Lecture capture has begun, available from Lectures page.

First two and a half weeks are packed.

Testing lab done, HTML started, and Stars due next Friday.

Department has a lot of student research opportunities
  ▶ MURA Office hours – Thurs at 12-1pm in CIT 546
  ▶ Contact mura@cs.brown.edu for more info.
Engineering with Interfaces

John Jannotti

/course/cs0320/www/lectures/

Feb 1, 2018
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 do not 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...
Interfaces specify, they do not 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>
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 do not 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?

If you look it up, you find:
```java
void sort(List<T> list)
```

Where did T come from? What should T mean? sort() shouldn't need to know how every T is ordered. So it requires that its argument implement Comparable.

Technically:
```java
<T extends Comparable<? super T>>
```

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?
  - If you look it up, you find: void sort(List<T> list)
  - Where did T come from? What should T mean?

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?
  - If you look it up, you find: void sort(List<T> list)
  - Where did T come from? What should T mean?
- sort() shouldn’t need to know how every T is ordered.
- So it requires that its argument implement Comparable.
- Technically: <T extends Comparable<? super T>>
- 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?
▶ If you look it up, you find: void sort(List<T> list)
▶ Where did T come from? What should T mean?

sort() shouldn’t need to know how every T is ordered.

So it requires that its argument implement Comparable.

Technically: <T extends Comparable<? super T>>

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?
- If you look it up, you find: void sort(List<T> list)
- Where did T come from? What should T mean?

sort() shouldn’t need to know how every T is ordered.
So it requires that its argument implement Comparable.
Technically: <T extends Comparable<? super T>>

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 be being compared to this.
     */

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

```java
1 public interface Comparator<T> {
2     int compare(T o1, T o2);
3 }
4 public NorthFirst implements Comparator<LatLng> {
5     int compare(LatLng a, LatLng b) {
6         return Double.compare(a.getLatitude(),
7         b.getLatitude());
8     }
9 }
10 Collections.sort(somePoints, new NorthFirst());
11 // somePoints is now sorted. How?
12 // What happens if some points have the same latitude?
```
A Comparator can profitably use state

```java
public Nearest implements Comparator<LatLng> {
    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
1 interface Vehicle { void move(); }
2 class Car implements Vehicle {
3     public void move() { drive(); }
4     void drive() { /* */ }
5 }
6 class Boat implements Vehicle {
7     public void move() { float(); }
8     void float() { /* */ }
9 }
10 Vehicle v = new Vehicle();
11 Vehicle car = new Car();
12 Vehicle boat = new Boat();
13 car.drive();
14 boat.move();
```

Which lines contain compile-time errors?
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.
  - Get Player’s ship positions.
  - 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. Not ETU.
  - Reinitializing objects “smells bad.” Not SFB.
  - Doomed to run the tournament on only one core. Not RFC.
- 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> decents = new ArrayList<>();
        for (int i = 0; i < 100; i++) {
            randoms.add(new RandomPlayer());
            decents.add(new DecentPlayer());
        }
        t.addPlayer("Randy", randoms);
        t.addPlayer("Joe", decents);
        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.

```java
public static main(String[] args) {
    Tournament t = new Tournament();
    t.addPlayer("Randy", new RandomFactory());
    t.addPlayer("Dumbo", new DumbFactory());
    t.run();
}
```

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. (ETU, SFB)
    - More flexible in handling evolution. (RFC)
  - If a design is “bad” in one way, it had better have something else to recommend it.
  - You *might* sacrifice ETU for a large performance gain, but you had better be *sure* you need to. Hint: profile.
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()`. 