Programming Languages and Techniques (CIS1200)

Lecture 36

Semester Recap

CIS 1200 Final Exam

- Wednesday, May 7th 9:00-11:00 AM
  - Location: Chem 102

- Students who need accommodations should schedule their exams (ASAP) through the Weingarten Center

- Review Session / Mock exam
  - Time and Location – Sunday, May 4th at 12pm in Towne 100
  - 2 hour mock exam (Spring 2024) followed by 2 hour review session
  - (The review session will be recorded)
  - Look for details on Ed

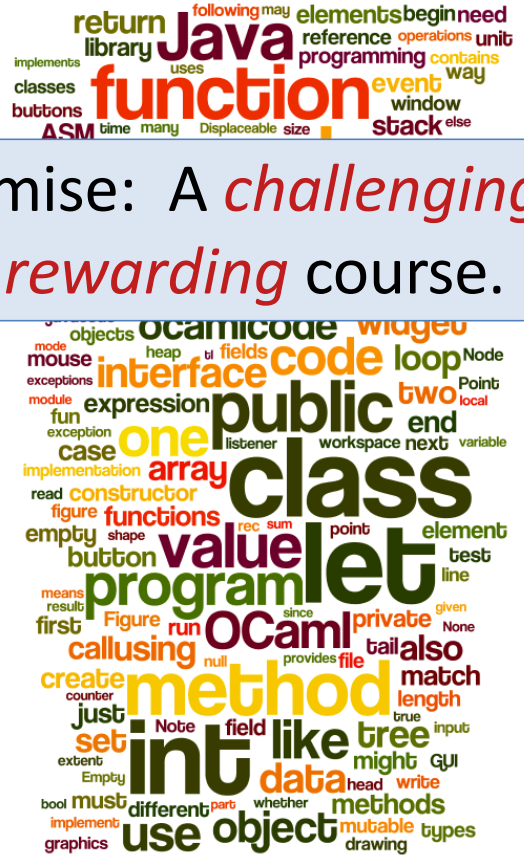
Exam Preparation

- *Comprehensive* exam covering the entire course:
  - *Ideas* from OCaml material (but no need to write OCaml)
  - All Java material
    - emphasizing material since midterm 2: subtyping, dynamic dispatch, collections, equality & overriding, exceptions, I/O, inner classes, swing
  - All course content
    - *except:* Bonus Lectures *(Code is Data, CIS Sustainability, OCaml at Jane Street)*
    - *Only simple/shallow questions about Advanced Topics*
- Closed book, but…
  - You may use one letter-sized, two-sided, *handwritten* sheet of notes during the exam.

CIS 1200 Recap

From Day 1

- CIS 1200 is a course in **program design**
- Practical skills:
  - ability to write larger (~1000 lines) programs
  - increased independence ("working without a recipe")
  - test-driven development, principled debugging
- Conceptual foundations:
  - common data structures and algorithms
  - several different programming idioms
  - focus on modularity and compositionality
  - derived from first principles throughout
- It will be fun!



Promise: A *challenging* but *rewarding* course.

Which assignment was the most *challenging*?

0

OCaml finger exercises 0%

DNA 0%

Sets and Maps 0%

Queues 0%

GUI 0%

Images 0%

Chat 0%

TwitterBot 0%

Game 0%

Which assignment was the most *rewarding*?

0

OCaml finger exercises 0%

DNA 0%

Sets and Maps 0%

Queues 0%

GUI 0%

Images 0%

Chat 0%

TwitterBot 0%

Game 0%

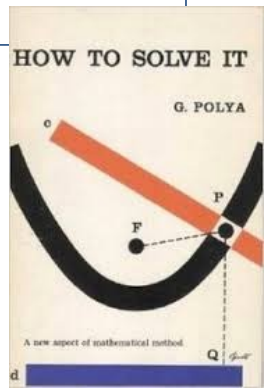
CIS 1200 Concepts

13 concepts in 36 lectures

Concept: Design Recipe

1. Understand the problem
   What are the relevant concepts and how do they relate?
2. Formalize the interface
   How should the program interact with its environment?
3. Write test cases
   How does the program behave on typical inputs? On unusual ones?
   On erroneous ones?
4. Implement the required behavior
   Often by decomposing the problem into simpler ones and applying the same recipe to each

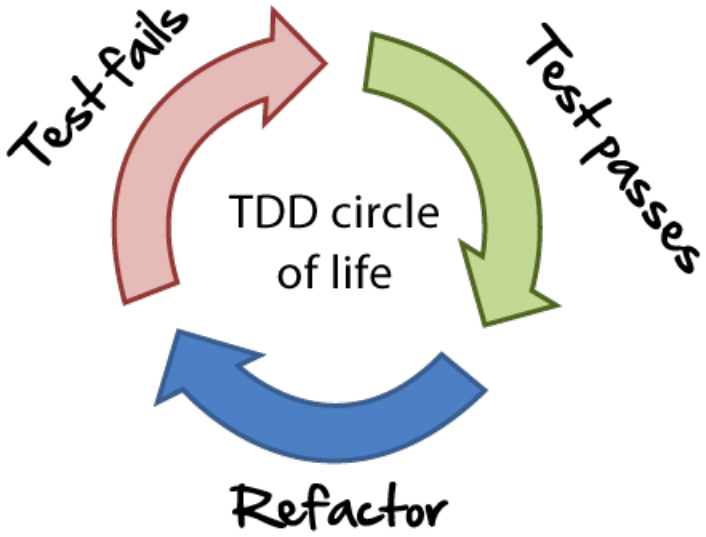
"Solving problems", wrote Polya, "is a practical art, like swimming, or skiing, or playing the piano: You can learn it only by imitation and practice."



Concept: Testing

- We use a *"test first"* methodology - write tests *before* coding

- Examples:
  - Simple assertions and properties for declarative programs (or subprograms)
  - Longer (and more) tests for stateful programs / subprograms
  - Informal tests for GUIs (can be automated through tools)



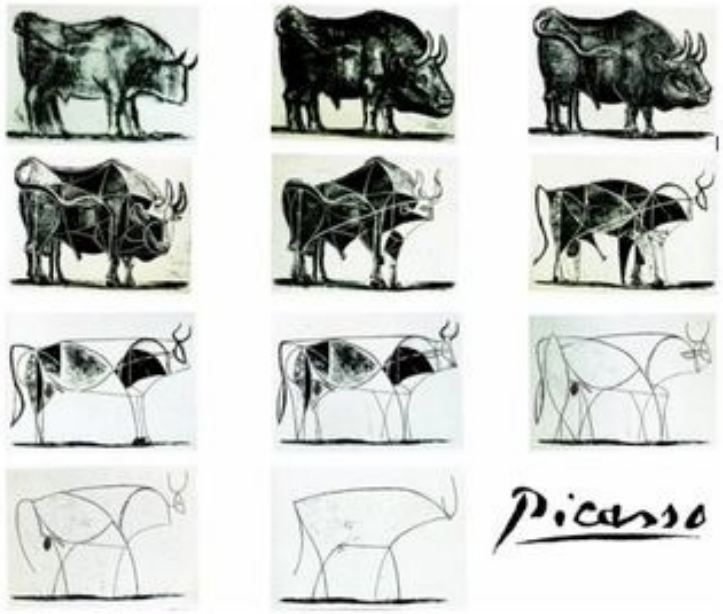
- Why?
  - Tests clarify the specification of the problem
  - Helps you understand the *invariants*
  - Thinking about tests informs the implementation
  - Tests help with extending and refactoring code later
  - Industry practice; useful for coordinating teams

Concept: Abstraction

- Generalize code so it can be reused in multiple situations.

Don't Repeat Yourself!

- Examples: Functions/methods, generics, higher-order functions, interfaces, subtyping, abstract classes inner classes



Pablo Picasso, Bull (plates I - XI) 1945

- Why?
  - Duplicated functionality = duplicated bugs
  - Duplicated functionality = more bugs waiting to happen
  - Good abstractions make code easier to read, modify, maintain

Concept: Persistent data structures

- Store data in *persistent, immutable* str… as *transformations* of those structu…
- Examples: immutable lists and tre… Strings, Streams in Java (HW 6/8)
- Why?
  - Simple model of computati…
  - Simple interface: De… between variou… …ts of the program
  - *Recursion* amenable to mathematical analysis (CIS 1600/1210)
  - Plays well with concurrency

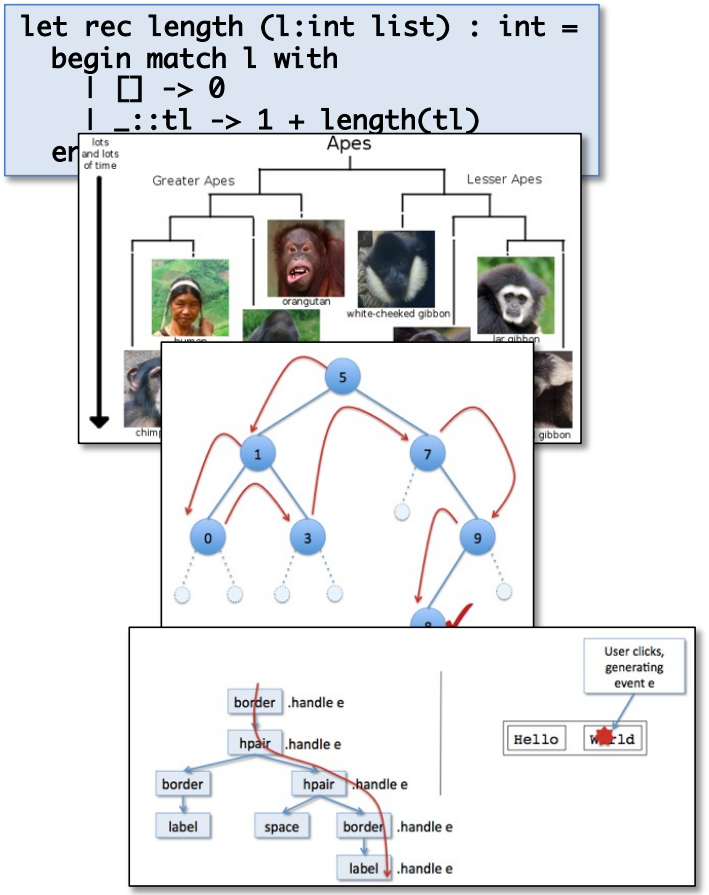
Recursion is the natural way of computing a function $f(t)$ when t belongs to an *inductive data* type:

1. Determine the value of f for the base case(s).
2. Compute f for larger cases by combining the results of recursively calling f on smaller cases.
3. Same idea as mathematical induction (a la CIS 1600)

Concept: Tree Structured data

- Examples:
  - Lists (i.e., “unary” trees)
  - Simple binary trees (evolutionary trees)
  - Trees with invariants: e.g., binary search trees
  - TreeSet and TreeMap collections in Java
  - Widget trees: screen layout + event routing
  - Swing components

- Why?
  - Trees are ubiquitous in computer science!
  - Organized data leads to efficient divide and conquer algorithms



Concept: First-class computation

- *Code is a form of data* that can be defined by functions, methods, or objects (including anonymous ones), stored in data structures, and passed to other functions
- Examples: map, filter, fold (HW4), pixel transformers (HW6), event listeners (HW5, 7, 9)

```
cell.addMouseListener(new MouseAdapter() {
    public voi
        selectC
    }
});
```

```
cell.addMouseListener(e ->
                      selectCell(cell));
```

- Why?
  - Powerful tool for abstraction: can factor out design patterns that differ only in certain computations

Concept: Static Types, Generics, and Subtyping

- *Static type systems* can detect many errors early. Every expression has a static type, and OCaml/Java use the types to rule out buggy programs. *Generics* and *subtyping* make types more flexible and allow for better code reuse.

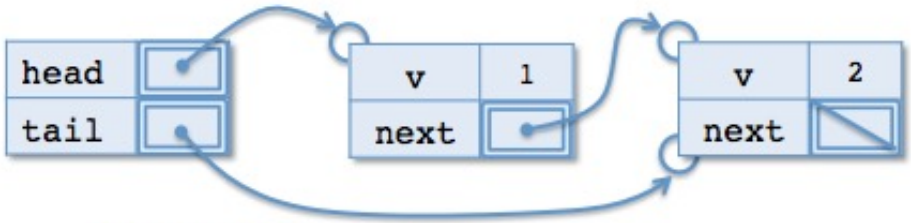
```
let rec contains (x:'a) (l:'a list) : bool =
  begin match l with
    | [] -> false
    | h::tl -> x = a || (contains x tl)
  end
```

- Why?
  - Let's the language enforce (programmer-defined) abstraction
  - Easier to fix problems indicated by a type error than to write a test case and then figure out why the test case fails
  - Promotes refactoring: type checking ensures that basic invariants about the program are maintained

Concept: Mutable data

- Some data structures are *ephemeral*: computations mutate them over time

- Examples: queues, deques (HW4), GUI state (HW5, 9), arrays (HW 6), iterators (HW8)



A queue with two elements

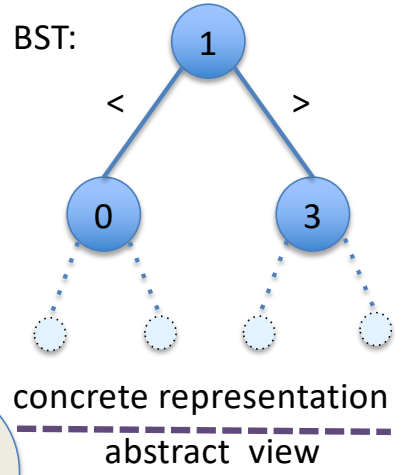
- Why?
  - Common in OO programming, which simulates the transformations that objects undergo when interacting with their environment
  - Heavily used for event-based programming, where different parts of the application communicate via shared state
  - Default style for Java libraries (collections, etc.)

Concept: Interface vs. Implementation

- *Type abstraction* hides the actual implementation of a data structure, describes a data structure by its interface (what it does vs. how it is represented), supports reasoning with invariants

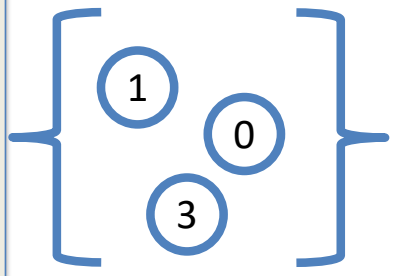
- Examples: Set/Map interface (HW3), queues in OCaml and Java, encapsulation and access control

- Why?
  - **Flexibility**: Can
    without modify
  - **Correctness**:
    implementa



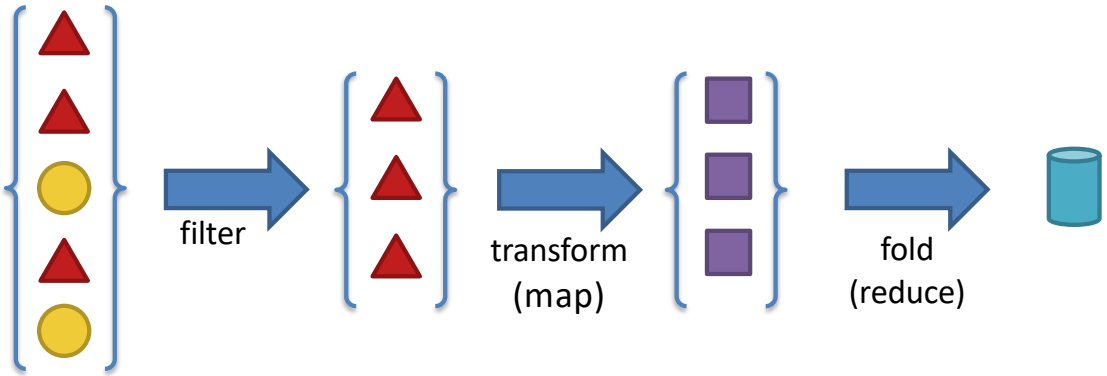
Invariants are a crucial tool for reasoning about data structures:

1. *Establish* the invariants when you create the structure.
2. *Preserve* the invariants when **you** modify the structure.
3. *Protect* the structure from external modification through *encapsulation*.



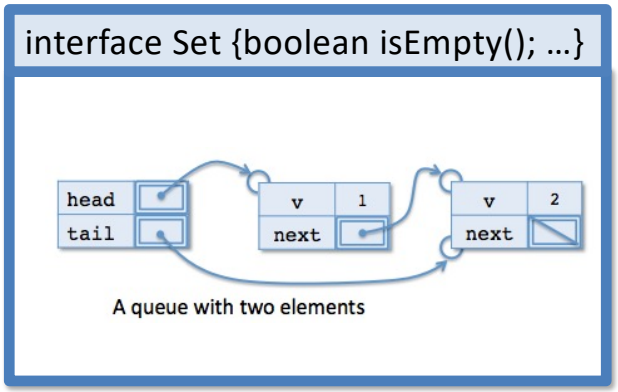
Concept: Collection types--Sequences, Sets, Maps

- Examples: HW3, Java Collections, HW 7, 8
- Why?
  - These abstract data types come up again and again
  - Need *aggregate* data structures (collections) no matter what language you are programming in
  - Need to be able to choose the data structure with the right semantics

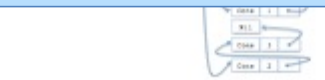


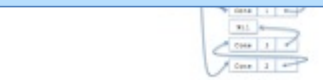
Concept: Linked Lists, Trees, BSTs, Queues, and Arrays

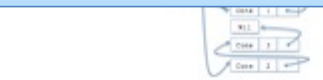
- There are *implementation trade-offs* for abstract types
- Examples:
  - Binary Search Trees vs. (linked) Lists vs. Hashing for sets and maps
  - Linked lists vs. Arrays for sequential data
- Why?
  - Abstract types have multiple implementations
  - Different implementations have different trade-offs. Need to understand these trade-offs to use them well.
  - For example: BSTs use their invariants to speed up lookup operations compared to linked lists.



Concept: Abstract Stack Machine







Do the Function call

Save Workspace; push l1, l2

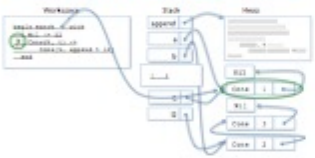
Lookup l1

Lookup l1

Match Expression

Nil case Doesn't Match

Cons case *Does* Match



Simplify the Branch: push h, t

Lookup 'h'

Lookup 'h'

Lookup 'append'

Lookup 'append'

Concept: Abstract Stack Machine

- The *Abstract Stack Machine* is a detailed model of how programs execute in OCaml/Java

- Example: Many, throughout the semester!

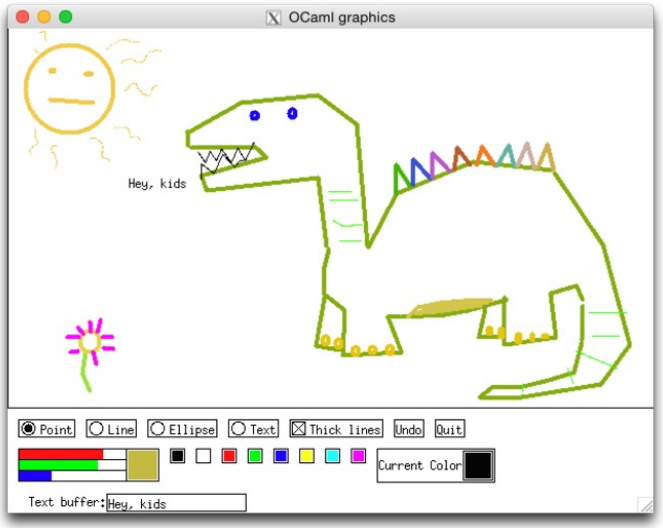
- Why?
  - To know what your program does without running it
  - To understand tricky features of Java/OCaml language (aliasing, first-class functions, exceptions, dynamic dispatch)
  - To help understand the programming models of other languages: Javascript, Python, C++, C#, …
  - To help predict performance and space usage
  - To implement a compiler or interpreter

Concept: Event-Driven programming

- Structure a program by associating "handlers" that *react to events*. Handlers typically interact with the rest of the program by modifying shared state.

- Examples: GUI programming in OCaml (HW 5) and Java (HW 9)

- Why?
  - Practice with reasoning about shared state
  - Practice with first-class functions
  - Basis for programming with Swing
  - Common in GUI applications



Why OCaml?

Why some other language than Java?

- Level playing field for students with varying backgrounds coming into the same class
- Two points of comparison — OCaml and Java — allows us to emphasize language-independent concepts
- Learn concepts that generalize *across* diverse languages.
- "OCaml-style" type systems have influenced many modern language designs

…but why *specifically* OCaml?

Rich, orthogonal vocabulary

- In Java: int, A[], Object, Interfaces
- In OCaml:
  - primitives
  - arrays
  - objects
  - datatypes (including lists, trees, and options)
  - records
  - refs
  - first-class functions
  - abstract types

- All of the above *can* be implemented in Java, but untangling various use cases of objects is subtle
- Concepts like generics can be studied in isolation in OCaml with fewer intricate interactions with the rest of the language

Functional Programming

- In Java, every reference is mutable and optional by default
- In OCaml, persistent data structures are the default. Furthermore, the type system keeps track of what is and is not mutable, and what is and is not optional

- Advantages of immutable/persistent data structures
  - Don't have to keep track of aliasing. Interface to the data structure is simpler
  - Often easier to think in terms of "transforming" data structures than "modifying" data structures
  - Simpler implementation
    (compare lists and trees to queues and deques)
  - Simple but powerful evaluation model
    (substitution + recursion)

Why Java?

Object Oriented Programming

- An important way of decomposing / structuring programs
- Basic principles
  - Encapsulation of local, mutable state
  - Inheritance to share code
  - Dynamic dispatch to select which code gets run
  - Subtyping to capture statically known information about inheritance and the "*is a*" relationship
- Fundamental to most major programming languages (Python, JavaScript, C++, C#, etc.)

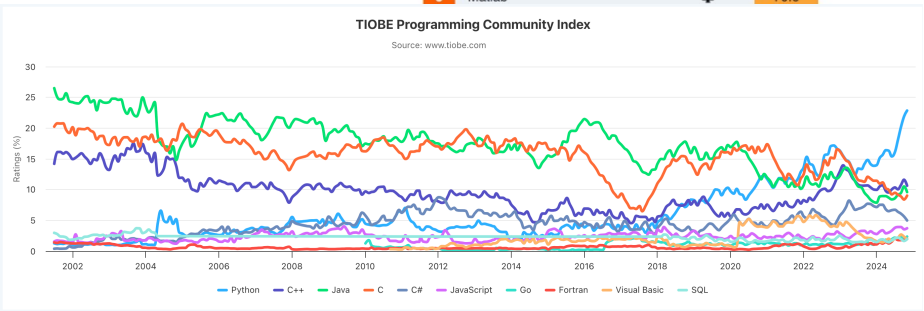
- but why *specifically* Java?

Important Ecosystem

- Canonical example of OO language design
- Widely used: Desktop / Server / Android / etc.
- Gateway to C/C++/C#/Kotlin/Scala/Rust
- Industrial strength tools
  - IntelliJ / Eclipse
  - JUnit testing framework
  - Profilers, debuggers, …
- Libraries:
  - Collections / I/O libraries/ Swing
- In-demand job skill
  - IEEE Spectrum: 2nd
  - TIOBE: 4rd

IEEE Spectrum Rank

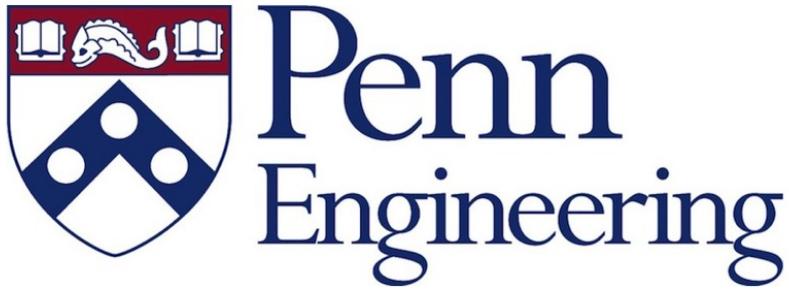
Rank	Language	Type	Score
1	Python		100.0
2	Java		96.3
3	C		94.4
4	C++		87.5
5	R		81.5
6	JavaScript		79.4
7	C#		74.5
8	Matlab		70.6



Onward…

What Next?

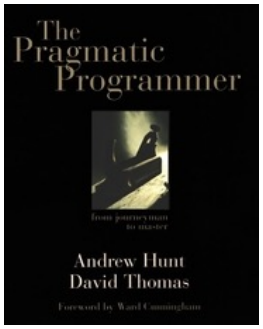
- Classes:
  - CIS 1210, 2620, 3200 – data structures, performance, computational complexity
  - CIS 19xx – programming languages
    - C++, Python, Haskell, Ruby on Rails, iPhone programming, Android, Javascript, Rust, Go
  - CIS 2400 – lower-level: hardware, gates, assembly, C programming
  - CIS 4710, 4480 – hardware and OS's
  - CIS 5520 – advanced functional programming in Haskell
  - CIS 5521 – compilers (projects in OCaml)
  - And many more!

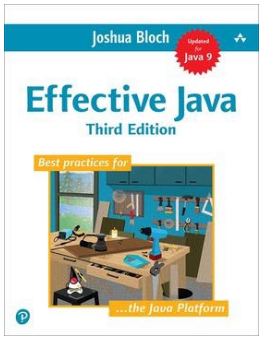


The Craft of Programming

- ***The Pragmatic Programmer:***
  From Journeyman to Master

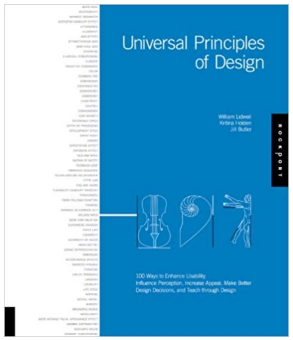
  by Andrew Hunt and David Thomas
  - Not about a particular programming language, it covers style, effective use of tools, and good practices for developing programs





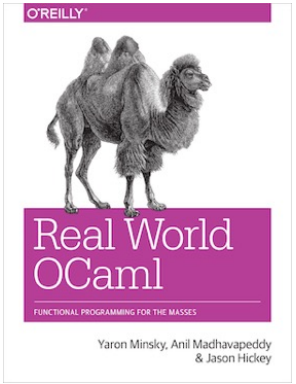
- ***Effective Java***
  by Joshua Bloch
  - Technical advice and wisdom about using Java for building software. The views we have espoused in this course share much of the same design philosophy

- **Universal Principles of Design**
  by William Lidwell, Kritina Holden, Jill Butler
  - General principles about good design with examples and applications ranging across software and user interfaces, to physical objects, to traditional graphic design.



Functional Programming

- *Real World OCaml*
  by Yaron Minsky, Anil Madhavpeddy,
  and Jason Hickey
  - Using OCaml in practice: learn how to leverage its rich types, module system, libraries, and tools to build reliable, efficient software.
  - https://realworldocaml.org/



- Explore related Languages:

Haskell Scala REASON

Kotlin Clojure F# Swift

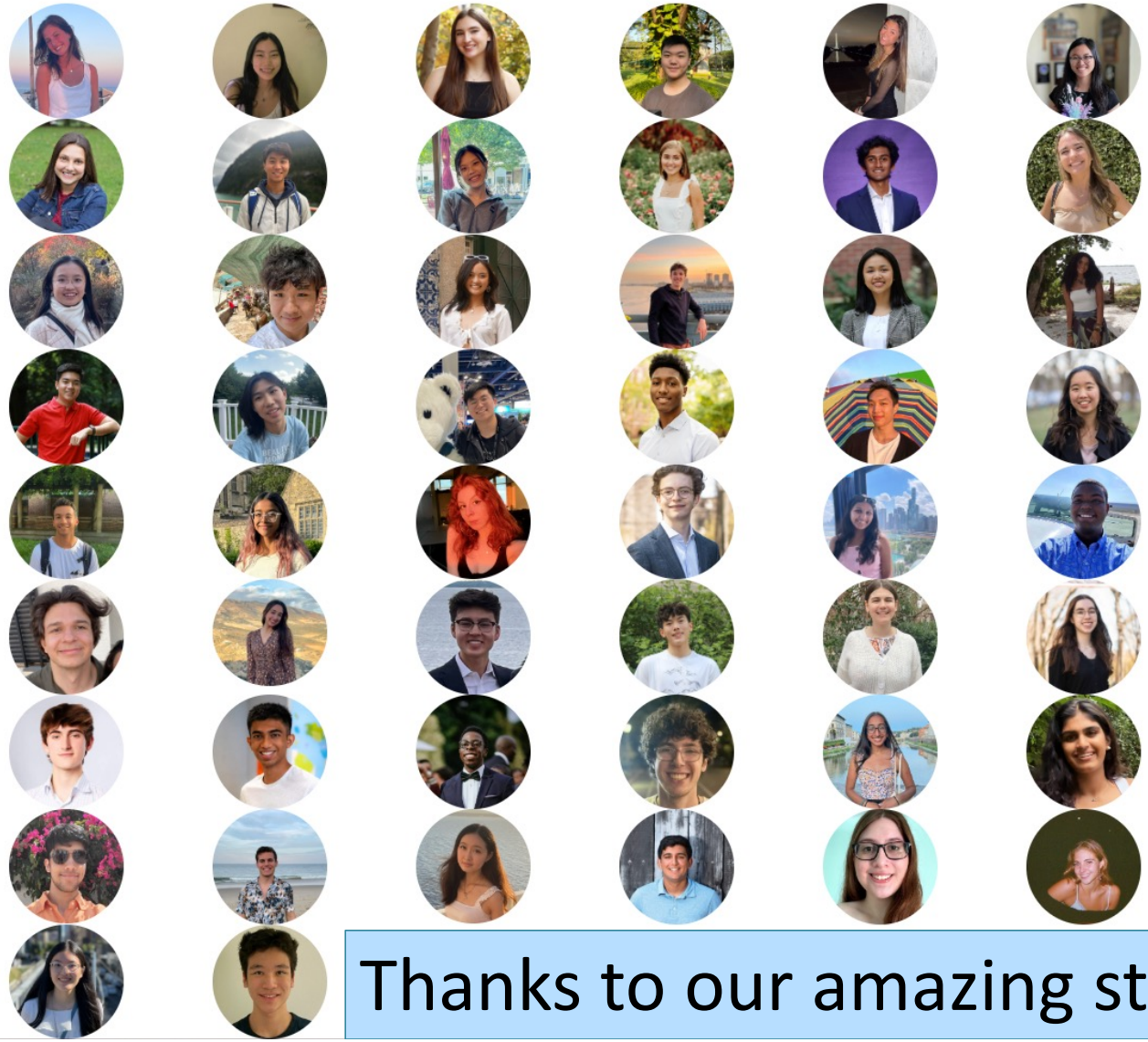
Conferences / Videos / Blogs

- Many blogs / tutorials about Java
- curry-on.org
- cufp.org Commercial Users of Functional Programming
  - See e.g. Manuel Chakravarty's talk "A Type is Worth a Thousand Tests"
- Jane Street Tech Blog
  - OCaml in practice
  - "Building better software" podcast
- Join us! Penn's PL Club plclub.org

Ways to get Involved

Become a TA!

Undergraduate Research



Thanks to our amazing staff!

Thanks to you!