I messed up

This is an apology.

I research the heck out of my talks and take pride in the quality of the product of that work. I tend to pick topics that require a fair bit of research because that's what motivates me. **I missed the mark**, however, **when talking about autoboxing and unboxing of primitive types in Java**.

The object conversion issue is one that can hurt you in most languages, but I didn't want this to be just a C# talk. Remembering back to my two years as a Java programmer, this issue came to mind. I searched the web to see if the situation had changed, but **my terminology was not precise enough to give me the answer I needed**. Instead, **I gravitated toward the answer I wanted** (to make my job easier).

Just so I can be 100% clear. **Boxing and unboxing primitives in Java happens if the parameter type is mismatched with the variable**. **It is not a normal problem in Java** but can be an insidious one when it occurs. **It certainly doesn't happen as I described it.**

Anyway, because I owe it to the community, I'll leave this deck otherwise untouched, but will follow up with a detailed blog post for wonks like me. Check https://m.eado.ws/blog for this and other upcoming posts.

Ignoring Nulls

And Five Other Horrible Things Most Developers Do

AKA

The Seven Deadly Sins of Programming

But Wait...

About Me

- Michael Meadows (he/him)
- Father, husband, trans rights ally, advocate for the unhoused
- Architect at Insight Consulting
- Aspirational sea kayaker

NOT the Venue for a Language Holy War

Seven Deadly Programming Sins

- Language-Driven Bad Practices
  1. Misusing Exceptions
  2. Underestimating the Impact of Objects on Memory Management
  3. Everything Related to Strings
- High Mental Load
  4. Ignoring the Complexity of Null
  5. Trying to Simplify Date/Time
  6. Everything Related to Logging
- And…
  7. Being dogmatic about any of the above

No. First, Let's Talk About

Seven Deadly Programming Sins

- Language-Driven Bad Practices
  1. Misusing Exceptions
  2. Underestimating the Impact of Objects on Memory Management
  3. Everything Related to Strings
- High Mental Load
  4. Ignoring the Complexity of Null
  5. Trying to Simplify Date/Time
  6. Everything Related to Logging
- And…
  7. Being dogmatic about any of the above

Language-Driven Bad Practices

Not just for n00b$!

Exceptions Are Your Enemy

… except when you follow this advice

… except when you need an escape route

- Special (exceptional) control flow
- Interrupts current execution when there is no better way out
- Built-in signaling and filtering in most languages

HCF

Exceptions be like...

… except when they burn you

- Use in non-exceptional control flow
- Reliance on the message
- Catch-log-throw
- Swallowing whole (and its variants)
- As a substitution for a return value

Exception Example: So Many Bads

```
function Find(values) {
  let retval;
  try {
    for (i = 0; i < values.length; i++) {
      retval = FormatValue(values[i]);
    }
  } catch (error) {
    if (typeof retval === 'undefined') {
      throw error;
    }
  }
  return retval;
}
```

```
function FormatValue(value){
  if (value) {
    return `Value is: ${value}`;
  }
  throw "The value is empty";
}

console.log(Find(["muh", "buh", "wuh"]));
// "wuh"
console.log(Find(["duh", null, "guh"]));
// "duh"
console.log(Find([null, "yuh"]));
// exception "The value is empty"
```

Exception Example: Many Less Bads

```
function Find(values) {
  let retval;
  for (i = 0; i < values.length; i++) {
    let value = values[i];
    if (!value) {
      return retval ? retval : null;
    }
    retval = FormatValue(value);
  }
  return retval;
}
```

```
function FormatValue(value) {
  return `Value is: ${value}`;
}

console.log(Find(["muh", "buh", "wuh"]));
// "wuh"
console.log(Find(["duh", null, "guh"]));
// "duh"
console.log(Find([null, "yuh"]));
// null
```

… except when you do it right

- Only handle ASAP or last-chance
- Use to indicate to the program that a situation was
  - Unforeseen and therefore not coded
  - Foreseen, but beyond the scope of the executing code

Underestimating the Impact of Objects on Memory Management

Objects are a heap of trouble

Objection, hearsay

- Everything is an object – Java 1996
- Object-oriented programming offers a sustainable way to write spaghetti code. It lets you accrete programs as a series of patches. – Paul Graham 2004
- Actually, I made up the term "object-oriented", and I can tell you I did not have C++ in mind. – Alan Kay 1997
- Fowler’s law is invoked when you have a penetrating insight into object-oriented programming.

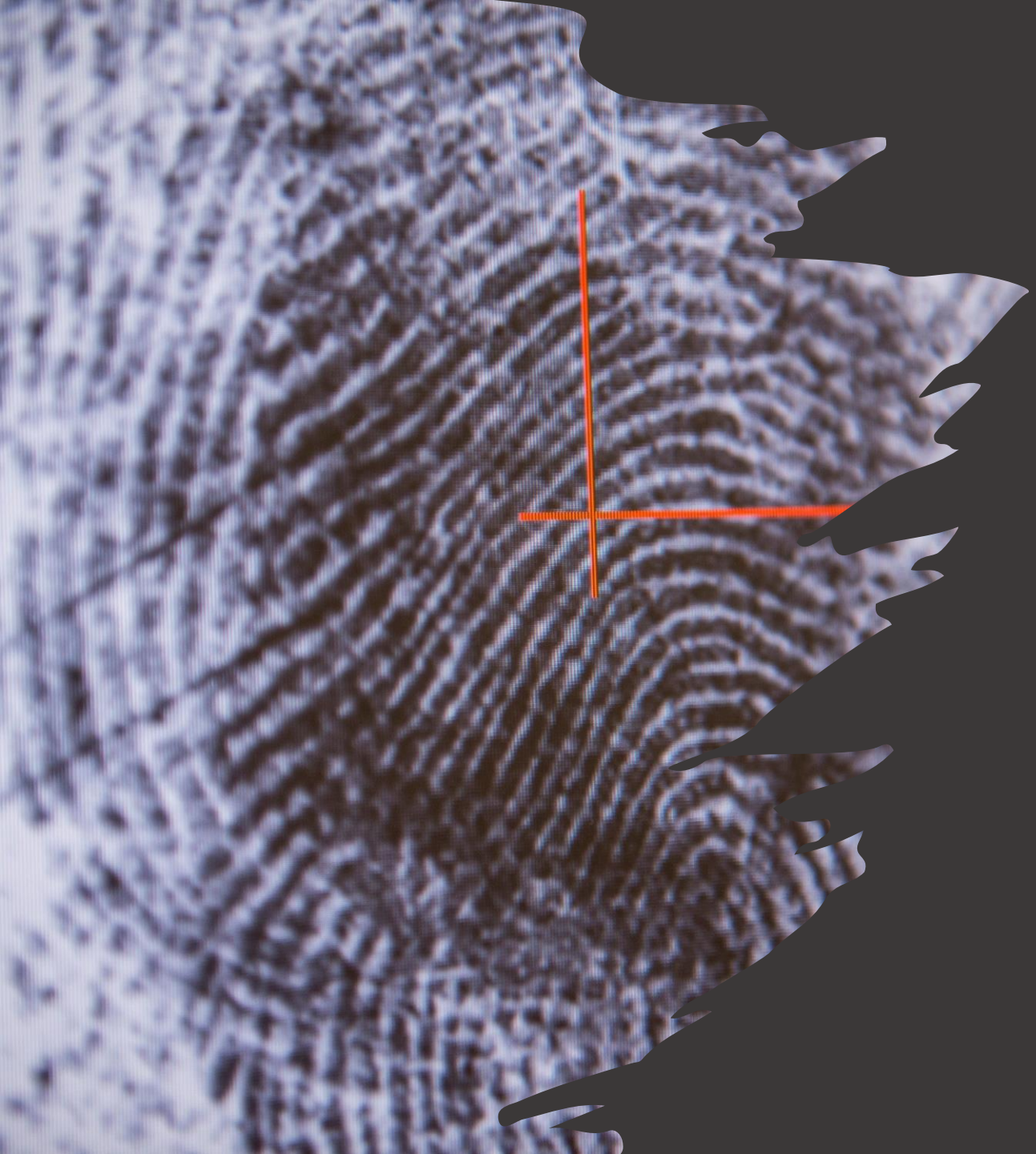
  If the quality of your insight is very high, you realise (*sic*) that Martin Fowler published the idea only five years ago. If the idea is poor, you realise (*sic*) that he published your idea more than 10 years ago… to demonstrate how many people still don’t get it. – nomorehacks 2009

Objection, irrelevant

- In languages where this is an issue, your use case may not demand optimization
- *Some* languages are evolving to move more state bags to the stack. Use with caution
- If you're not sure, side with simplicity, plan a load test, and let the profiler tell you what you need to do

Objection, speculative

- Garbage collection is your best friend and your worst enemy!
- In high scale systems, memory is the long pole in the performance tent
- Frequent garbage collection also taxes CPU



Objection, best evidence

- Don’t be afraid to use singletons where they make sense
- If you’re using dependency injection, take some time to understand the lifecycle of injected objects
- If your language offers structured value types (e.g., structs), use them when they make sense
- Transient objects should truly be transient so that your garbage collection can more easily free up memory
- Understand the consequences of primitive boxing

Examples: Everything is an Object!

- Read this blog post by Cameron McKenzie. He talks about the cost of autoboxing
  https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/Performance-cost-of-Java-autoboxing-and-unboxing-of-primitive-types

- A mountain a pile of heap of garbage to be collected

```
public string doThisIneffeiciently(int foo, boolean bar)
{
  return getResult(foo, bar, "steve");
}
public String getResult(int foo, boolean bar, String baz)
{
  return bar ? String.format("%s", foo) : baz;
}
```

Example: If Everything is an Object, Lean In

```
class FooHelper {
  boolean bar;
  int foo;

  public FooHelper(boolean bar, int foo) {
    this.bar = bar;
    this.foo = foo;
  }
  public String doThisIneffeiciently() {
    return getResult("steve");
  }
  private String getResult(String baz) {
    return bar ? String.format("%s", foo) : baz;
  }
}
```

Example: Or Just Get More Functional

```
private const string Steve = "steve";
public static string DoThisMoreEfficiently(int foo, bool bar)
{
    return bar switch
    {
        true => FormatFoo(foo),
        _ => FormatBaz(Steve)
    };
}

private static string FormatFoo(int foo) => foo.ToString();
private static string FormatBaz(string baz) => baz;
```

Strings are Objects…

… that are arrays of characters

… that all trace back to C

- A string in ANSI C

```
char string1[20] = {
   "A",
   " ",
   "s", "t", "r", "i", "n", "g",
   "\0"
};
```

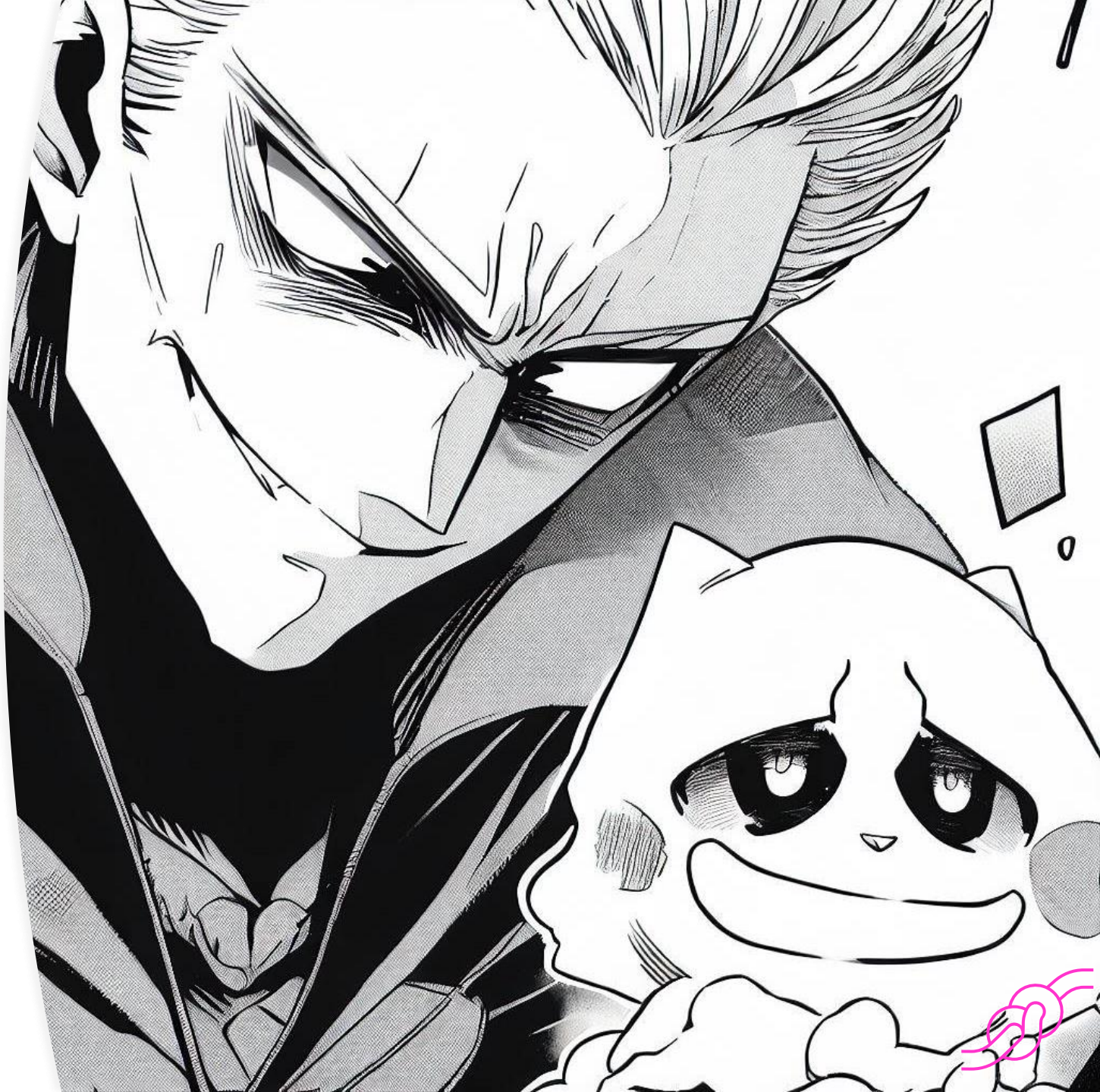
- Or

```
char string2[20] = "A string";
```

- #ThankYouAnsiC

… but Worse (and better)

- Modern languages optimize strings to make them immutable
- Every variable assignment to string allocates memory
- This is powerful in many cases, but can be *evil* if you're not careful

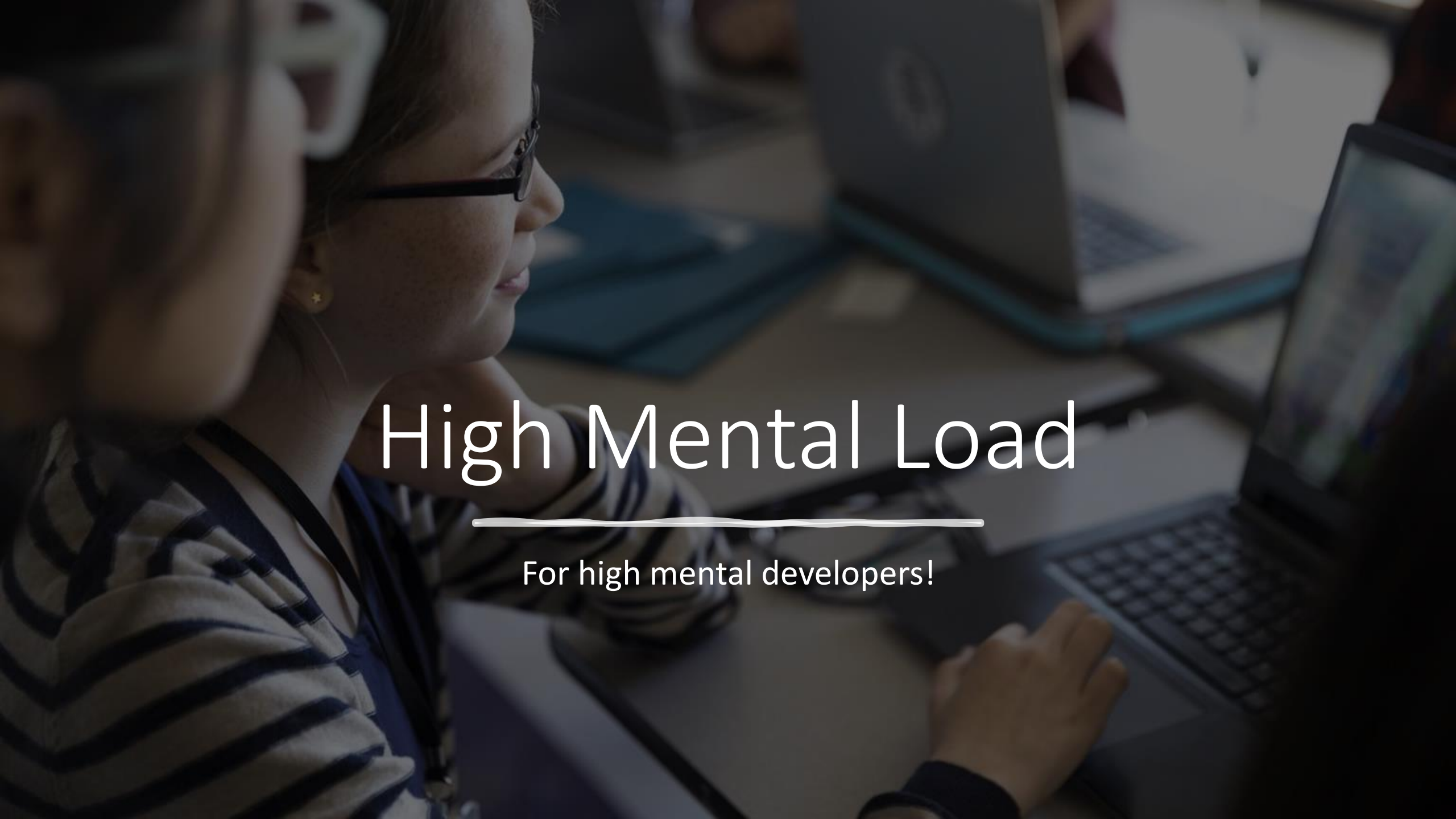


… that are Often Used Inefficiently

- Transitional states (string building)
- Frequent reassignment
- Not using symbolic constants
- Formatting

… that Need Some Love and Care

- Use a string builder if available
- Avoid creating unique strings in high volume methods/functions
- Don’t use strings to convey unique data or state unless they’re constants
- Be aware of the impact of logging on string creation



High Mental Load

For high mental developers!

Null is Never your Friend

Except when it is, but we'll talk about that later

Also… Except when cribbed from Tony Hoare's 2009 talk at Qcon (https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/)

background: null

- Called the “billion-dollar mistake” by Tony Hoare, who was credited for the creation of null.
- Originally, null was a pointer without a value
- Has evolved to mean “no value assigned” in memory managed languages
- Does not apply to value types (see section on objects)

problem := null

- Lacks any semantic meaning (other than that the pointer has no reference)
- Requires null checks everywhere
- What do you do if it’s null?

language.solve(null)

- Null vs Undefined
- Structs
- Compile-time null checking
- Discriminated unions

Traditional (pseudocode)

```
function get_book(id:number): book
  retrieve record from database by id
  if record does not exist return null
  else return book
end function

my_book = get_book(1)
if my_book is null print “book not found”
else print “book found”
```

Discriminated Union (pseudocode)

```
function get_book(id:number): no_book or book
  retrieve record from database by id
  if record does not exist return no_book
  else return book
end function

get_book(1)
  when no_book print “book not found”
  when book print “book found”
```

Discriminated Union(ish) C#

```
public OneOf<Book, NoBook> GetBook(string id)
{
    Book book = _repo.GetBookById(id);
    if (book == null)
    {
        return NoBook.Instance;
    }
    return book;
}
var result = GetBook(1);
Console.WriteLine(result.Match(
    book => "book found",
    noBook => "book not found"));
```

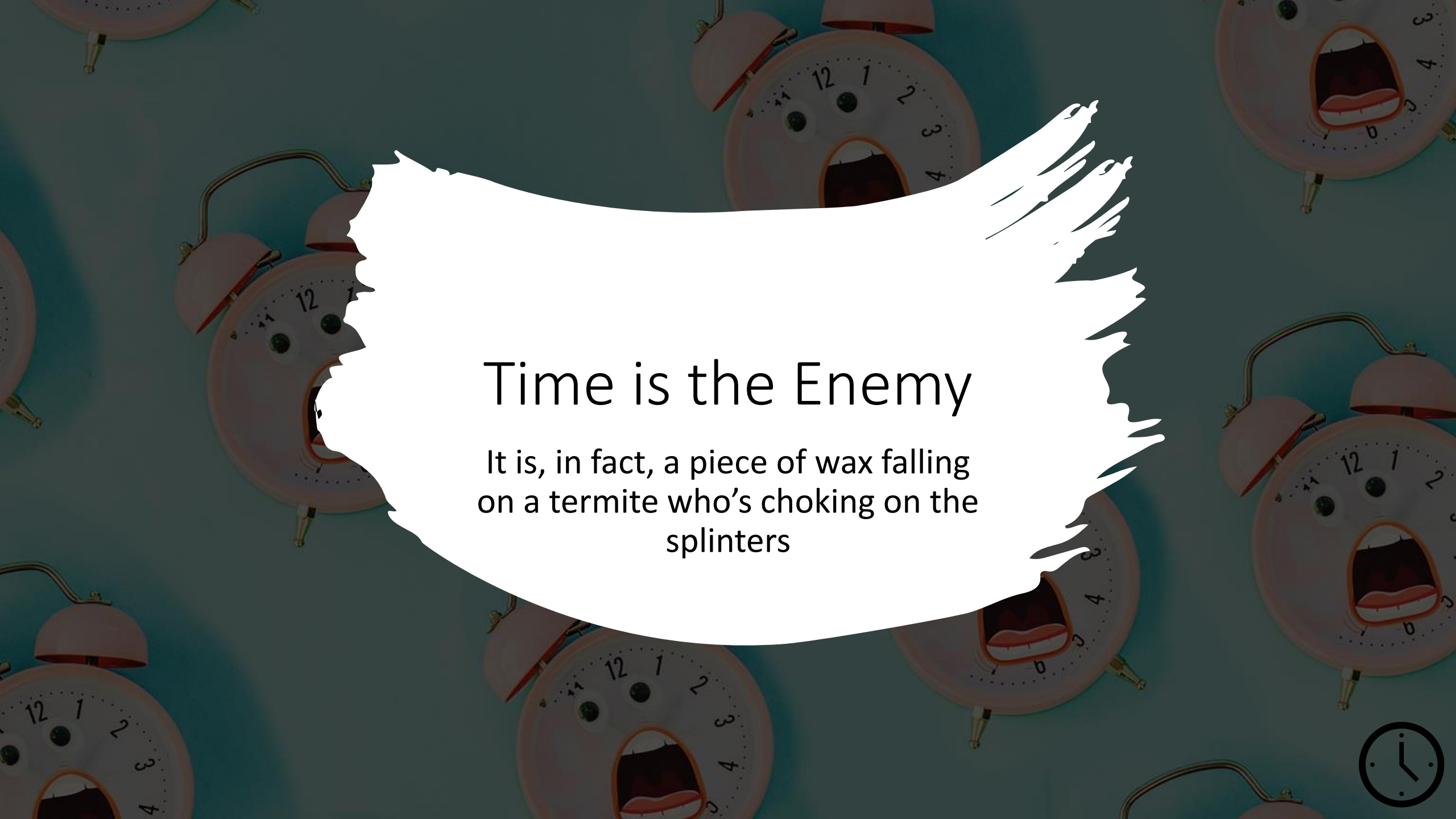
null < best_practices

- Use what the language brings
- Check for nulls at system boundaries, avoid them internally
- Interrogate and remediate every line of code that returns null
- Assign default object references instead of null (see null object pattern)

Null Object Pattern (c#)

```
public interface IBookshelf
{
    Book FindBook(string id);
}

public class NullBookshelf : IBookshelf
{
    public Book FindBook(string id) => Book.NoBook;
}
```



Time is the Enemy

It is, in fact, a piece of wax falling on a termite who’s choking on the splinters

Is [not] On My Side

- Instant in time
  - Impossible when accounting for Clock Skew and Relativity
- An instant in time at a specific place
  - Geographic time
- An instant in time in a specific time zone
  - Time zones are political
- An instant in time as an offset of UTC
  - DateTime with offset
- Conceptual time
  - Loose communication of time

Keeps on Slippin’ Slippin’

- We rely on our intuition to reason about time
- We don’t have a good [human] language for expressing time precisely
- Most programming languages/frameworks do a terrible job clarifying
- Time zones are political
- Cultural differences in things like age calculations

Does Anybody Really Know What Time it Is

Commissions are paid every Thursday

- Which time zone? UTC, head office TZ, employee's TZ

That occurs daily at exactly 4:45AM EST

- Historically, this is a mess because geographic affinity to a time zone changes over time

First commit wins

- Clock skew and relativistic time make this an ugly mess

That transaction happened at 01:21 -07:00

- Offsets are no better than UTC when stored on disk. The further ahead or behind we do math on this, the less likely it is to be relevant to the observer who recorded it (i.e., time zone changes, daylight saving)

Children can create an account when they turn 13

- Under which reckoning (e.g., Western, East Asian)

Waiting for someone or something to show you the way

- Within systems, use epoch timestamps (long integers) if you're able
- Always store time as UTC or epoch timestamp (long integer)
- Time zones are a localization concern and should be left to the front-end
- If needed, store event Time Zone information as separate field with the record
- Never use offsets
- Don't underestimate the complexity of dealing with time in your designs

Logging and Logging and Logging

{"date":"05/05/23 12:45:56","severity":"ANY","message":“starting logging"}
{"date":"05/05/23 12:45:57","severity":"ANY","message":“logging"}
{"date":"05/05/23 12:45:57","severity":"ANY","message":“finishing logging"}
{"date":"05/05/23 12:45:57","severity":"INFO","message":“logging complete"}

[12:45:12 INF] We need to log this

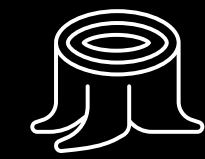
[12:45:50 WRN] Troubleshooting a specific issue

[12:51:30 INF] Capture context for inspection

[12:49:00 VRB] Trace a request across services

[12:55:02 FTL] Build metrics (after the fact)

[12:47:12 ERR] Log that one error in every method in the stack



[13:05:00] TRACE Have a Plan

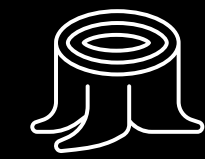
[13:05:00] WARN Log consistently (not profusely)

[13:05:01] ERROR Don’t log errors reflexively

[13:05:01] INFO Think about performance

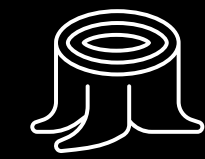
[13:05:01] INFO Use log context scope rather than formatted log message

[13:05:05] TRACE Event ids > message

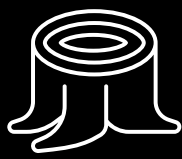


Don’t Count Your Eggs (Unless They’re Aggregated)

```
{
  "series": [
    {
      "metric": "pay_attention",
      "points": [
        {
          "cheaper_than_logging": 1,
          "proves_system_function": 2,
          "fancy_gauges_look_cool": 3
        }
      ]
    }
  ]
}
```



- Traces and distributed traces provide far better visibility across large codebases or multiple services
- Flame charts make it easier to spot bottlenecks
- A trace is the “vertical” execution – a stack of dependencies
- A span is the “horizontal” execution – the traces over time
- Most modern runtimes allow transparent configuration of tracing



You Don’t Have to Reinvent the Wheel

- Most application performance monitoring (APM) products provide agents that can inject metrics and traces
- OpenTelemetry is on the rise
- Many packages exist to help minimize the mental load

https://m.eado.ws/s/2023/stirtrek

Conclusion

Also, DALL-E doesn’t always hit the mark.