Lecture capture has begun, available from Lectures page.

First two and a half weeks are packed.

Testing lab now, HTML next week, and Stars due on its heels.

Next week
  ▶ Try to have a functional Stars repl/cli by next Tuesday’s class.
  ▶ Lab 2 is on HTML. Then you can work on UI and details.

CS Department “Town Hall”, today 4:30pm.
An interface is a collection of related method signatures.

Interface methods contain no code.

- This has changed slightly in Java 8. See default methods.

A class implements an interface by

- Saying so: implements Runnable
- And: Implementing every method from the interface. (Use @Override)

An interface is used as a type, just like a class.

- An object of any class that implements the type can be assigned.
- Animal cardinal = new Bird(Color.RED, location);
- You can only call the Animal methods on cardinal, not Bird’s.
- bird.move() perhaps, but not bird.fly().
- List<Cat> cats = new ArrayList<>();
- cats.ensureCapacity(1000); is a compile error.
Interfaces specify, they don’t implement

- Export functionality without revealing/promising details.
  - List<String> names = Collections.emptyList();
  - What is the class of names?

- Constrain objects taken as arguments.
  - Arguments to methods: Collections.shuffle(List)
  - Arguments to generic types, like...
Export functionality without revealing/promising details.
  ▶ List<String> names = Collections.emptyList();
  ▶ What is the class of names?

Constrain objects taken as arguments.
  ▶ Arguments to methods: Collections.shuffle(List)
  ▶ Arguments to generic types, like...
  ▶ public class KdTree<T extends Cartesian>
Interfaces specify, they don’t implement

- Export functionality without revealing/promising details.
  - `List<String> names = Collections.emptyList();`  
  - What is the class of `names`?

- Constrain objects taken as arguments.
  - Arguments to methods: `Collections.shuffle(List)`
  - Arguments to generic types, like...
  - `public class KdTree<T extends Cartesian>`

- Or both in one example.
  - `List safe = Collections.unmodifiableList(lst)`
Interfaces specify, they don’t implement

- Export functionality without revealing/promising details.
  - List<String> names = Collections.emptyList();
  - What is the class of names?

- Constrain objects taken as arguments.
  - Arguments to methods: Collections.shuffle(List)
  - Arguments to generic types, like...
  - public class KdTree<T extends Cartesian>

- Or both in one example.
  - List safe = Collections.unmodifiableList(lst)
  - Your first Pattern: Decorator (But a particularly vacuous example)
Example: Comparable

- Java supplies a Collections.sort() method.
- What can you pass it?
  - How would you sort Strings? LatLng? Chairs?

The sort() routine doesn't want to make these choices. So it requires that its argument implement Comparable. To be Comparable, objects implement one method, compareTo(). Collections.sort calls compareTo() as needed.

Back to LatLng... What might LatLng.compareTo() do? If there's no LatLng.compareTo(), can I sort them? The two argument version of sort() takes another interface, Comparator, to add more flexibility.
Example: Comparable

- Java supplies a Collections.sort() method.
- What can you pass it?
  - How would you sort Strings? LatLng? Chairs?
- The sort() routine doesn’t want to make these choices.
- So it requires that its argument implement Comparable.
- To be Comparable, objects implement one method, compareTo().
- Collections.sort calls compareTo() as needed.
- Back to LatLng... What might LatLng.compareTo() do?
Java supplies a Collections.sort() method.
What can you pass it?
  ▶ How would you sort Strings? LatLng? Chairs?
The sort() routine doesn’t want to make these choices.
So it requires that its argument implement Comparable.
To be Comparable, objects implement one method, compareTo().
Collections.sort calls compareTo() as needed.
Back to LatLng... What might LatLng.compareTo() do?
If there's no LatLng.compareTo(), can I sort them?
Java supplies a Collections.sort() method. What can you pass it?

- How would you sort Strings? LatLng? Chairs?

The sort() routine doesn’t want to make these choices. So it requires that its argument implement Comparable.

To be Comparable, objects implement one method, compareTo().

Collections.sort calls compareTo() as needed.

Back to LatLng. . . What might LatLng.compareTo() do?

If there’s no LatLng.compareTo(), can I sort them?

The two argument version of sort() takes another interface, Comparator, to add more flexibility.
public interface Comparable<T> {

    /**
     * ... 
     * @param o the object to be compared.
     * @return a negative integer, zero, or a positive integer as this object
     * is less than, equal to, or greater than the specified object.
     * @throws NullPointerException if 'o' is null
     * @throws ClassCastException if the type of 'o'
     * cannot being compared to this.
     */

    public int compareTo(T o);
}
Comparator separates a class and a sort order

```java
public interface Comparator<T> {
    int compare(T o1, T o2);
}

public NorthFirst implements Comparator<LatLong> {
    int compare(LatLng a, LatLng b) {
        return Double.compare(a.getLatitude(), b.getLatitude());
    }
}

Collections.sort(somePoints, new NorthFirst());
// somePoints is now sorted. How?
// What happens if some points have the same latitude?
```
A Comparator can profitably use state

```java
public Nearest implements Comparator<LatLong> {
  private final LatLng pt;

  public Nearest(LatLng pt) {
    this.pt = pt;
  }

  int compare(LatLng a, LatLng b) {
    return Double.compare(pt.distance(a), pt.distance(b));
  }
}

LatLng pvd = new LatLng(42.36, -71.09);
Collections.sort(somePoints, new Nearest(pvd));
// somePoints are now in order from Providence.
// With lambda expressions, this becomes one line.
```
Using interfaces

```java
interface Vehicle { void move(); }

class Car implements Vehicle {
    public void move() { drive(); }
    void drive() { /* */ }
}

class Boat implements Vehicle {
    public void move() { float(); }
    void float() { /* */ }
}

Vehicle v = new Vehicle();
Vehicle car = new Car();
Vehicle boat = new Boat();

car.drive();
boat.move();

Which option contains only lines with compile-time errors?
Assume lines 8-12 are in a method (same package).
A) 2, 5      B) 3, 6      C) 9, 10      D) 4, 8      E) 6, 11
```
Case study: Battleship Tournament

Consider a library for running games, or entire tournaments.
  ▶ User implements some strategies, library tries them out.

How can the library avoid impacting player design?
  ▶ Good libraries do not impose their world-view on callers.

The library provides a Game and Tournament abstraction.
  ▶ How would you implement Game? How does a caller construct one?
  ▶ How about Tournament? How can the library make games on demand?
What are the Classes?

- What information do we need?
  - To describe the competitors?
  - To enforce rules?
  - To determine a winner (of a game? of a tournament?)

- How can callers supply that information?
  - Without undue inconvenience.
  - Or constraint on their design.
First, imagine you are the caller.

- In a library, your first concern should be how your code will be used.
- How would you like the code to look?
- What would you pass to a “Game” object?
- A “Tournament” object?

```java
public class Main {
    public static main(String[] args) {
        Game g = new Game(...);
        ...
        Tournament t = new Tournament(...);
    }
}
```

What “makes sense” in the ellipses?
Now, realize you are not the caller.

- You should not force callers to use your classes.
- You should not expose your internal classes.
- Interfaces are much more flexible.
- Consider exactly what Game needs from Player.
  - Start the game, convey rules.
  - Ask the Player to take a turn.
  - Report to the Player what happened.
- Define an interface with exactly those requirements.
- A caller may implement the interface using their own existing abstractions.
package edu.brown.cs.staff.battleship;

import java.util.Map;

public interface Player {
    Map<Ship, Placement> setup(Position max);
    Position fire();
    void radar(Position pos, Impact impact);
}
How much design is dictated?

- Note that there’s no concept of Board, or “Game State”
  - Yes, Game will surely have one internally.
  - But clients do not use Game’s representation.
  - Game need not even be a public class.
  - General principle: sharing state is error-prone.

- Clients are not required to inherit from a class.
  - A Java class extends only one superclass.
  - But it implements as many interfaces as desired.
  - What if a caller also wants to use a UI or persistence library?

- Why expose Placement, Position, and Impact?
  - These are “small” concepts.
  - And are mostly information for the caller.
  - All are immutable.
  - You could make them interfaces too, but little gain.
A Player is largely unconstrained

Let’s look at...

- DumbPlayer
- RandomPlayer
- DecentPlayer
- MemoPlayer extents WrapPlayer

Note that none contain a Board. They have their own notions of important state.
What about Tournaments?

- Can you run a Tournament with the Player interface?
- Maybe. Tournament could call `setup()` for each new game.
- But there are drawbacks
  - Explaining the Player “lifecycle” becomes more complex.
  - Reinitializing objects “smells bad.”
  - Doomed to run the tournament on only one core.
- Let’s have the Tournament start each Game with fresh Players.
- How can callers provide all those Players?
A bad idea...

```java
public class Main {
    public static main(String[] args) {
        Tournament t = new Tournament();
        List<Player> randoms = new ArrayList<>();
        List<Player> decent = new ArrayList<>();
        for (int i = 0; i < 100; i++) {
            randoms.add(new RandomPlayer());
            decent.add(new DecentPlayer());
        }
        t.addPlayer("Randy", randoms);
        t.addPlayer("Joe", decent);
        t.run();
    }
}
```

Could we let the Tournament instantiate the Players as needed, instead?
Factories abstract creation

- Code in one module needs to create objects best made by another.
- Tournament needs to create Players to run more Games.
- The second module can supply a Factory instead of the objects themselves. (Your second Pattern)

```java
package edu.brown.cs32.staff.battleship;

public interface PlayerFactory {
    public Player create();
}
```
A better idea.

```
1 public static main(String[] args) {
2     Tournament t = new Tournament();
3     t.addPlayer("Randy", new RandomFactory());
4     t.addPlayer("Dumbo", new DumbFactory());
5     t.run();
6 }
```

As needed, the Tournament invokes `create()` on the supplied Factory.
Review

- Design in terms of interfaces
- Keep the interfaces short and abstract, only what is *required*.
- Only expose classes if you must, or they are *very* simple.
- Think about different approaches
  - Although there are no completely “right” or “wrong” answers,
  - Some *are* better than others
    - Simpler, easier to use
    - More flexible in handling evolution
  - If a design is “bad” in one way, it had better have something else to recommend it.
Before Java 8, interfaces can be wordy

- Recall Tournament.addPlayer(PlayerFactory pf).
- Defined a new interface, PlayerFactory, to let callers do creation as needed.
- Callers created a PlayerFactory implementation, then constructed it.
- t.addPlayer("Randy", new RandomFactory())
- Wordy, and “hides” the implementation (from the callsite)
Lambda expressions to the rescue

- Prefer `Tournament.addPlayer(Supplier<Player> sp)`\(^1\)
  - Call with `t.addPlayer("Randy", () => new RandomPlayer())`
- Compiler sees that `Tournament.addPlayer` expects a “Functional” or “Single Abstract Method” interface.
- So it creates one.
- Handy interfaces like `Supplier` avoid declaring a lot of new interfaces.
- Lambdas avoid the creation of tiny interface implementations.
- You can also use a “method reference” to implement a SAM.
  - Then call with `t.addPlayer("Randy", RandomPlayer::new)`

---
\(^1\) `java.util.function.Supplier`, which contains only `get()`.