Languages (+ end of code review)

Martin Kellogg

How to write code review comments

How to write code review comments

- **Be kind**, courteous, and respectful.

How to write code review comments

- **Be kind**, courteous, and respectful.
- **Explain** your reasoning.

How to write code review comments

- **Be kind**, courteous, and respectful.
- **Explain** your reasoning.
- **Balance** giving explicit directions with just pointing out problems and letting the developer decide.

How to write code review comments

- **Be kind**, courteous, and respectful.
- **Explain** your reasoning.
- **Balance** giving explicit directions with just pointing out problems and letting the developer decide.

> “In general it is the **developer’s responsibility** to fix a CL, not the reviewer’s”

How to write code review comments

- **Be kind**, courteous, and respectful.
- **Explain** your reasoning.
- **Balance** giving explicit directions with just pointing out problems and letting the developer decide.
- **Insist** developers simplify code or add code comments instead of just explaining the complexity to you.

How to write code review comments

- **Be kind**, courteous, and respectful.
- **Explain** your reasoning.
- **Balance** giving explicit directions with just pointing out problems and letting the developer decide.
- **Insist** developers simplify code or add code comments instead of just explaining the complexity to you.

> “Explanations written only in the code review tool are not helpful to **future code readers**”

How to write code review comments: severity

- Label comments with their **severity**, to avoid misunderstandings:

How to write code review comments: severity

- Label comments with their **severity**, to avoid misunderstandings:
  - **Must Fix**: I don't think I can approve this CL until this problem is fixed, even if everything else is perfect.

How to write code review comments: severity

- Label comments with their **severity**, to avoid misunderstandings:
  - **Must Fix**: I don't think I can approve this CL until this problem is fixed, even if everything else is perfect.

Usually authors treat comments without a severity level as **must fix**

How to write code review comments: severity

- Label comments with their **severity**, to avoid misunderstandings:
  - **Must Fix**: I don't think I can approve this CL until this problem is fixed, even if everything else is perfect.
  - **Nit**: This is a minor thing. Technically you should do it, but it won't hugely impact things.

How to write code review comments: severity

- Label comments with their **severity**, to avoid misunderstandings:
  - **Must Fix**: I don't think I can approve this CL until this problem is fixed, even if everything else is perfect.
  - **Nit**: This is a minor thing. Technically you should do it, but it won't hugely impact things.
  - **Optional**: I think this may be a good idea, but it's not strictly required.

How to write code review comments: severity

- Label comments with their **severity**, to avoid misunderstandings:
  - **Must Fix**: I don't think I can approve this CL until this problem is fixed, even if everything else is perfect.
  - **Nit**: This is a minor thing. Technically you should do it, but it won't hugely impact things.
  - **Optional**: I think this may be a good idea, but it's not strictly required.
  - **FYI**: I don't expect you to do this in this CL, but you may find this interesting to think about for the future.

Common mistakes to avoid as a reviewer

Common mistakes to avoid as a reviewer

- Being **condescending**, especially if you’re wrong.

Common mistakes to avoid as a reviewer

- Being **condescending**, especially if you're wrong.

If you get **pushback** on a suggestion, take the time to understand why

Common mistakes to avoid as a reviewer

- Being **condescending**, especially if you're wrong.
- Taking **too long** to complete a review.

Common mistakes to avoid as a reviewer

- Being **condescending**, especially if you're wrong.
- Taking **too long** to complete a review.

Try to get back to the author within "**one business day**", whatever that means for your team

Common mistakes to avoid as a reviewer

- Being **condescending**, especially if you're wrong.
- Taking **too long** to complete a review.
- Being **too lax**

Common mistakes to avoid as a reviewer

- Being **condescending**, especially if you're wrong.
- Taking **too long** to complete a review.
- Being **too lax**

Common mistake: "LGTM" everything for the sake of speed

Common mistakes to avoid as a reviewer

- Being **condescending**, especially if you're wrong.
- Taking **too long** to complete a review.
- Being **too lax**
- Being **inconsistent**

Common mistakes to avoid as a reviewer

- Being **condescending**, especially if you're wrong.
- Taking **too long** to complete a review.
- Being **too lax**
- Being **inconsistent**

I've had reviewers ask for one thing, which I do, and then ask for something completely different a week later. **Read your previous review!**

Common mistakes to avoid as a reviewer

- Being **condescending**, especially if you're wrong.
- Taking **too long** to complete a review.
- Being **too lax**
- Being **inconsistent**
- Letting complexity through with a promise to **clean up** later

Common mistakes to avoid as a reviewer

- Being **condescending**, especially if you're wrong.
- Taking **too long** to complete a review.
- Being **too lax**
- Being **inconsistent**
- Letting complexity through with a promise to **clean up** later

Doesn't usually happen! If the problem is serious, insist on fixing it now!

Common mistakes to avoid as an author

Common mistakes to avoid as an author

- Respond to **every comment**

Making a code change counts as a response! Don’t write “fixed” or similar on every comment.

Common mistakes to avoid as an author

- Respond to **every comment**
- If you fix something in one place, **fix it everywhere**

As a reviewer, it is very tedious to point out every place that an author has made the same mistake.

Common mistakes to avoid as an author

- Respond to **every comment**
- If you fix something in one place, **fix it everywhere**
- Assume **good faith**

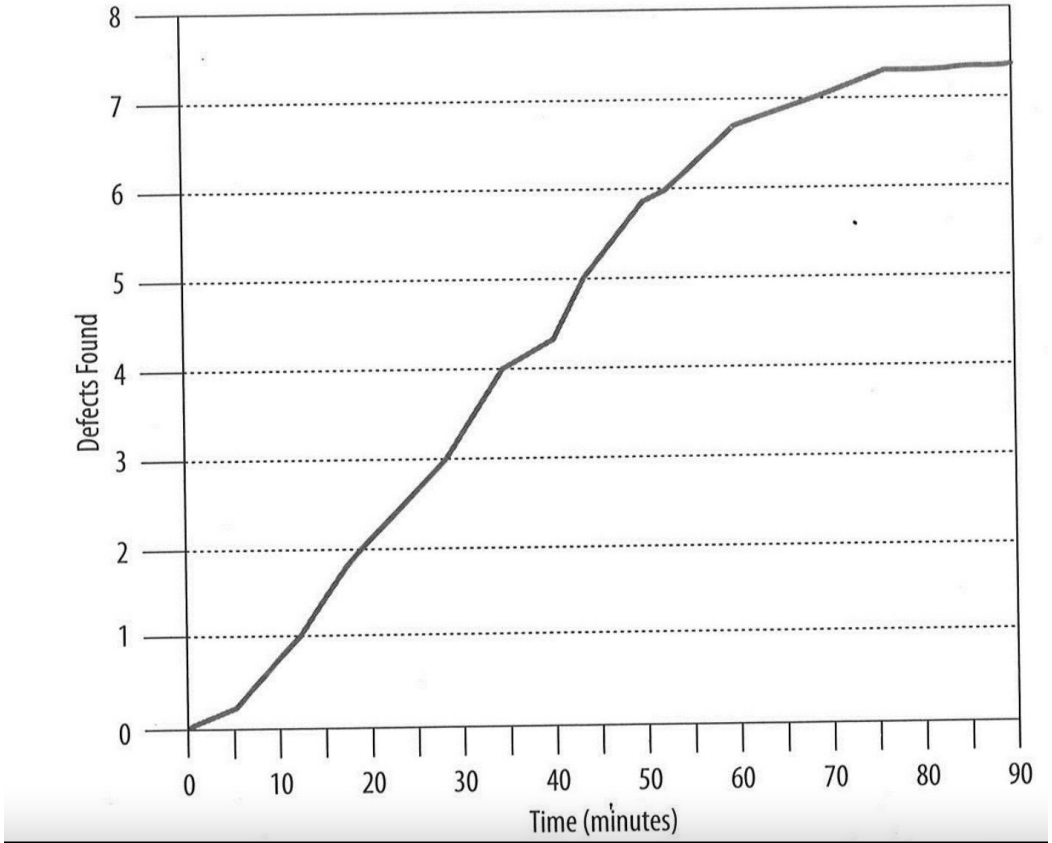
Common mistakes to avoid as an author

- Respond to **every comment**
- If you fix something in one place, **fix it everywhere**
- Assume **good faith**
- Address comments **by changing the code**, not by explaining in the review tool

Empirical guidelines for code review

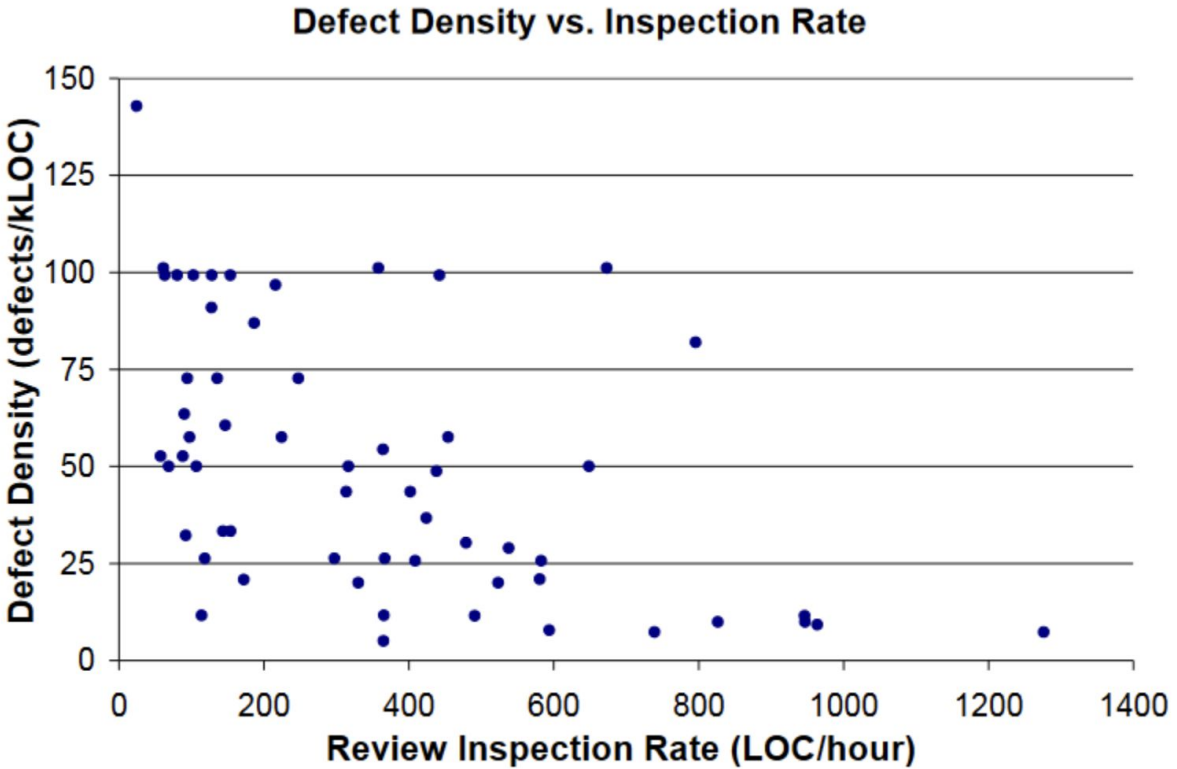
Empirical guidelines for code review

- **Recommendation**: Do not exceed 60 minute session
- **Reason**: focus fatigue



Empirical guidelines for code review

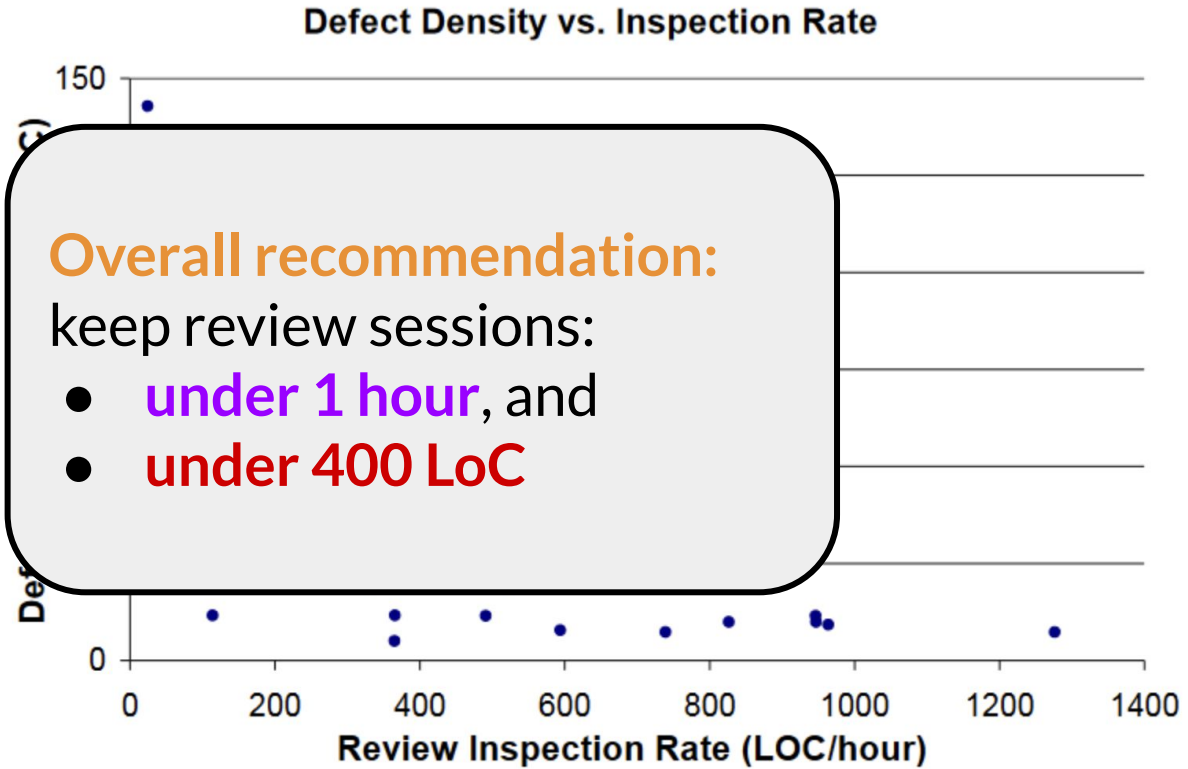
- **Recommendation**: Don’t review more than 400 LoC per hour
- **Reason**: at faster paces, reviews get too shallow



[Code Review at Cisco Systems. In J A Cohen et al.'s Best Kept Secrets of Peer Code Review, 2013.]

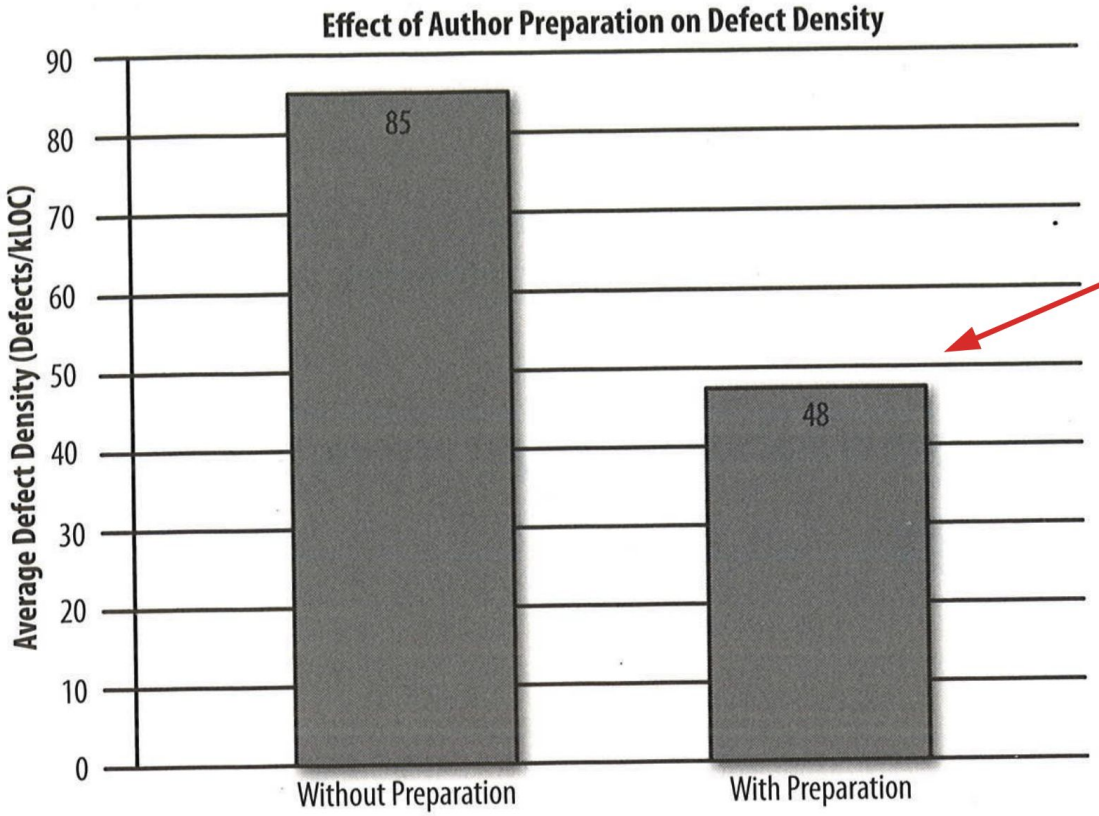
Empirical guidelines for code review

- **Recommendation**: Don't review more than 400 LoC per hour
- **Reason**: at faster paces, reviews get too shallow



[Code Review at Cisco Systems. In J A Cohen et al.'s Best Kept Secrets of Peer Code Review, 2013.]

Empirical guidelines for code review



Important to review your own code before giving it to others

Code Review

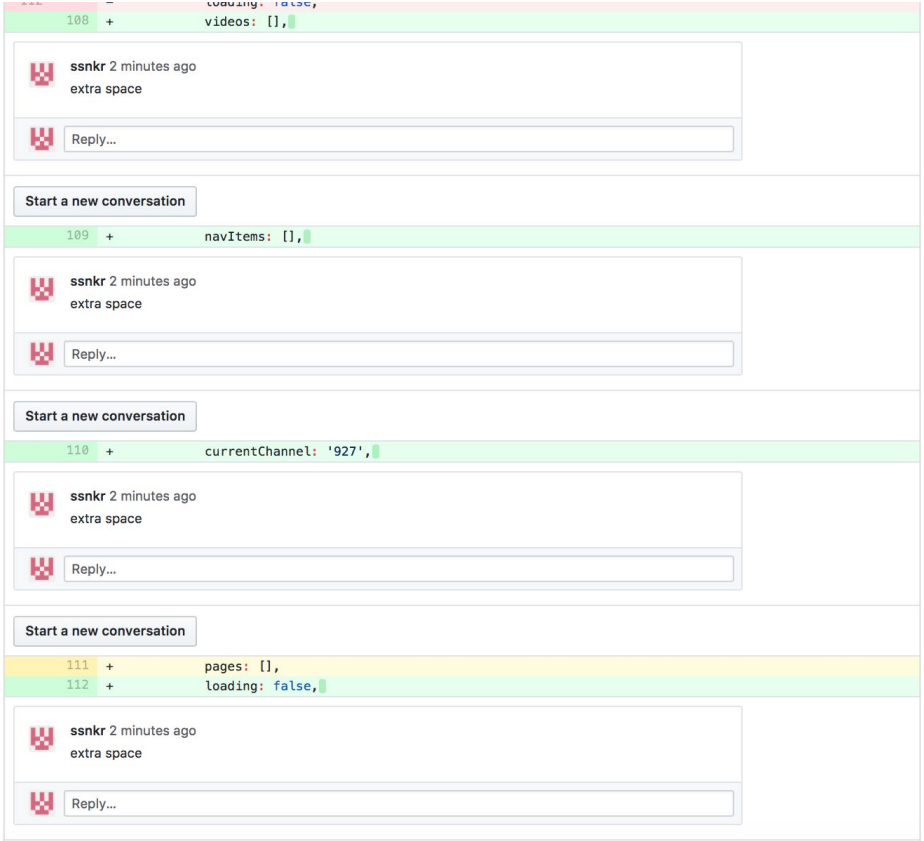
Today's agenda:

- Finish slides on interviewing
- Reading Quiz
- What is code review (and why we do it)
- How to do a code review (with empirical evidence)
- **Good and bad examples of code review comments**

Example comment: good or bad?

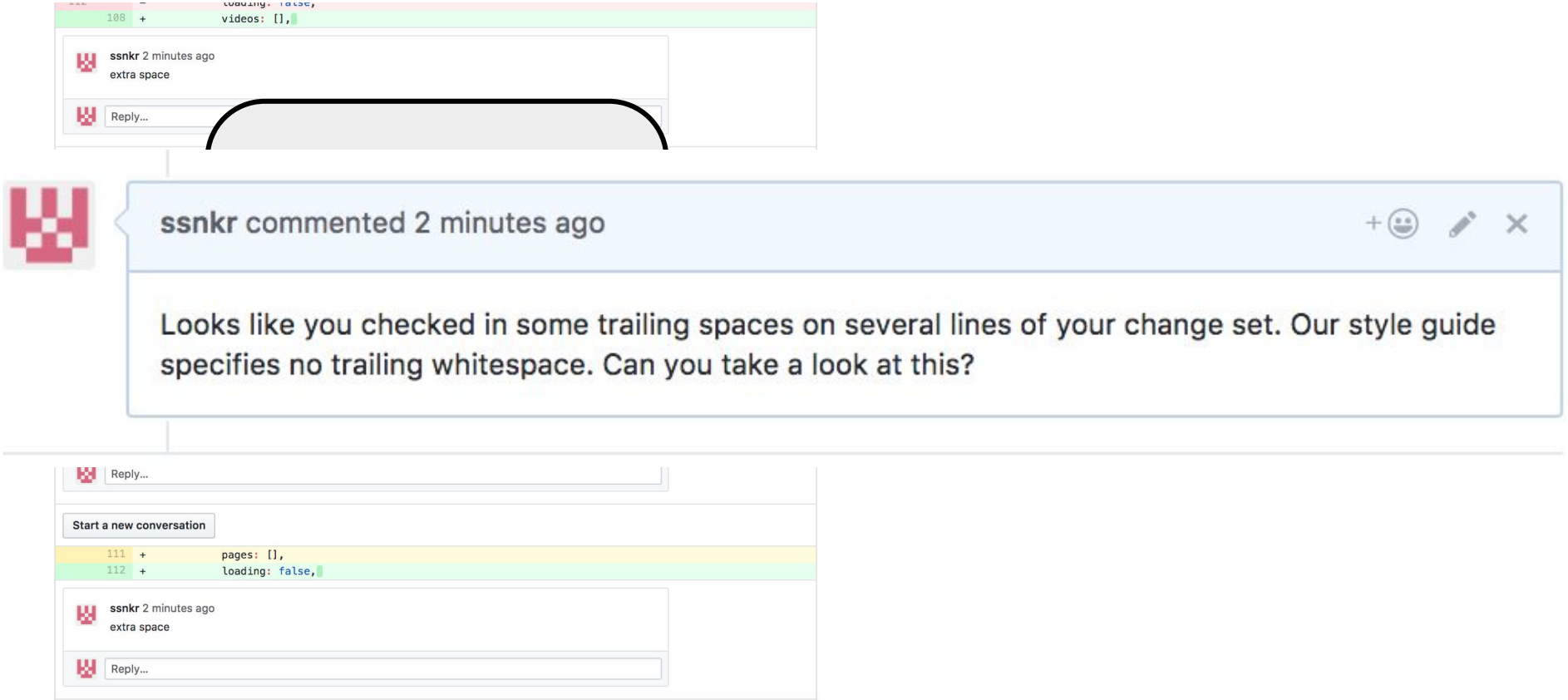
[Many of the examples in the following slides borrowed from Sandya Sankarram's "Unlearning toxic behaviors in a code review culture"]

Example comment: good or bad?

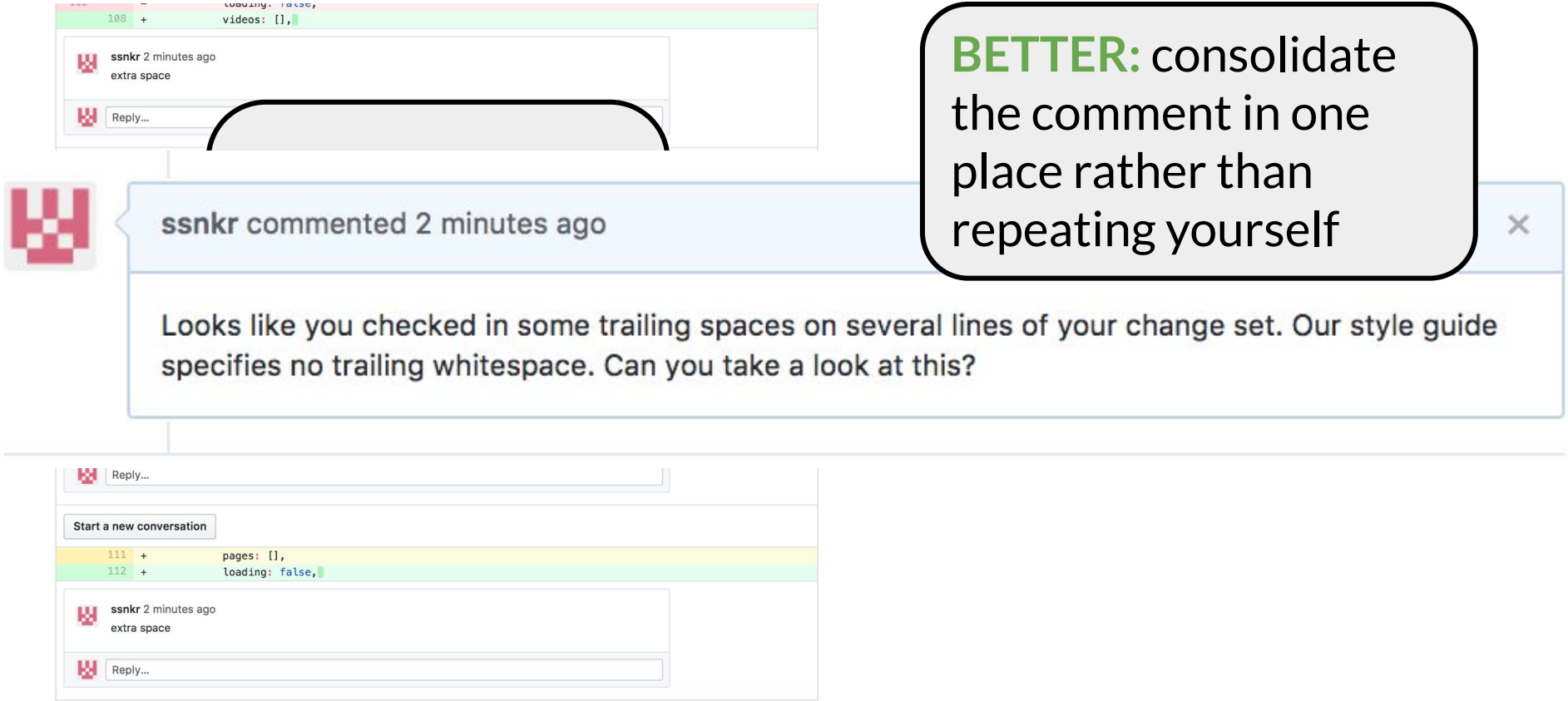


Example comment: good or bad?

Example comment: good or bad?

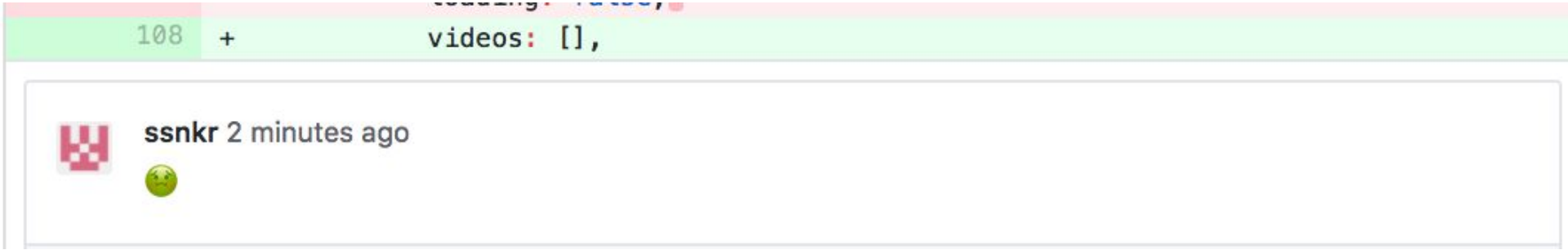


Example comment: good or bad?

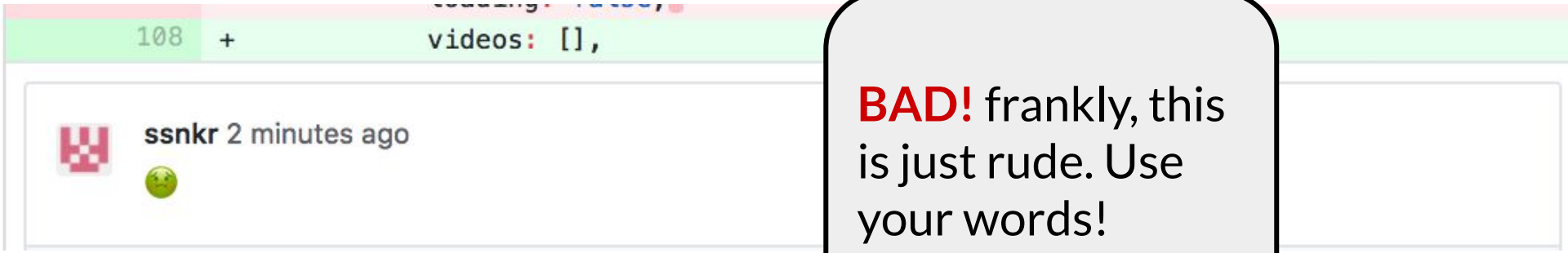


BETTER: consolidate the comment in one place rather than repeating yourself

Example comment: good or bad?



Example comment: good or bad?



BAD! frankly, this is just rude. Use your words!

Example comment: good or bad?

Example comment: good or bad?

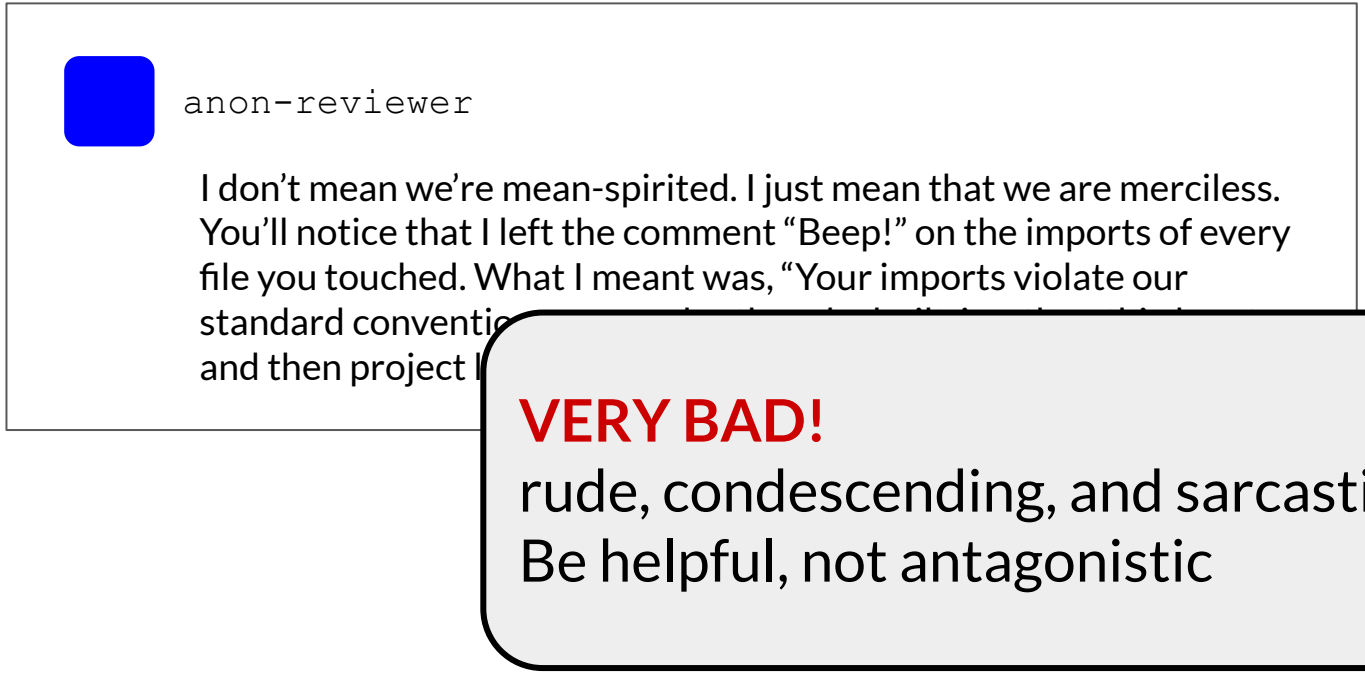
OK: emojis and similar “casual” language should only be used to praise, never to criticize

Example comment: good or bad?

anon-reviewer

I don’t mean we’re mean-spirited. I just mean that we are merciless. You’ll notice that I left the comment “Beep!” on the imports of every file you touched. What I meant was, “Your imports violate our standard convention — we order them by built-ins, then third party, and then project level,” but that was too much to type on every file.

Example comment: good or bad?

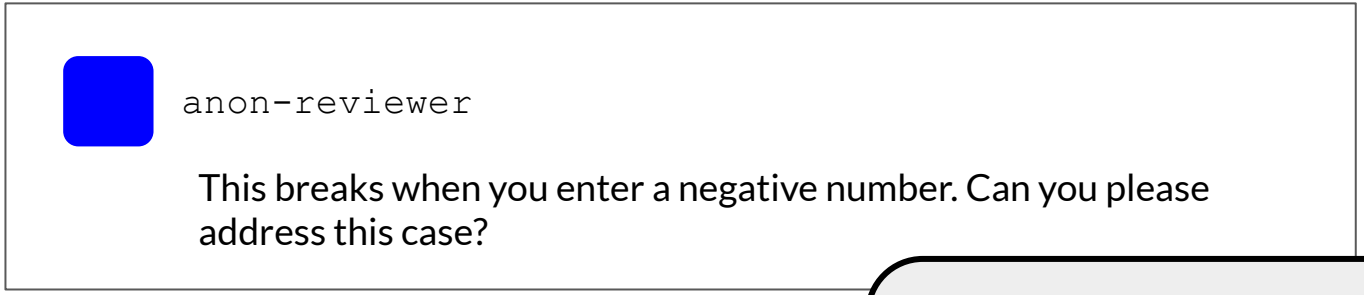


Example comment: good or bad?

`anon-reviewer`

This breaks when you enter a negative number. Can you please address this case?

Example comment: good or bad?



GOOD: straight to the point, politely points out a technical problem

Takeaways: code review

- Code review is one of the best ways to prevent defects
  - You must do it during the course project (I will check!)
- Be nice as both an author and a reviewer
  - Respect each other and each other's time
- One thing I'll look for when assessing your group project is the quality of your code reviews
  - If you're unsure, you can ask the course staff to review your reviews (in office hours)

Reading Quiz: Languages

Reading Quiz: Languages

Q1: What was the source of the latency spikes in the original Go implementation of the Read States service?

A. Go's default hash table had too many collisions
B. Go is an interpreted language
C. Go arrays require expensive copy operations if they get too big
D. Go's garbage collector runs every 2 minutes

Q2: **TRUE** or **FALSE:** Even with just basic optimization, the Rust version of the Read States service outperformed the hyper hand-tuned Go version.

Reading Quiz: Languages

Q1: What was the source of the latency spikes in the original Go implementation of the Read States service?

A. Go's default hash table had too many collisions
B. Go is an interpreted language
C. Go arrays require expensive copy operations if they get too big
D. Go's garbage collector runs every 2 minutes

Q2: **TRUE** or **FALSE:** Even with just basic optimization, the Rust version of the Read States service outperformed the hyper hand-tuned Go version.

Reading Quiz: Languages

Q1: What was the source of the latency spikes in the original Go implementation of the Read States service?

A. Go's default hash table had too many collisions
B. Go is an interpreted language
C. Go arrays require expensive copy operations if they get too big
D. Go's garbage collector runs every 2 minutes

Q2: **TRUE** or **FALSE:** Even with just basic optimization, the Rust version of the Read States service outperformed the hyper hand-tuned Go version.

Why discuss programming languages at all?

Why discuss programming languages at all?

- the language a project is written in has a big impact on how the project goes

Why discuss programming languages at all?

- the language a project is written in has a big impact on how the project goes
  - as always, **choose the right tool for the job**

Why discuss programming languages at all?

- the language a project is written in has a big impact on how the project goes
  - as always, **choose the right tool for the job**
- it's fairly rare that you get to choose a language, but when you do, it's a big responsibility!

Why discuss programming languages at all?

- the language a project is written in has a big impact on how the project goes
  - as always, **choose the right tool for the job**
- it's fairly rare that you get to choose a language, but when you do, it's a big responsibility!
  - lecture goal: give you tools to **evaluate the trade-offs** between different languages

Why discuss programming languages at all?

- the language a project is written in has a big impact on how the project goes
  - as always, **choose the right tool for the job**
- it's fairly rare that you get to choose a language, but when you do, it's a big responsibility!
  - lecture goal: give you tool
    between different languag

Advice before we go further: when you inherit a code base, **don't try to rewrite** it right away in a "better" language: it's usually not worth it

How can programming languages differ?

How can programming languages differ?

- programming paradigm
- whether they have a type system
  - and, if they do, what kind of type system they have
- library support
  - the standard library is especially important
- performance
- team/process factors
  - how well do you know the language
  - how easy it'll be to hire other developers who do

How can programming languages differ?

- **programming paradigm**
- whether they have a type system
  - and, if they do, what kind of type system they have
- library support
  - the standard library is especially important
- performance
- team/process factors
  - how well do you know the language
  - how easy it'll be to hire other developers who do

Programming language paradigms

Definition: a language ***paradigm*** is a way to classify programming languages, usually by their style of structuring programs

Programming language paradigms

Definition: a language ***paradigm*** is a way to classify programming languages, usually by their style of structuring programs

- usually based on some kind of mathematical foundation

Programming language paradigms

Definition: a language ***paradigm*** is a way to classify programming languages, usually by their style of structuring programs

- usually based on some kind of mathematical foundation
- common, important paradigms we'll discuss today:
  - imperative
  - functional
  - object-oriented

Imperative programming

Definition: in the ***imperative*** paradigm, programs are sequences of commands that destructively update one or more arrays

Imperative programming

Definition: in the ***imperative*** paradigm, programs are sequences of commands that destructively update one or more arrays

- key mathematical formalism: **???**

Imperative programming

Definition: in the ***imperative*** paradigm, programs are sequences of commands that destructively update one or more arrays

- key mathematical formalism: **Turing machines**

Imperative programming

Definition: in the ***imperative*** paradigm, programs are sequences of commands that destructively update one or more arrays

- key mathematical formalism: **Turing machines**
  - review: what's a Turing machine (on the whiteboard)?

Imperative programming

Definition: in the ***imperative*** paradigm, programs are sequences of commands that destructively update one or more arrays

- key mathematical formalism: **Turing machines**
  - review: what's a Turing machine (on the whiteboard)?
- this is the single **most-common** programming paradigm

Imperative programming

Definition: in the ***imperative*** paradigm, programs are sequences of commands that destructively update one or more arrays

- key mathematical formalism: **Turing machines**
  - review: what's a Turing machine (on the whiteboard)?
- this is the single **most-common** programming paradigm
- models **actual computers** very well:
  - commands = ?
  - array that is destructively updated = ?

Imperative programming

Definition: in the ***imperative*** paradigm, programs are sequences of commands that destructively update one or more arrays

- key mathematical formalism: **Turing machines**
  - review: what's a Turing machine (on the whiteboard)?
- this is the single **most-common** programming paradigm
- models **actual computers** very well:
  - commands = instructions to the processor
  - array that is destructively updated = ?

Imperative programming

Definition: in the ***imperative*** paradigm, programs are sequences of commands that destructively update one or more arrays

- key mathematical formalism: **Turing machines**
  - review: what's a Turing machine (on the whiteboard)?
- this is the single **most-common** programming paradigm
- models **actual computers** very well:
  - commands = instructions to the processor
  - array that is destructively updated = registers/memory/disk

Imperative programming: examples

Languages with imperative programming (non-exhaustive list):

Imperative programming: examples

Languages with imperative programming (non-exhaustive list):

- FORTRAN
- C
- C++
- Python
- Java
- JavaScript/TypeScript
- many, many others!

Imperative programming: examples

Consider the following C program:

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

Imperative programming: examples

Consider the following C program:

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

semicolons separate commands, program is a list of commands

Imperative programming: examples

Consider the following C program:

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

destructive updates of memory cells

Functional programming

Definition: in the ***functional*** paradigm, programs are compositions of mathematical expressions (especially functions)

Functional programming

Definition: in the ***functional*** paradigm, programs are compositions of mathematical expressions (especially functions)

- key mathematical formalism: **?**

Functional programming

Definition: in the ***functional*** paradigm, programs are compositions of mathematical expressions (especially functions)

- key mathematical formalism: **lambda calculus**

Functional programming

Definition: in the ***functional*** paradigm, programs are compositions of mathematical expressions (especially functions)

- key mathematical formalism: **lambda calculus**
  - in the lambda calculus, **everything is a function**

Functional programming

Definition: in the ***functional*** paradigm, programs are compositions of mathematical expressions (especially functions)

- key mathematical formalism: **lambda calculus**
  - in the lambda calculus, **everything is a function**
  - lambda calculus is **as powerful** as Turing machines
    - “as powerful” = anything you can compute with a Turing machine can also be computed with the lambda calculus

Functional programming

Definition: in the ***functional*** paradigm, programs are compositions of mathematical expressions (especially functions)

- key mathematical formalism: **lambda calculus**
  - in the lambda calculus, **everything is a function**
  - lambda calculus is **as powerful** as Turing machines
    - "as powerful" = anything you can compute with a Turing machine can also be computed with the lambda calculus
- functional programming **models math** well
  - it is easier to formally reason about functional programs

Functional programming: characteristics

- Computation = **evaluating** (math) functions

Functional programming: characteristics

- Computation = **evaluating** (math) functions
- Avoid “global state” and “mutable data”

Functional programming: characteristics

- Computation = **evaluating** (math) functions
- Avoid “global state” and “mutable data”
- Get stuff done = apply (**higher-order**) functions

Functional programming: characteristics

- Computation = **evaluating** (math) functions
- Avoid “global state” and “mutable data”
- Get stuff done = apply (**higher-order**) functions
- Avoid sequential commands

Functional programming: characteristics

- Computation = **evaluating** (math) functions
- Avoid “global state” and “mutable data”
- Get stuff done = apply (**higher-order**) functions
- Avoid sequential commands
- Important Features of functional languages:
  - **Higher-order, first-class** functions
  - Closures and **recursion**
  - **Lists** and list processing

Functional programming: characteristics

- Computation = **evaluating** (math) functions
- Avoid “global state” and “mutable data”
- Get stuff done = apply (**higher-order**) functions
- Avoid sequential commands
- Important Features of functional languages:
  - **Higher-order, first-class** f
  - Closures and **recursion**
  - **Lists** and list processing

Let’s look at how imperative and functional languages **manage state** in a bit more detail

State management: functional vs imperative

Definition: The ***state*** of a program is all of the current variable and heap values

State management: functional vs imperative

Definition: The ***state*** of a program is all of the current variable and heap values

- **Imperative** programs **destructively update** the state.

State management: functional vs imperative

Definition: The ***state*** of a program is all of the current variable and heap values

- **Imperative** programs **destructively update** the state.
  - e.g., after executing `*x = y` (in a C program), the memory cell that `x` points to now holds the value `y`. Its old value is gone.

State management: functional vs imperative

Definition: The ***state*** of a program is all of the current variable and heap values

- **Imperative** programs **destructively update** the state.
  - e.g., after executing `*x = y` (in a C program), the memory cell that `x` points to now holds the value `y`. Its old value is gone.
- **Functional** programs yield **new similar states** over time.

State management: functional vs imperative

Definition: The ***state*** of a program is all of the current variable and heap values

- **Imperative** programs **destructively update** the state.
  - e.g., after executing `*x = y` (in a C program), the memory cell that `x` points to now holds the value `y`. Its old value is gone.
- **Functional** programs yield **new similar states** over time.
  - `let x = y in …` , however, only changes `x`'s value **within** the scope of the …

Example: functional vs. imperative

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

Example: functional vs. imperative

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

Example: functional vs. imperative

```
double avg(int x, int y) {
   double z = (double)(x + y);
   z = z / 2;
   printf("Answer: %g\n", z);
   return z;
}
```

```
let avg (x:int) (y:int) : float = begin




end
```

Example: functional vs. imperative

```
double avg(int x, int y) {
   double z = (double)(x + y);
   z = z / 2;
   printf("Answer: %g\n", z);
   return z;
}
```

```
let avg (x:int) (y:int) : float = begin
   let z = float_of_int (x + y) in



end
```

NOT the same as a semi-colon: commands vs expressions

Example: functional vs. imperative

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

```
let avg (x:int) (y:int) : float = begin
  let z = float_of_int (x + y) in
  let z = z /. 2.0 in



end
```

Example: functional vs. imperative

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

```
let avg (x:int) (y:int) : float = begin
  let z = float_of_int (x + y) in
  let z = z /. 2.0 in


end
```

even the operators are type-safe (in OCaml)

Example: functional vs. imperative

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

```
let avg (x:int) (y:int) : float = begin
  let z = float_of_int (x + y) in
  let z = z /. 2.0 in
  printf "Answer: %g\n" z ;

end
```

Example: functional vs. imperative

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

commands still exist, but limited to inherently "imperative" operations (I/O, saving to disk, etc.)

```
let avg (x:int) (y:int) : float = begin
  let z = float_of_int (x + y) in
  let z = z /. 2.0 in
  printf "Answer: %g\n" z ;

end
```

Example: functional vs. imperative

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

```
let avg (x:int) (y:int) : float = begin
  let z = float_of_int (x + y) in
  let z = z /. 2.0 in
  printf "Answer: %g\n" z ;
  z
end
```

Example: functional vs. imperative

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

no "return" statement, because everything is an expression

```
let avg (x:int) (y:int) : float = begin
  let z = float_of_int (x + y) in
  let z = z /. 2.0 in
  printf "Answer: %g\n" z ;
  z
end
```

Examples of functional languages

Examples of functional languages

- Lisp
- OCaml/SML
- Haskell

Examples of functional languages

- Lisp
- OCaml/SML
- Haskell
- Python
- JavaScript/TypeScript
- Java (???)
- Closure
- Ruby
- etc.

Examples of functional languages

- Lisp
- OCaml/SML
- Haskell
- Python
- JavaScript/TypeScript
- **Java (???)**
- Closure
- Ruby
- etc.

15.27. Lambda Expressions

Here are some examples of lambda expressions:

```
() -> {}                  // No parameters; result is void
() -> 42                  // No parameters, expression body
() -> null                // No parameters, expression body
() -> { return 42; }      // No parameters, block body with return
() -> { System.gc(); }    // No parameters, void block body
```

Functional advantages

Functional advantages

- Tractable program semantics
  - Procedures are functions (simplifies reasoning)
  - Formulate and prove assertions about code more easily
  - More readable (if you like math)

Functional advantages

- Tractable program semantics
  - Procedures are functions (simplifies reasoning)
  - Formulate and prove assertions about code more easily
  - More readable (if you like math)
- ***Referential transparency***
  - Replace any expression by its value without changing the result

Functional advantages

- Tractable program semantics
  - Procedures are functions (simplifies reasoning)
  - Formulate and prove assertions about code more easily
  - More readable (if you like math)
- ***Referential transparency***
  - Replace any expression by its value without changing the result
- “No” side-effects
  - Fewer errors

Functional disadvantages

Functional disadvantages

- Efficiency
  - Copying takes time

Functional disadvantages

- Efficiency
  - Copying takes time

Language	Speed	Space
C (gcc)	1.0	1.1
C++ (g++)	1.0	1.6
OCaml	1.5	2.9
Java (JDK -server)	1.7	9.1
Lisp	1.7	11
C# (mono)	2.4	5.6
Python	6.5	3.9
Ruby	16	5.0

17 small benchmarks

Functional disadvantages

- Efficiency
  - Copying takes time
- Compiler implementation
  - Frequent memory allocation

Language	Speed	Space
C (gcc)	1.0	1.1
C++ (g++)	1.0	1.6
OCaml	1.5	2.9
Java (JDK -server)	1.7	9.1
Lisp	1.7	11
C# (mono)	2.4	5.6
Python	6.5	3.9
Ruby	16	5.0

17 small benchmarks

Functional disadvantages

- Efficiency
  - Copying takes time
- Compiler implementation
  - Frequent memory allocation
- Unfamiliar (to you, and maybe those you're hiring!)
  - New programming style

Language	Speed	Space
C (gcc)	1.0	1.1
C++ (g++)	1.0	1.6
OCaml	1.5	2.9
Java (JDK -server)	1.7	9.1
Lisp	1.7	11
C# (mono)	2.4	5.6
Python	6.5	3.9
Ruby	16	5.0

17 small benchmarks

Functional disadvantages

- Efficiency
  - Copying takes time
- Compiler implementation
  - Frequent memory allocation
- Unfamiliar (to you, and maybe those you're hiring!)
  - New programming style
- Not appropriate for every program
  - Some programs are inherently stateful

Language	Speed	Space
C (gcc)	1.0	1.1
C++ (g++)	1.0	1.6
OCaml	1.5	2.9
Java (JDK -server)	1.7	9.1
Lisp	1.7	11
C# (mono)	2.4	5.6
Python	6.5	3.9
Ruby	16	5.0

17 small benchmarks

Object-oriented programming

Definition: in the ***object-oriented*** paradigm, programs are composed of interacting objects, each of which is responsible for some well-defined part of the program's state

Object-oriented programming

Definition: in the ***object-oriented*** paradigm, programs are composed of interacting objects, each of which is responsible for some well-defined part of the program's state

- underlying mathematical formalism:

Object-oriented programming

Definition: in the ***object-oriented*** paradigm, programs are composed of interacting objects, each of which is responsible for some well-defined part of the program's state

- underlying mathematical formalism: type systems? dictionaries?
  - still something of an open research problem

Object-oriented programming

Definition: in the ***object-oriented*** paradigm, programs are composed of interacting objects, each of which is responsible for some well-defined part of the program's state

- underlying mathematical formalism: type systems? dictionaries?
  - still something of an open research problem
- **extraordinarily common**

Object-oriented programming

Definition: in the ***object-oriented*** paradigm, programs are composed of interacting objects, each of which is responsible for some well-defined part of the program's state

- underlying mathematical formalism: type systems? dictionaries?
  - still something of an open research problem
- **extraordinarily common**
- models **the real world** well
  - objects are good abstractions for real-world entities and concepts

Object-oriented programming gotchas

- classes vs prototypes

Object-oriented programming gotchas

- classes vs prototypes
  - a ***class*** is a template for building objects (but is not itself an object!)
  - a ***prototype*** is an object that is used as a template for building other objects

Object-oriented programming gotchas

- classes vs prototypes
  - a ***class*** is a template for building objects (but is not itself an object!)
  - a ***prototype*** is an object that is used as a template for building other objects
- similar, but lead to **subtle differences**
  - prototypes can be modified at run time!

Object-oriented programming gotchas

- classes vs prototypes
  - a ***class*** is a template for building objects (but is not itself an object!)
  - a ***prototype*** is an object that is used as a template for building other objects
- similar, but lead to **subtle differenc**
  - prototypes can be modified at

Which of the two does Java use? What about JavaScript?

Object-oriented programming languages

Object-oriented programming languages

- Smalltalk
- Java
- C++
- C#
- Python
- JavaScript/TypeScript
- Swift
- R
- etc.

How can programming languages differ?

- programming paradigm
- **whether they have a type system**
  - and, if they do, what kind of type system they have
- library support
  - the standard library is especially important
- performance
- team/process factors
  - how well do you know the language
  - how easy it'll be to hire other developers who do

What is a type system, anyway?

What is a type system, anyway?

Definition: a ***type system*** is a set of rules that give every program element a ***type***, which is an upper bound on the set of possible values that that element can take on at run time

What is a type system, anyway?

Definition: a ***type system*** is a set of rules that give every program element a ***type***, which is an upper bound on the set of possible values that that element can take on at run time

- goal of a type system: **prevent errors** at run time due to unexpected values

What is a type system, anyway?

Definition: a ***type system*** is a set of rules that give every program element a ***type***, which is an upper bound on the set of possible values that that element can take on at run time

- goal of a type system: **prevent errors** at run time due to unexpected values
- **type theory** is the discipline of math (yes!) that studies the formal properties of type systems

What is a type system, anyway?

Definition: a ***type system*** is a set of rules that give every program element a ***type***, which is an upper bound on the set of possible values that that element can take on at run time

- goal of a type system: **prevent errors** at run time due to unexpected values
- **type theory** is the discipline of math (yes!) that studies the formal properties of type systems
- most programming languages include some kind of type system
  - exceptions: assembly, Lisp, a few others

Kinds of type systems

- Static vs dynamic checking

Kinds of type systems

- Static vs dynamic checking
  - ***statically typed*** languages have their types checked before the program runs, typically **at compile time**

Kinds of type systems

- Static vs dynamic checking
  - ***statically typed*** languages have their types checked before the program runs, typically **at compile time**
    - shares advantages/disadvantages with other static analyses

Kinds of type systems

- Static vs dynamic checking
  - ***statically typed*** languages have their types checked before the program runs, typically **at compile time**
    - shares advantages/disadvantages with other static analyses
  - ***dynamically typed*** languages have their types checked **at run time**, typically by a special interpreter or language runtime

Kinds of type systems

- Static vs dynamic checking
  - ***statically typed*** languages have their types checked before the program runs, typically **at compile time**
    - shares advantages/disadvantages with other static analyses
  - ***dynamically typed*** languages have their types checked **at run time**, typically by a special interpreter or language runtime
    - shares advantages/disadvantages with other dynamic analyses

Kinds of type systems

- Static vs dynamic checking
  - ***statically typed*** languages have their types checked before the program runs, typically **at compile time**
    - shares advantages/disadvantages with other static analyses
  - ***dynamically typed*** languages have their types checked **at run time**, typically by a special interpreter or language runtime
    - shares advantages/disadvantages with other dynamic analyses
- **Insight**: typechecking is just another program analysis

Static vs dynamic types

- Both are **common in practice**

Static vs dynamic types

- Both are **common in practice**
  - examples of each?

Static vs dynamic types

- Both are **common in practice**
  - examples of each?
    - Static: Java, C, Rust, OCaml, TypeScript, etc.
    - Dynamic: Python, Ruby, JavaScript, etc.

Static vs dynamic types

- Both are **common in practice**
  - examples of each?
    - Static: Java, C, Rust, OCaml, TypeScript, etc.
    - Dynamic: Python, Ruby, JavaScript, etc.
- **Ongoing debate** about the benefits

Static vs dynamic types

- Both are **common in practice**
  - examples of each?
    - Static: Java, C, Rust, OCaml, TypeScript, etc.
    - Dynamic: Python, Ruby, JavaScript, etc.
- **Ongoing debate** about the benefits
  - Benefits of static typing:
    - ???
  - Benefits of dynamic typing:
    - ???

Static vs dynamic types

- Both are **common in practice**
  - examples of each?
    - Static: Java, C, Rust, OCaml, TypeScript, etc.
    - Dynamic: Python, Ruby, JavaScript, etc.
- **Ongoing debate** about the benefits
  - Benefits of static typing:
    - early detection of errors, types are documentation
  - Benefits of dynamic typing:
    - faster prototyping, no false positives

Other ways type systems differ

Other ways type systems differ

- **Implicit** vs **explicit**

Other ways type systems differ

- **Implicit** vs **explicit**
  - “do you write the types yourself”
  - almost all mainstream, static languages are explicit

Other ways type systems differ

- **Implicit** vs **explicit**
  - “do you write the types yourself”
  - almost all mainstream, static languages are explicit
- **Strength** of the type system
  - not all type systems can prove the same properties

Other ways type systems differ

- **Implicit** vs **explicit**
  - “do you write the types yourself”
  - almost all mainstream, static languages are explicit
- **Strength** of the type system
  - not all type systems can prove the same properties
  - e.g., Kotlin **guarantees no null-pointer dereferences**, but Java doesn’t (both compile to Java bytecode)

Other ways type systems differ

- **Implicit** vs **explicit**
  - “do you write the types yourself”
  - almost all mainstream, static languages are explicit
- **Strength** of the type system
  - not all type systems can prove the same properties
  - e.g., Kotlin **guarantees no null-pointer dereferences**, but Java doesn’t (both compile to Java bytecode)
  - stronger types can be added to a language (**ask me more**)
    - “pluggable types”

How can programming languages differ?

- programming paradigm
- whether they have a type system
  - and, if they do, what kind of type system they have
- **library support**
  - the standard library is especially important
- performance
- team/process factors
  - how well do you know the language
  - how easy it'll be to hire other developers who do

Library support

- **Key question**: do the right tools for the job you need to do exist in the language?

Library support

- **Key question**: do the right tools for the job you need to do exist in the language?

Remember: **Don't Repeat Yourself**
If someone else has already built what you need, don't build it again

Library support

- **Key question**: do the right tools for the job you need to do exist in the language?
- Tied to **language popularity**: languages that are more popular have better libraries, so people are more likely to use them
  - positive feedback loop!

Library support

- **Key question**: do the right tools for the job you need to do exist in the language?
- Tied to **language popularity**: languages that are more popular have better libraries, so people are more likely to use them
  - positive feedback loop!
- Common situation: you need library A and library B, but A is written in language L and B is written in language M
  - What to do?

Multi-language projects

- In a given project, not all code needs to be written in the same language!

Multi-language projects

- In a given project, not all code needs to be written in the same language!

Multi-language projects are common!
Developer quote: ““My last 4 jobs have been apps that called: Java from C#, and C# from F#; Java from Ruby; Python from Tcl, C++ from Python, and C from Tcl; Java from Python, and Java from Scheme (And that's not even counting SQL, JS, OQL, etc.)””

Multi-language projects

- In a given project, not all code needs to be written in the same language!
- Multi-language projects allow you to **choose the right language** for each part of your application

Multi-language projects

- In a given project, not all code needs to be written in the same language!
- Multi-language projects allow you to **choose the right language** for each part of your application

For example, concurrency might be better handled in F#/OCaml (immutable functional) or Ruby (designed to hide such details), while low-level OS or hardware access is much easier in C or C++, while rapid prototyping is much easier in Python or Lua, etc.

Multi-language projects

- In a given project, not all code needs to be written in the same language!
- Multi-language projects allow you to **choose the right language** for each part of your application
  - but **complicate** many parts of software engineering

Multi-language projects

- In a given project, not all code needs to be written in the same language!
- Multi-language projects allow you to **choose the right language** for each part of your application
  - but **complicate** many parts of software engineering
- Traditional architecture:

Multi-language projects

- In a given project, not all code needs to be written in the same language!
- Multi-language projects allow you to **choose the right language** for each part of your application
  - but **complicate** many parts of software engineering
- Traditional architecture:
  - Application **kernel** is written in a statically typed, optimized, compiled language

Multi-language projects

- In a given project, not all code needs to be written in the same language!
- Multi-language projects allow you to **choose the right language** for each part of your application
  - but **complicate** many parts of software engineering
- Traditional architecture:
  - Application **kernel** is written in a statically typed, optimized, compiled language
  - **Scripts** are written in a dynamically typed, interpreted language

Multi-language projects

- In a given project, not all code needs to be written in the same l
- N anguage for e
  - but **complicate** many parts of software engineering
- Traditional architecture:
  - Application **kernel** is written in a statically typed, optimized, compiled language
  - **Scripts** are written in a dynamically typed, interpreted language

Examples: Emacs (C / Lisp), Adobe Lightroom (C++ / Lua), NRAO Telescope (C / Python), Google Android (C / Java), most games (C++ / Lua), etc.

Multi-language projects

C/C++ is a lingua franca

- In a given project, not all code needs to be written in the same l
- N anguage for e
  - but **complicate** many parts of software engineering
- Traditional architecture:
  - Application **kernel** is written in a statically typed, optimized, compiled language
  - **Scripts** are written in a dynamically typed, interpreted language

Examples: Emacs (**C** / Lisp), Adobe Lightroom (**C++** / Lua), NRAO Telescope (**C** / Python), Google Android (**C** / Java), most games (**C++** / Lua), etc.

Multi-language projects

- Another common approach: ***common language infrastructure***
  - enables easy integration and interoperability

Multi-language projects

- Another common approach: ***common language infrastructure***
  - enables easy integration and interoperability
- Examples:
  - .NET framework (Microsoft)
    - C++, C#, J#, F#, Visual Basic, etc.
  - Java bytecode + Java virtual machine
    - Java, Scala, Kotlin, Closure, etc.
  - LLVM bytecode
  - etc.

Multi-language projects: complications

Multi-language projects: complications

- **Integrating data and control flow** across languages can be difficult

Multi-language projects: complications

- **Integrating data and control flow** across languages can be difficult
- **Debugging** can be harder
  - Especially as values flow and control flow from language A to language B

Multi-language projects: complications

- **Integrating data and control flow** across languages can be difficult
- **Debugging** can be harder
  - Especially as values flow and control flow from language A to language B
- **Build process** (next week) becomes more complicated

Multi-language projects: complications

- **Integrating data and control flow** across languages can be difficult
- **Debugging** can be harder
  - Especially as values flow and control flow from language A to language B
- **Build process** (next class) becomes more complicated
- **Developer expertise** is required in multiple languages
  - Must understand types (etc.) in **all** languages

Multi-language projects: complications

- **Integrating data and control flow** across languages can be difficult
- **Debugging** can be harder
  - Especially as values flow and control flow from language A to language B
- **Build process** (next class) becomes more complicated
- **Developer expertise** is required in multiple languages
  - Must understand types (etc.) in **all** languages
- Most **tools are language specific**: testing frameworks (+ generation, coverage, etc.), static analysis, build systems, debuggers, etc.

How can programming languages differ?

- programming paradigm
- whether they have a type system
  - and, if they do, what kind of type system they have
- library support
  - the standard library is especially important
- **performance**
- team/process factors
  - how well do you know the language
  - how easy it'll be to hire other developers who do

Language performance

- Three **main axes** to trade-off between languages:

Language performance

- Three **main axes** to trade-off between languages:
  - **Performance** (“how fast do programs run”)

Language performance

- Three **main axes** to trade-off between languages:
  - **Performance** (“how fast do programs run”)
  - **Safety** (“how easy is it to make mistakes”)

Language performance

- Three **main axes** to trade-off between languages:
  - **Performance** (“how fast do programs run”)
  - **Safety** (“how easy is it to make mistakes”)
  - Developer **Effort** (“how hard do I have to think to write a program in this language”)

Language performance

- Three **main axes** to trade-off between languages:
  - **Performance** (“how fast do programs run”)
  - **Safety** (“how easy is it to make mistakes”)
  - Developer **Effort** (“how hard do I have to think to write a program in this language”)
- Different languages choose different trade-offs. Examples:

Language performance

- Three **main axes** to trade-off between languages:
  - **Performance** (“how fast do programs run”)
  - **Safety** (“how easy is it to make mistakes”)
  - Developer **Effort** (“how hard do I have to think to write a program in this language”)
- Different languages choose different trade-offs. Examples:
  - Rust: good performance and safety, hard to write

Language performance

- Three **main axes** to trade-off between languages:
  - **Performance** (“how fast do programs run”)
  - **Safety** (“how easy is it to make mistakes”)
  - Developer **Effort** (“how hard do I have to think to write a program in this language”)
- Different languages choose different trade-offs. Examples:
  - Rust: good performance and safety, hard to write
  - Python: easy to write, okay safety, slow

Language performance

- Three **main axes** to trade-off between languages:
  - **Performance** (“how fast do programs run”)
  - **Safety** (“how easy is it to make mistakes”)
  - Developer **Effort** (“how hard do I have to think to write a program in this language”)
- Different languages choose different trade-offs. Examples:
  - Rust: good performance and safety, hard to write
  - Python: easy to write, okay safety, slow
  - C: good performance, easy-ish to write, very unsafe

What impacts performance

What impacts performance

- #1: **safety features enforced at run time**

What impacts performance

- #1: **safety features enforced at run time**
  - dynamic type checking: type safety
  - **garbage collection**: memory safety
  - exceptions: segfault safety

What impacts performance

- #1: **safety features enforced at run time**
  - dynamic type checking: type safety
  - **garbage collection**: memory safety
  - exceptions: segfault safety
- Also relevant: **optimizations**

What impacts performance

- #1: **safety features enforced at run time**
  - dynamic type checking: type safety
  - **garbage collection**: memory safety
  - exceptions: segfault safety
- Also relevant: **optimizations**
  - **interpreted** languages almost always slower: no optimizing compiler

What impacts performance

- #1: **safety features enforced at run time**
  - dynamic type checking: type safety
  - **garbage collection**: memory safety
  - exceptions: segfault safety
- Also relevant: **optimizations**
  - **interpreted** languages almost always slower: no optimizing compiler
  - JITs (***just-in-time compilers***) can produce surprisingly fast code
    - e.g., Java Virtual Machine

Trade-off: safety features

- #1 performance problem: **safety features enforced at run time**

Trade-off: safety features

- #1 performance problem: **safety features enforced at run time**
- So, why not **enforce safety at compile time** instead?

Trade-off: safety features

- #1 performance problem: **safety features enforced at run time**
- So, why not **enforce safety at compile time** instead?
  - requires **static analysis** (= there will be false positives)

Trade-off: safety features

- #1 performance problem: **safety features enforced at run time**
- So, why not **enforce safety at compile time** instead?
  - requires **static analysis** (= there will be false positives)
  - harder for programmers (trades off against **effort**)

Trade-off: safety features

- #1 performance problem: **safety features enforced at run time**
- So, why not **enforce safety at compile time** instead?
  - requires **static analysis** (= there will be false positives)
  - harder for programmers (trades off against **effort**)
    - the garbage collector in Java/Go/etc. is automatic
    - but writing Rust code requires follows its (complex) type discipline

Trade-off: safety features

- #1 performance problem: **safety features enforced at run time**
- So, why not **enforce safety at compile time** instead?
  - requires **static analysis** (= there will be false positives)
  - harder for programmers (trades off against **effort**)
    - the garbage collector in Java/Go/etc. is automatic
    - but writing Rust code requires follows its (complex) type discipline
  - bottom line: statically safe languages **can be faster**, but are **generally harder to program in**

How can programming languages differ?

- programming paradigm
- whether they have a type system
  - and, if they do, what kind of type system they have
- library support
  - the standard library is especially important
- performance
- **team/process factors**
  - how well do you know the language
  - how easy it'll be to hire other developers who do

Team/process factors

- **Learning** a new programming language takes time

Team/process factors

- **Learning** a new programming language takes time
  - Becoming productive shouldn't take that long
    - but, this scales with how hard the language is to program in (+ access to mentors, etc.)

Team/process factors

- **Learning** a new programming language takes time
  - Becoming productive shouldn't take that long
    - but, this scales with how hard the language is to program in (+ access to mentors, etc.)
  - Becoming an expert takes a long time!

Team/process factors

- **Learning** a new programming language takes time
  - Becoming productive shouldn't take that long
    - but, this scales with how hard the language is to program in (+ access to mentors, etc.)
  - Becoming an expert takes a long time!
- If you need performance, you usually need **at least one expert**
  - cf. AWS employs some JVM experts to tune the garbage collector for AWS services that use Java

Team/process factors

- Learning a new programming language takes time
  - program
  - Becoming an expert takes a long time!
- If you need performance, you usually need **at least one expert**
  - cf. AWS employs some JVM experts to tune the garbage collector for AWS services that use Java

Implication: if you're going to need an expert, make sure you have one! This often seriously limits your choice of languages in practice :(

Team/process factors

- Because learning a new language takes time, the **popularity** of a language is also a plus:

Team/process factors

- Because learning a new language takes time, the **popularity** of a language is also a plus:
  - it's **easier to hire** new engineers who already know the language, and therefore can ramp up faster

Team/process factors

- Because learning a new language takes time, the **popularity** of a language is also a plus:
  - it's **easier to hire** new engineers who already know the language, and therefore can ramp up faster
  - but this impact is relatively small over a typical engineer's tenure at a company

Team/process factors

- Because learning a new language takes time, the **popularity** of a language is also a plus:
  - it's **easier to hire** new engineers who already know the language, and therefore can ramp up faster
  - but this impact is relatively small over a typical engineer's tenure at a company
- Implication: if all else is equal, **choose the more popular** language

When to rewrite

- the reading talked about moving a service from one language to another
  - why?

When to rewrite

- the reading talked about moving a service from one language to another
  - why? **Performance problems**.

When to rewrite

- the reading talked about moving a service from one language to another
  - why? **Performance problems**.
- This is usually a **risky thing** to do:
  - you're not building new features
  - integration problems
  - will the benefits be worth it?

When to rewrite

- the reading talked about moving a service from one language to another
  - why? **Performance problems**.
- This is usually a **risky thing** to do:
  - you're not building new features
  - integration problems
  - will the b

Implication: rewriting is a good idea if you're confident that the benefits of the new language are worthwhile, but be cautious: it can expensive!

Takeaways

- there is a wider world of languages than just imperative and object-oriented (but those are the most popular)
  - learning to write functional code can make you a better programmer
- different programming languages have different trade-offs
  - performance vs safety vs ease of use vs …
- when starting a new project, think carefully about the requirements before choosing a language
- rewrite a project in a new language only after careful consideration