Principles of Software Construction: Objects, Design, and Concurrency

IDEs, Build system, Continuous Integration, Libraries

Christian Kästner **Vincent Hellendoorn**

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Outline

- On Homework 1
- Abstraction, Reuse, and Programming Tools
- For each in {IDE, Build systems, libraries, CI}:
  - What is it today?
  - What is under the hood?
  - What is next?

Homework 1

Welcome to the deep end!

- Java/TS + IDE + Maven/Npm + GitHub + Travis + linter!?

Homework 1

Welcome to the deep end!

- Java/TS + IDE + Maven/Npm + GitHub + Travis + linter!?
- We're here to help:
  - Recitation tomorrow, walks through all this setup
  - Find some clarifications on Piazza
    - E.g., only implement what is asked for; all other functionality (repeating, flipping question/answer) is already there.
  - Use office hours (see course calendar)

Homework 1

Welcome to the deep end!

- Java/TS + IDE + Maven/Npm + GitHub + Travis + linter!?
- We're here to help:
  - Recitation tomorrow, walks through all this setup
  - Find some clarifications on Piazza
    - E.g., only implement what is asked for; all other functionality (repeating, flipping question/answer) is already there.
  - Use office hours (see course calendar)
- Actual coding effort is small -- reading & setting up is the point
- Small typo detected on Piazza in `mostmistakes.ts`; fixed now. Not essential to your HW.

Mini-quiz

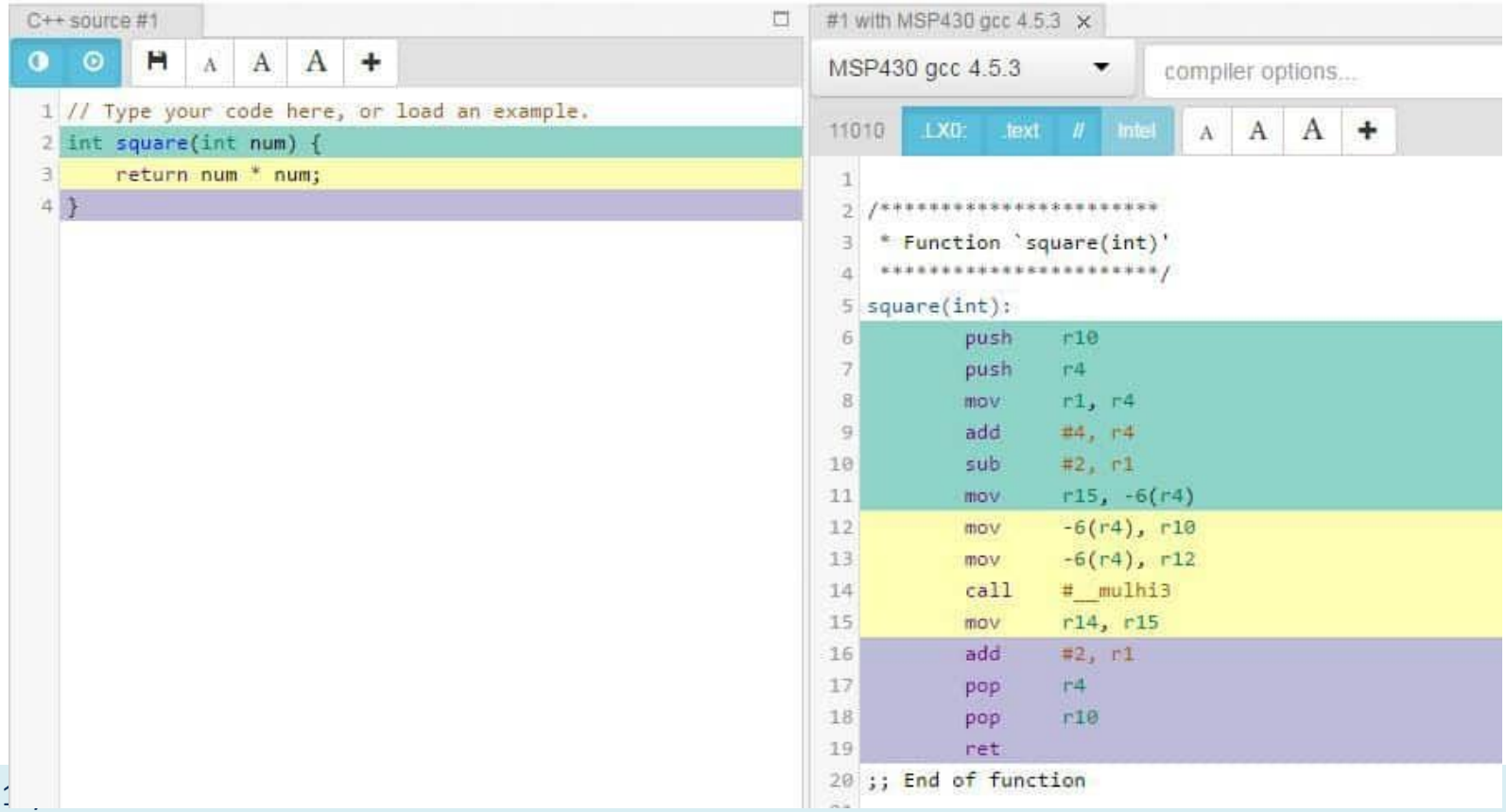
https://forms.gle/9tnB5BszVz9KTY7r5

Outline

- On Homework 1
- **Abstraction, Reuse, and Programming Tools**
- For each in {IDE, Build systems, libraries, CI}:
  - What is it today?
  - What is under the hood?
  - What is next?

Automation Requires Abstraction

Automation Requires Abstraction



Automation Requires Abstraction

We all treat familiar levels of abstraction as normal/natural

- That's fine if you only drive your car
  - Not so much if you are a mechanic
  - How to debug a broken transmission?
- Also slow to evolve
  - *Conf.* people adamantly refusing to use an automatic
- Engineers seek out abstractions that simplify their work, help focus on the hard parts
  - They also know what is beneath the abstractions

Automation Requires Abstraction

Today's "normal":

- Integrated-development environments (IDEs) galore
  - Web-based too! Press "." on a GitHub (file) page 😲
- Frequent build, test, release
  - In some companies, every commit is a "release"
- Never write code for which there is a useful library
  - Define "useful" (we will)
- All of the above, entangled

Outline

- On Homework 1
- Abstraction, Reuse, and Programming Tools
- For each in {IDE, Build systems, libraries, CI}:
  - **What is it today?**
  - What is under the hood?
  - What is next?

Today’s toolchain: a quick overview

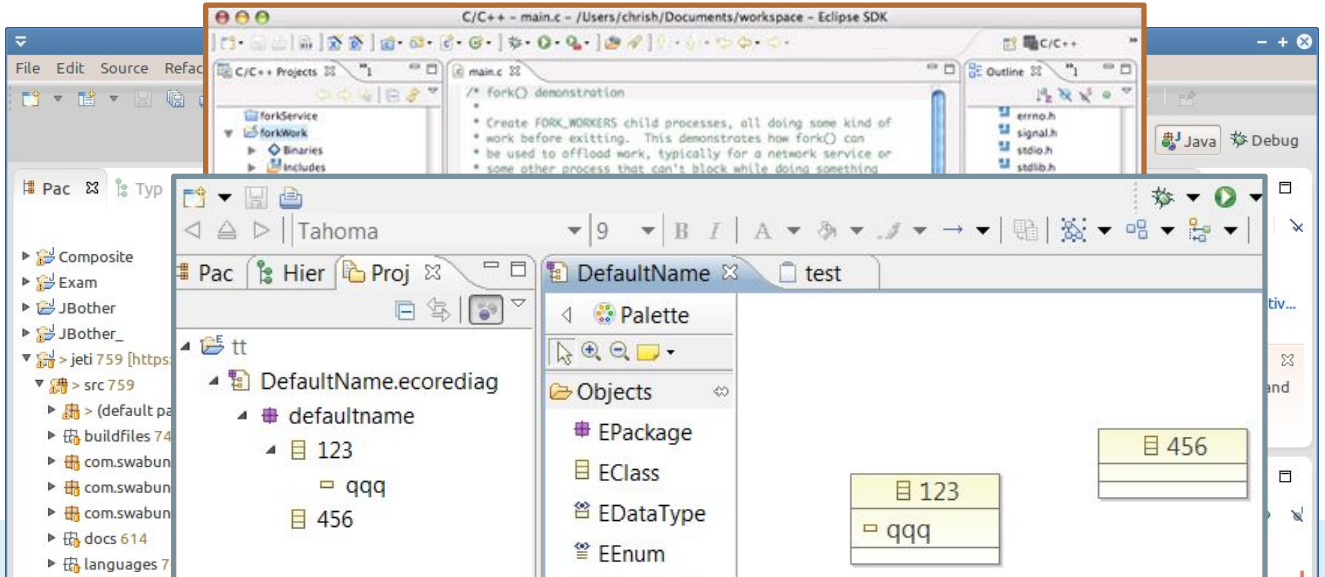
IDEs:

- Integrated Development Environments, bundle development workflows in a single UI
  - Editing, refactoring, running & debugging, adding dependencies, compiling, deploying, plugins, you name it
  - They often try to be everything, with mixed results
  - Leverage them to the fullest extent, to automate and check your work

Today's toolchain: a quick overview

IDEs:

- Eclipse was the dominant player in Java for 20-odd years, owing to its powerful backbone and plugin architecture



Today’s toolchain: a quick overview

IDEs:

- Recently, IntelliJ has been more dominant
  - Packs a lot of “recipes” to create certain types of projects (e.g., web-app with Spring & Maven)

Today’s toolchain: a quick overview

IDEs:

- Recently, IntelliJ has been more dominant
  - Packs a lot of “recipes” to create certain types of projects (e.g., web-app with Spring & Maven)
- VSCode is surging in popularity
  - Local & web, lightweight but with a massive plugin ecosystem
    - Quick tangent: if you can build either a large product or a platform, build a platform

Today's toolchain: a quick overview

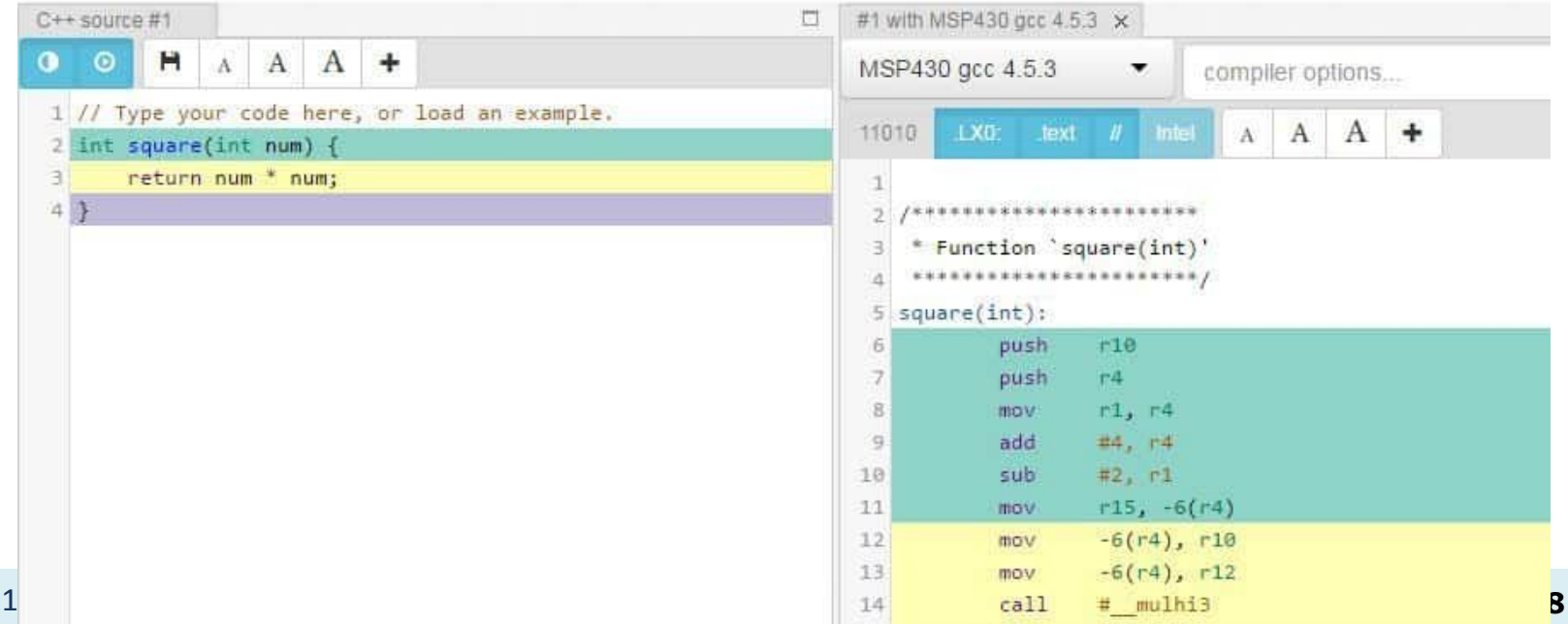
IDEs:

- Recently, IntelliJ has been more dominant
  - Packs a lot of "recipes" to create certain types of projects (e.g., web-app with Spring & Maven)
- VSCode is surging in popularity
  - Local & web, lightweight but with a massive plugin ecosystem
    - Quick tangent: if you can build either a large product or a platform, build a platform
- But choose based on need!
  - You can relearn key-bindings; "killer features" are rare and temporary
  - E.g., Android: might want Android Studio (itself built on IntelliJ) since Google supports it
  - For this homework, choose what you'd like. We suggest IntelliJ for Java, VSCode for TS

Today's toolchain: a quick overview

Build Systems:

- How does this happen?



Today’s toolchain: a quick overview

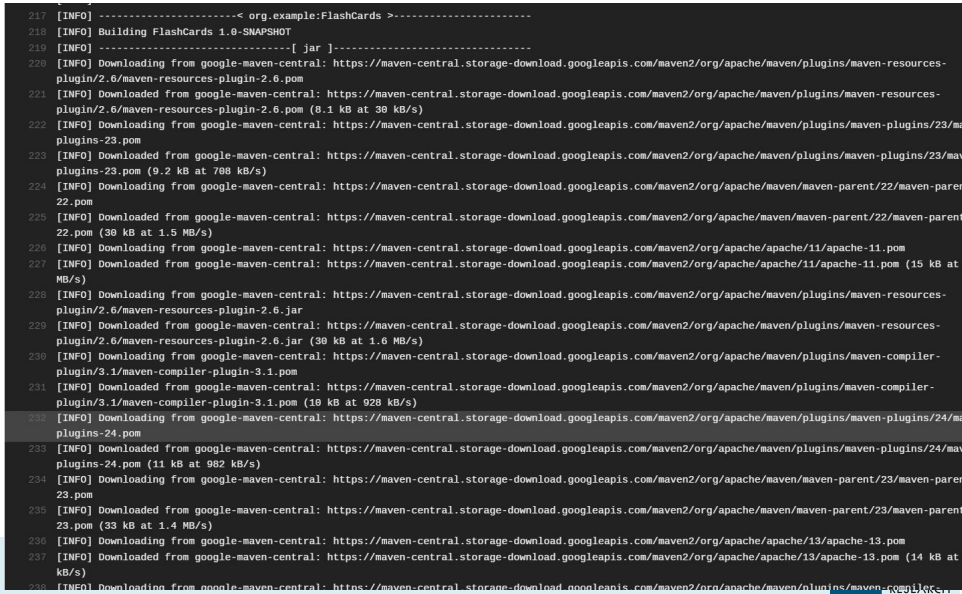
Build Systems:

- Compiling is “easy” when all your source code is here
  - (Please don’t tell a compiler expert I said that)
- Nowadays, your code is not “here”
  - Even libraries that you use in the IDE!
  - Interfaces make that possible

Today’s toolchain: a quick overview

Build Systems:

- Compiling is “easy” when all your source code is here
  - (Please don’t tell a compiler expert I said that)
- Nowadays, your code is not “here”
  - Even libraries that you use in the IDE!
  - Interfaces make that possible
- Study the Travis log:
  - What is it doing?
  - Downloading, compiling, running checks
  - Most of this is “building”, using Maven
  - More on Travis later



Today’s toolchain: a quick overview

Build Systems:

- Has a few basic tasks:
  - Compiling & linking, to produce an executable
  - Creating secondary *artifacts*, e.g. documentation-pages, linter reports, test suite reports
  - Different levels of “depth” may be appropriate, for large code bases (e.g. Google)

Today's toolchain: a quick overview

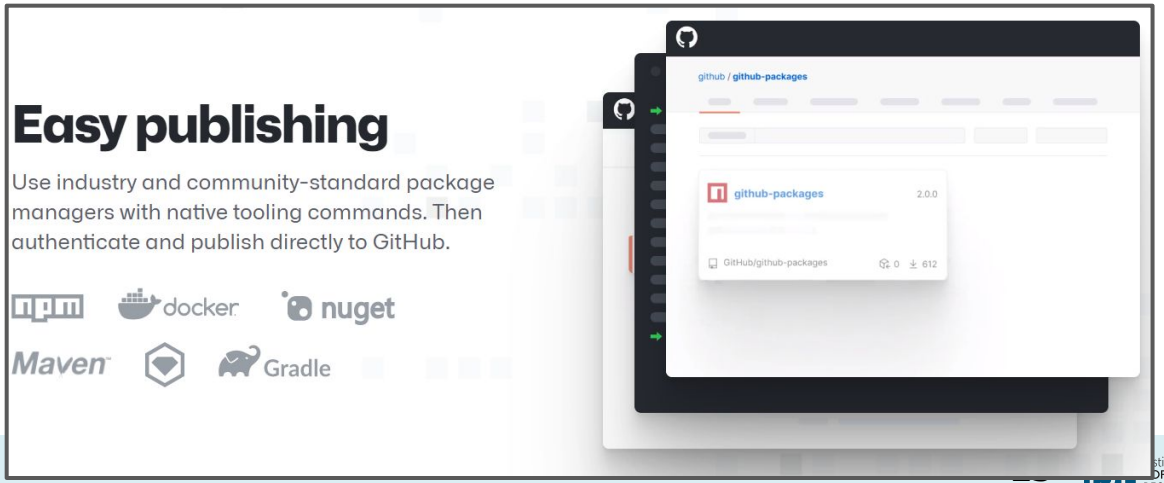
Build Systems:

- Has a few basic tasks:
  - Compiling & linking, to produce an executable
  - Creating secondary *artifacts*, e.g. documentation-pages, linter reports, test suite reports
  - Different levels of "depth" may be appropriate, for large code bases (e.g. Google)
- Popular options:
  - For Java: Maven and Gradle -- historically Ant.
    - You could do any homework in either; we're not attached to one
  - For JS/TS: Node(JS)
    - Generally coupled with the Node Package Manager (NPM)
  - Often built into IDEs, as plugins

Today’s toolchain: a quick overview

Libraries:

- Myriad. Publicly hosted on various *package managers*
  - Often tied, but not inextricably linked, to build tools, and languages
  - Maven/Gradle for Java, NPM for JS/TS, Nuget for C#, ...
  - Registries of managers, e.g., GitHub Packages



Today’s toolchain: a quick overview

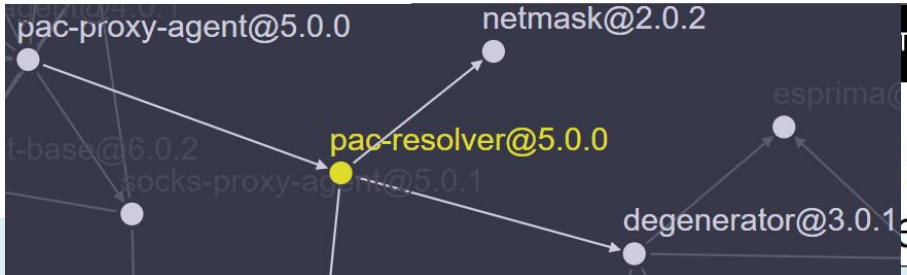
Libraries:

- Myriad. Publicly hosted on various *managers*
  - Often tied, but not inextricably linked, to build tools, and languages
  - Maven, Gradle, NPM, Nuget, Docker, …
  - Registries of managers, e.g., GitHub Packages
- Releases are generally fast-paced or frigid
  - Almost all volunteer-based, so support waivers, as does documentation quality
  - Often open-source, so you can check out the status & details on GitHub
  - Beware of vulnerabilities and bugs, esp. with minor-releases and nightly’s, old packages

Today’s toolchain: a quick overview

Libraries:

- A Case-Study:
  - ‘pac-resolver’ (3M weekly downloads) has a major security vulnerability
    - Uses ‘degenerator’ (same author), which misuses a Node module
    - “The vm module is not a security mechanism. Do not use it to run untrusted code.”
    - (a mistake that’s been made before: people rarely read disclaimers)
  - ‘pac-proxy-agent’ (**2M weekly** downloads, same author) uses the above
    - Is widely popular, the main reason people use ‘degenerator’
    - Most people using this package have never heard of the latter -- many never will



Today’s toolchain: a quick overview

Continuous Integration:

- Automates standard build, test, deploy pipelines
  - Technically, the latter is “CD”
  - Typically builds from scratch in a clean *container*
  - Often tied to code-review; triggers on new commits, pull requests
    - Ideally, official releases pass the build
  - Produces (long) logs with debugging outputs

Today’s toolchain: a quick overview

Not mentioned:

- Docker: containerize applications for coarse-grained reuse
- Cloud: deploy and scale rapidly, release seamlessly
- Bug/Issue trackers, often integrated with reviews

Outline

- On Homework 1
- **Abstraction, Reuse, and Programming Tools**
- For each in {IDE, Build systems, libraries, CI}:
  - What is it today?
  - What is under the hood?
  - What is next?

Behind the Abstraction

First, a bit of nuance:

- Automation vs. Reuse
  - We tend to automate common chains of actions
    - Gear-up := {Press clutch, switch gear, release clutch while accelerating}
  - To facilitate reusing such “subroutines”, we introduce abstractions
    - Accelerate in ‘D’ => Gear-up when needed

Behind the Abstraction

First, a bit of nuance:

- Automation vs. Reuse
  - We tend to automate common chains of actions
    - Gear-up := {Press clutch, switch gear, release clutch while accelerating}
  - To facilitate reusing such “subroutines”, we introduce abstractions
    - Accelerate in ‘D’ => Gear-up when needed
- Reuse vs. Interfaces
  - Interfaces facilitate reuse through abstraction
    - Allow upgrading implementation without breaking things
    - Provide explicit & transparent contract

Behind the Abstraction

First, a bit of nuance:

- Most tools are abstractions of common commands
  - Typically operated via GUI and/or a DSL
  - Obvious for Travis: just read the Yaml
    - Script-like languages are common
    - Involving a vocabulary of “targets”
    - E.g., `mvn site`

Behind the Abstraction

First, a bit of nuance:

- Most tools are abstractions of common commands
  - Typically operated via GUI and/or a DSL
  - Obvious for Travis: just read the Yaml
    - Script-like languages are common
    - Involving a vocabulary of “targets”
    - E.g., `mvn site`
- Abstraction can also “trap” us
  - When/how do we leave the abstraction?
  - Command-line comes built into IDEs for a reason
  - Non-trivial in general! May require switching/“patching” libraries
    - E.g., Maven → Gradle for more unusual build routines

Outline

- On Homework 1
- Abstraction, Reuse, and Programming Tools
- For each in {IDE, Build systems, libraries, CI}:
  - What is it today?
  - **What is under the hood?**
  - What is next?

IDEs

Automate common programming actions:

- Handy refactorings, suggestions
  - E.g., just press `alt+enter` in IntelliJ while highlighting nearly any code
    - Keyboard shortcuts are super useful: explore your IDE!
  - These can make you a better programmer: encode a lot of best-practices
    - Though, don't read into them too much

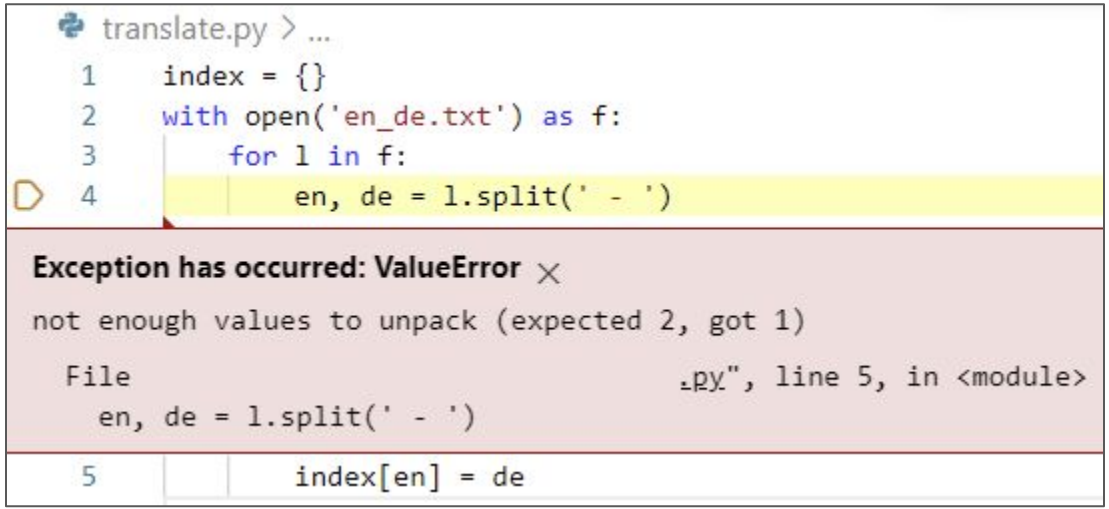
IDEs

- The engine: continuous parsing, building
  - Key feature: most partial programs don't parse, but IDEs make sense of them
  - That allows quickly relaying compile warnings/errors and useful suggestions
  - Same with API resolution
- Powered by rapid incremental compilation
  - Only build what has been updated
    - Virtually every edit you make triggers a compilation, re-linking
    - Of just the changed code and its dependencies
  - Works because *very little* of the code changes most of the time
    - But no free lunch: tends to drop optimizations (mostly fine), may struggle with big projects
  - Just try it: call an API with the wrong parameters & see how fast it triggers an alert; contrast with running a full Maven build (e.g., with `mvn install`)

IDEs

Automate common programming actions:

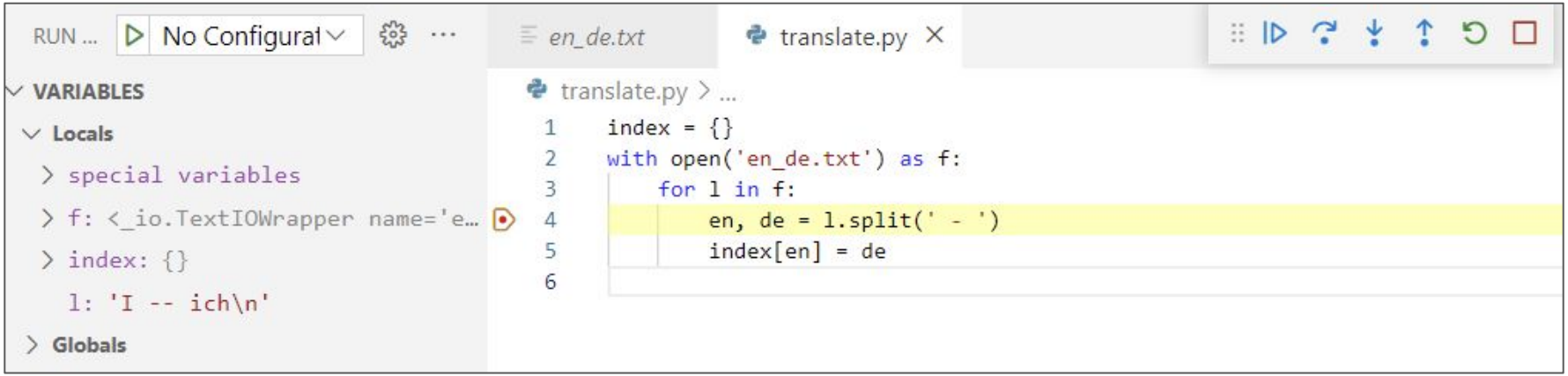
- Debugging
  - Often the default mode when you run in the IDE



IDEs

Automate common programming actions:

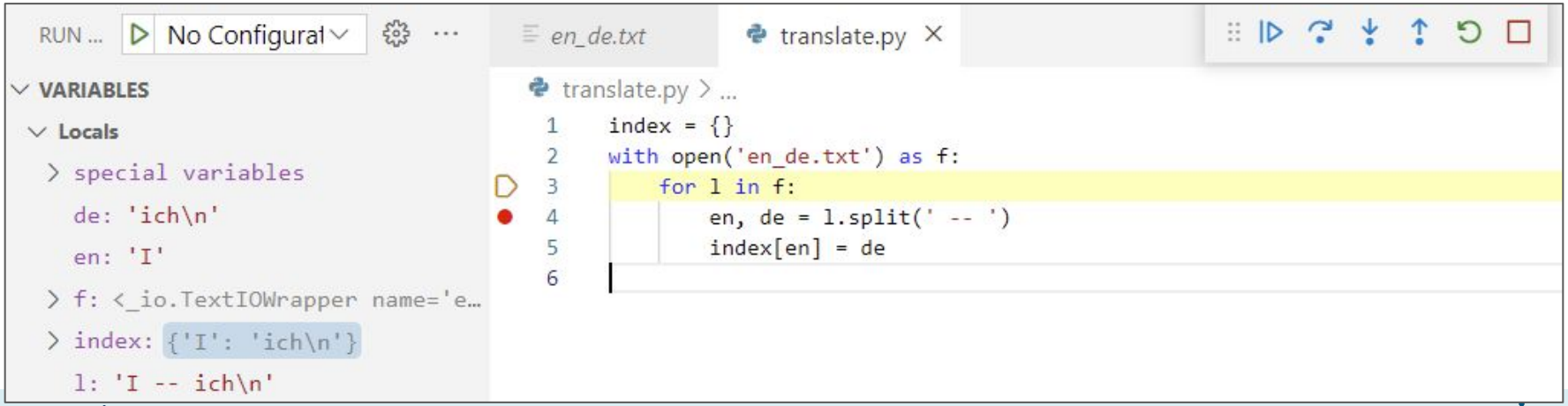
- Debugging
  - Often the default mode when you run in the IDE
  - Allows setting breakpoints
    - Which give you rich insight into execution



IDEs

Automate common programming actions:

- Debugging
  - Often the default mode when you run in the IDE
  - Allows setting breakpoints
    - Which give you rich insight into execution



IDEs

- IDE designers spend a lot of time automating common development tasks
  - Sometimes they get a little too helpful (modifying pom's)
  - Many plugins provide customized experience
  - Mostly evolve with new tools, prioritizing emerging routines
- Useful to know how these actions work
  - Often not much more than invoking commands for you
    - VSCode, IntelliJ are very explicit about this in the terminal -- great for customization

```
"C:\Program Files\Java\jdk-16.0.1\bin\java.exe" -ea -Didea.test.cyclic.buffer.size=1048576 "-javaagent:C:\Program Files\JetBrains\IntelliJ ID

Process finished with exit code 0
```

Build Systems

- These days: intricately tied with IDEs, package managers
- Projects often come with a build config file or two
  - ‘pom.xml’ for Maven
  - ‘tsconfig.json’ + ‘package.json’ for TypeScript+NPM -- the second deals with packages
  - These can be nested, one per (sub-)directory, to compose larger systems
    - On GitHub, you can create links across repositories

Build Systems

- These days: intricately tied with IDEs, package managers
- Projects often come with a build config file or two
  - ‘pom.xml’ for Maven
  - ‘tsconfig.json’ + ‘package.json’ for TypeScript+NPM -- the second deals with packages
  - These can be nested, one per (sub-)directory, to compose larger systems
    - On GitHub, you can create links across repositories
  - Specifies:
    - Compilation source and target version
    - High-level configuration options
    - Targets for various phases in development
      - “lifecycle” in Maven; e.g. ‘compile’, ‘test’, ‘deploy’
    - Often involving plugins
    - Dependencies with versions
      - Not shown: in package.json

```
{
    "compilerOptions": {
      "target": "es2016",
      "module": "commonjs",
      "sourceMap": true,
      "strict": true,
      "esModuleInterop": true,
      "moduleResolution": "node",
      "outDir": "dist"
    }
  }
```

Libraries & Frameworks

Packages can be either:

- Libraries:
  - A set of classes and methods that provide reusable functionality
  - Typically: programmer calls, library returns data, that's it.

Libraries & Frameworks

Packages can be either:

- Libraries:
  - A set of classes and methods that provide reusable functionality
  - Typically: programmer calls, library returns data, that's it.
- Frameworks:
  - Reusable skeleton code that can be customized into an application
  - Framework calls back into client code
    - The Hollywood principle: "Don't call us. We'll call you."
  - E.g., Android development: you declare your UI elements, activities to be composed
  - Principle: inversion of control

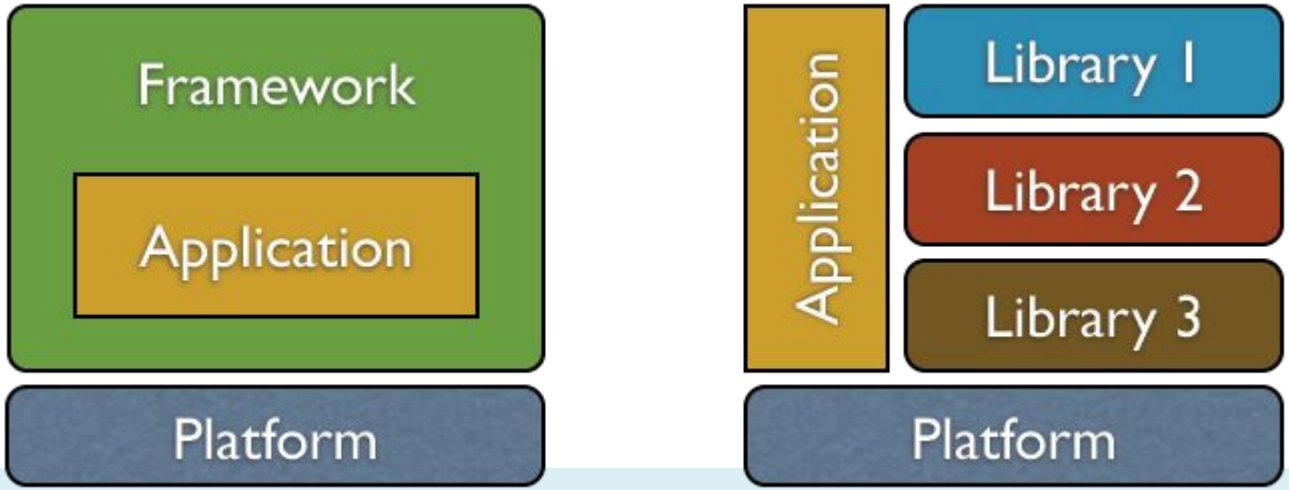
Libraries & Frameworks

Packages can be either:

- Libraries:
  - A set of classes and methods that provide reusable functionality
  - Typically: programmer calls, library returns data, that's it.
- Frameworks:
  - Reusable skeleton code that can be customized into an application
  - Framework calls back into client code
    - The Hollywood principle: "Don't call us. We'll call you."
  - E.g., Android development: you declare your UI elements, activities to be composed
  - Principle: inversion of control
- You typically use zero/one framework and many libraries
  - Frameworks might be especially constraining, but for good reason.
  - Some tools are a bit of both, and not all frameworks quite invert control

Libraries & Frameworks

Which kind is a command-line parsing package?



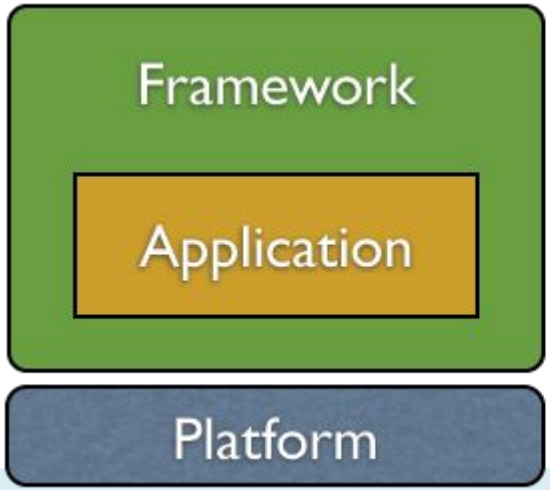
http://tom.lokhorst.eu/2010/09/why-libraries-are-better-than-frameworks`

Libraries & Frameworks

Which kind is a command-line parsing package?

How about a tool that runs tests based on annotations you add in your code?

- More on Thursday



http://tom.lokhorst.eu/2010/09/why-libraries-are-better-than-frameworks`

Libraries

Look into:

- Stated Goal:
  - A simple interface (“get started in one line!”) also means lots of abstraction
  - That’s neither good nor bad; know what you need
  - Docs with “advanced use cases” are always neat
- Maintenance:
  - Active release cycle, recent updates to documentation
  - GitHub build status, issue tracker (filled with unmerged ‘dependabot’ PRs?)
  - Lots of companies deliberately lag by one minor (or even major) version
- Recursive dependencies
  - Myriad, beyond inspection. Using OSS in corporate environments is a headache

Frameworks

Whitebox:

- Extension via subclassing and overriding methods
- Common design pattern(s):
  - Template method
- Subclass has main method but gives control to framework

Blackbox:

- Extension via implementing a plugin interface
- Common design pattern(s):
  - Command
  - Observer
- Plugin-loading mechanism loads plugins and gives control to the framework

Continuous Integration

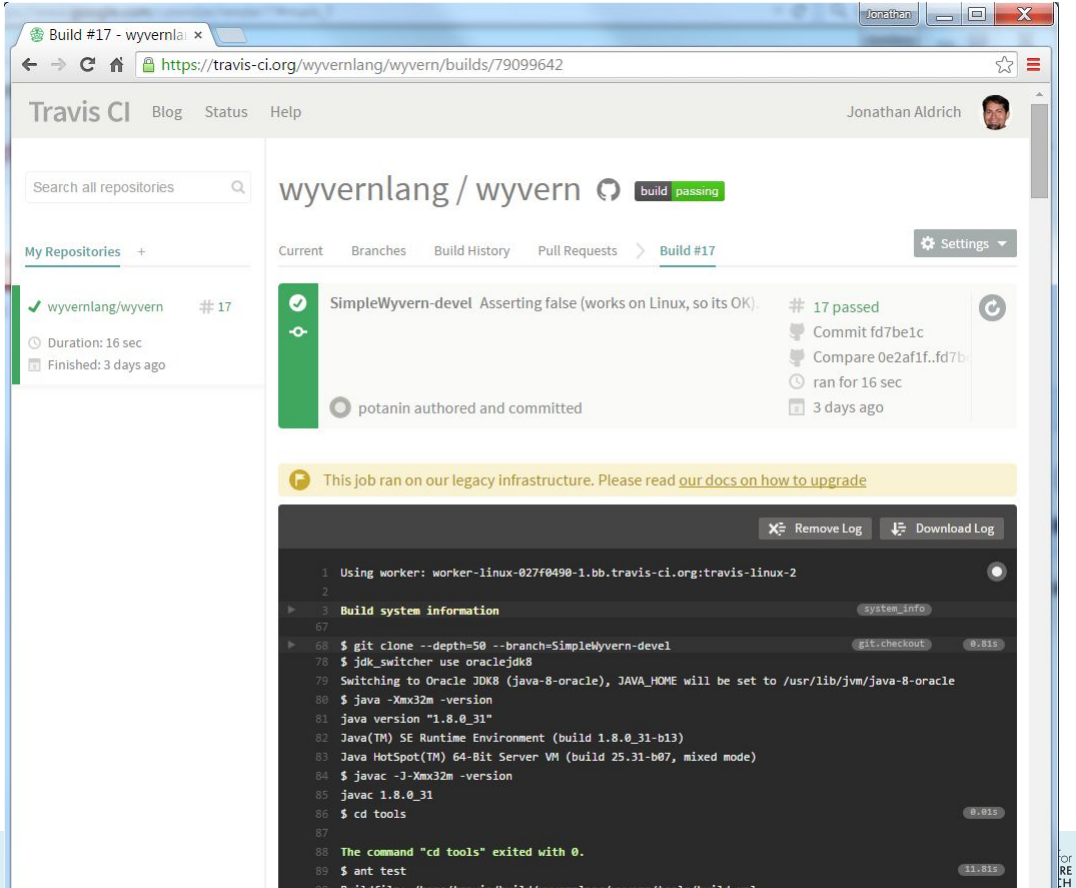
Defines a series of actions to be run in a clean build:

- Actions start from the very top:
  - Clone repository, checkout branch
  - Download & install Java/Node
  - Invoke commands with timeouts
- Travis allocates a new (Docker) container for each build
  - Think of this like a fresh, temporary computer
  - Usually with a few default libraries present (i.e., based on an *image*)
- That means: **fully replicable builds**

```
Installing SSH key from: default repository key                                          ssh_key
Using /home/travis/.netrc to clone repository.

$ git clone --depth=50 --branch=TypeScript https://github.com/CMU-17-214/template-21f-hw1.git CMU-17-214/template-21f-hw1    git.checkout   1.75s
Cloning into 'CMU-17-214/template-21f-hw1'...
remote: Enumerating objects: 117, done.
remote: Counting objects: 100% (117/117), done.
remote: Compressing objects: 100% (73/73), done.
remote: Total 117 (delta 50), reused 104 (delta 37), pack-reused 0
Receiving objects: 100% (117/117), 69.89 KiB | 2.25 MiB/s, done.
Resolving deltas: 100% (50/50), done.
$ cd CMU-17-214/template-21f-hw1
$ git checkout -qf 0d657225c8cbdd52751c2f88527f93f4099b041e

                                                                                                  0.01s
$ nvm install 16                                                                    nvm.install   3.65s
Downloading and installing node v16.8.0...
Downloading https://nodejs.org/dist/v16.8.0/node-v16.8.0-linux-x64.tar.xz...
Computing checksum with sha256sum
Checksums matched!
Now using node v16.8.0 (npm v7.21.0)

Setting up build cache                                                                  cache.1

                                                                                        cache.npm
$ node --version
v16.8.0
$ npm --version
7.21.0
$ nvm --version
0.38.0

$ npm ci                                                                            install.npm   3.04s

$ timeout 5m npm run compile                                                                      4.93s

> hw1-flashcards@1.0.0 compile
> tsc
```

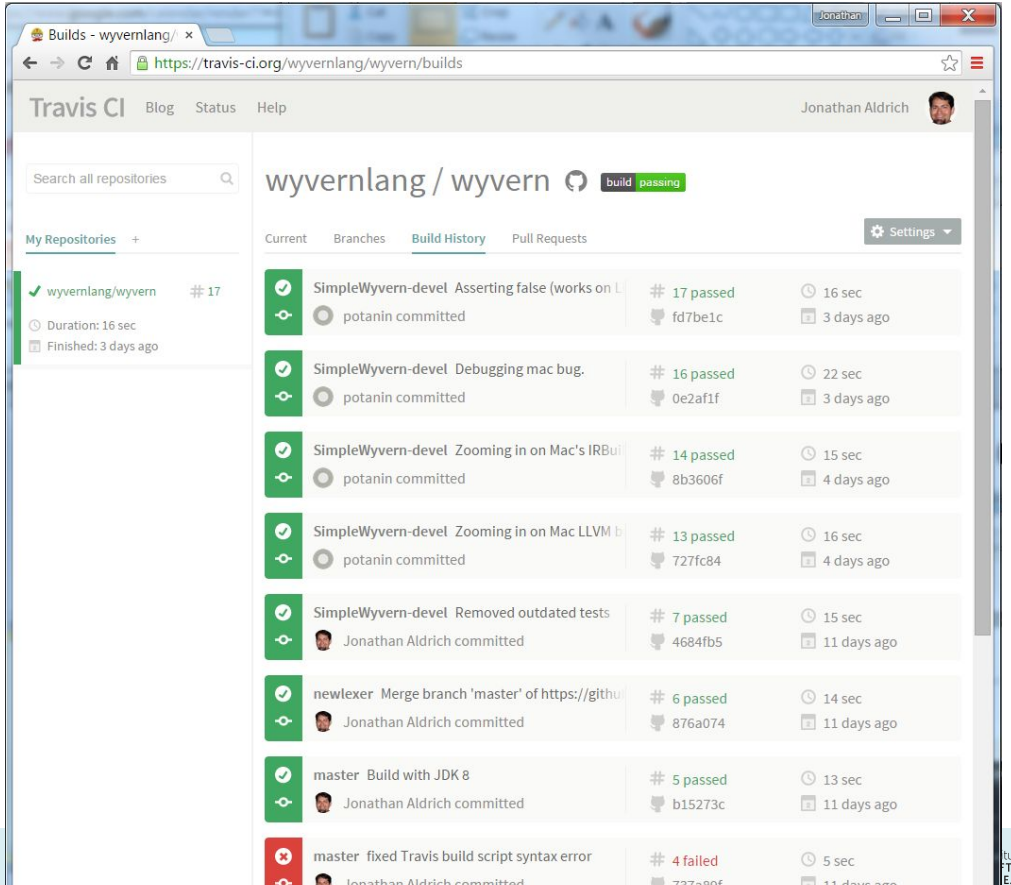
Continuous integration – Travis CI

Automatically builds, tests,
and displays the result



Continuous integration – Travis CI

You can see the results of builds over time



Outline

- On Homework 1
- Abstraction, Reuse, and Programming Tools
- For each in {IDE, Build systems, libraries, CI}:
  - What is it today?
  - What is under the hood?
  - **What is next?**

What's Next

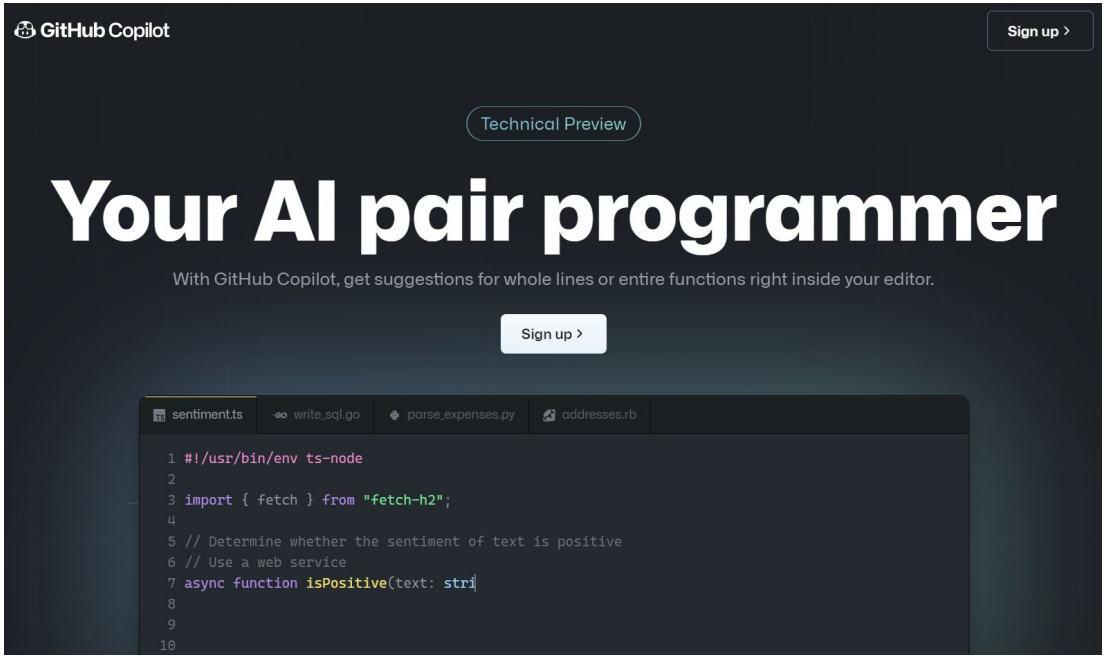
Anyone care to guess?

- Can be based on something you've seen, but think will boom

What's Next

AI Powered Programming

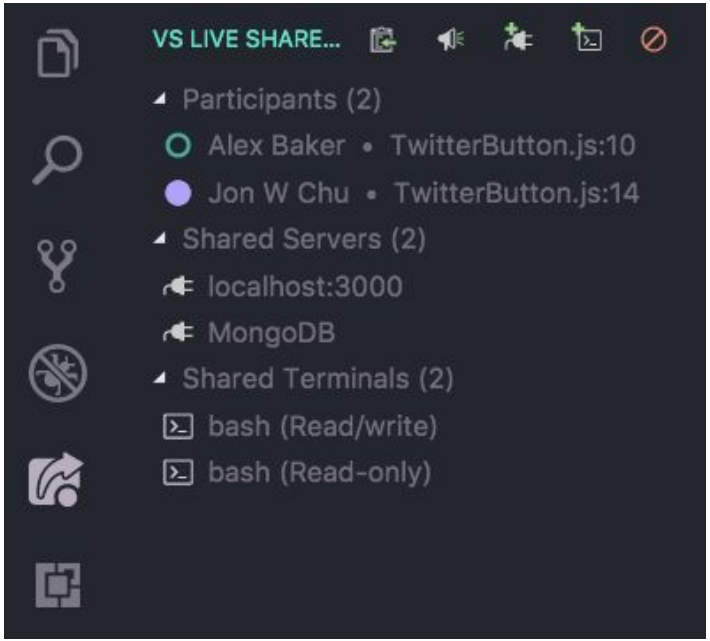
- Easier in Web IDEs
  - Which are themselves "next"



What's Next

Collaborative online coding

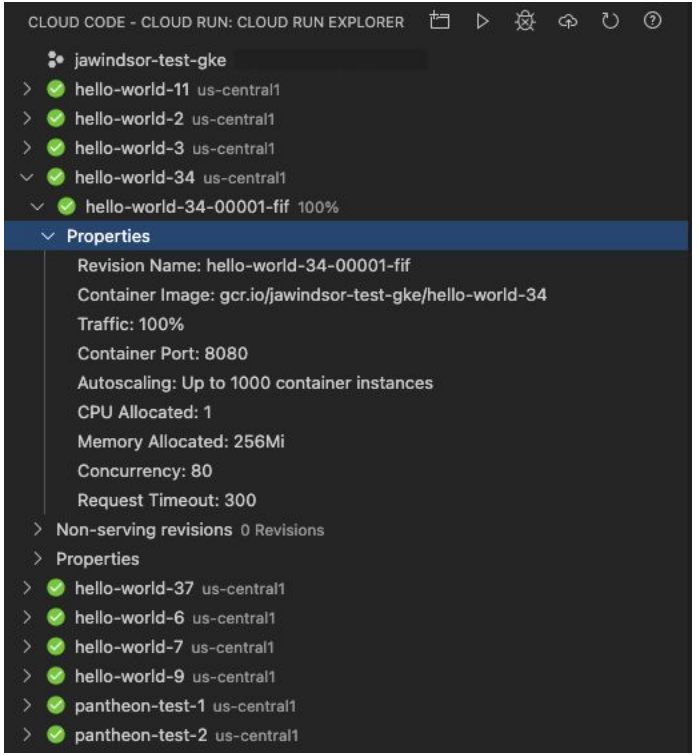
- Think: Google Docs for code
- E.g. VS Life Share
- How will this change "commits"?



What's Next

Tighter IDE-to-cloud integration

- Google Cloud is pushing on this with VSCode
- We will (lightly) touch on Containers & Clouds in this course



Summary

- Programming Tools are abundant, and rapidly evolving
  - Learn multiple; you will have to inevitably
- They rely on abstractions through interfaces to facilitate reuse
  - Which come in many shapes: GUI, API, DSL
  - And can be a limitation -- choose wisely
- Your HW1 toolchain sets you up for all homeworks
  - With modest variations (frameworks, new build targets)
  - Self-discovery is a big asset
  - Tomorrow's recitation offers help