Principles of Software Construction: Objects, Design, and Concurrency

Refactoring & Anti-patterns

Claire Le Goues Bogdan Vasilescu

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

slido

Join at slido.com #124127

ⓘ Start presenting to display the joining instructions on this slide.

Administrivia

I promise we will sync Canvas grades with Gradescope.

Submit the correct link on Canvas.

- If you notice you submitted the wrong link, send us a DM and then resubmit the right link. We won't take late days if you submitted the wrong link on time.

HW4 will be released today, due a either Tuesday or Wednesday depending on what the TAs need for grading.

- …but promise me you'll start HW5 early, it's much longer.

Reading quiz is on Canvas!

…go there and do it it'll take 30 seconds…

slido

Audience Q&A Session

ⓘ Start presenting to display the audience questions on this slide.

Today: Patterns, anti-patterns, and refactoring

- Patterns: using and choosing between them.
- Antipatterns and refactoring
  - Sidequest on equals, toString, typecasting
- Several other useful patterns

Refactoring: Any functionality-preserving rewrite or restructure.

Midterm Scenario: Shape drawing software

Assume you have a complex drawing that consists of many shapes, and you want to save it. Some logic of the saving functionality is always the same (e.g., going through all shapes, reducing them to drawable lines), but others you want to vary to support saving in different file formats (e.g., as png, as svg, as pdf). You want to support different file formats later.

Which pattern makes the most sense? How can we tell?

- Strategy Pattern
- Template Method Pattern
- Composite Pattern
- Decorator Pattern
- Observer Pattern

Drawing Example -- Basics

```
class Line {
    // TODO
}

interface Shape {
    toLines(): Line[];
}

class Triangle implements Shape {
    public toLines(): Line[] {
        return ...;
    }
}
```

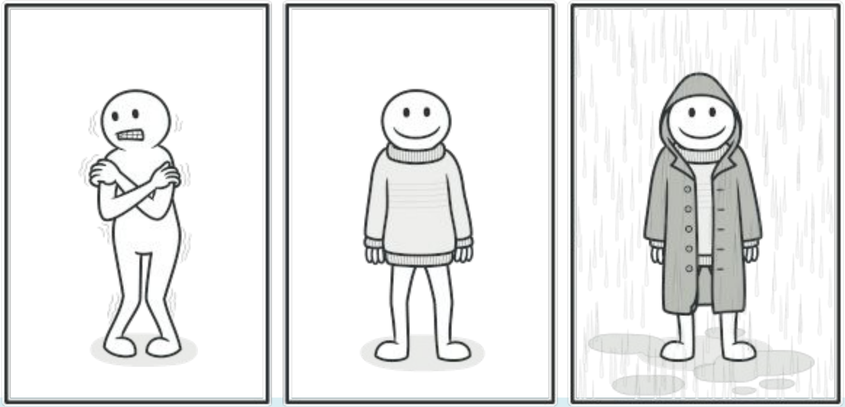
```
// A drawing consists of many shapes.
class Drawing {
    shapes: Shape[]
    constructor(shapes: Shape[]) {
        this.shapes = shapes;
    }
    public toLines() {
        let lines: Line[] = []
        for (let shape of this.shapes) {
            lines.push(shape.toLines());
        }
        return lines;
    }
}
```

Midterm Scenario: Shape drawing software

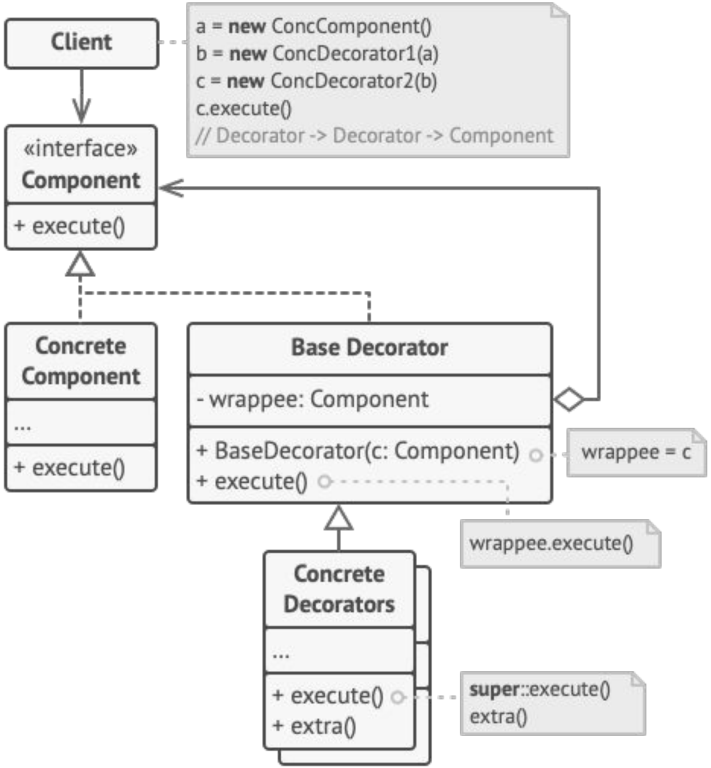
Assume you have a complex drawing that consists of many shapes, and you want to save it. Some logic of the saving functionality is always the same (e.g., going through all shapes, reducing them to drawable lines), but others you want to vary to support saving in different file formats (e.g., as png, as svg, as pdf). You want to support different file formats later.

Which pattern makes the most sense? How can we tell?

- **Strategy Pattern**
- **Template Method Pattern**
- Composite Pattern
- **Decorator Pattern??**
- Observer Pattern



Drawing Example -- Decorator?



Drawing Example -- Decorator?

```
interface DrawingSaver {
    saveDrawing(drawing: Drawing, path: string): void;
}

class BasicSaver implements DrawingSaver {
    public saveDrawing(drawing: Drawing, path: string): void {
        let lines: Line[] = drawing.toLines();
        // Now what?
    }
}
```

Drawing Example -- Decorator?

```
class DrawingSaverDecorator implements DrawingSaver {
    wrappee: DrawingSaver
    constructor(source: DrawingSaver) { this.wrappee = source; }

    public saveDrawing(drawing: Drawing, path: string): void {
        this.wrappee.saveDrawing(drawing, path);
    }
}
class JPEGDecorator extends DrawingSaverDecorator {
    public saveDrawing(drawing: Drawing, path: string): void {
        let lines: Line[] = drawing.toLines();
        // Internally store in JPEG
        super.saveDrawing(drawing, path);
    }
}
```

Midterm Scenario: Shape drawing software

Assume you have a complex drawing that consists of many shapes, and you want to save it. Some logic of the saving functionality is always the same (e.g., going through all shapes, reducing them to drawable lines), but others you want to vary to support saving in different file formats (e.g., as png, as svg, as pdf). You want to support different file formats later.

Which pattern makes the most sense? How can we tell?

- **Strategy Pattern**
- **Template Method Pattern**
- Composite Pattern
- ~~**Decorator Pattern**~~
- Observer Pattern

Drawing Example -- Strategy

```
interface LineFormatter {
    write(lines: Line[], writer: Writer): void;
}

class DrawingSaver {
    public save(drawing: Drawing, formatter: LineFormatter, path: string) {
        let lines: Line[] = drawing.toLines();
        let writer: Writer = new Writer(path);
        formatter.write(lines, writer);
    }
}

class JPEGFormatter implements LineFormatter {
    public write(lines: Line[], writer: Writer) { // Store JPEG data. }
}
```

Drawing Example -- Template Method

```
abstract class DrawingSaver {
    public save(drawing: Drawing, path: string) {
        let lines = drawing.toLines();
        let formatted = this.toFormat(lines);
        let writer: Writer = new Writer(path);
        writer.write(formatted);
    }

    abstract toFormat(lines: Line[]): any[];
}

class JPEGSaver extends DrawingSaver {
    public toFormat(lines: Line[]): any[] { // Store JPEG data. }
}
```

Notice how we can get basically the same functionality but with different structure/patterns applied!

(strategy is better if you want to implement this per-shape rather than per-drawing).

A good use of Decorator

```java
public interface Shape {
   void draw();
}

public class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Shape: Rectangle");
   }
}

public class Circle implements Shape {
   @Override
   public void draw() {
      System.out.println("Shape: Circle");
    }
}

public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

```java
public class RedShapeDecorator extends ShapeDecorator {
   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}

public class DecoratorPatternDemo {
   public static void main(String[] args) {
      Shape circle = new Circle();
      Shape redRectangle = new RedShapeDecorator(new Rectangle());
      System.out.println("Circle with normal border");
      circle.draw();

      System.out.println("\nRectangle with red border");
      redRectangle.draw();
   }
}
```

slido

Audience Q&A Session

ⓘ Start presenting to display the audience questions on this slide.

Revisiting: type-casting

- Sometimes you want a different type than you have
  - e.g.,
    ```
    double pi = 3.14;
    int indianaPi = (int) pi;
    ```

In TS:

```
(dog as Animal).identify()
```

- Useful if you know you have a more specific subtype:

  ```
  Account acct = ...;
  CheckingAccount checkingAcct = (CheckingAccount) acct;
  long fee = checkingAcct.getFee();
  ```

  - Will get a `ClassCastException` if types are incompatible
- Advice: avoid downcasting types
  - Never(?) downcast within superclass to a subclass

Typecasting revisited

```
class Animal {
   name : string;

   constructor(name : string) {
       this.name = name;
   }

   identify() : string {
       return this.name;
   }
}
```

```
class Dog extends Animal {
   public constructor() {
       super("dog");
   }
}

…
function printAnimal(animal : Animal) {
    if (animal instanceof Dog) {
       console.log(“dog”);
    }
    else if(animal instance of Cat) {
       console.log(“cat”);
    }
}
```

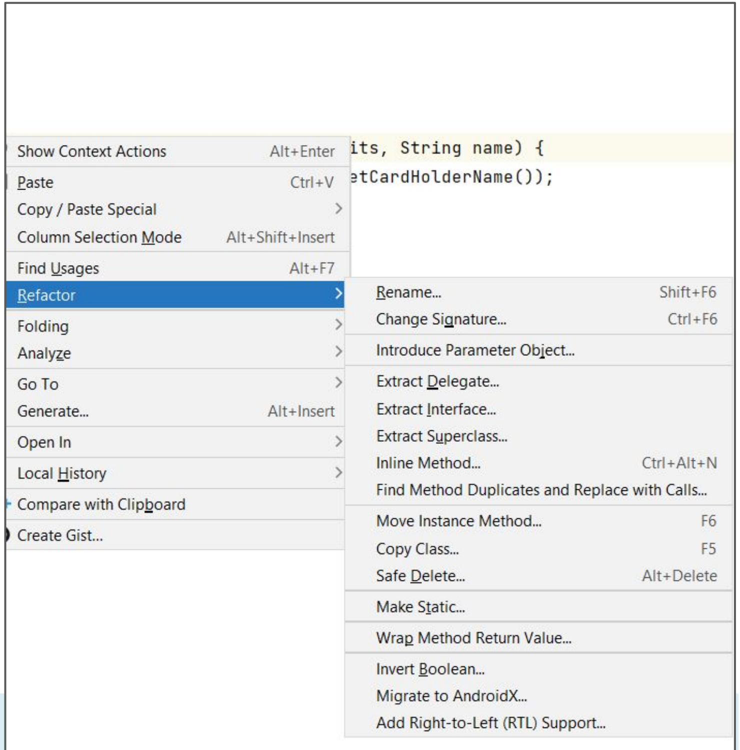
We all agree that’s bad, right?

Refactoring

- Any functionality-preserving restructuring
  - Typically automated by IDE
  - Ideas?

Refactoring: IDE support

- Rename class, method, variable to something not in-scope
- Extract method/inline method
- Extract interface
- Move method (up, down, laterally)
- Replace duplicates



Refactoring and Anti-Patterns

- Often, all the functionality is correct, but the organization is bad
  - High coupling, high redundancy, poor cohesion, god classes, …
- Refactoring is the principal tool to improve structure
  - Automated refactorings even guarantee correctness
    - But you can't always count on those being right
  - A series of refactorings is usually enough to introduce design patterns

Refactoring and Anti-Patterns

- Often, all the functionality is correct, but the organization is bad
  - High coupling, high redundancy, poor cohesion, god classes, …
- Refactoring is the principal tool to improve structure
  - Automated refactorings even guarantee correctness
    - But you can't always count on those being right
  - A series of refactorings is usually enough to introduce design patterns
- HW4 involves analyzing such a system and making primarily refactoring changes
  - "primarily", because sometimes you do need to alter things slightly.

Anti-patterns

Anti-patterns are *common* forms of bad/no-design

- Can you think of examples?
- Where do they come from?

Anti-patterns

- We have talked a fair bit about bad design heuristics
  - High coupling, low cohesion, law of demeter, …
- You will see a much larger vocabulary of related issues
  - Commonly called code/design “smells”
  - Worthwhile reads:
    - **A short overview: https://refactoring.guru/refactoring/smells**
    - Wikipedia: https://en.wikipedia.org/wiki/Anti-pattern#Software_engineering
    - Book on the topic (no required reading): Refactoring for Software Design Smells: Managing Technical Debt, Suryanarayana, Samarthyam and Sharma
      - S.O. summary: https://stackoverflow.com/a/27567960

Anti-patterns

- Two ways of looking at this:
  - Design issues that manifest as bad/unmaintainable code
  - Poorly written/evolved code that leads to bad design

(switch to IntelliJ for example)

slido

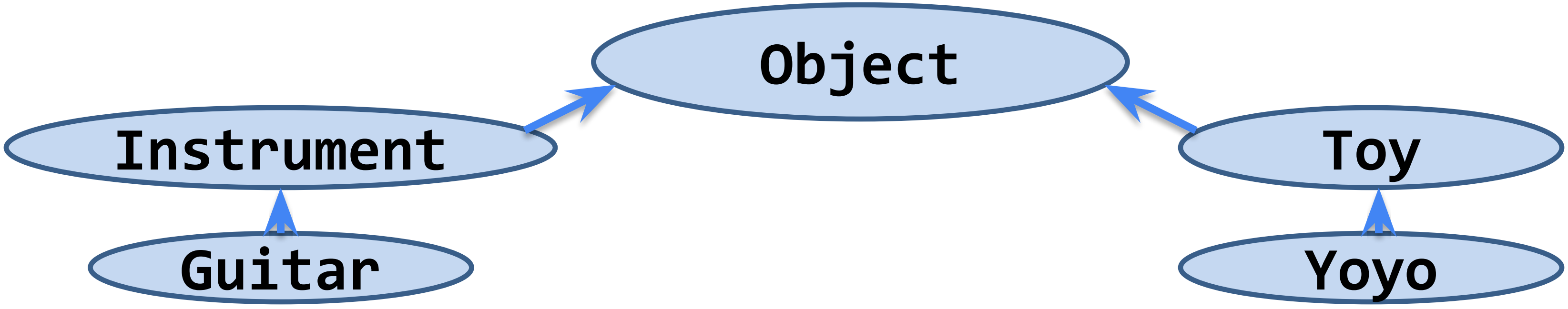
Audience Q&A Session

ⓘ Start presenting to display the audience questions on this slide.

While we’re on the subject of objects and equality.

The Java class hierarchy

- The root is Object (all non-primitives are objects)
- All classes except Object have one parent class
  - Specified with an `extends` clause
    ```
    class Guitar extends Instrument { ... }
    ```
  - If `extends` clause omitted, defaults to `Object`
- A class is an instance of all its superclasses



Methods common to all objects

- How do collections know how to test objects for **equality**?
  - Why did this work:

```
for(Person p: this.records) {
   if(p.equals(newP)) {
     …
```

- How do they know how to **hash** and **print** them?

- The relevant methods are all present on `Object`
  - `equals` - returns true if the two objects are “equal”
  - `hashCode` - returns an `int` that must be equal for equal objects, and is likely to differ on unequal objects
  - `toString` - returns a printable string representation (default is gross: Type and hashcode)

Comparing values

`x == y` compares x and y "directly":

- **primitive values:** returns true if x and y **have the same value**
- **objects references:** returns true if x and y **refer to same object**

`x.equals(y)` typically compares the ***values*** *of the objects referred to* by x and `y*`

* Assuming it makes sense to do so for the objects in question

True or false?

```
int i = 5;
int j = 5;
System.out.println(i == j);
---------------------------
```

True or false?

```
int i = 5;
int j = 5;
System.out.println(i == j);
---------------------------
```

true i $\boxed{5}$

j $\boxed{5}$

True or false?

```
int i = 5;
int j = 5;
System.out.println(i == j);
----------------------------
true
```

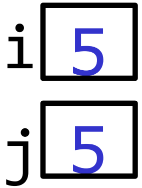
i $\boxed{5}$

j $\boxed{5}$

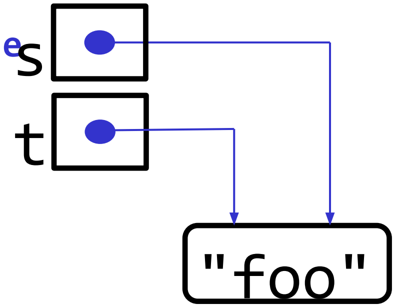
```
String s = "foo";
String t = s;
System.out.println(s == t);
----------------------------
```

True or false?

```
int i = 5;
int j = 5;
System.out.println(i == j);
----------------------------
true
```

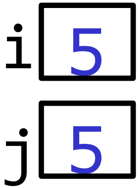



```
String s = "foo";
String t = s;
System.out.println(s == t);
----------------------------
true
```

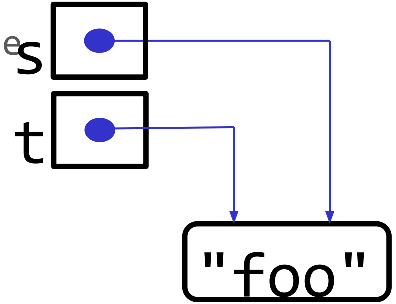


True or false?

```
int i = 5;
int j = 5;
System.out.println(i == j);
---------------------------
true
```



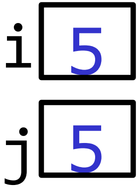

```
String s = "foo";
String t = s;
System.out.println(s == t);
---------------------------
true
```



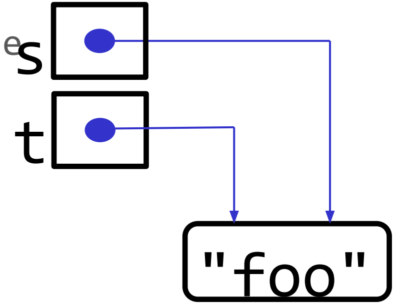

```
String u = "iPhone";
String v = u.toLowerCase();
String w = "iphone";
System.out.println(v == w);
---------------------------
```

True or false?

```
int i = 5;
int j = 5;
System.out.println(i == j);
-------------------------------
true
```

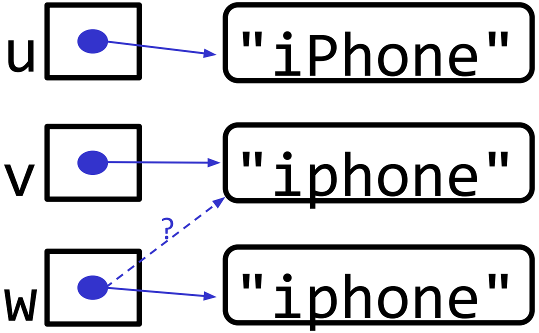



```
String s = "foo";
String t = s;
System.out.println(s == t);
-------------------------------
true
```




```
String u = "iPhone";
String v = u.toLowerCase();
String w = "iphone";
System.out.println(v == w);
-------------------------------
```

false (in practice)



The moral

- **Always use `.equals` to compare object refs!**
  - (Except for enums, which are special)
  - **The == operator can fail silently and unpredictably when applied to object references**
  - Same goes for the `!=` operator

Overriding `Object` implementations

- **No need to override `equals` and `hashCode` if you want identity semantics**
  - **When in doubt, don't override them**
  - It's easy to get it wrong
  - Records give you equals for free, neato!

- **Nearly always override `toString`**
  - `println` invokes it automatically
  - Why settle for ugly?

Overriding `toString` is easy and beneficial

```
final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;
      ...
    @Override public String toString() {
        return String.format("(%03d) %03d-%04d",
            areaCode, prefix, lineNumber);
    }
}

Number jenny = ...;
System.out.println(jenny);
```

Prints: `(707) 867-5309`

Typescript notes.

There is also a `toString`.

Equality is a funny thing: == (equality) vs === (strict equality)

- Typescript requires that you compare things that are the same type, so this distinction is SLIGHTLY less important.
- Javascript lets you do `10 == ‘10’ // true`

Equivalent behavior for, say, presence of an object in a Collection, is a bit trickier (no off-the-shelf equivalent of `equals`, but many ways to get it).

slido

Audience Q&A Session

ⓘ Start presenting to display the audience questions on this slide.

Back to antipatterns/refactoring

Anti-patterns

- Common system-level anti-patterns
    - Bad encapsulation, violates information hiding
        - public fields should be private; interface leaks implementation details; lack of interface
    - Bad modularization, violates coupling
        - related methods in different places, or vice versa; very large interface; “god” class
    - Bad abstraction, violates cohesion
        - Not exposing relevant functionality; near-identical classes; too many responsibilities
    - Bad inheritance/hierarchy
        - Violating behavioral subtyping; unnecessary inheritance; very large hierarchies (too wide or too deep)

Anti-patterns

- Zooming in: common code smells
  - Not necessarily bad, but worthwhile indicators to check
    - When problematic, often point to design problems
  - Long methods, large classes, and the likes. Suggests bad abstraction
    - Tend to evolve over time; requires restructuring
  - Inheritance despite low coupling (“refused bequest”)
    - Replace with delegation, or rebalance hierarchy
  - ‘instanceof’ (or ‘switch’) instead of polymorphism
  - Overly similar classes, hierarchies
  - Any change requires lots of edits
    - High coupling across classes (“shotgun surgery”), or heavily entangled implementation (intra-class)

Anti-patterns

- Zooming in: common code smells
  - Not necessarily bad, but worthwhile indicators to check
    - When problematic, often point to design problems
  - Excessive, unused hierarchies
  - Operations posing as classes
  - Data classes
    - Tricky: not always bad, but ideally distinguish from regular classes (e.g., ‘record’), and assign responsibilities if any exist (think: FlashCard did equality checking)
  - Heavy usage of one class’ data from another (“feature envy”, “inappropriate intimacy”; poor coupling)
  - Long chains of calls needed to do anything (law of demeter)
  - A class that only delegates work

Anti-patterns

- You can detect them from either side
  - Pick a design principle, look for violations
  - Identify “weird” code and isolate design flaw

Anti-patterns

- You can detect them from either side
  - Pick a design principle, look for violations
  - Identify “weird” code and isolate design flaw
- All fairly easy to spot on their own
  - But in HW4, there are multiple, tangled up
    - We actually provide way more guidance than you’ll get in the wild!
  - How do you approach that?

Refactoring and Anti-patterns

- Identifying multiple design problems
  - Make a list
    - Read the code, record anything that stands out
      - Pay attention to class names and their (apparent) interfaces
      - Make note of repetitive code (esp. across methods)
    - Draw a diagram, using a tool or by hand
      - Spot duplication, (lack of) interfaces, strange inheritance
    - This takes **practice**
  - <u>Don’t solve every problem</u>
    - Many issues are orthogonal
      - Or, at least, you can improve things somewhat
    - When issues intersect, prioritize fixing interfaces

Refactoring

- So where is “refactoring” in all this?
  - It’s what comes next.
  - Most design issues can be resolved with functionality-preserving transformation(s)
    - Too many parameters? Merge relevant ones into object, and/or replace with method calls.
    - Two near-identical classes? Merge their signatures using renamings, parameterization, then delete one or extract super-class

slido

Audience Q&A Session

ⓘ Start presenting to display the audience questions on this slide.

More useful patterns! Remember that long parameter list?

Fluent APIs / Cascade Pattern

Setting up Complex Objects

Long constructors, lots of optional parameters, long lists of statements

```
Option find = OptionBuilder
    .withArgName("file")
    .hasArg()
    .withDescription("search..." )
    .create("find");
```

```
client.getItem('user-table')
    .setHashKey('userId', 'userA')
    .setRangeKey('column', '@')
    .execute()
    .then(function(data) {
        ...
     })
```

Liquid APIs

Each method changes state,
then returns `this`

(Immutable version: Return modified copy)

```
class OptBuilder {
    private String argName = "";
    private boolean hasArg = false;
    ...
    OptBuilder withArgName(String n) {
        this.argName = n;
        return this;
    }
    OptBuilder hasArg() {
        this.hasArg = true;
        return this;
    }
    ...
    Option create() {
        return new Option(argName,
             hasArgs, ...)
    }
}
```

Python: Named parameters

```
parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                     help='sum the integers (default: find the max)')
```

JavaScript: JSON Objects

```
var argv = require('yargs/yargs')(process.argv.slice(2))
  .option('size', {
    alias: 's',
    describe: 'choose a size',
    choices: ['xs', 's', 'm', 'l', 'xl']
  })
   .argv
```

Notice the combination of cascading and complex JSON parameters

Fluent APIs: Discussion and Tradeoffs

Problem: Complex initialization and configuration

Advantages:

- Fairly readable code
- Can check individual arguments
- Avoid untyped complex arguments

Disadvantages:

- Runtime error checking of constraints and mandatory arguments
- Extra complexity in implementation
- Not always obvious how to terminate
- Possibly harder to debug

Iterator Pattern & Streams

(what's up with `for(Person p : this.records)`?)

Traversing a collection

- Since Java 1.0:

```
  Vector arguments = …;
  for (int i = 0; i < arguments.size(); ++i) {
    System.out.println(arguments.get(i));
  }
```

- Java 1.5: enhanced for loop

```
List<String> arguments = …;
for (String s : arguments) {
  System.out.println(s);
}
```

- Works for every implementation of `Iterable`

```
public interface Iterable<E> {
  public Iterator<E> iterator();
}
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
```

- In JavaScript (ES6)

```
let arguments = …
for (const s of arguments) {
  console.log(s)
}
```

- Works for every implementation with a "magic" function `[Symbol.iterator]` providing an iterator

```
interface Iterator<T> {
  next(value?: any): IteratorResult<T>;
  return?(value?: any): IteratorResult<T>;
  throw?(e?: any): IteratorResult<T>;
}
interface IteratorReturnResult<TReturn> {
  done: true;
  value: TReturn;
}
```

The Iterator Idea

Iterate over elements in arbitrary data structures (lists, sets, trees) without having to know internals

Typical interface:

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
}
```

(in Java also remove)

Using an iterator

Can be used explicitly

```
List<String> arguments = …;
for (Iterator<String> it = arguments.iterator(); it.hasNext();  ) {
  String s = it.next();
  System.out.println(s);
}
```

Often used with magic syntax:

```
for (String s : arguments)
for (const s of arguments)
```

Java: Getting an Iterator

```
public interface Collection<E> extends Iterable<E> {
  boolean     add(E e);
  boolean     addAll(Collection<? extends E> c);
  boolean     remove(Object e);
  boolean     removeAll(Collection<?> c);
  boolean     retainAll(Collection<?> c);
  boolean     contains(Object e);
  boolean     containsAll(Collection<?> c);
  void        clear();
  int         size();
  boolean     isEmpty();
  Iterator<E> iterator();
  Object[]    toArray()
  <T> T[]     toArray(T[] a);
  …
}
```

Defines an interface for creating an Iterator,
but allows Collection implementation to decide which Iterator to create.

Iterators for everything

```
public class Pair<E> {
  private final E first, second;
  public Pair(E f, E s) { first = f; second = s; }









}
```

```
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { … }
```

An Iterator implementation for Pairs

```
public class Pair<E> implements Iterable<E> {
  private final E first, second;
  public Pair(E f, E s) { first = f; second = s; }
  public Iterator<E> iterator() {
    return new PairIterator();
  }
  private class PairIterator implements Iterator<E> {
    private boolean seenFirst = false, seenSecond = false;
    public  boolean hasNext() { return !seenSecond; }
    public  E next() {
      if (!seenFirst)  { seenFirst  = true; return first;  }
      if (!seenSecond) { seenSecond = true; return second; }
      throw new NoSuchElementException();
    }
    public void remove() {
      throw new UnsupportedOperationException();
    }
  }
}
```

```
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { … }
```

Iterator design pattern

- Problem: Clients need uniform strategy to access all elements in a container, independent of the container type
  - Order is unspecified, but access every element once
- Solution: A strategy pattern for iteration
- Consequences:
  - Hides internal implementation of underlying container
  - Easy to change container type
  - Facilitates communication between parts of the program

Iterator and FlashCards?

Streams

Stream ~ Iterator -- a sequence of objects

Typically provide operations to produce new stream from old stream (map, flatMap, filter) and operations on all elements (fold, sum) -- using higher-order functions/strategy

> Often provide efficient/parallel implementations (subtype polymorphism)

Built-in in Java since Java 8; basics in Node libraries in JavaScript

```java
List<String>results = stream.map(Object::toString)
                .filter(s -> pattern.matcher(s).matches())
                .collect(Collectors.toList());
```

```java
int sum = numbers.parallelStream().reduce(0, Integer::sum);
```

```javascript
for (let [odd, even] in numbers.split(n => n % 2, n => !(n % 2)).zip()) {
    console.log(`odd = ${odd}, even = ${even}`);  // [1, 2], [3, 4], ...
 }
```

```javascript
Stream(people).filter({age: 23}).flatMap("children").map("firstName")
   .distinct().filter(/a.*/i).join(", ");
```

Summary

- Practice applying design patterns, recognizing anti-patterns
  - Create scenarios and try to write code
  - Find examples in public projects
  - Use this time to gain experience
  - Read lots of code, think about alternatives, like in HW4
  - Learn a vocabulary of anti-patterns (even if imperfect)

slido

Audience Q&A Session

ⓘ Start presenting to display the audience questions on this slide.