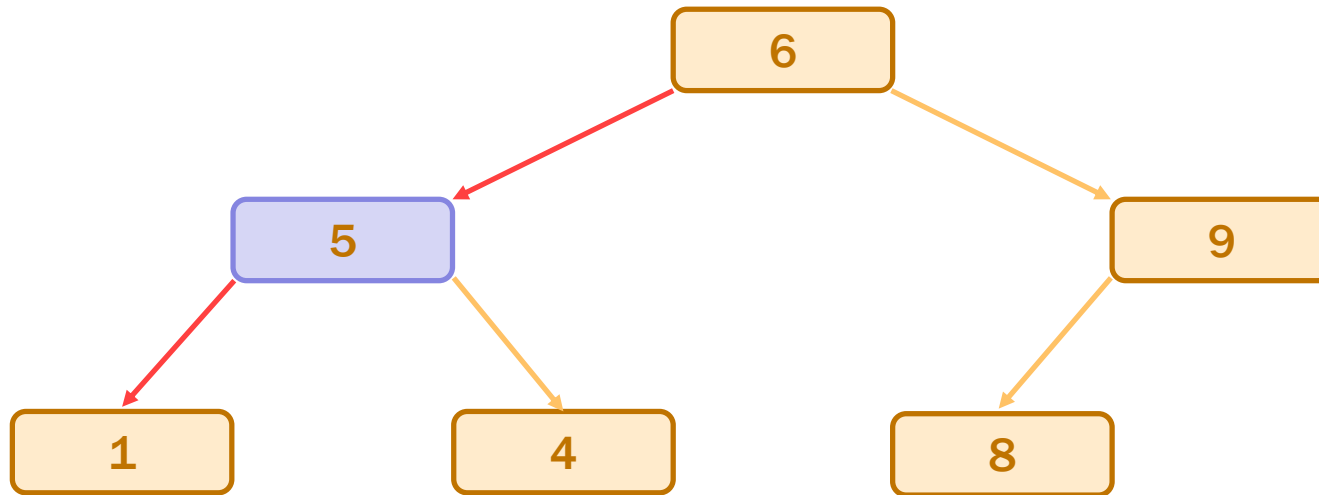
- Famous bug due to mixing up argument order!

```
    memset(A, 0, A.length)    // set A to zeros
vs  memset(A, A.length, 0)    // does nothing!
```

- If you program long enough, you will see this one
 - easy case to miss in unit tests

Recall: Binary Search Trees

- Suppose someone changed "3" into "5"...
 - now this happens when we search for "4":



- It can no longer be found!

Bug 3: Key Mutation

```
Map<A, B> map;
```

- **Bug if any A used as a key in this map is mutated**
 - Java type checker will not catch this
 - possible only with aliasing to mutable A
- **Debugging will be very painful**
 - mutation only happen occasionally
 - could take weeks to find it

Impossible By *Design*

Method Call Order

```
public interface A {  
    void foo();  
    int bar(); // ONLY call this after foo!!  
}
```

- Possible to prevent this by better design...

```
public interface A {  
    B foo();  
}
```

Not possible to call `bar`
(no longer has that method)

```
public interface B extends A {  
    int bar();  
}
```

Can only get a `B` by calling `foo`
(do not provide any other way)

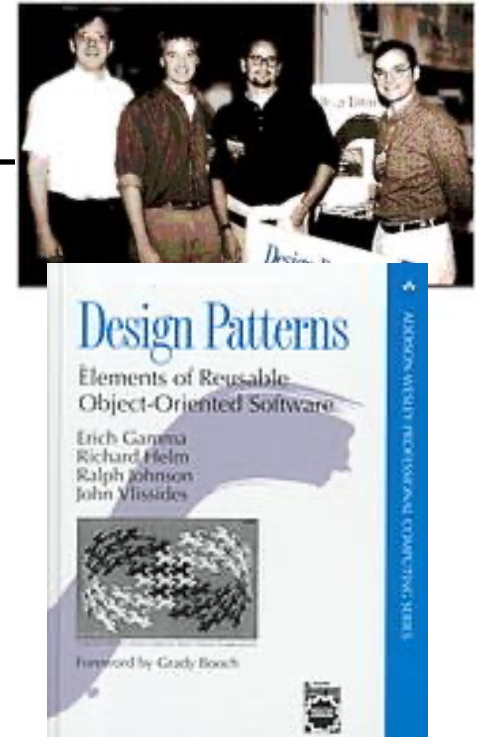
Impossible by Design

- The other bugs can also be prevented by design, but require more complex solutions...

Design Patterns

Design Patterns

- Introduced in the book of that name
 - written by the “Gang of Four”
Gamma, Helm, Johnson, Vlissides
 - worked in C++ and SmallTalk
- Found that they independently developed many of the same solutions to recurring problems
 - wrote a book about them
- Many are problems with OO languages
 - authors worked in C++ and SmallTalk
 - some things are not easy to do in those languages



Parts of a Design Patterns

Each pattern in the book includes

- **Problem** to be solved
- **Description** of the solution
- **Name** of the pattern

Java Example: Iterator

- **Java Collections use the **Iterator** Design Pattern**
 - enumerate a collection while hiding data structure details
 - return another ADT that outputs the items
 - that object knows how to walk through the data structure
 - operations for retrieving the current item and moving on to the next one
- **Clever idea that is now used everywhere**
 - Kevin remembers when C++ introduced iterators
 - huge improvement over code we were writing before

Categories of Design Patterns

The book has three categories of patterns

- **Creational:** **factory function**, factory object, builder, prototype, **singleton**, ...
- **Structural:** **adapter**, bridge, composite, decorator, façade, flyweight, proxy
- **Behavioral:** command, interpreter, **iterator**, mediator, observer, state, strategy, visitor, ...
 - **green** = mentioned already
 - we will just cover a few of these for now...

Creational Patterns

- One third of the patterns deal with object **creation**
 - why?
- Because constructors are terrible!
- Already saw factory functions and singleton
 - yet we still need more creational patterns

Constructors

Public Constructors

- **Most Java classes have public constructors**
 - e.g., create an `ArrayList` with “`new ArrayList<String>()`”
- **Constructors have undesirable properties**
 - not a good idea to do any real work in them
 - cannot test a method without also testing the constructor
 - surprisingly error-prone
 - several important limitations

Recall: Tight Coupling Example 3

```
class WorkList {  
    // RI: len(names) = len(times) and total = sum(times)  
    protected ArrayList<String> names;  
    protected ArrayList<Integer> times;  
    protected int total;  
  
    public addWork(Job job) {  
        int time = job.getTime(); // just one call  
        total += time;  
        addToLists(job.getName(), time);  
    }  
}
```

RI is not true in method call!

Method Calls from Constructors

- Any method call from a constructor is dangerous!
- Almost always calling with RI false
 - usually, the RI does not hold until all fields are assigned
typically, that is the last line of the constructor
 - hence, any methods are called with the RI still false
- Asking for trouble!
 - method needs to know that some parts of RI may be false
 - eventually, someone changing code will mess this up
 - better to avoid method calls in the constructor

Limitations of Constructors

- **Constructor is called *after* the object is created**
 - can't decide, in the constructor, not to create it
- **Limitations of constructors**
 1. **Cannot return an existing object**
 2. **Cannot return a different class**
 3. **Does not have a name!**

Factory Functions & Singleton

- Factory functions can return an existing object
- Common case: there is only one instance!
 - called the “**singleton**” design pattern
- Examples from before...

Recall: Example 2: Point in 2D Space

```
/** Creates a point at the given coordinates. */  
public static Point makePoint(double x, double y) {  
    if (Runtime.getRuntime().totalMemory() > MIN_MEM) {  
        return new PolarPoint(x, y);  
    } else {  
        return new SimplePoint(x, y);  
    }  
}
```

**Can return instances of different
classes in different cases**

- This is a "factory function"
 - an example of a **design pattern**
 - Java SDK includes many, e.g., `Arrays.asList(..)`

Recall: Creator of FastLists

```
public static FastList EMPTY_LIST =  
    new FastBackList(null);
```

```
/** @return nil */  
public static FastList emptyList() {  
    return EMPTY_LIST;  
}
```

Do not need to create a new object

- This is the "singleton" **design pattern**
 - note: this is only possible since `FastList` is immutable!

Returning a Subtype

- Factory functions can return a subtype
 - declared to return **A** but returns subtype **B** instead
 - allowed since every **B** is an **A**

- Example:

```
// @returns an empty NumberSet that can be used to
//      store numbers between min and max (inclusive)
public NumberSet makeNumberSet(int min, int max) {
    if (0 <= min && max <= 100) {
        return makeArrayNumberSet(); // only supports small sets
    } else {
        return makeSortedNumberSet(); // use a tree instead
    }
}
```

Multiple Constructors

- Java classes allow multiple constructors

```
class HashMap {  
    public HashMap() { ... } // initial capacity of 16  
    public HashMap(int initialCapacity) { ... }  
}
```

- Multiple methods with the same name is "overloading"
- Methods are distinguished by the argument types
 - name + arg types is called the "signature" of the method
 - type checker figures out which method you are calling

Constructors Have No Name

- **Do not get to name constructors**
 - in Java, same name as the class
 - likewise in JavaScript, where it is called “constructor”
- **Names are useful!**
 1. Let you distinguish between different cases
 - use names to distinguish cases that otherwise look the same
 2. Let you explain what it does
 - the only thing you know the client will read!

Example: Distinguishing Constructors

- JavaScript's Array has multiple constructors

```
new Array()           // creates []
```

```
new Array(a1, ..., aN) // creates [a1, ..., aN]
```

```
new Array(2)          // creates [undefined, undefined]
```

- what does “`new Array(a1)`” return when `a1` is a number?
- how to make a 1-element array containing just `a1`

```
const A = new Array(1);  
A[0] = a1;
```

- don't have a name to distinguish these cases!

Example: Distinguishing Constructors

- Factory functions have names
 - allow us to distinguish these cases

```
// @returns []  
public int[] makeEmptyArray()  
  
// @returns A with A.length = len and  
//      A[j] = undefined for any 0 <= j < len  
public int[] makeArray(int len)  
  
// @returns [args[0], ..., args[N-1]]  
public int[] makeArrayContaining(int... args)
```

- function name is also the one thing you know clients read!
best chance to tell them how to use it correctly

Example: Distinguishing Constructors

- Factory functions have names
 - allow us to distinguish these cases

```
// @returns []  
public int[] makeEmptyArray()
```

```
// @returns A with A.length = len and  
//      A[j] = undefined for any 0 <= j < len  
public int[] makeArray(int len)
```

```
// @returns A with A.length = len and  
//      A[j] = val for any 0 <= j < len  
public int[] makeFilledArray(int len, int val)
```

Be careful...

Creational Pattern: Builder

- Object that helps with creation of another object
 - constructor / factory requires you to give info all at once
 - builder lets you describe what you want bit by bit

- **Example:** `StringBuilder`

```
StringBuilder buf = new StringBuilder();  
buf.append("Total distance: ");  
buf.append(distance);  
buf.append(" meters.");  
return buf.toString();
```

- each call adds more text / number to the final string
- we can't do this with strings because strings are *immutable*

Creational Pattern: **Builder**

- Object that helps with creation of another object
 - constructor / factory requires you to give info all at once
 - builder lets you describe what you want bit by bit
- Good pairing: **mutable** Builder for an **immutable** type
 - must avoid aliasing with the mutable builder
 - e.g., never use it as a key in a BST or Map
 - **immutable object can be shared arbitrarily**
 - no worries about aliasing

Creational Pattern: Builder

- Builder is often written like this:

```
class FooBuilder {  
    ...  
    public FooBuilder setX(int x) {  
        this.x = x;  
        return this;  
    }  
    ...  
    public Foo build() { ... }  
}
```

- can then use them like this

```
Foo f = new FooBuilder().setX(1).setY(2).build();
```


avoids worries about argument order

Recall: Argument Order Bugs

```
// @returns A with A.length = len and
//      A[j] = val for any 0 <= j < len
public int[] makeFilledArray(int len, int val)
```

Be careful...

- Some famous bugs due to mixing up argument order!
- If you program long enough, you will see this one
- Can fix this with a **Builder**
(in other languages, there is a simpler solution)

Argument Builder

```
// Returns an array with length & value given in args.
```

```
public int[] makeFilledArray(Args args)
```

```
class Args {  
    public int length;  
    public int value;  
}
```

```
Args args = new Args();  
args.length = 10;  
args.value = 5;  
... = makeFilledArray(args);
```

Arguments distinguished by name
not by order

- code using the function is now more verbose...
can make this easier by giving them a Builder

Argument Builder

// Returns an array with length & value given in args.

```
public int[] makeFilledArray(Args args) { ... }
```

```
class ArgsBuilder {
```

```
    ...
```

```
    public ArgsBuilder setLength(int length) {
```

```
        this.length = length;
```

```
        return this;
```

```
    }
```

```
    ...
```

```
    public Args toArgs() { ... }
```

```
}
```

```
... = makeFilledArray(new ArgsBuilder()  
    .setLength(10).setValue(5).toArgs());
```

Categories of Design Patterns

The book has three categories of patterns

- **Creational:** **factory function**, factory object, builder, prototype, **singleton**, ...
- **Structural:** adapter, bridge, composite, decorator, façade, flyweight, proxy
- **Behavioral:** command, interpreter, **iterator**, mediator, observer, state, strategy, visitor, ...
 - **green** = mentioned already

Structural Pattern: Adapter

- In Java, these two classes are not interoperable:

```
interface Duration {  
    int getMinutes();  
    int getSeconds();  
}
```

```
interface AmountOfTime {  
    int getMinutes();  
    int getSeconds();  
}
```

- cannot pass one where the other is expected
- in Java, different names means they are different
we say it has a "nominal" type system

Structural Pattern: Adapter

- Get around this by creating an adapter

```
class DurationAdapter implements AmountOfTime {  
    private Duration d;  
  
    public DurationAdapter(Duration d) {  
        this.d = d;  
    }  
  
    int getMinutes() { return d.getMinutes(); }  
    int getSeconds() { return d.getSeconds(); }  
}
```

- makes a Duration into an AmountOfTime

Alternative Languages

Alternative Languages

- Last time, we looked at ways to prevent bugs by working *within* Java's type system
- Next, we will look at alternative languages
 1. Python
 2. TypeScript
 3. Rust
- Goal is to **understand** how the type system works
 - know which bugs it catches and which it misses
 - we will not be writing any code in this language

Alternative Languages

- Last time, we looked at ways to prevent bugs by working *within* Java's type system
- Next, we will look at alternative languages
 1. Python
 2. TypeScript
 3. Rust
- Java's type system made one set of choices
 - none of these are sacrosanct
 - easier to see once you know a few different languages

Python

Background on Python

- By some measures the most popular language
- Sometimes called a "scripting" language
 - for writing "scripts" that call other programs
 - especially easy to call into C libraries
- Designed for ease-of-use not performance
 - python interpreter is *extremely* slow
- Widely used today for AI
 - hard work is done inside of libraries and on GPUs

Background on Python

- Does not require type declarations
 - any variable can store type
 - same variable can switch types during execution

```
x = 1          # I'm an integer
x = "abc"      # I'm a string now!
```

- Called a "dynamically typed" language
 - type checks occur at run-time (in some cases)
 - it often tries to convert types that are not correct
 - no type checker means more testing & reasoning

Named Arguments

```
# @modifies A
# @effects A[j] = val for 0 <= j < len
def memset(vals, count, value):
    ... # implementation
```

- Can use argument names in calls:

```
memset(A, value=0, count=100)
```

- **latter two arguments are identified by name**
does not matter what order they are in
- **eliminates most argument order bugs**

Built-In Dictionary

- **Java creates maps like this:**

```
Map<Integer, String> m = new HashMap<>();  
m.put(3, "foo");  
m.put(4, "bar");
```

- **Can create the same in Python like this**

```
m = {3: "foo", 4: "bar"}
```

- much more concise
- can also create records without classes

Built-In Dictionaries

- Can add and remove from dictionaries

```
m = {}  
m[1] = "foo"  
m["bar"] = 2          # can use strings as keys
```

- But cannot use mutable types as keys:

```
L = [1, 2]  
m[L] = "baz"          # runtime type error
```


Built-In Dictionaries

- Cannot use lists as keys:

```
L = [1, 2]
m[L] = "baz"           # runtime type error
```

- Can use tuples (immutable lists) as keys

```
L = (1, 2)
m[L] = "baz"           # okay!
```

- Only immutable keys is less efficient
 - requires extra copying of data in many cases
 - worthwhile for the elimination of key mutation bugs

JavaScript

History of JavaScript

- **By some measures, second most popular language**
- **Incredibly simple language**
 - created in 10 days by Brendan Eich in 1995
 - often difficult to use because it is so simple
- **Features added later to fix problem areas**
 - imports (ES6)
 - classes (ES6)
 - integers (ES2020)

Relationship to Java

- **Initially had no relation to Java**
 - picked the name because Java was popular then
 - added Java's Math library to JS also
 - e.g., `Math.sqrt` is available in JS, just like Java
 - copied *some* of Java's String functions to JS string

JavaScript Syntax

- Both are in the “C family” of languages
- Much of the syntax is the same
 - most expressions (+, −, *, /, ? :, function calls, etc.)
 - `if, for, while, break, continue, return`
 - comments with `//` or `/* .. */`

Java vs JavaScript Syntax

- The following code is legal in both languages:
 - assume “s” and “j” are already declared

```
s = 0;
j = 0;
while (j < 10) {                OR for (j = 0; j < 10; j++)
    s += j;
    j++;
}

// Now s == 45
```

JavaScript Syntax

- Both are in the “C family” of languages
- Much of the syntax is the same
 - most expressions (+, −, *, /, ?:, function calls, etc.)
 - `if, for, while, break, continue, return`
 - comments with `//` or `/* .. */`
- Different syntax for a few things
 - declaring variables
 - declaring functions
 - equality (===)

no declared types

"==" is not transitive

Differences from Java: Type Declarations

- JavaScript variables have no declared types
 - it is "dynamically typed"
- Declare variables in one of these ways:

```
const x = 1;  
let y = "foo";
```

- “**const**” cannot be changed; “**let**” can be changed
- use “**const**” whenever possible!

Basic Data Types of JavaScript

- JavaScript includes the following runtime types

number

bigint

string

boolean

undefined

null

(another undefined)

Object

Array

(special subtype of Object)

Checking Types at Run Time

Condition	Code
x is undefined	<code>x === undefined</code>
x is null	<code>x === null</code>
x is a number	<code>typeof x === "number"</code>
x is an integer	<code>typeof x === "bigint"</code>
x is a string	<code>typeof x === "string"</code>
x is an object or array (or null)	<code>typeof x === "object"</code>
x is an array	<code>Array.isArray(x)</code>

Programmers must write their own type checks!

TypeScript

TypeScript Adds Declared Types to JavaScript

- TypeScript includes declared types for variables
- Compiler checks that the types are valid
 - produces JS just by *removing* the types

TypeScript Adds Declared Types

- Type is declared after the variable name:

```
const u: bigint = 3n;  
const v: bigint = 4n;
```

```
function add(x: bigint, y: bigint): bigint {  
    return x + y;  
};
```

```
console.log(add(u, v)); // prints 7n
```

- return type is declared after the argument list (...) and before {
- “Where types go” is the main syntax difference vs Java

Literal Types

- Any literal value is also a type:

```
let x: "foo" = "foo";  
let y: 16n = 16n;
```

- Variable can only hold that specific value!
 - can assign it again, but only with the same value
 - seems silly, but turns out to be useful...

Ways to Create New Types in TypeScript

- **Union Types** `string | bigint`
 - can be either one of these
- **Not possible in Java!**
 - TS can describe types of code that Java cannot

Enumerations

- Use **unions of literals** are “enums”

```
function dist (dir: "left"|"right", amt: bigint): bigint {  
  if (dir === "right") {  
    return amt;  
  } else {  
    return -amt;  
  }  
};
```

- TypeScript ensures that callers will only pass one of those two strings (“left” or “right”)
 - impossible to do this in Java
(must fake it with the enumeration **design pattern**)

Java Enums

- Another design pattern built into Java:

```
enum Dir {  
    LEFT, RIGHT  
}
```

- `Dir.LEFT` and `Dir.RIGHT` are the only 2 instances
- Cannot pass a `Dir` where `String` is expected
 - must add methods to convert between them

Ways to Create New Types in TypeScript

- **Union Types** `string | bigint`

- can be either one of these

- How do we work with this code?

```
const x: string | bigint = ...;
```

```
// how to check if can I call isPrime(x: bigint)?
```

- We can check the type of `x` using “`typeof`”
 - TypeScript understands these expressions
 - will “`narrow`” the type of `x` to reflect that information

Type Narrowing With “If” Statements

- **Union Types** `string | bigint`
 - can be either one of these
- How do we work with this code?

```
const x: string | bigint = ...;

if (typeof x === "bigint") {
  console.log(isPrime(x))    // okay! x is a bigint
} else {
  ...                        // x is a string
}
```

Programmer must write type checks only for unions
and the type checker will make sure they do!

Type Narrowing vs Casting

```
const x: string | bigint = ...;

if (typeof x === "bigint") {
  console.log(isPrime(x))    // okay! x is a bigint
} else {
  ...                        // x is a string
}
```

- Note that this does not require a **type cast**
 - TypeScript knows `x` is a `bigint` inside the “if” (narrowing)
- TypeScript has casts but they are completely unsafe!

Type Narrowing vs Casting

```
Object x = ...;  
System.out.println(isPrime( (Integer) x ) )
```

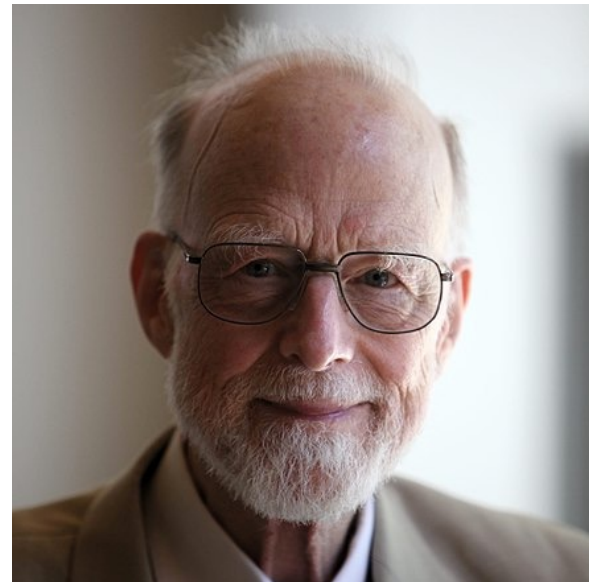
- Java will check this at runtime
 - throws an exception if `x` is not an `Integer`

```
const x: string | bigint = ...;  
console.log(isPrime(<bigint>x))    // trust me!
```

- TypeScript casts are unchecked at runtime!
 - seem designed to create extremely **painful** debugging

Sir Anthony Hoare

- Recall that Floyd & Hoare invented Floyd Logic
 - won the Turing award in 1980
- Also invented "null"…
 - ... and null pointer exception
 - "billion-dollar **mistake**"



Tony Hoare

Java Reference Types

- In Java, these type declarations

```
String s;  
A x;
```

really mean

```
... s ... // is a String or null  
... x ... // is an A or null
```

- All reference types implicitly allow null also!
 - use of fields or methods are checked at runtime
throws an NPE if it is null
 - no help from the **type checker**

TypeScript Types

- In Java, these type declarations

```
let s: string;  
let x: A;
```

mean what they say

```
... S ... // is a string  
... x ... // is an A
```

- Not possible for these to be null at runtime

TypeScript Types

- Must include null *explicitly*

```
let n: bigint | null;
```

- The type checker will make sure you handle it

```
isPrime(n);    // error: could be null
```

```
if (typeof n == "bigint")  
    isPrime(n); // this is okay
```

Ways to Create New Types in TypeScript

- Can create **compound** types in multiple ways
 - put multiple types together into one larger type

- **Record Types** {x: **bigint**, s: **string**}
 - anything with *at least* fields “x” and “s”

```
const p: {x: bigint, s: string} = {x: 1n, s: "hi"};  
console.log(p.x); // prints 1n
```

Ways to Create New Types In TypeScript

- Can create **compound** types in multiple ways
 - put multiple types together into one larger type

- Tuple Types `[bigint, string]`
 - create them like this

```
const p: [bigint, string] = [1n, "hi"]; // an array
```

- give names to the parts (“destructuring”) to use them

```
const [x, y] = p;  
console.log(x); // prints 1n
```

Use Records for Named Arguments

```
// @modifies A
// @effects A[j] = val for 0 <= j < len
function memset(vals: number[],
               desc: {count: number, value: number})
```

- **Client invokes it like this**

```
memset(A, {value: 0, count: 100})
```

- **latter two arguments are identified by name**
does not matter what order they are in
- **much easier than Java (less easy than Python)**
Python turns named arguments into a record for you

Type Aliases

- TypeScript lets you give shorthand names for types

```
type Point = {x: bigint, y: bigint};
```

```
const p: Point = {x: 1n, y: 2n};  
console.log(p.x); // prints 1n
```

- Usually nicer but not necessary
 - e.g., this does the same thing

```
const p: {x: bigint, y: bigint} = {x: 1n, y: 2n};  
console.log(x); // prints 1n
```

Structural vs Nominal Typing

- Deep difference between TypeScript and Java types
- TypeScript uses “**structural typing**”
 - sometimes called “duck typing”

“if it walks like a duck and quacks like a duck, it’s a duck”

```
type T1 = {a: bigint, b: string};
```

```
type T2 = {a: bigint, b: string};
```

```
const x: T1 = {a: 1n, b: "two"};
```

- can pass “`x`” to a function expecting a “`T2`”!

Recall: Structural Pattern: Adapter

- In Java, these two classes are not interoperable:

```
interface Duration {  
    int getMinutes();  
    int getSeconds();  
}
```

```
interface AmountOfTime {  
    int getMinutes();  
    int getSeconds();  
}
```

- cannot pass one where the other is expected
- in Java, different names means they are different
we say it has a "nominal" type system

Structural vs Nominal Typing

- Adapters are often needed with **nominal** typing
 - design pattern working around a language issue
- With **structural** typing, these two interoperate:

```
type Duration = {min: number, sec: number};
```

```
type AmountOfTime = {min: number, sec: number};
```

- can pass either where the other is expected
- not an issue of fields vs methods
 - still interoperable if we have `getMinutes` and `getSeconds` methods

Rust

Rust

- **Language for systems programming**
 - operating systems
 - real-time systems
 - servers of various kinds
- **Type system has extra safety properties**
 - regarding aliases, multi-threading, allocation
 - much safer than C or C++
- **Only available since 2015**

Warnings

- Rust is not a language Kevin is fluent in
 - written 100+k lines in every other language discussed
- Much of the code shown here is **illegal**
 - ignores additional Rust rules on
 1. multi-threading
 2. memory allocation
 - no time for multi-threading (see 451)
 - memory allocation rules are annoying (more later)

Rust Type System

- Rust ensures every value on the heap has either
 1. a single **mutable** reference
 2. any number of **immutable** references
 - cannot have multiple mutable references
 - cannot mix mutable and immutable reference
- Why do this?
 - it prevents the mutable aliases
 - exact same advice we gave earlier about aliases

these are the two "easy ways" to avoid unsafe aliases

Mutable and Immutable References

- Variables must be declared mutable
 - default is immutable

```
let s = String::from("hello");  
s.push_str(", world"); // error!
```

- requires an explicit declaration to call mutators

```
let mut s = String::from("hello");  
s.push_str(", world"); // okay
```

- This is good!
 - but far from perfect (as we will see later)

Mutable References

- Cannot create a second reference:

```
let mut s = String::from("hello");  
let mut t = s;  
t.push_str(", world");  
s.push_str("!"); // error!
```

- Assignment to `t` invalidates `s`
 - ownership "move" from `s` to `t`
 - can no longer use `s`

Function Calls

- Creates problems for function calls:

```
fn add_text(t: mut String) -> {  
    t.push_str(", world");  
}
```

```
let mut s = String::from("hello");  
add_text(s);  
s.push_str("!"); // error!
```

- Call to `add_text` implicitly assigns "`t = s`"
- So how do we call functions?

Function Calls

- Rust allows a "**borrow**" by using "&"

```
fn add_text(t: &mut String) -> {  
    t.push_str(", world");  
}
```

```
let mut s = String::from("hello");  
add_text(&mut s);  
s.push_str("!");    // okay
```

- ownership is moved but then returned after the call

Function Calls

- If a function attempts to keep an alias, it will cause an error:

```
static ref strs = HashMap::new();

fn add_text(t: &mut String) -> {
    t.push_str(", world");
    strs.insert(t, strs.length()); // error!
}

let mut s = String::from("hello");
add_text(&mut t);
```

- Rust knows ownership must be returned at the end, so you cannot move ownership into `strs`

Structs (a.k.a. Records)

- Everything above used `String`
- Can declare your own types of records like this:

```
struct MyRecord {  
    value: i32;    // 32-bit integer  
    count: i32;  
}
```

```
let r = MyRecord {value: 1, count: 10};
```

- declares a new record and then instantiates one
- this will be used in the homework

Other Thoughts on Rust

- Some parts of Rust are not good for general use
- Notion of immutable is too low-level
 - "mutable" means any change to the fields
- Not every field change is really a mutation
 - only a mutation if it changes the **abstract state**
 - if the abstract state is unchanged, then all methods return the same values, so clients cannot observe any difference
 - see the homework for an example of this!

Other Thoughts on Rust

- **Some parts of Rust are not good for general use**
- **Memory allocation is suboptimal**
 - uses ownership to avoid garbage collection
- **Garbage collection is useful**
 - making everyone pay for problems of real-time systems
 - there are type systems that can solve this problem
 - see "fully in-place functional programming"
- **There is no perfect programming language**

Final Thoughts

- **Saw language features that prevent bugs**
 - Java's choices are not sacrosanct
- **Saw variety of type systems**
 - TypeScript prevents null pointer exceptions
 - Rust prevents multiple aliases to mutable state
 - structural vs nominal types
- **Kevin expects more work here in response to AI**
 - type system can add reliability to LLM codegen
 - very few limits on what can be done (see 311)