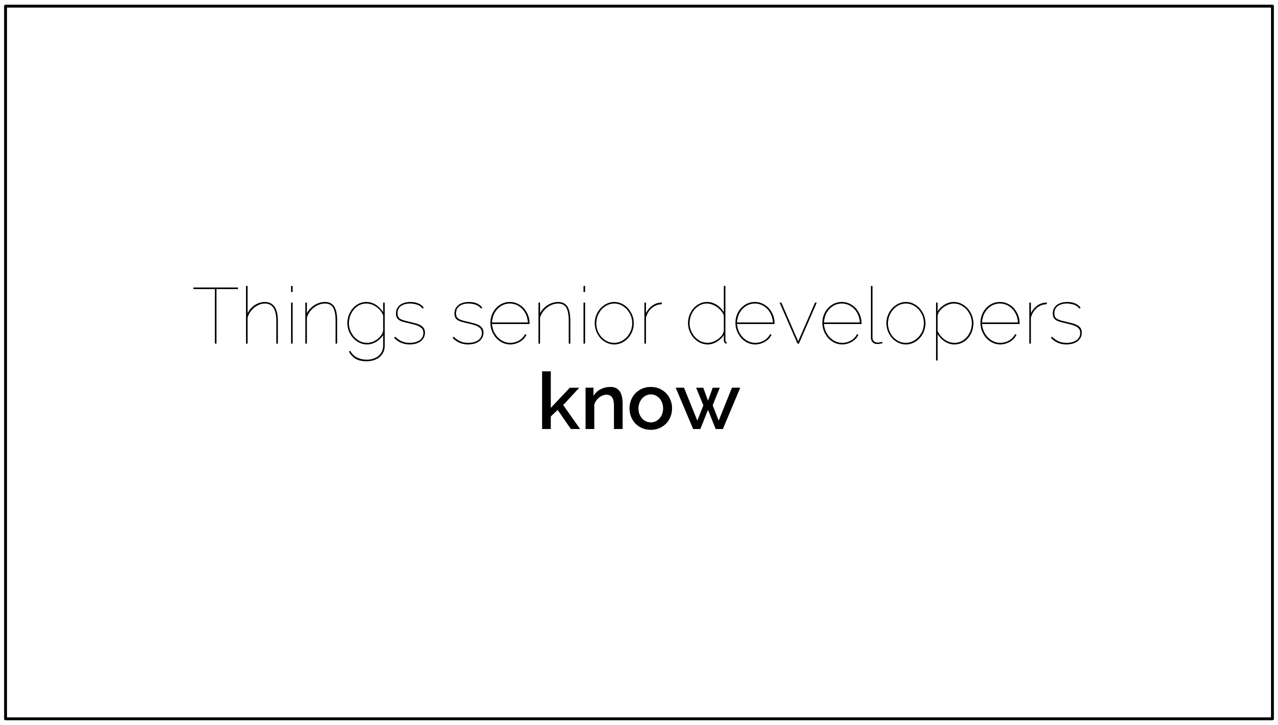
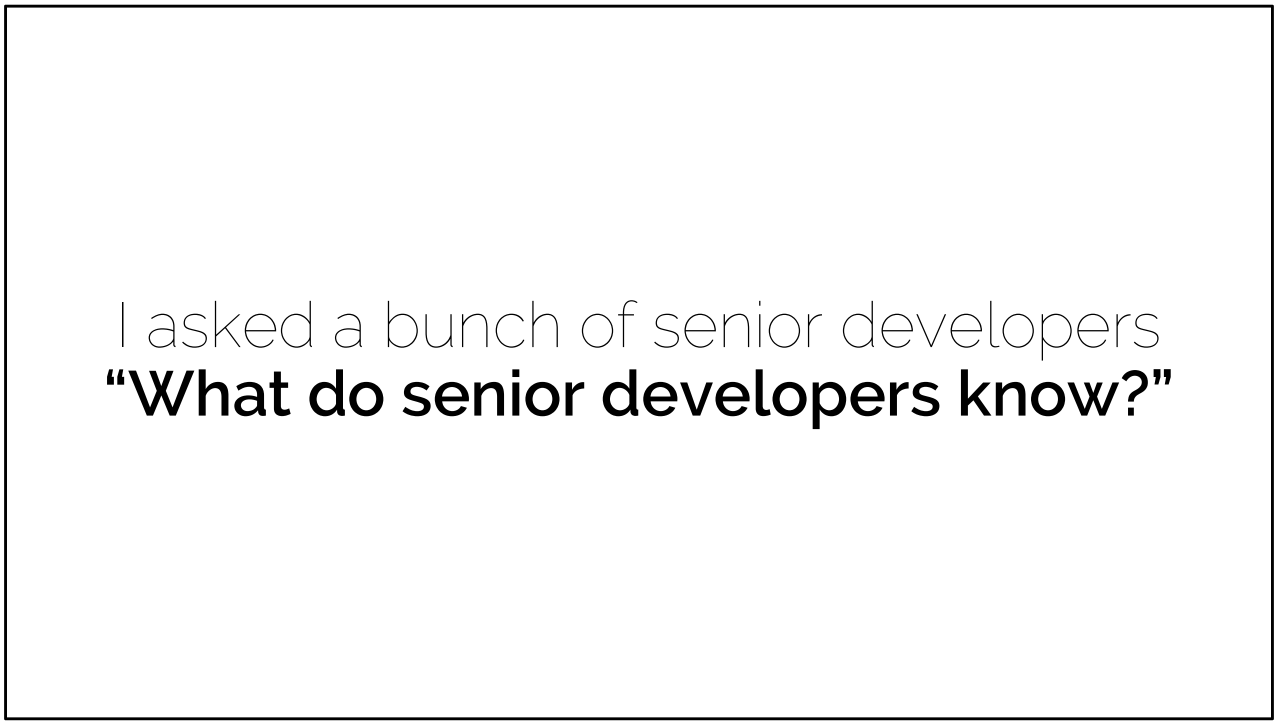


Things senior developers **know**

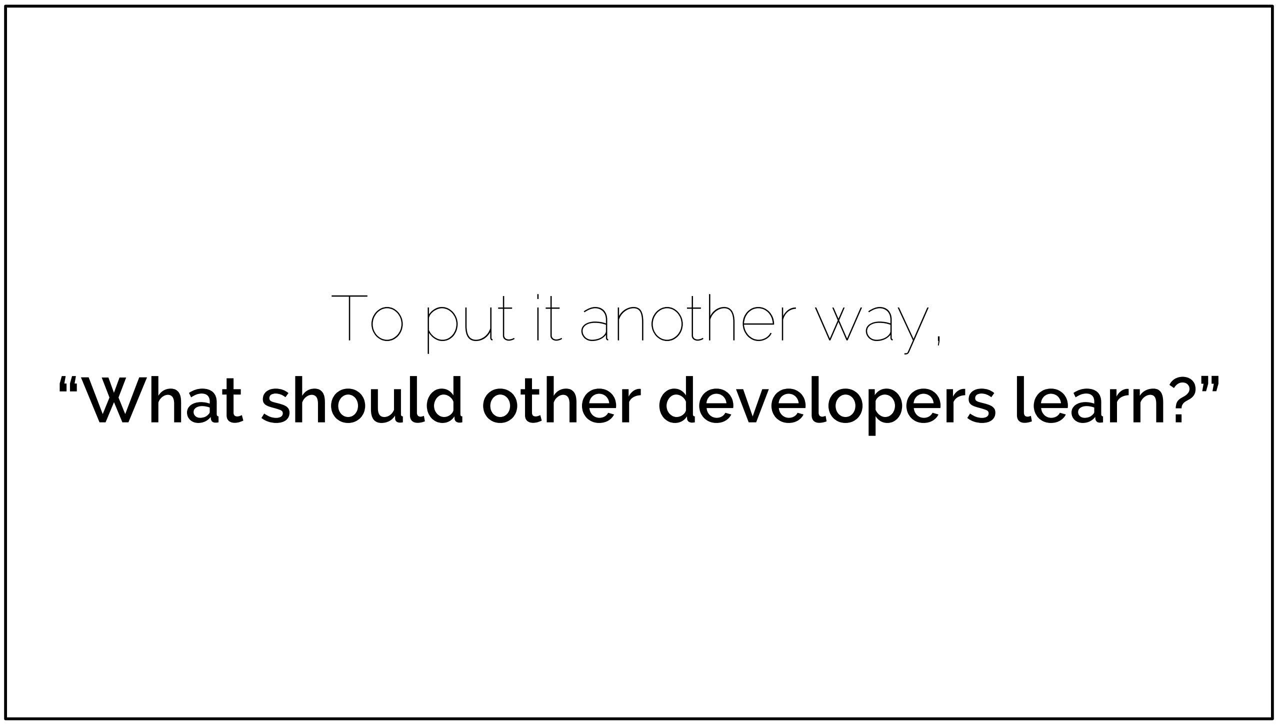


I asked a bunch of senior developers

“What do senior developers know?”

What do Senior Developers know that junior developers don’t

Because a Senior Developer isn’t just a junior developer with more expertise, there’s a qualitative difference



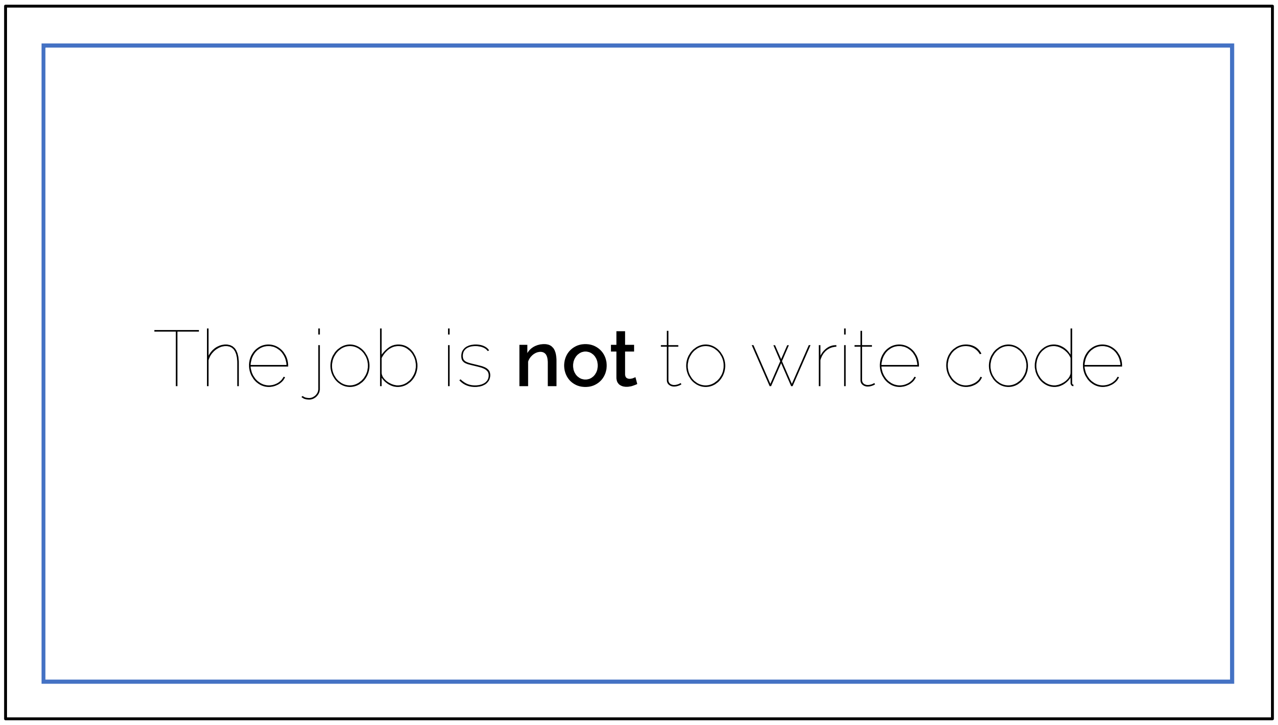
To put it another way,

“What should other developers learn?”

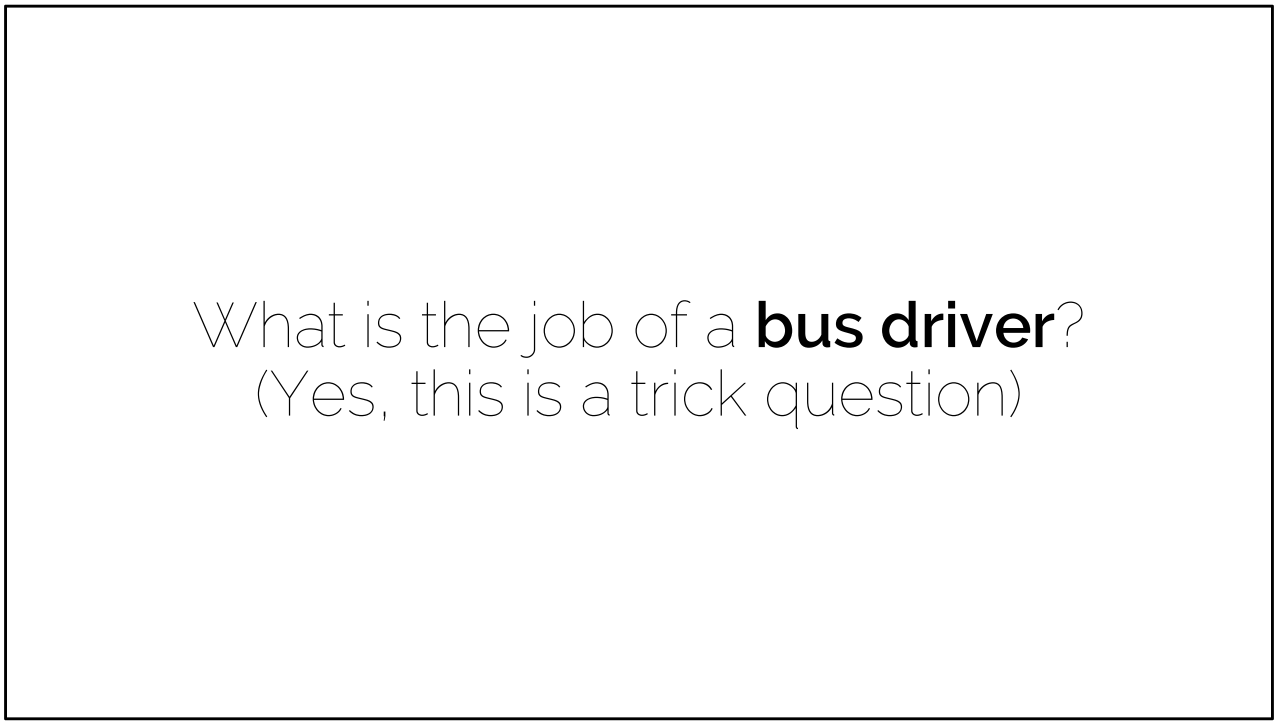
I’ve seen too many good developers leave the profession to do something else after 5-7 years because they plateau and run out of challenges.

Yet, I’ve been writing code for over 30 years and I’ve never been learning so much – or having so much fun in my job.

Reverend Billy Hollis – has said that he will one day be discovered with his nose between the keys because it’s just too much fun to build new stuff.


The job is **not** to write code

Much as we like to write code, that’s not the job



What is the job of a **bus driver**?
(Yes, this is a trick question)

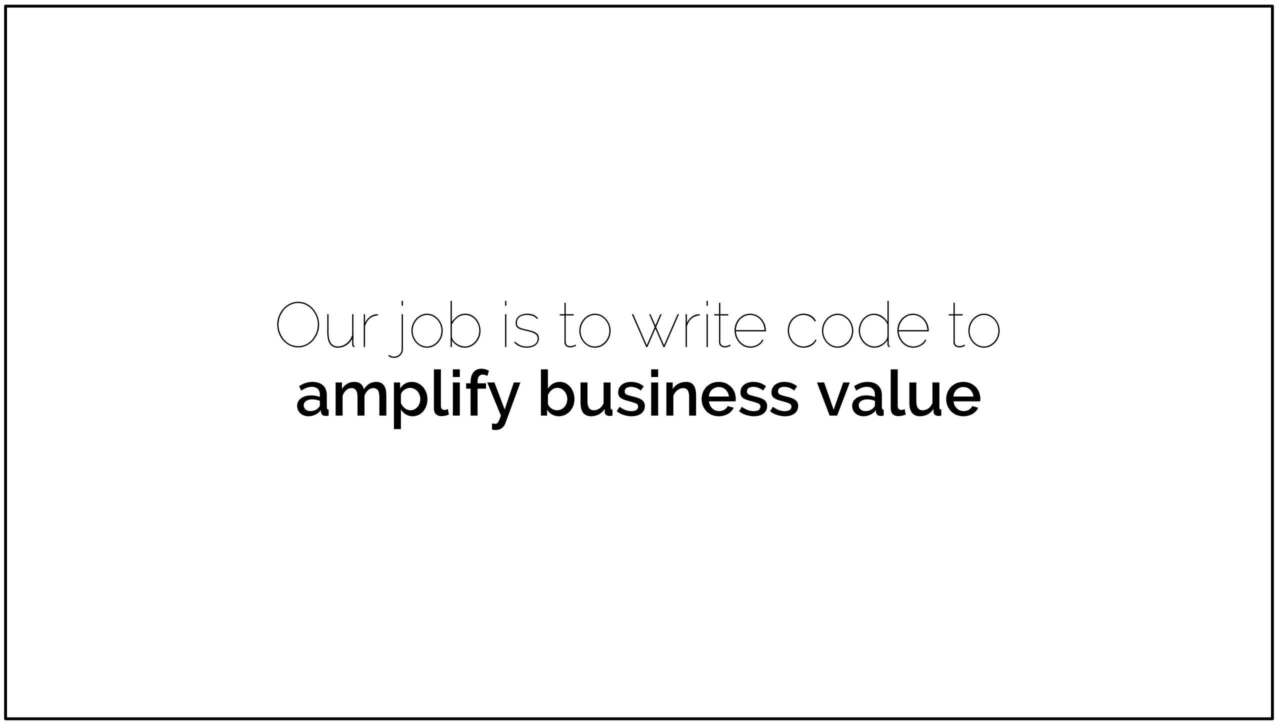
A bus driver’s job isn’t to drive a bus.

A bus driver who drove around all day without picking up passengers isn’t doing his job.

A bus driver who drove around picking passengers all day without letting them off isn’t doing his job … and would likely end up on the news.

A bus driver’s job is **to help people get where they need to go.**

This is why a bus driver who is 2 minutes early is really not doing his job.

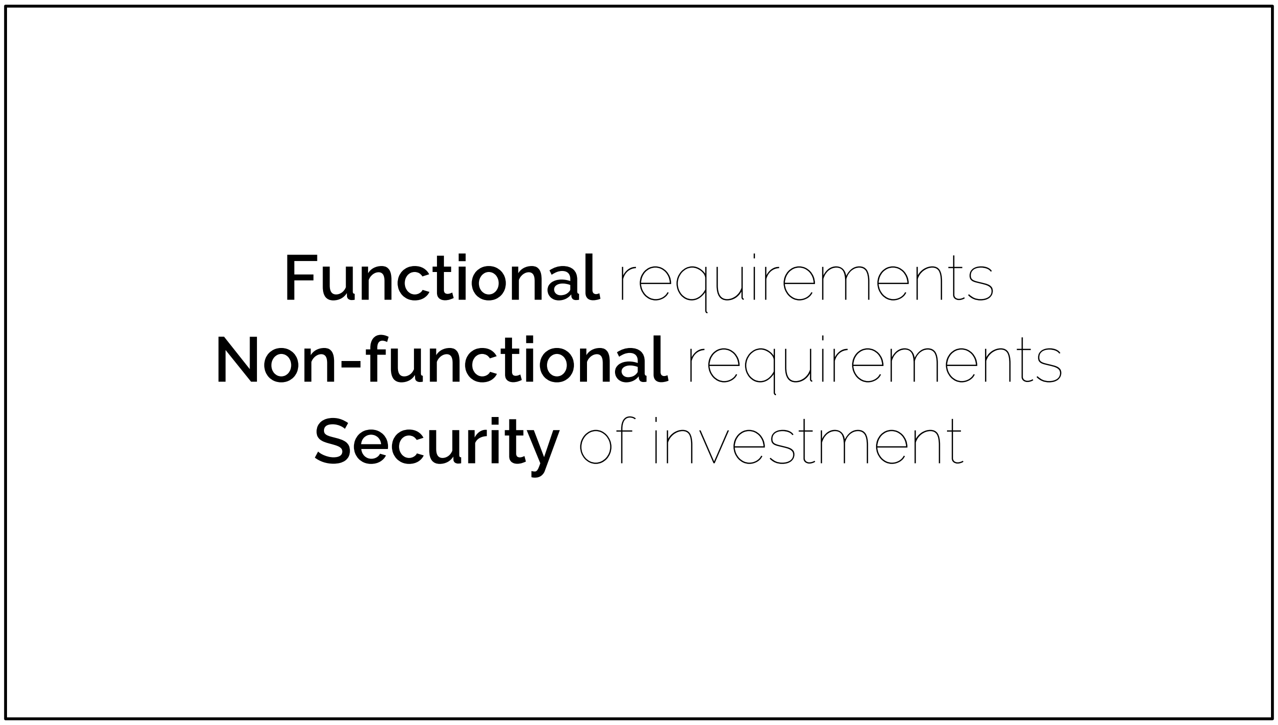

Our job is to write code to
amplify business value

Writing code is fun.

But if the code doesn’t impact on business value, it’s a waste of time

Developers are expensive (figure 3 x Annual Salary for an approximate all up cost, including support staff, equipment plus rent/heat/power etc)

No one is going to pay me to write code that doesn’t have a significant impact on the business



Functional requirements
Non-functional requirements
Security of investment

Three different areas to consider

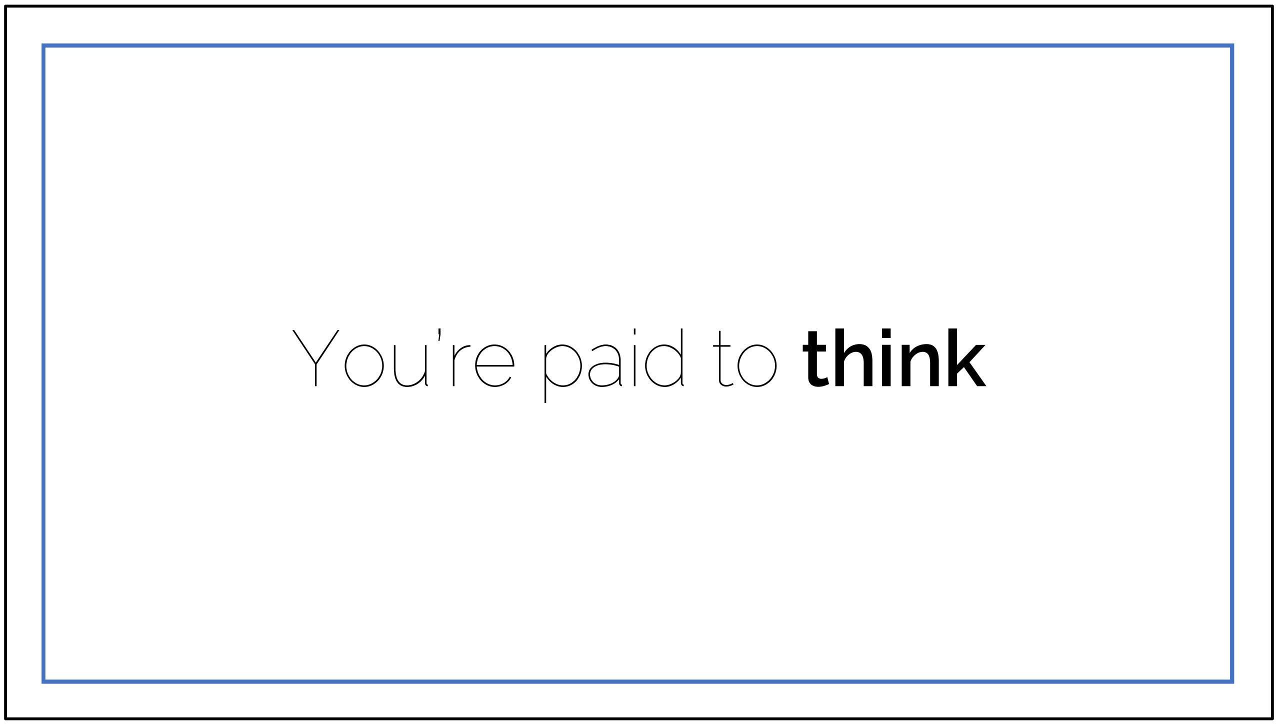
People think that Functional + Non-Functional cover everything
But that ignores the passage of time

We are building a **business asset**

Code doesn't go away when we release it.

In fact, that's when it starts delivering value.

Finished features that aren't in production are inventory

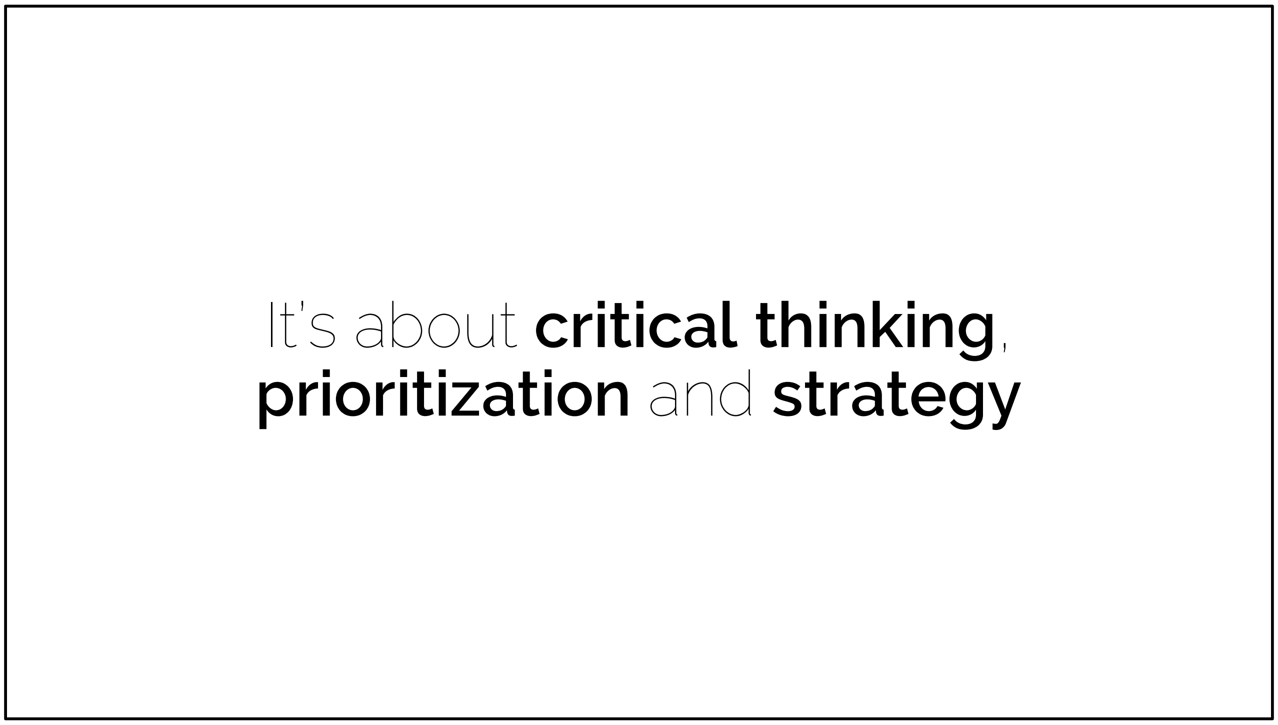


You're paid to **think**

This means thinking clearly, logically and well

It also means being able to clearly articulate what you think, to make a persuasive argument

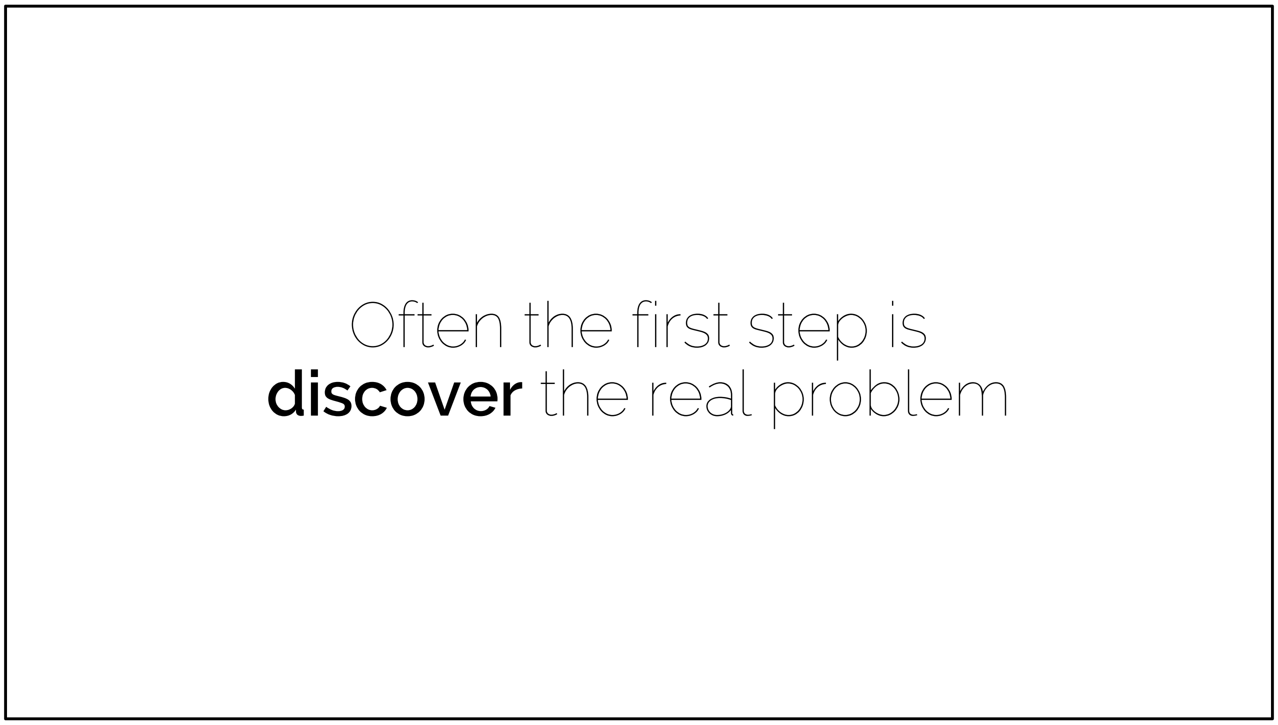
It's not about being a **code monkey** churning out working code



It's about **critical thinking**, **prioritization** and **strategy**

You need to think carefully about what you do and deliver what the business really needs

Sometimes this involves refusing to do the wrong thing



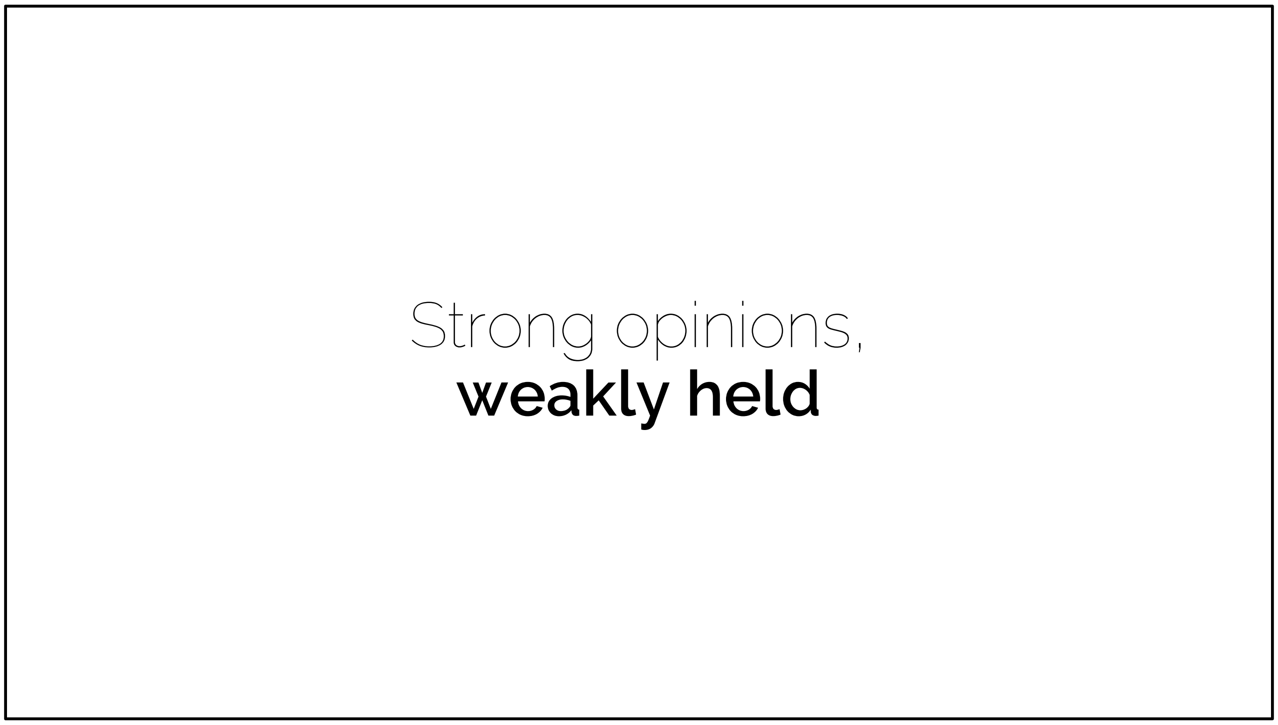
Often the first step is
discover the real problem

Users are really good at coming to us with solutions

But their ideas for solutions are constrained by what they know and believe about technology

Once we discover the real problem, we can provide better solutions

Junior Developers are asked “Please implement this design”
Intermediate Developers are asked “Please build me an X” (e.g. Website)
Senior Developers are told “I have this pain …”



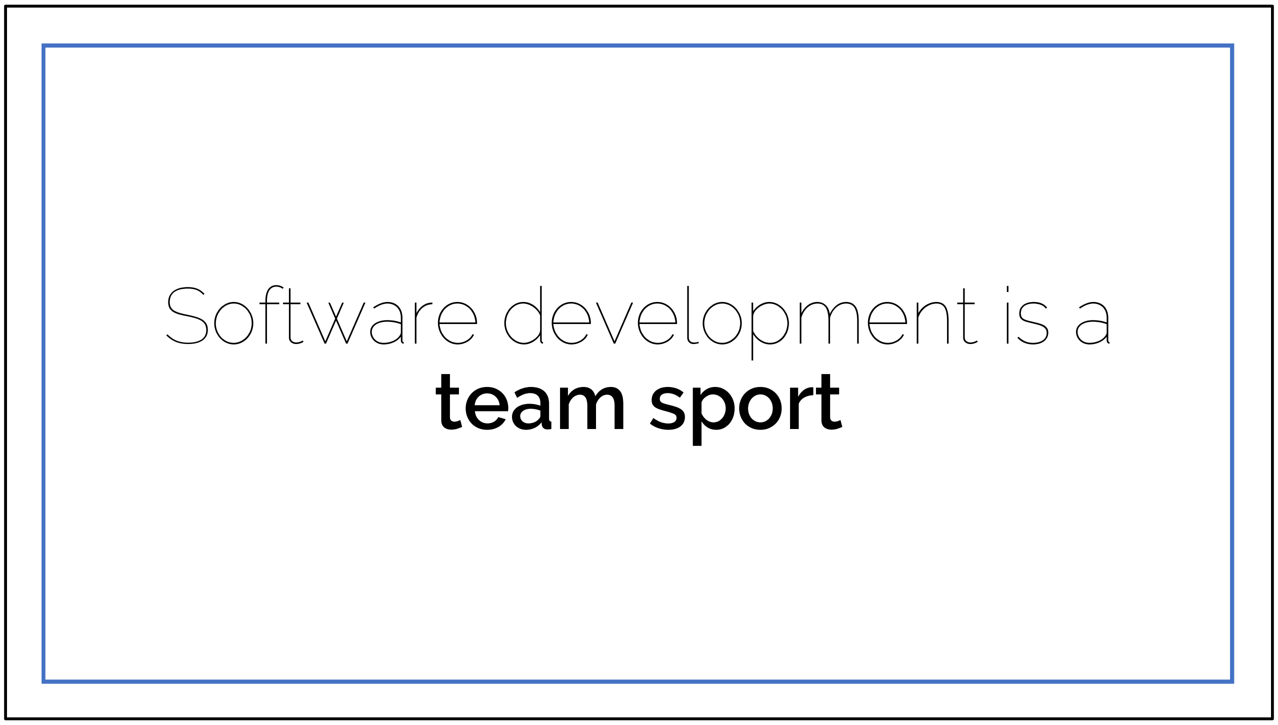
Strong opinions, **weakly held**

Know the evidence for the opinion you have
Be prepared to argue for it – to defend your point of view

BUT NEVER STOP LISTENING

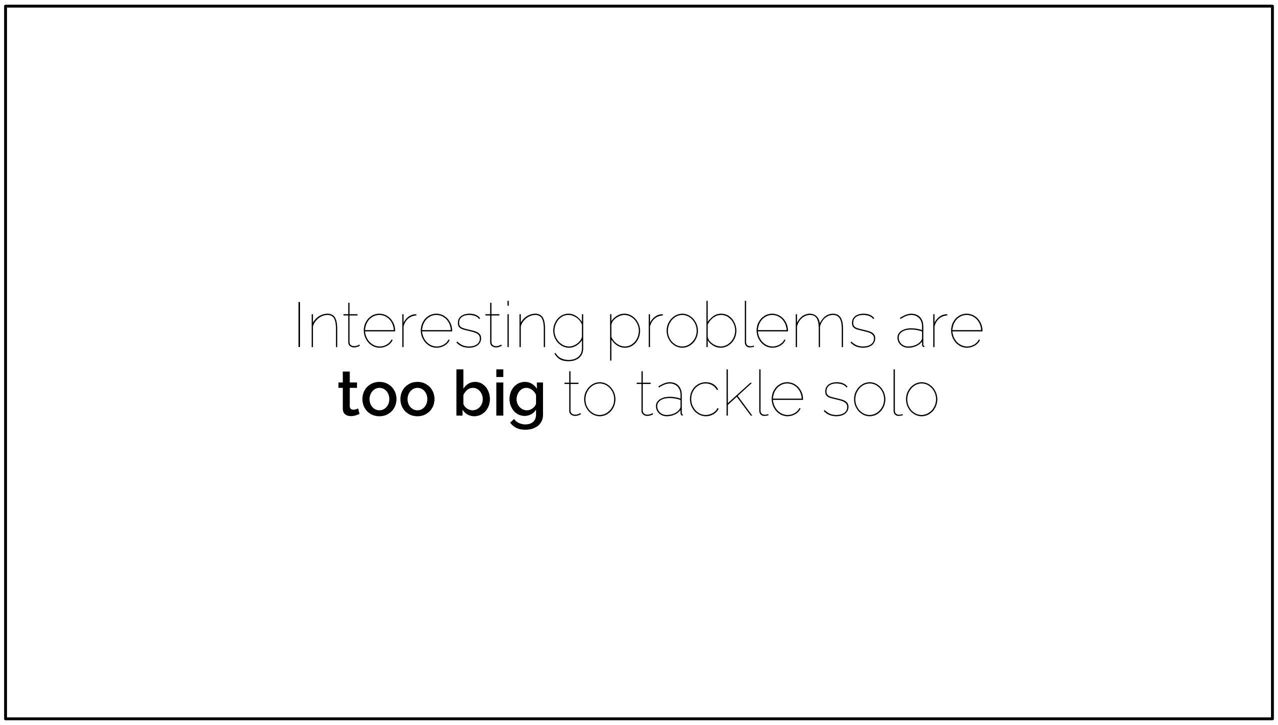
And if the evidence for an alternative position is stronger than your own
Be prepared to change your mind

But, if you pivot in the middle of a meeting, be prepared to for resistance.
Ask me how I know



Software development is a
team sport

Odd for an industry where the (admittedly fading) stereotype is of people with no social skills lurking in basements

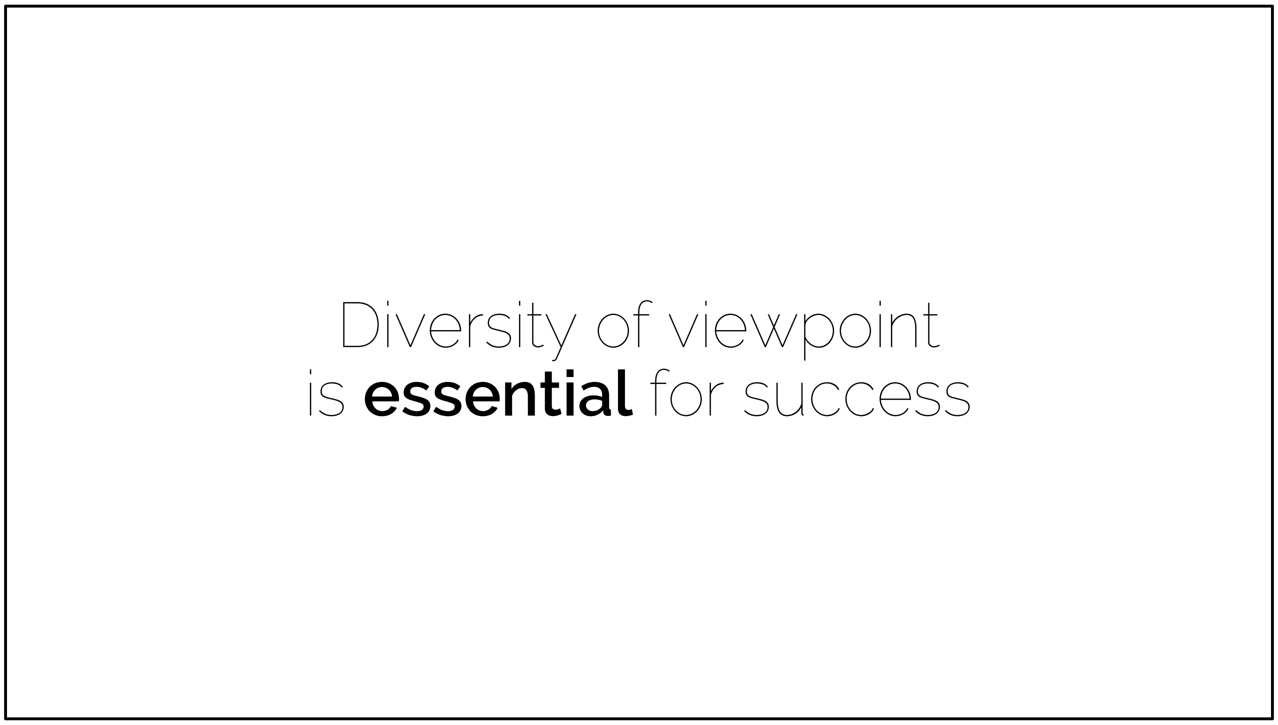


Interesting problems are
too big to tackle solo

The average developer delivers 1500 lines of working code into production each year

Interestingly, this is pretty constant regardless of the choice of language: Assembler, C#, Ruby, Scala, Go, Rust, Java or Prolog

Even if you're WAY above average and deliver 10,000 lines in a year … you're not going to build a 600,000 line system by yourself.



Diversity of viewpoint
is **essential** for success

Hiring a team of people just like yourself doesn't work.

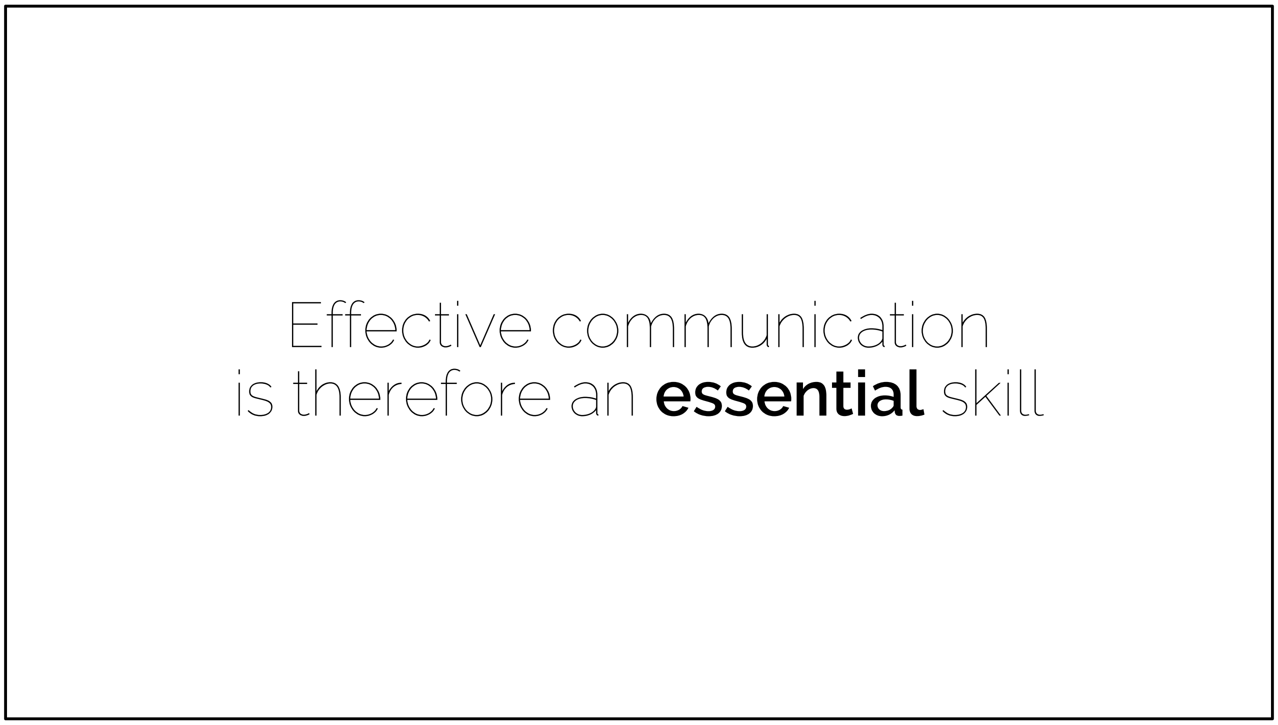
If the whole world were like me, it would be a very strange place.

This isn't just a good idea, but an actual fact backed up by proper research.
(Not an alternative fact)

Not everyone on the team
will be **technical**

This means learning to talk with different people in different ways

Learn about the Myers-Briggs type indicator, about Belbin team profiles, about the Kolbe Conative Index

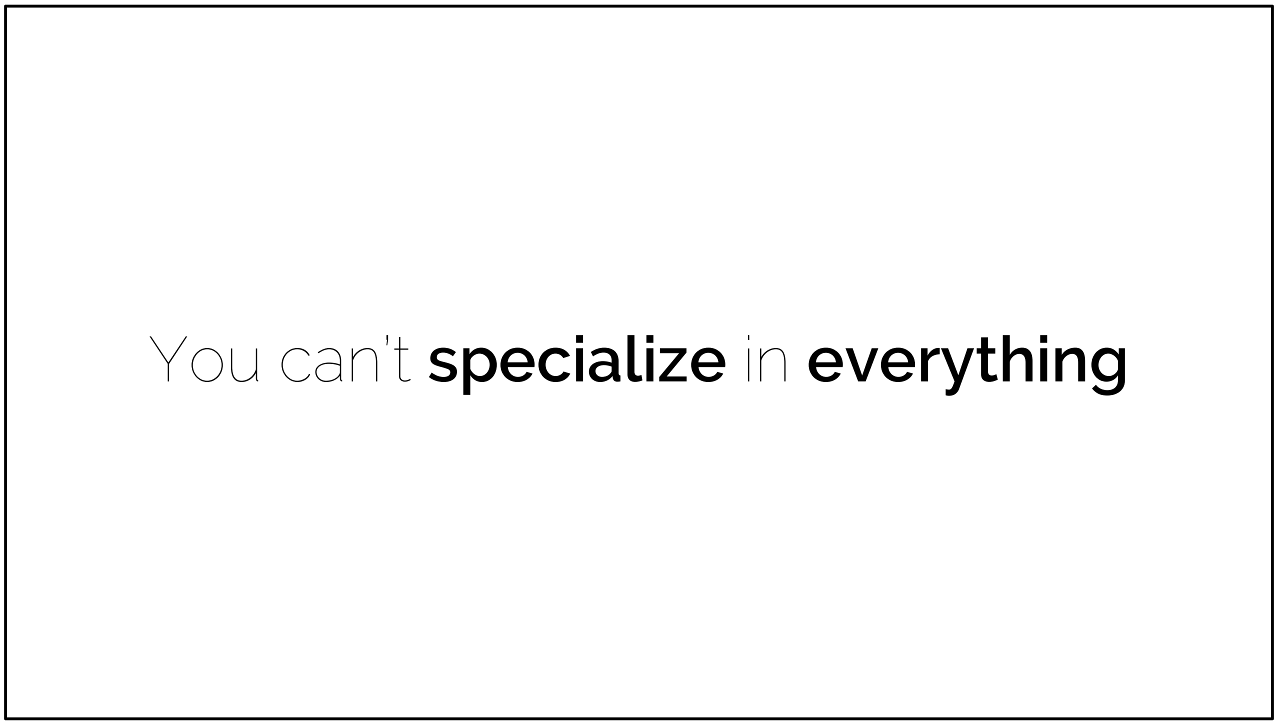

Effective communication
is therefore an **essential** skill

Key point – it’s not good communication unless the other person understands

Any judgement about the quality of your explanation is not yours to make

Military approach “Brief Back”
Instruct, listen to student’s explanation, correct

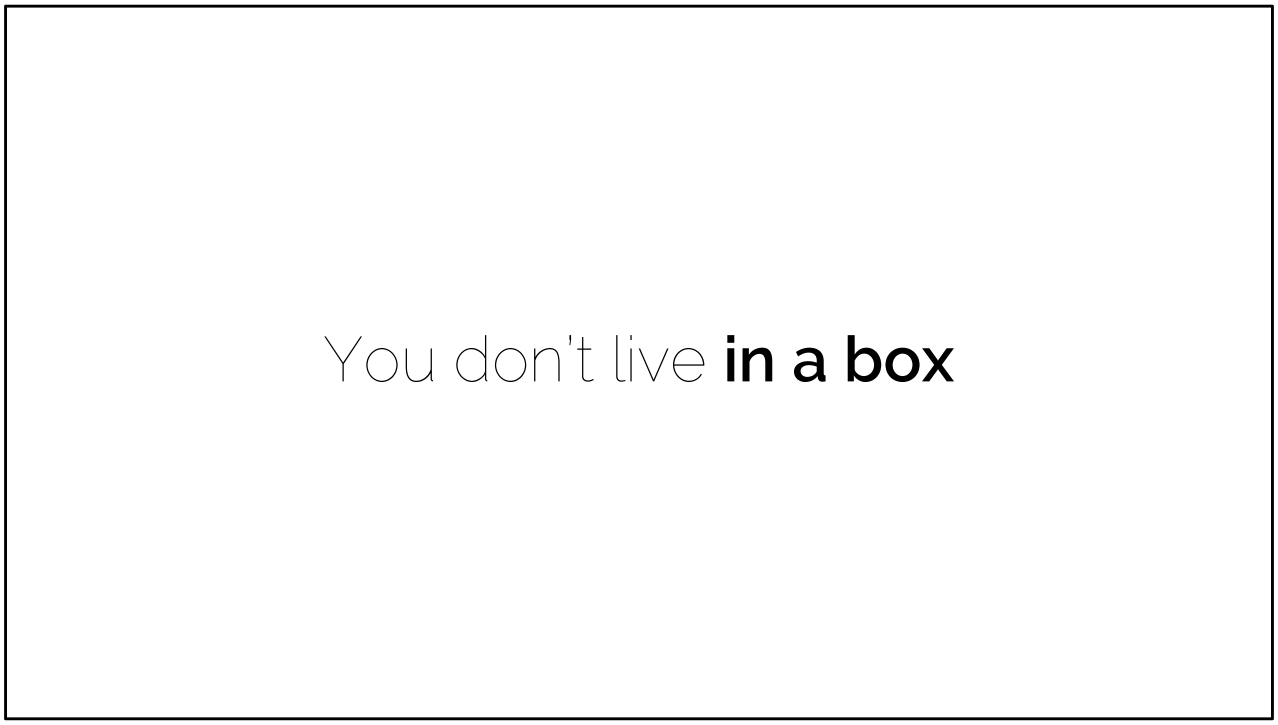
Avoid “Yes, yes, I understand”



You can't **specialize** in **everything**

But trying can be a lot of fun

There is no magic
Only a lack of time to understand



You don't live **in a box**

You have to be aware of what’s going on around you

Doing your job well means making it easier for other people to do theirs

This is where devops comes from

But there’s more to it too.

No matter how careful you are, you will make **mistakes**

Coding conventions
help us **avoid** common mistakes

By writing down the known good ways of doing things, we can avoid mistakes we've seen before

Code reviews help us **find** mistakes

Other people – with fresh eyes – will spot things we don't.

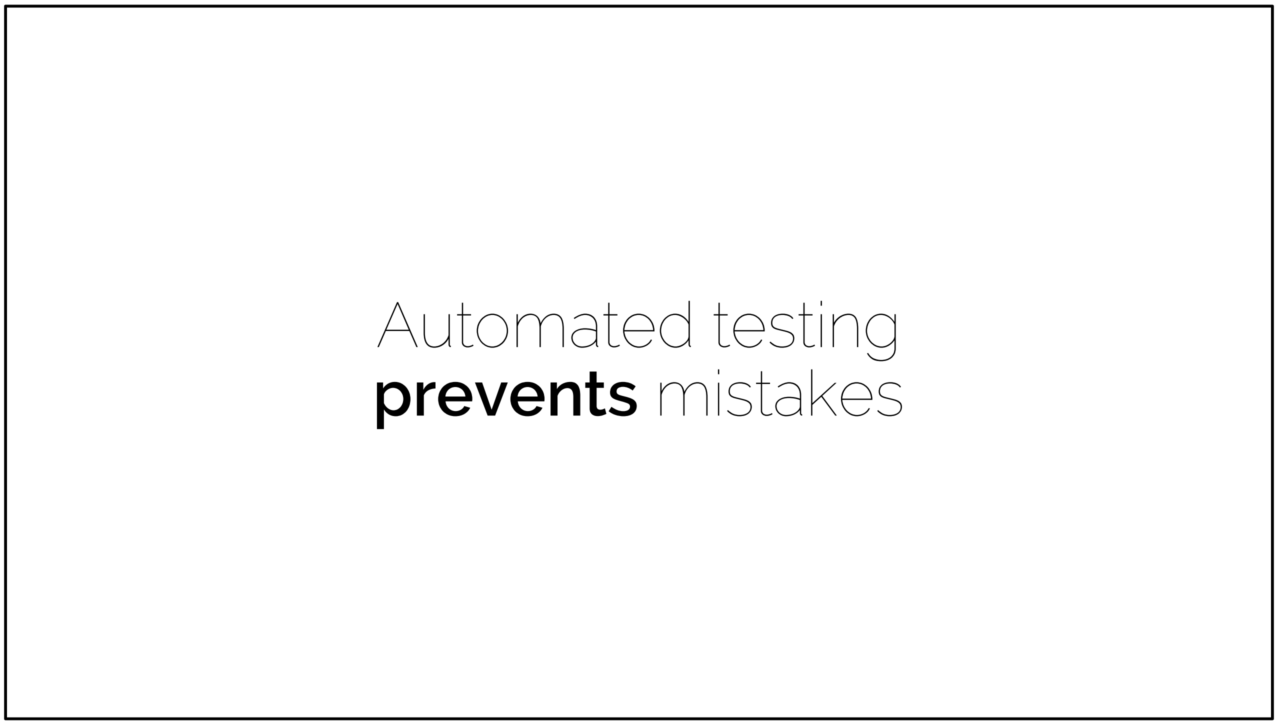
Code reviewing is a learned skill, persist with it.

It can be hard to have other people criticising your code, but as long as their intention is to improve the code it's all good

Static code analysis tools
detect potential mistakes

Source code control helps us **recover** from mistakes

The big red UNDO button that takes us back to what we had last week … or to what’s currently in production


Automated testing
prevents mistakes

Tests specify the behaviour the system should exhibit and dynamically check it

If the tests fail, either the system is broken or the ~~test~~ specification needs updating

Continuous Integration
alerts us to mistakes early

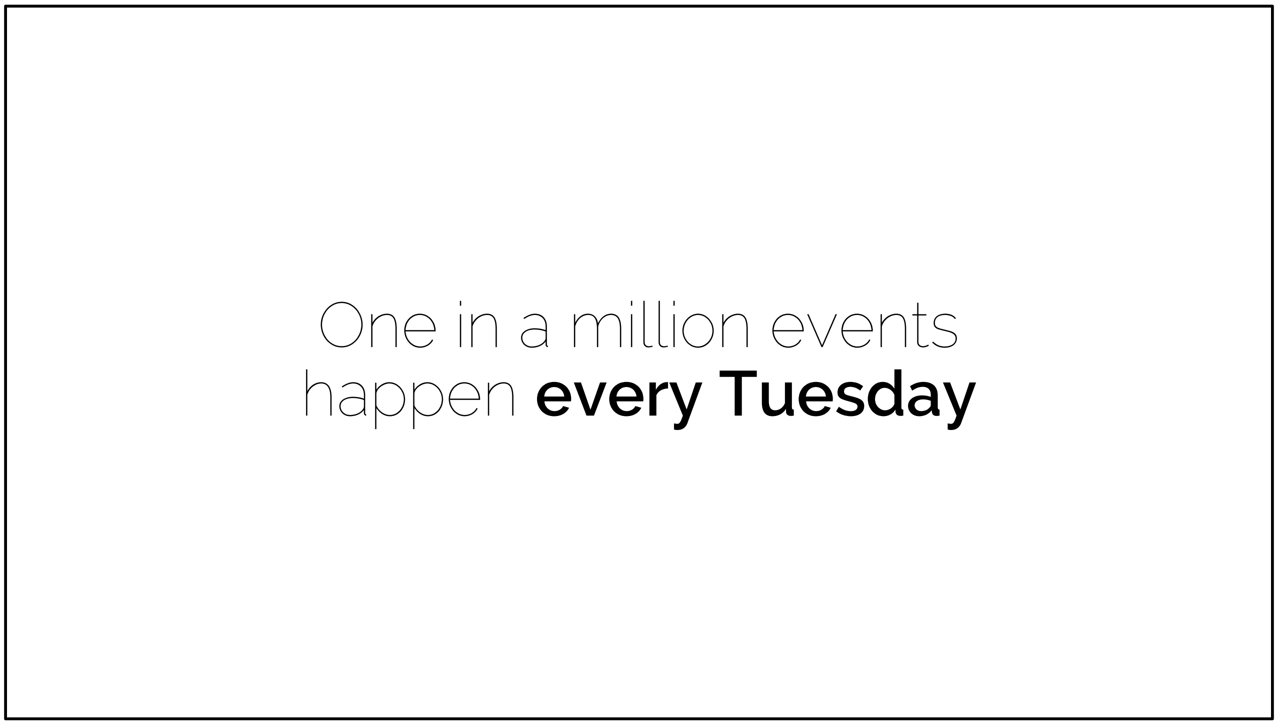
Run all of the analysis and tests frequently

Continuous Deployment helps us **fix** mistakes quickly

If it takes six months to deploy a fix, we're stuck with it for ages, so the consequences of a production release with a bug are high

But if we can deploy a fix 20 minutes after we confirm a fix for the problem …

The **impossible** happens



One in a million events
happen **every Tuesday**

That thing you were told would NEVER happen

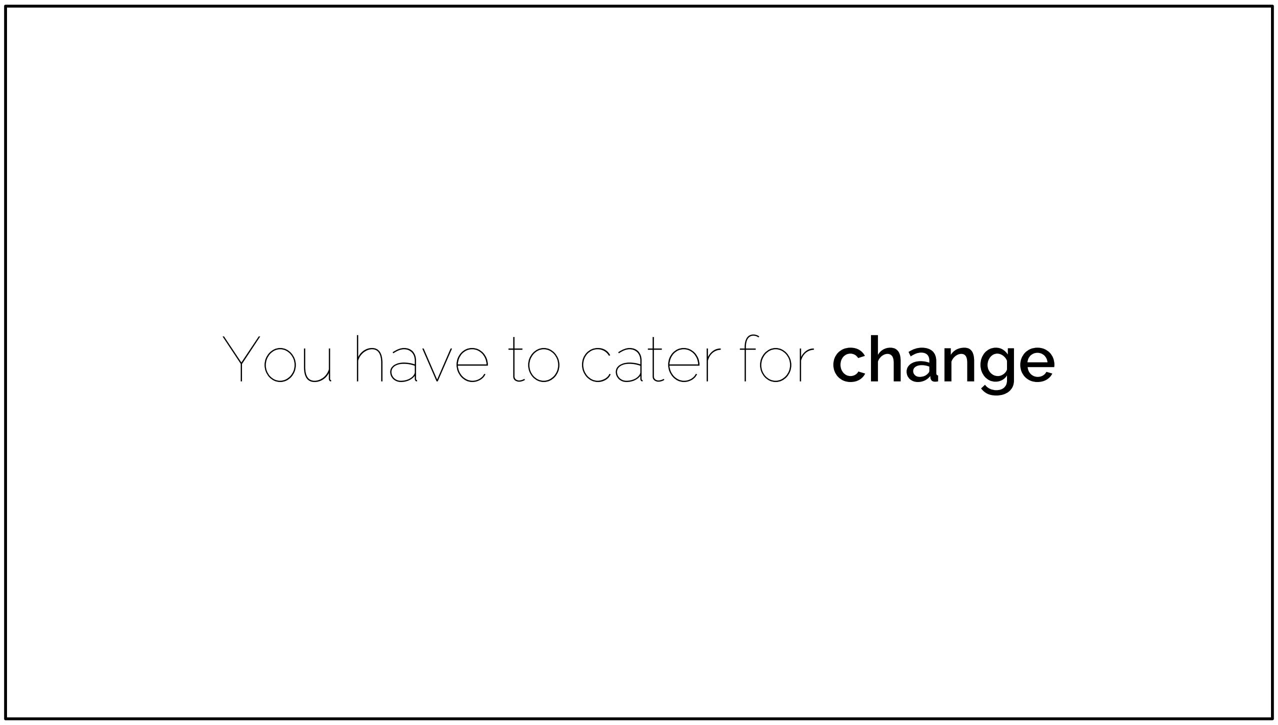
Yeah, it happens every Tuesday

Leave a **good looking** corpse

When things go wrong, leave enough debris lying around that someone can diagnose the problem

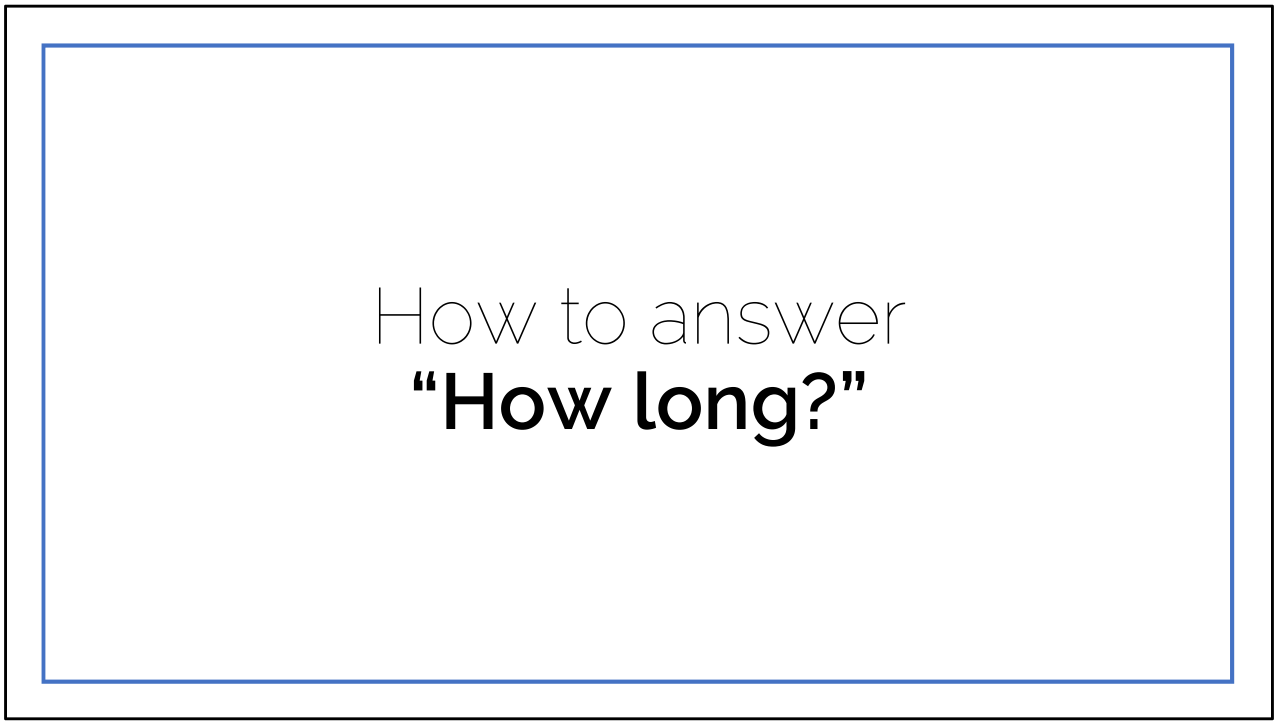
When your program is going to die anyway, the least it can do is tell you who killed it

The WinForms app with a dynamic UX that crashed only when showing a large survey form.



You have to cater for **change**

As far as possible, try to avoid backing yourself into a corner



How to answer
“How long?”

The difference between a **prediction**, a **target**, and a **commitment**

A **prediction** is based
on everything you **already know**

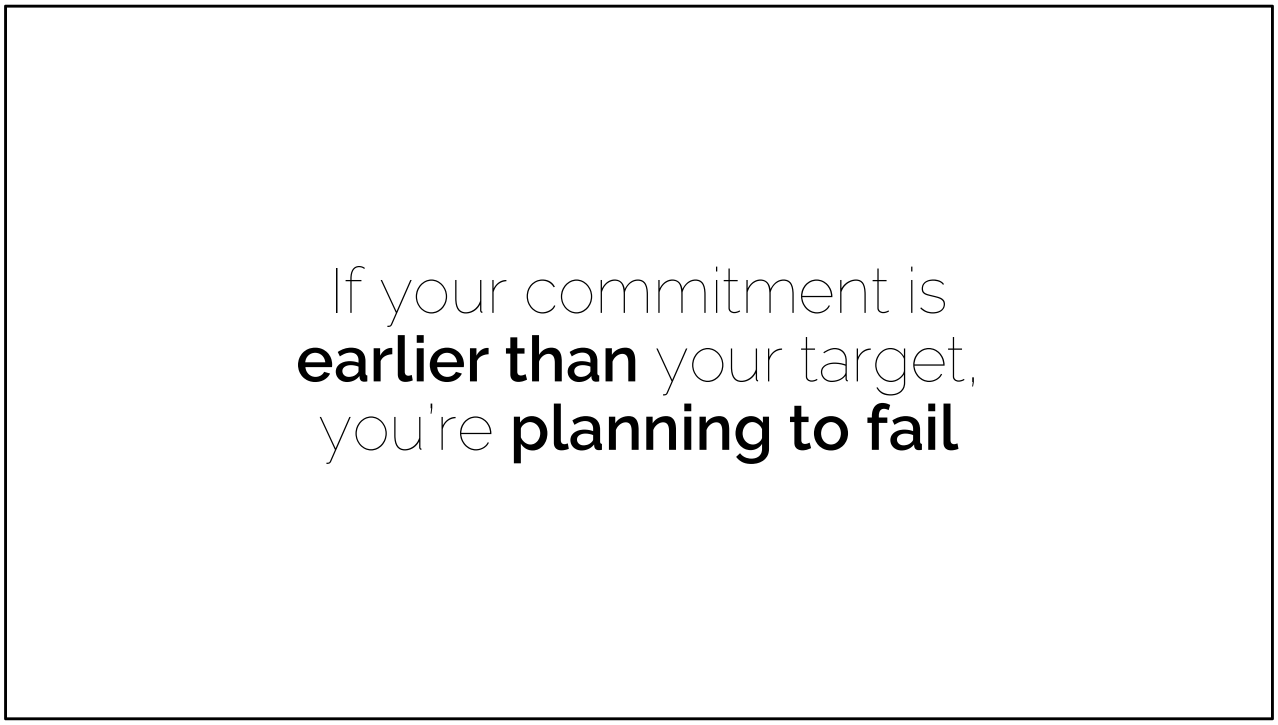
A **target** is when
you'd like to **finish**

A **commitment** is what
other people use in their plans

If your target is
later than your prediction,
you're **conservative**

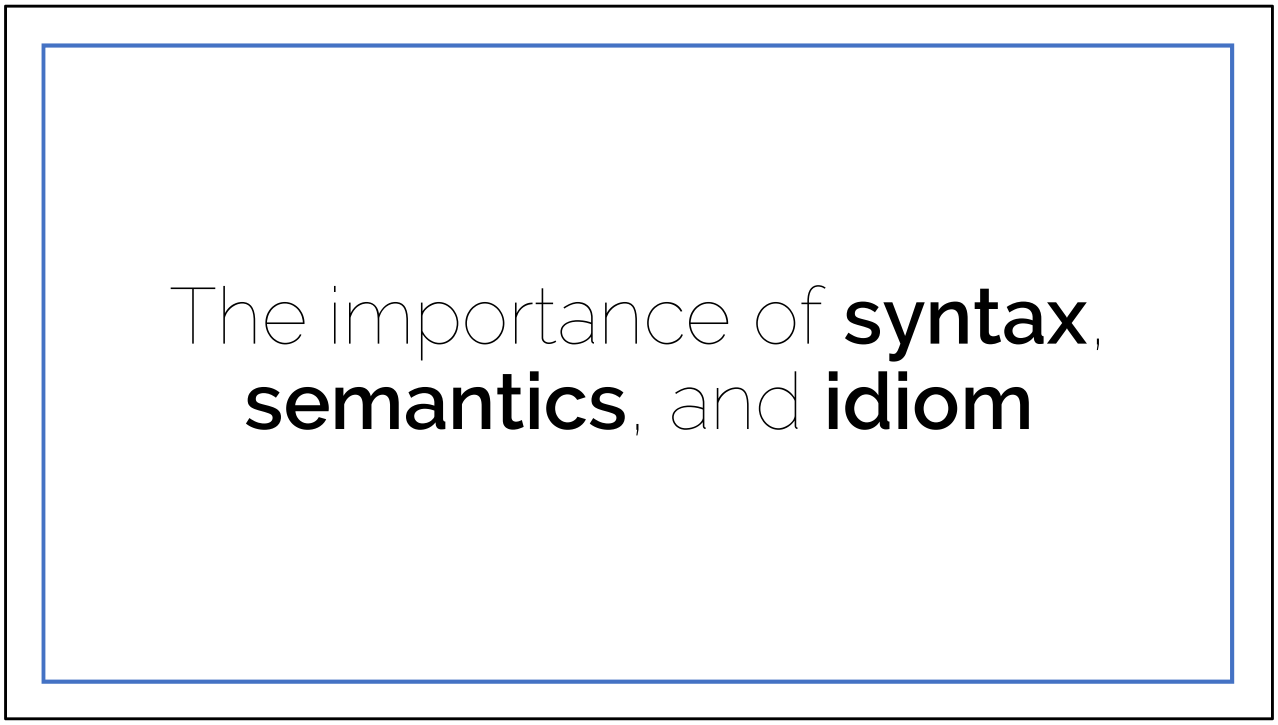
If your commitment is
later than your target,
you have **room to move**

If your target is
earlier than your prediction,
you're **optimistic**

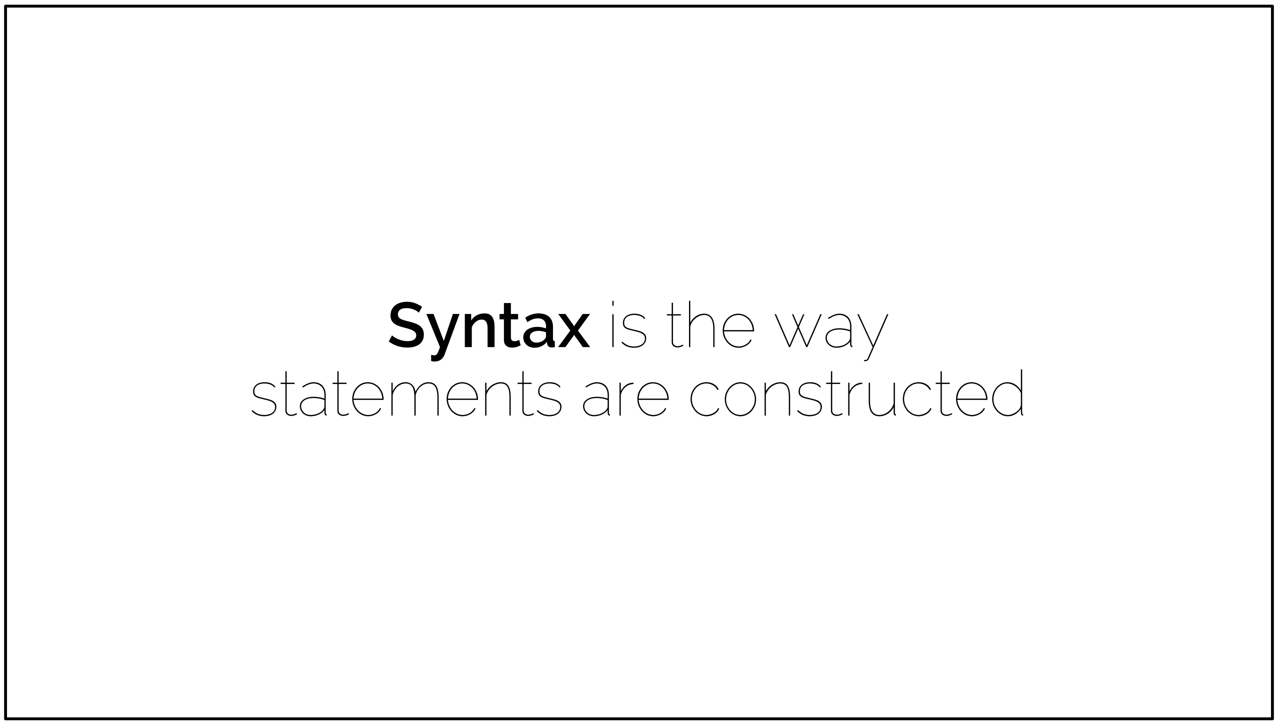


If your commitment is
earlier than your target,
you're **planning to fail**

If you **miss** your commitment,
other people will **fail** to meet their targets



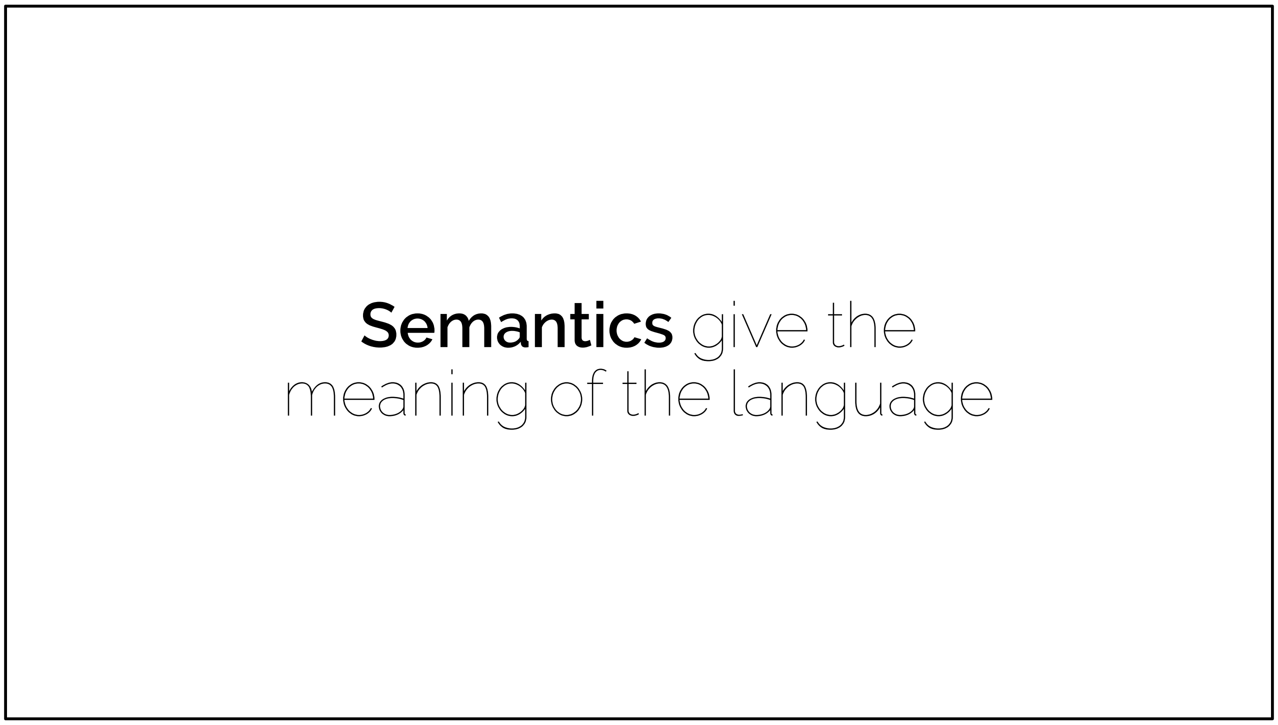
The importance of **syntax**, **semantics**, and **idiom**



Syntax is the way
statements are constructed

Syntax is easy to learn

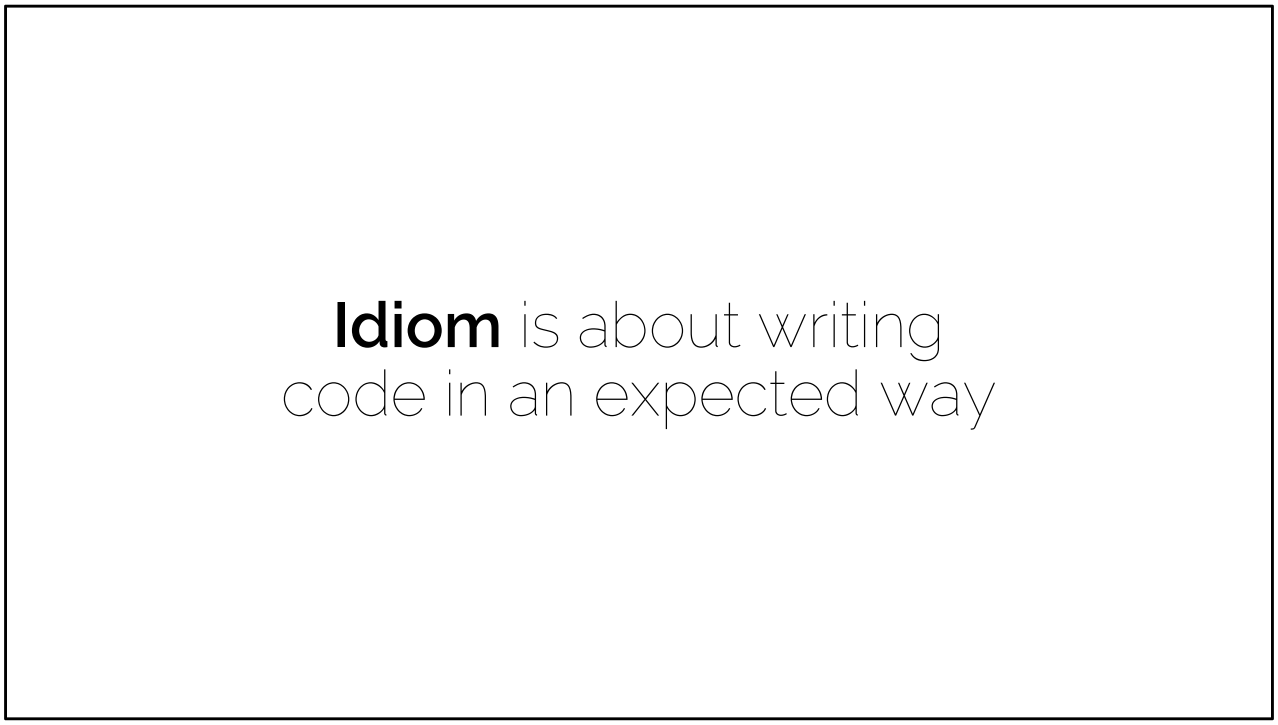
Correct **syntax** means
your code **will run**



Semantics give the meaning of the language

Semantics are not quite so easy to learn, but can be picked up quickly, especially since they are common across languages

Correct **semantics** mean
your code will **do what you expect**

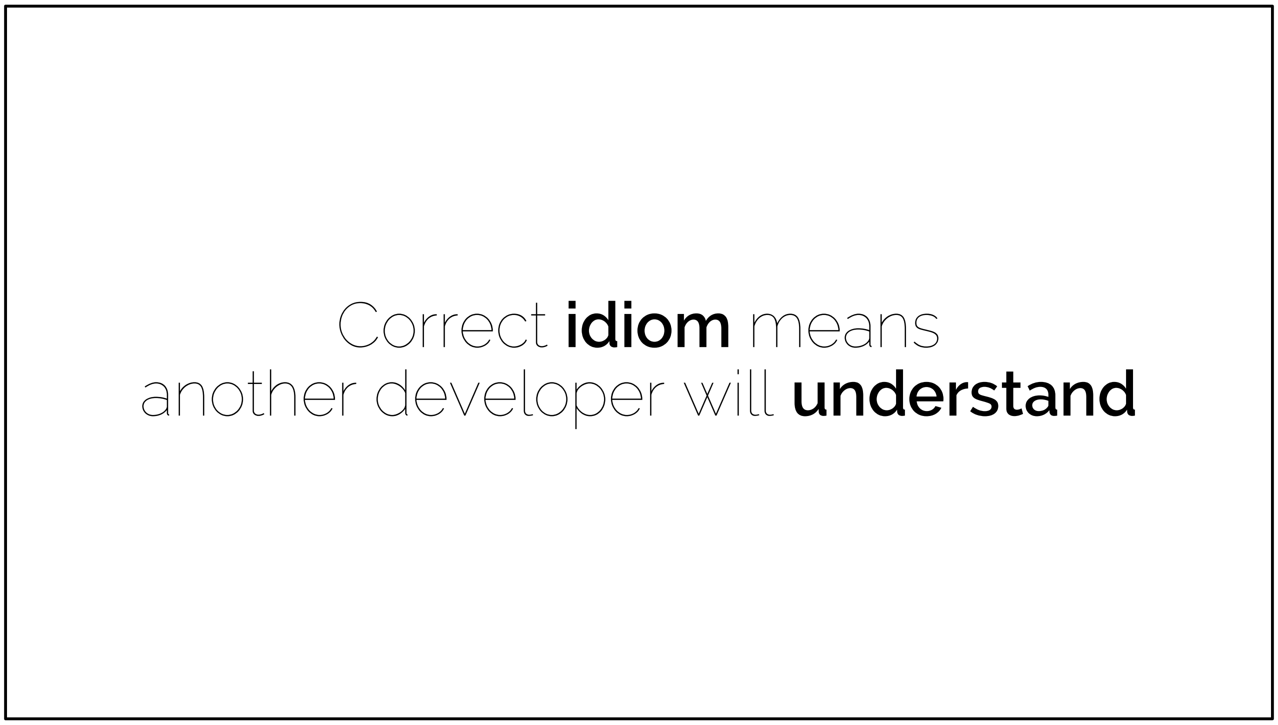


Idiom is about writing code in an expected way

Idiom is harder to learn and requires you to read other people's code

This can be hard

I've known plenty of developers who don't really read anyone else's code

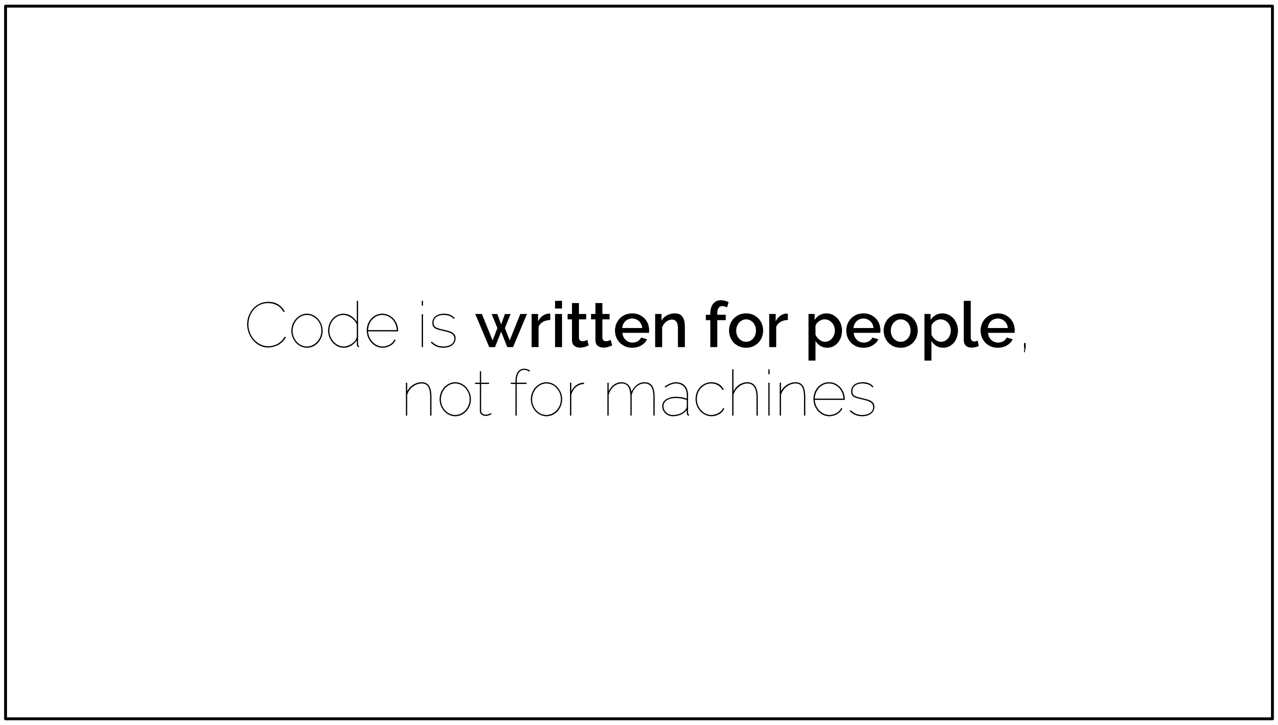


Correct **idiom** means another developer will **understand**

If you write things in the **native idiom** for your platform, other developers will already know those patterns

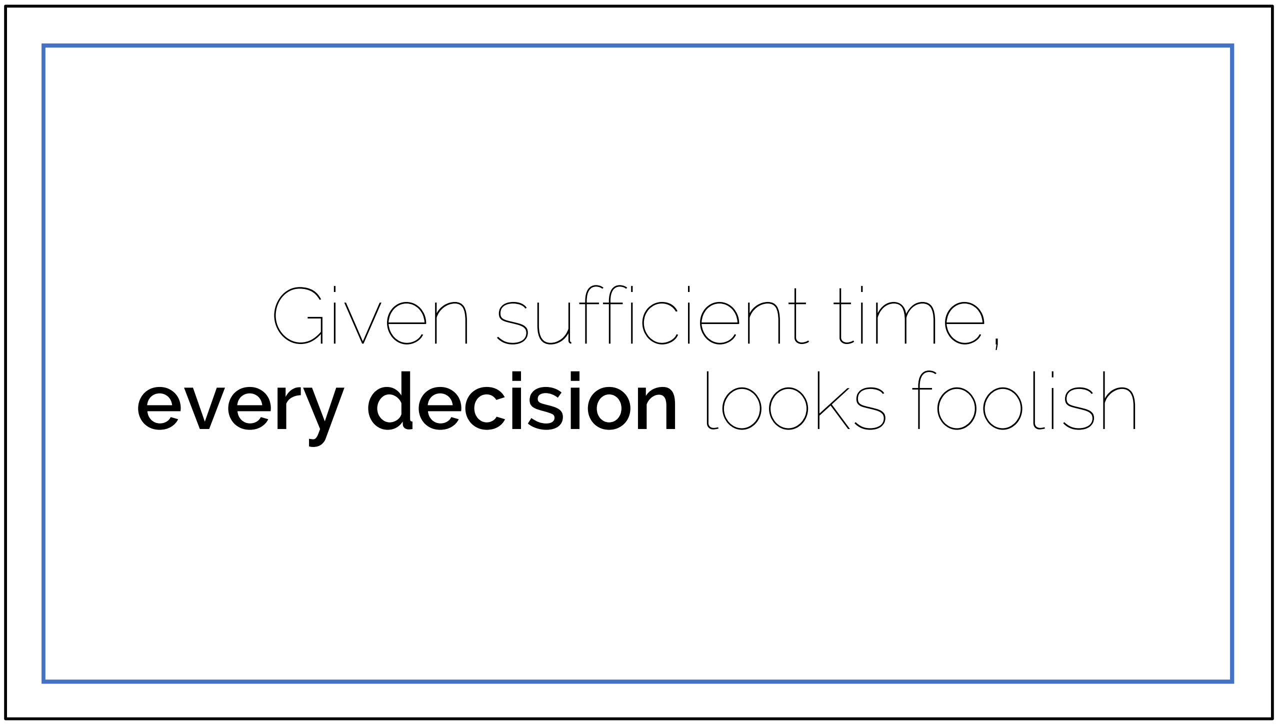
Write things in a different way and either
… they assume you're an idiot and waste time rewriting it
… or they assume you're not and waste time trying to understand why you did it differently

I've worked with a lot of C# code written with C++ idioms,
Was hard to work with until I understood
Often was working too hard to do something that C# handled in a simpler way (e.g. LINQ to objects)



Code is **written for people**, not for machines

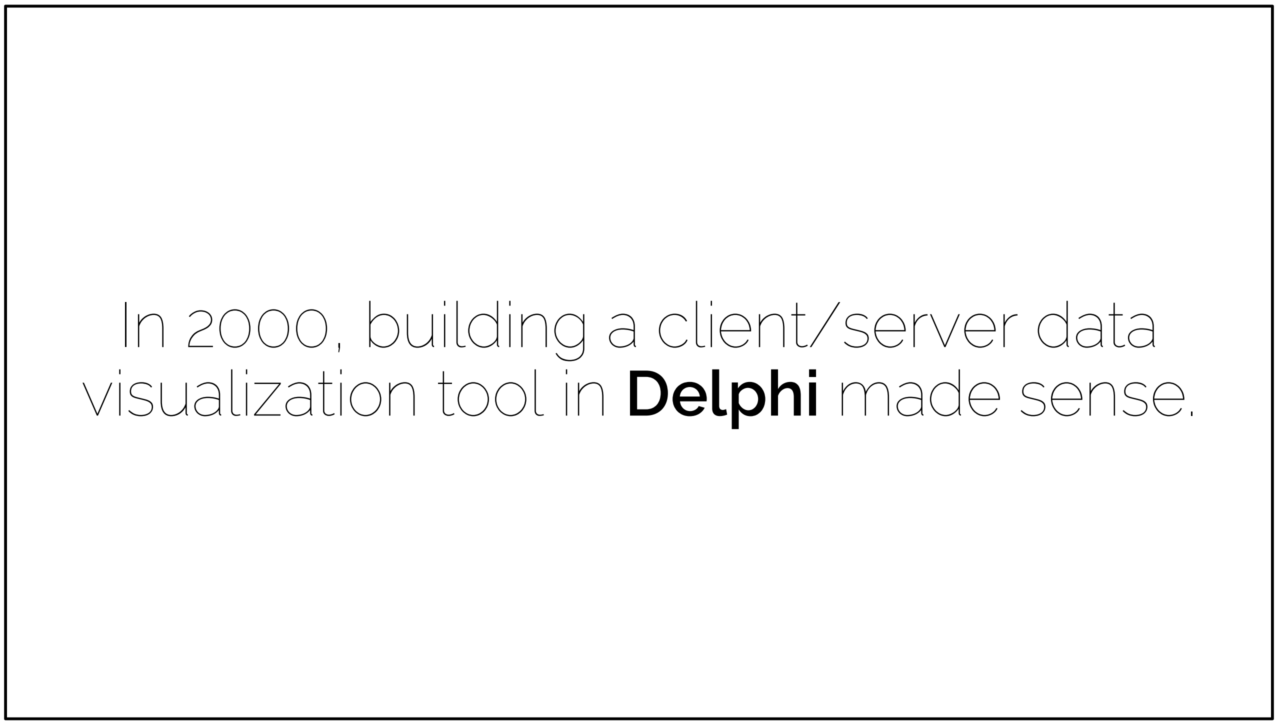
Machines can understand any rubbish
And they don’t care if it’s right or not



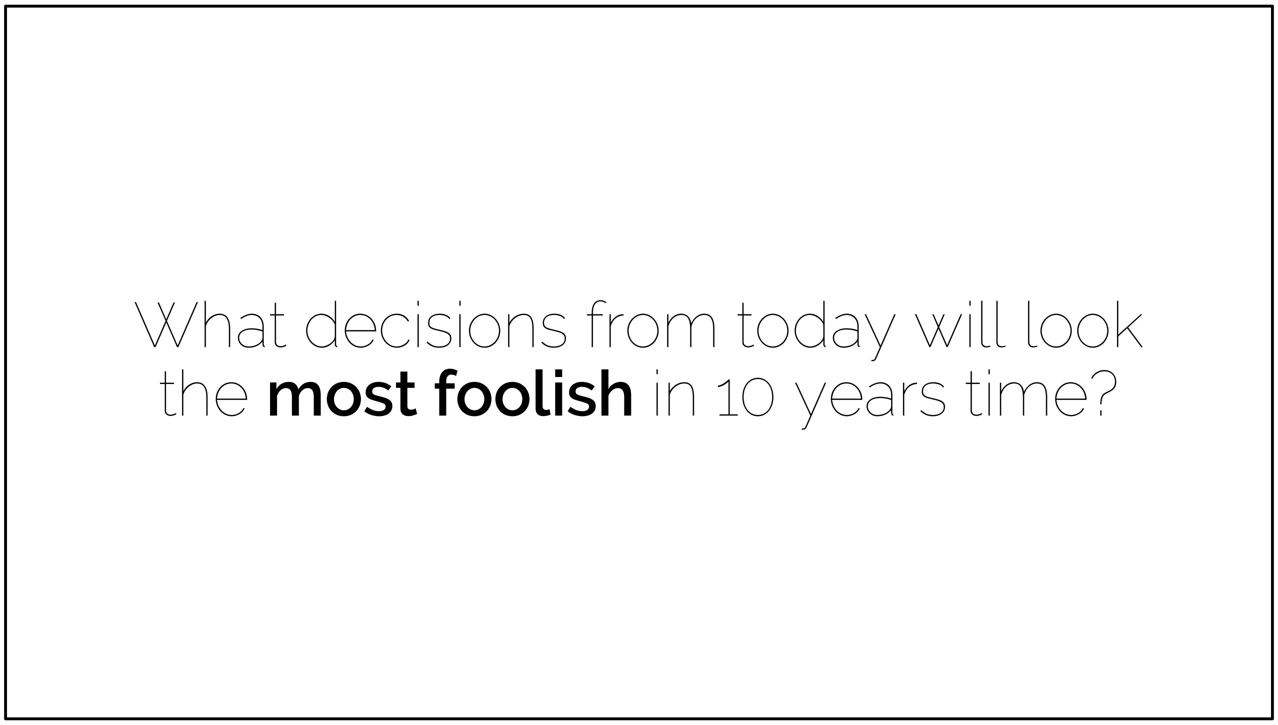
Given sufficient time,
every decision looks foolish

In 1996, the future of web apps was
Java applets running in browsers

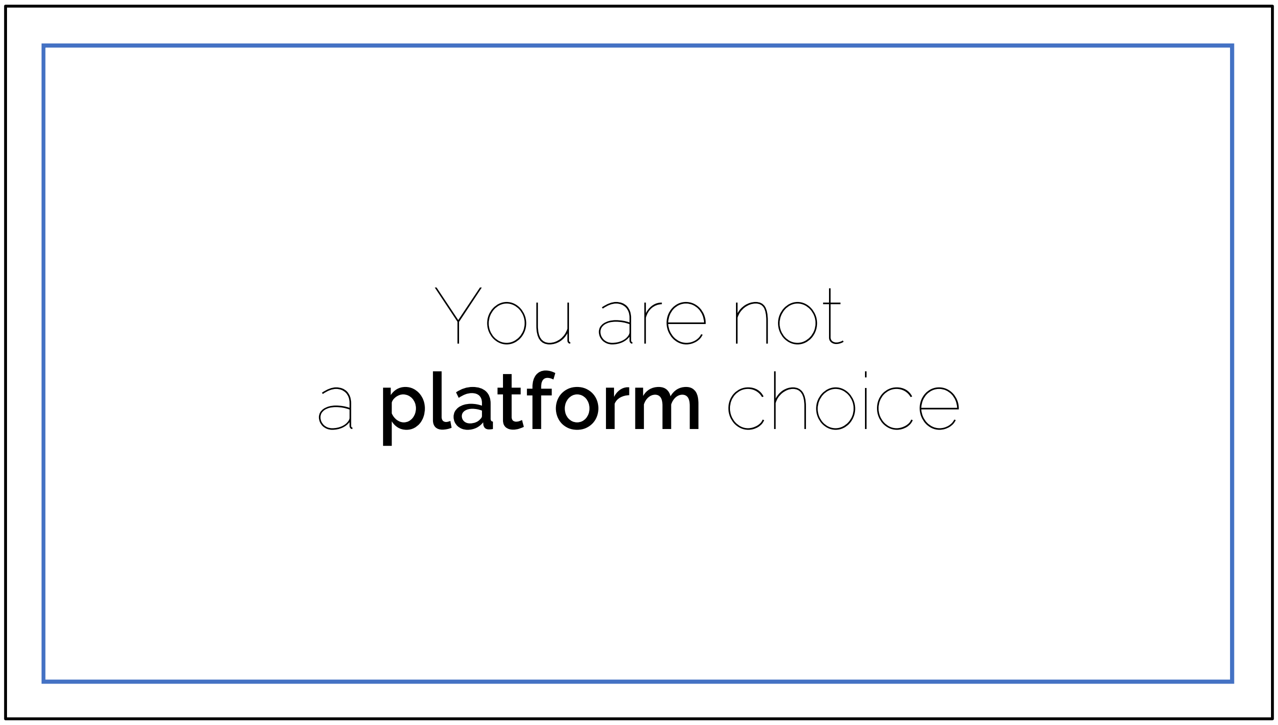
In 1999, building your own **data centre** was necessary for large scale



In 2000, building a client/server data visualization tool in **Delphi** made sense.



What decisions from today will look the **most foolish** in 10 years time?

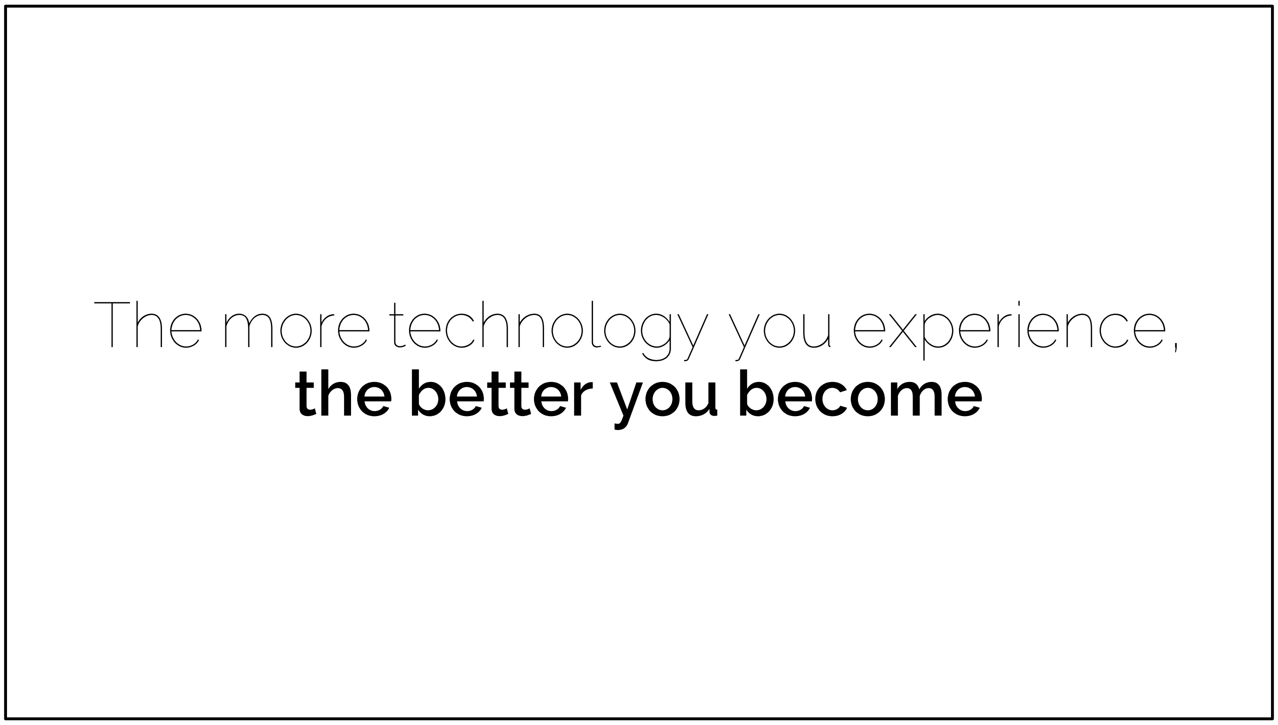


You are not
a **platform** choice

While many recruiters would like to tell you otherwise, you are not a C# developer… or a JavaScript, Delphi, Python, Java, Ruby, C++, TypeScript, Scala, Rust, Go or any other kind of developer.

You are a **developer**

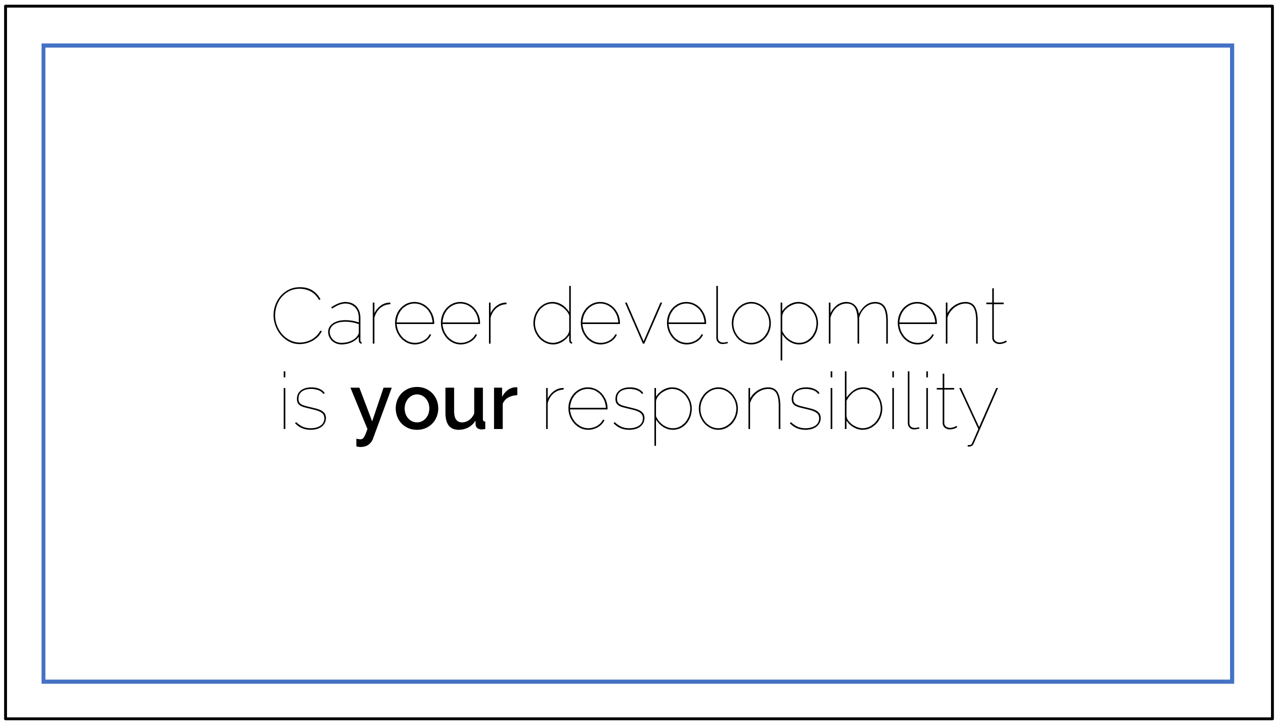
Know **Ruby**? Learn some **Rust**
Know **Java**? Learn some **Python**
Know **C#**? Learn some **Ruby**
Know **Visual Basic**? Learn some **F#**
Know **JavaScript**? Learn some **C**



The more technology you experience,
the better you become

Learning new technologies will positively impact your capability with technologies you already know

Learning a little Ruby made me a better C# developer

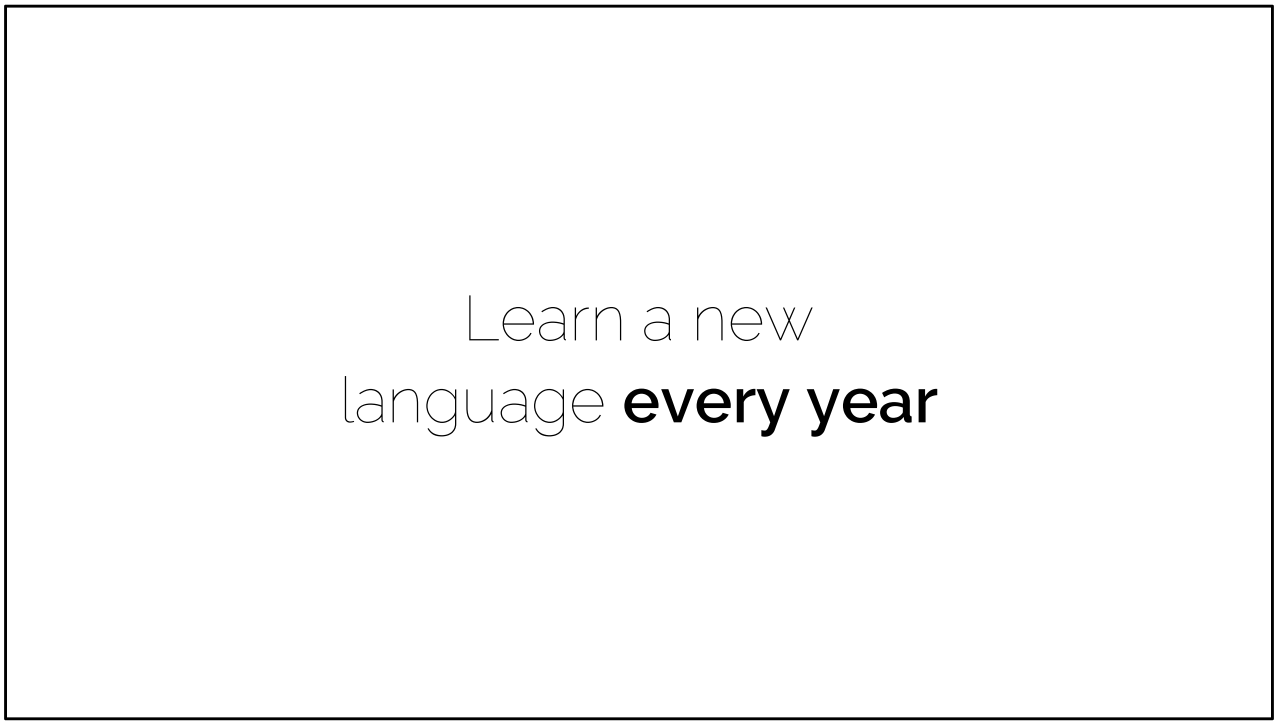


Career development
is **your** responsibility

It's not your bosses problem

Though, if your boss is willing to finance it, that's good

Read a book for
career growth **every month**



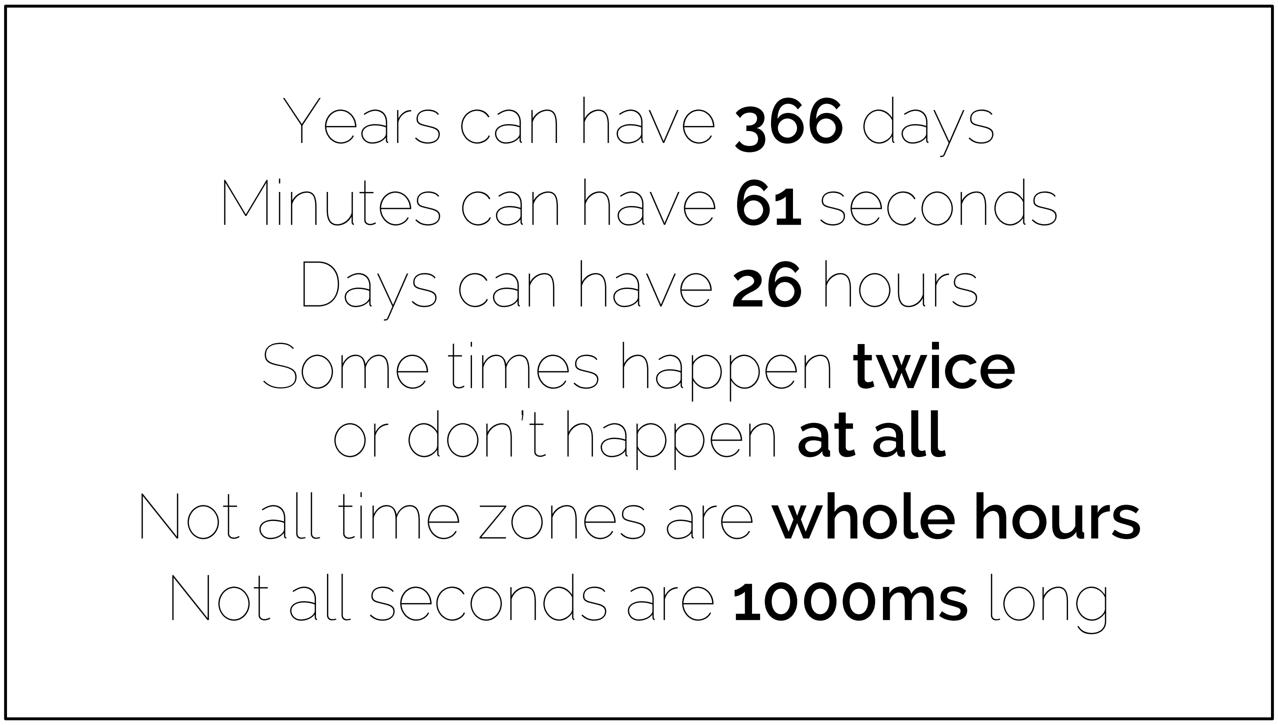
Learn a new language **every year**

Programming language, that is

If you're not **embarrassed**
by code you wrote **six months** ago,
you're not **growing**

The real world is
more complex than you think

Time is such a simple thing



Years can have **366** days
Minutes can have **61** seconds
Days can have **26** hours
Some times happen **twice**
or don't happen **at all**
Not all time zones are **whole hours**
Not all seconds are **1000ms** long

Leap years and leap seconds

Time Zones range from UTC-12 to UTC+14, so in one respect days can be up to 26 hours long

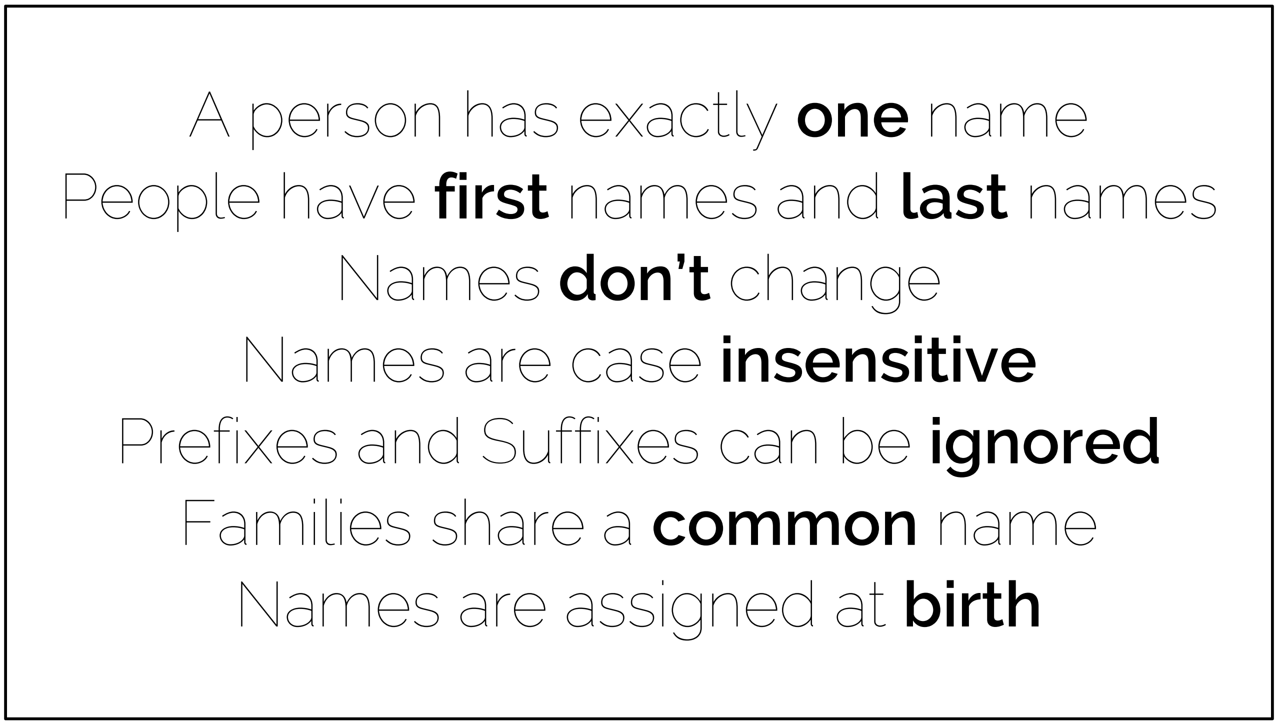
Daylight savings changes – but also when clocks resync with network time, might jump forward or backward

Pakistan standard time is UTC+5:00;
… Indian Standard Time is UTC+05:30
… Nepal Standard Time is UTC+05:45

Time zones are political, not geographic

Sometimes clocks are run fast or slow to sync to network time – avoids the discontinuity of a jump, but …

Names are such simple things



A person has exactly **one** name
People have **first** names and **last** names
Names **don't** change
Names are case **insensitive**
Prefixes and Suffixes can be **ignored**
Families share a **common** name
Names are assigned at **birth**

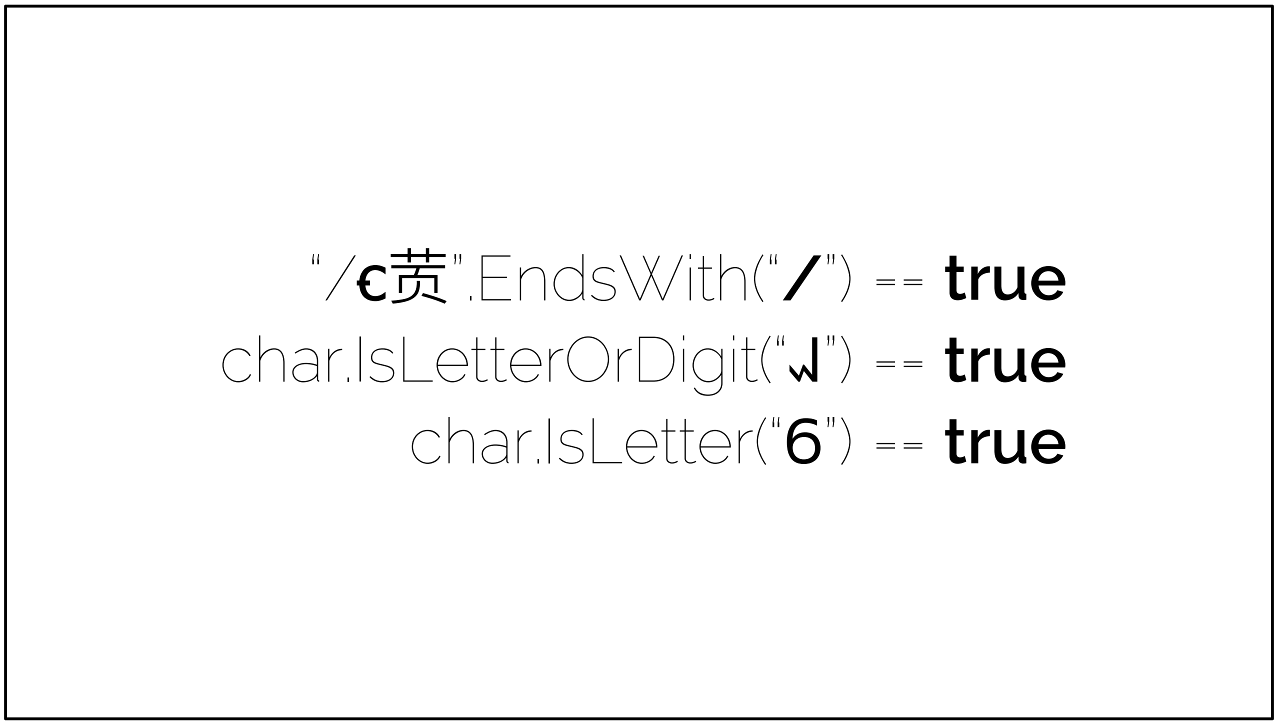
Some people use different names in different arenas of their life
E.g. my son uses Cameron at school, Cam for Theatre

It doesn't always take a life event for names to change
E.g. At school a friend decided to switch from her middle name (Tania) to her given name (Susan)

Patriarchal societies have names based on genealogy, so names are fluid from generation to generation

In areas of high child mortality (e.g. Edwardian England) it's common to get your "real" name well after birth, e.g. Puberty or Age 13

Characters are such simple things



“/€贲”.EndsWith(“/”) == **true**
char.IsLetterOrDigit(“𐐉”) == **true**
char.IsLetter(“𐐞”) == **true**

贲 does not affect the sort order, and EndsWith() uses .Compare() instead of .Equals()
(this is probably a bug in the framework)

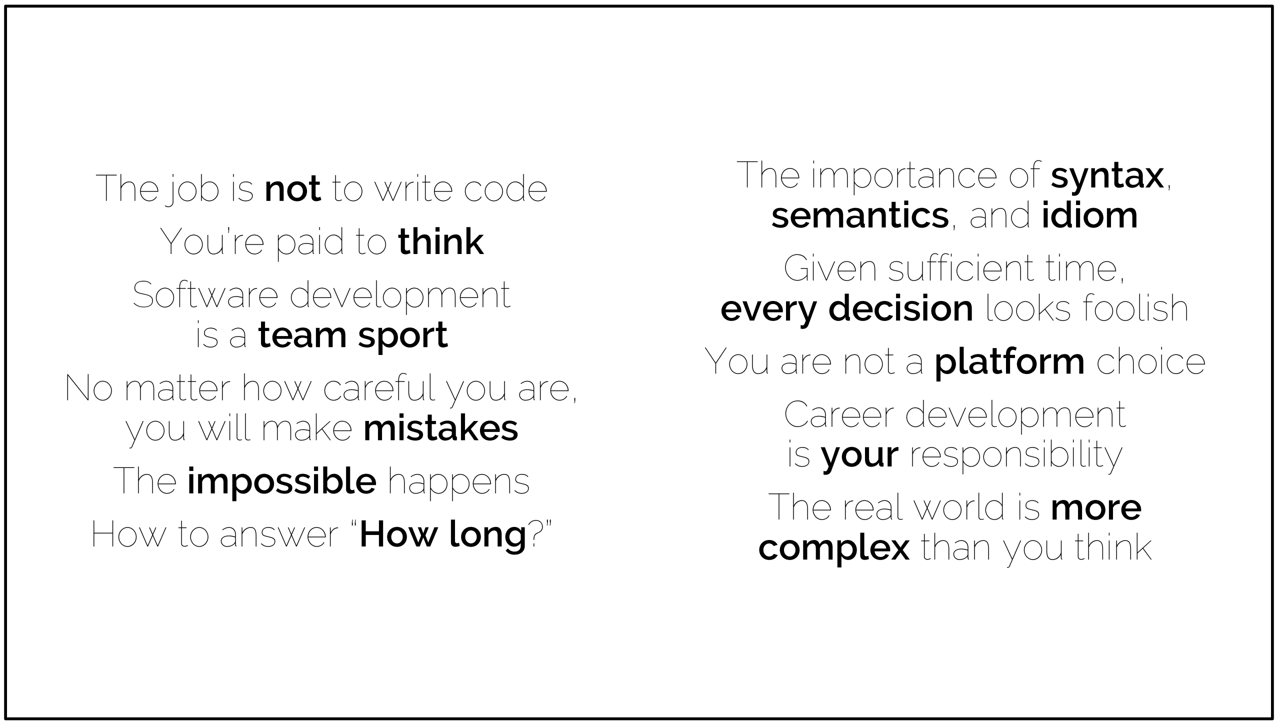
𐐉 = 'DESERET CAPITAL LETTER SHORT AH' (U+10409)
𐐞 = 'DESERET CAPITAL LETTER ZEE' (U+1041E)

Addresses are such simple things.

I think you **get the point**.

Consult with Professor Google or Doctor Bing for interesting facts about addresses and how they don’t work the way you do

In **Closing**



The job is **not** to write code

You're paid to **think**

Software development
is a **team sport**

No matter how careful you are,
you will make **mistakes**

The **impossible** happens

How to answer "**How long**?"

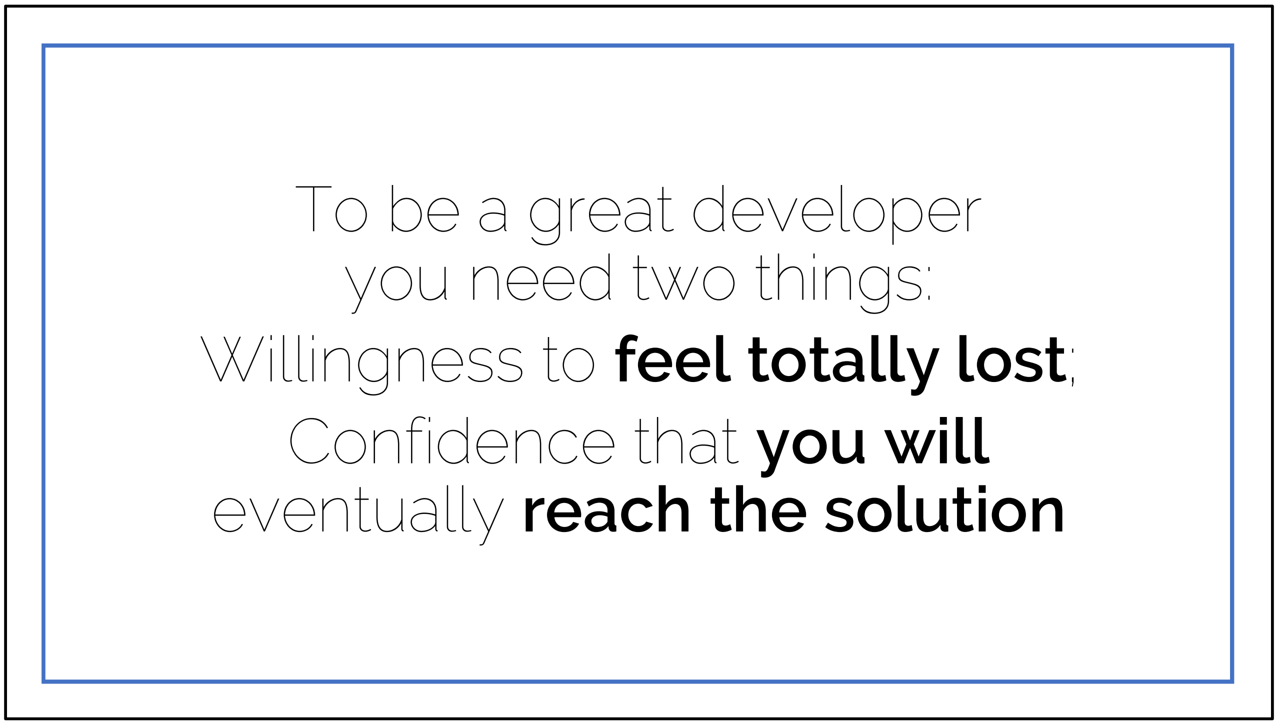
The importance of **syntax**,
semantics, and **idiom**

Given sufficient time,
every decision looks foolish

You are not a **platform** choice

Career development
is **your** responsibility

The real world is **more
complex** than you think



To be a great developer
you need two things:
Willingness to **feel totally lost**;
Confidence that **you will**
eventually **reach the solution**

Couldn’t find an attribution for this

Thanks

@unrepentantgeek
bevan@nichesoftware.co.nz
http://www.nichesoftware.co.nz