Announcements

- TA hours and labs start today.
- First lab is out and due next Wednesday, 1/31.
- “Getting started” lab is also out
  ▶ Get you setup for project/lab work.
  ▶ We’ll check it with the first lab.
- “Stars” is out and due in a little over 2 weeks (2/9 6pm)
- Gear-up session here (Metcalf Auditorium) Monday at 6:30pm 1/29
- Brown CS “town hall” meeting today at 4pm in Atrium.
Introductions

- John Jannotti (jj) — Networking, Sightpath, Foodler
  - CIT 449 (Hours: After class until around 6pm.)
- Tim Nelson (tn) — Logic for Systems, cs18
- HTAs — Adam DeHovitz, Alex Jang, Joe Romano
- UTAs
  - You’ll “meet” them soon.
  - See website for blurbs that may help you find the right mentor.
CSCI0320 focus

- We focus mainly on *Applications* programming.
  - CS33 focuses on *Systems* programming.
- We will spend most of our time on
  - Writing correct, understandable, and extensible code.
  - Building the right thing. *You* will make these decisions.
  - Integrating and connecting subsystems.
- The class is designed to work with code bases over time.
  - Projects work together.
  - Your pair project will marry your code to another student’s.
  - Labs will evolve the Boggle project.
  - Group project — lots of code, several contributors, over time.
Meetings

- Lectures (Metcalf Auditorium)
  - Emphasis on design, we’ll walk through many case studies.
  - We time lectures to help with projects, honest.
  - More emphasis on participation this year.
  - Tuesday & Thursday 1pm-2:20pm

- Project “gear-up” sessions.

- Labs
  - Hands-on training you’ll need for assignments.
  - Significant to your grade.

- Communication via the web-site, Piazza
  - Signup: piazza.com/brown/spring2018/cs32
  - Calendar and lectures slides will be available.
  - You are responsible for everything posted.
This course just keeps going. . .

There is very little slack in the schedule.

- Start (even if only to read) assignments when they come out.
- Labs: Due in lab hours by due date or *nothing*.
- Lateness is penalized 20% / day. (We forgive your first two.)
  - No credit at all after 48 hours, and yet. . .
  - Every assignment must be turned in and working to pass!
  - If you’re not truly done 47.9 hours after the deadline, turn *something* in.
  - Getting a zero on a project costs about one whole letter grade.
  - You still need a passing grade. “All working by semester end” is a necessary, but not sufficient, condition for credit.
Collaboration

- We have loosened some restrictions.
- You may discuss projects orally with other (current) 32 students.
- You may not look at each others’ code, or collaborate over a medium that allows direct sharing of others’s words or code (email, IM).
- No photos, recordings, written notes.
- Take a break before coding after oral collaboration.
- We do use MOSS. It finds shared code. Please don’t test it.
Term Project

● Team project — Four person teams.
● Project of your choice
  ▶ We let you choose, because we want it to matter to you.
  ▶ Some of you won’t stop at the end of the course.
  ▶ You will learn the most from this, we can’t guide you step-by-step.
● Concepts
  ▶ Pull together all your skills
  ▶ Software engineering, project management; software design
● We’re intentionally starting a bit later this year.
  ▶ Ideas not due for a while. Pay attention in daily life.
  ▶ We encourage “heterogeneous” teams (wrt Intro Sequence)
  ▶ No web frameworks beyond Spark.
  ▶ No (native) mobile apps.
Your illustrious TAs

Let’s meet your TAs.
Software Engineering

- **Software Engineering ≠ Programming**
  - Programming is often a surprisingly small part.
- **What else?**
  - Determining *what* to build
    - Requirements (what tasks should the Software accomplish?)
    - Specifications (exact operating behavior)
  - Determining *how* to build it (Design/Architecture)
  - Testing (correctness and user experience)
  - Debugging (functional and performance)
  - Maintaining the program (new APIs, platforms, minor features)
  - There are certainly dependencies, but not purely linear.
- **Hopefully**, software lifetime is mostly maintenance/enhancement.
And the programming is different, too

- We’re stressing certain aspects when we say, “engineering”.
  - What do you think matters most when we talk about “high-quality code”? 
Three first principles

As you learn about software engineering you’ll hear a lot of “best practices,” and “rules of thumb” or their opposite: “code smells.” While handy, at base we try to justify everything against three goals.

- Safe From Bugs (SFB) – is your code likely to stay bug-free as it’s changed?
- Easy To Understand (ETU) – can someone read your code and know what’s going on quickly, without false starts?
- Ready For Change (RFC) – if new features are needed, are the needed changes likely to be straightforward?

These goals are about code quality — meeting requirements (including correctness and performance) is an external property, and also matters.
Among “equivalence classes” language isn’t so important
  ▶ Should be able to pick up new ones on your own.
  ▶ We’ll teach universal concepts.

But, how you use the language is important
  ▶ By necessity, we’ll talk about language specific things.
  ▶ Others need to understand your code.
  ▶ Idioms vary from one language to another.
  ▶ We’ll expect (and teach) you to write Java “properly.”

Projects and lectures will all use Java (8) at their “core”.
But the user interfaces will be “webby” - HTML, CSS, Javascript es6.
Why Java?

- Easy to teach concepts.
  - The interface mechanism is simple and powerful (often a better abstraction mechanism than inheritance).
  - Encapsulation / Data-hiding support is excellent.
  - Garbage Collection allows cleaner interfaces than C/C++
  - Static typing catches many mistakes at compile-time.

- Strong “ecosystem”
  - IDEs and command-line tools widely available.
  - Extensive libraries available.
  - Common enough that you gain real-world experience.

- Fast enough for almost anything.
Why “Webby”? 

- User Interfaces are always a bit grungy.
- You might as well pick up reusable skills.
- HTML, CSS, and Javascript are “reasonable”
- Ecmascript, or “es6”, even more so.
- You’ll learn at least half of what matters to modern web apps.
- The same separation applies to mobile apps.
- We’ll be treating them as “display technology.”, we won’t worry about
  - Users: Logins, Passwords, Multiple users
  - Security: Cookies, XSS attacks, malicious clients
  - Browser details: “Back button”, multiple windows, browser quirks.
- I’ll spend a little time on them late, for the sake of projects.
Java vs Javascript

Almost the same name, and they look very similar.

```java
int n = 3;
while (n != 1) {
    System.out.println(n);
    if (n % 2 == 0) {
        n = n / 2;
    } else {
        n = 3 * n + 1;
    }
}
System.out.println(n);
```

```javascript
let n = 3;
while (n != 1) {
    console.log(n);
    if (n % 2 == 0) {
        n = n / 2;
    } else {
        n = 3 * n + 1;
    }
}
console.log(n);
```

Awfully similar, right? What's different? Do you expect the same results for all n?
Messy interlude

Let’s have a look at what can go wrong.
What went wrong?

- The programs used *types* in a way they were not intended.
- We may have imagined the values as representing abstract, mathematical numbers (integers).
- That wasn’t true in either language, though it was untrue in different ways in each.
- To write high-quality programs you must not only understand the built-in types of a language, you will become skilled at creating new types.
- Back to Java vs Javascript: The biggest single difference is that Java has an explicit typing system that it checks at *compile-time*.
Let’s have a look at what else can go wrong, and how these languages tell us about the problem.
Static Typing

Static typing is a great benefit in meeting our three core goals.

- Safe from bugs. — Catch type errors and other bugs before runtime.
- Easy to understand. — Types are explicitly stated in the code.
- Ready for change. — Identifies other places that need to change in tandem.

Next week, I’ll show you more tools and techniques to get similar benefits with *static analysis* of your code.
There’s much more to writing code than not having type errors.