Information Session
Thursday, September 22nd
5:00pm
Barus & Holley 161
shpe.brown@gmail.com
www.fb.com/shpebrown

CS15 & beyond

Gregory Chatzinoff ’15
Software Engineer, Chrome iOS

About Me
At Brown:
- CS A.B.
- Research Assistant in Andy’s group
- HTA/TA’d CS15 (played Joffrey in Game of Thrones skit)
- Involved with student government

Since graduation:
- Software Engineer on Chrome iOS team in Mountain View, CA
- UI improvements, stability/memory management experiments, back end work, data analysis, etc.

CS15 is a transformative course

Life of a Software Engineer in Industry
- Large codebase with many contributors
- Code lasts a long time
- Design Docs
- Code Reviews
Who am I?

Studied Computer Science (Class of ‘15)
- Started late, from Engineering
- Bachelor of Arts
- Took classes in 13 depts
- Musical Forum & Brown Debating Union

Worked as a TA
- CS15: 2013, 2014
- CS8, CS1951A, HIST97

“CS15 kicked my butt & changed my life”

I’m not a Software Engineer!

Associate Product Manager at Google
- PM on the Polymer team. OOP on Web!
- Create vision, manage development, oversee product launch, repeat.
- “Captain of the [Pirate] Ship”

Internships
- Web/Mobile Developer at DreamIt Ventures
- Software Engineer at Redfin
Recall: Declaring vs. Defining Methods

- What's the difference between declaring and defining a method?
  - Method declaration is the scope (public), return type (void), name and parameters (makeSounds())
  - Method definition is the body of the method – the actual implementation (the code that actually makes the sounds)

```java
public class Dog {
    //constructor elided
    public void makeSounds() {
        this.bark();
        this.whine();
        this.bark();
    }
    public void bark() {
        //code elided
    }
    public void whine() {
        //code elided
    }
}
```

Review of Association

- Does School contain Teacher?
  - Yes! School instantiated Teacher, therefore School contains Teacher.
- Can School send messages to Teacher?
  - Yes! Teacher knows about School that created it, but does not contain it

```java
public class School {
    private Teacher _teacher;
    public School() {
        _teacher = new Teacher(this);
        this.assignTeacher();
    }
    public void assignTeacher() {
        //code elided
    }
}
```

Using What You Know

- Imagine this program:
  - Sophia and Dan are racing from their dorms to CIT
    - whoever gets there first, wins!
    - catch: they don’t get to choose their method of transportation
- Design a program that
  - assigns mode of transportation to each racer
  - starts the race
- For now, assume transportation options are Car and Bike
Goal 1: Assign transportation to each racer

- Need transportation classes (something to give to racers)
- Let’s use **Car** and **Bike** classes
- Both classes will need to describe how the transportation moves
  - **Car** needs `drive` method
  - **Bike** needs `pedal` method

Coding the project

- Let’s build transportation classes

```java
public class Car {
    public Car() { //constructor
        //code elided
    }
    public void drive() {
        //code elided
    }
    //more methods elided
}
public class Bike {
    public Bike() { //constructor
        //code elided
    }
    public void pedal() {
        //code elided
    }
    //more methods elided
}
```

Goal 1: Assign transportation to each racer (cont.)

- Need racer classes that will use their type of transportation
  - **CarRacer**
  - **BikeRacer**
- What methods will we need? What capabilities should each **Racer** class have?
  - **CarRacer** needs to know when to use the car
    - write `useCar()` method
  - **BikeRacer** needs to know when to use the bike
    - write `useBike()` method

Coding the project (cont.)

- Let’s build the racer classes

```java
public class CarRacer {
    private Car _car;
    public CarRacer() {
        _car = new Car();
    }
    public void useCar() {
        _car.drive();
    }
    //more methods elided
}
public class BikeRacer {
    private Bike _bike;
    public BikeRacer() {
        _bike = new Bike();
    }
    public void useBike() {
        _bike.pedal();
    }
    //more methods elided
}
```

Goal 2: Tell the racers to start the race

- Race class contains Racers
  - **App** contains **Race**
  - Race class will have `startRace()` method
    - `startRace()` tells each racer to use their transportation
    - `startRace()` gets called in **App**

Coding the project (cont.)

- Let’s build the **Race** class

```java
public class Race {
    private CarRacer _dan;
    private BikeRacer _sophia;
    public Race() {
        _dan = new CarRacer();
        _sophia = new BikeRacer();
    }
    public void startRace() {
        _dan.useCar();
        _sophia.useBike();
    }
}
```

startRace:
Tell _dan to useCar
Tell _sophia to useBike
Coding the project (cont.)

- Now build the App class
- Now the race to the CIT!

Solution 1: Create one Racer class with methods!

- Create one `Racer` class
  - define different methods for each type of transportation
- _.dan is instance of Racer_ and elsewhere we have:
  - `Car` danCar = new Car();
  - `.dan.useCar(danCar);`
- `Car`'s `drive()` method will be invoked
- But any given instance of `Racer` will need a new method to accommodate every kind of transportation

Solution 1 Drawbacks

- Now imagine all the CS15 TAs join the race and there are 10 different modes of transportation
- Writing these similar `useX()` methods are a lot of work for you, the developer, and inefficient coding style

What does our design look like?

Things to think about

- Do we need two different `Racer` classes?
  - Want multiple instances of `Racer` that use different modes of transportation
  - But how?

Can we do better?
Is there another solution?

- Can we go from left to right?

Interfaces and Polymorphism

- In order to simplify code, need to learn
  - Interfaces
  - Polymorphism

public class Car implements Transporter {
    public Car() {
        // code elided
    }
    public void drive() {
        // code elided
    }
    @Override
    public void move() {
        this.drive();
    }
    // more methods elided
}

public class Racer {
    // previous code elided
    public void useTransportation(Transporter transport) {
        transport.move();
    }
}

Interfaces: Spot the Similarities

- What do cars and bikes have in common?
- What do cars and bikes not have in common?

Cars vs. Bikes

- Play radio
- Turn off/on headlights
- Turn off/on turn signal
- Lock/unlock doors

- Move
- Drop kickstand
- Change gears

Digging deeper into the similarities

- How similar are they when they move?
  - do they move in same way?
  - Not very similar
    - cars drive
    - bikes pedal
  - Both can move, but in different ways

Can we model this in code?

- Many real-world objects have several broad similarities
  - cars and bikes can move
  - cars and laptops can play radio
  - phones and Teslas can be charged

- Take Car and Bike class
  - how can their similar functionalities get enumerated in one place?
  - how can their broad relationship get portrayed through code?

- Play radio
- Lock/unlock doors
- Drive

- Drop kickstand
- Change gears
- Pedal
Introducing Interfaces

- Interfaces group similar capabilities/function of different classes together
- Model “acts-as” relationship
- Cars and Bikes could implement a Transporter interface
  - they can transport people from one place to another
  - “act as” transporters
    - objects that can move
    - have shared functionality, such as moving, braking, turning etc.
  - for this lecture, interfaces are green and classes that implement them pink

- Interfaces are contracts that classes agree to
  - If classes choose to implement given interface, it must define all methods declared in interface
    - if classes don’t implement one of interface’s methods, the compiler raises error
    - later we’ll discuss strong motivations for this contract enforcement
  - Interfaces don’t define their methods - implementing classes do
    - Interfaces only care about the fact that the methods get defined - not how – implementation-agnostic
  - Models similarities while ensuring consistency
    - What does this mean?

Models Similarities While Ensuring Consistency

- How does this help our program?
- We know Cars and Bikes both need to move
  - i.e., should all have some move() method
  - let compiler know that too!
- Let’s make the Transporter interface!
  - what methods should the Transporter interface declare?
    - move()
      - only using a move() for simplicity, but brake(), etc. would also be useful
    - compiler doesn’t care how method is defined, just that it’s been defined
    - general tip: methods that interface declares should model functionality all implementing classes share

Let’s break that down

1) Models Similarities

2) Ensures Consistency

Interface Example (1/4)

What does this look like?

```java
public interface Transporter {
    public void move();
}
```

- That’s it!
- Interfaces, just like classes, have their own .java file. This file would be Transporter.java

Interface Example (2/4)

What does this look like?

```java
public interface Transporter {
    public void move();
}
```

- Declare it as interface rather than class
Interface Example (3/4)

What does this look like?

```java
public interface Transporter {
    public void move();
}
```

- Declare methods - the contract
- In this case, only one method required: `move()`
- All classes that sign contract (implement this interface) must define actual implementation of any declared methods

Interface Example (4/4)

What does this look like?

```java
public interface Transporter {
    public void move();
}
```

- Interfaces can only declare methods - not define them
- Notice: method declaration end with semicolons, not curly braces!

Clicker Questions
Which line of this program is incorrect?

A. `public interface Colorable {
    public Color getColor();
    }
`  
B. `return Color.WHITE;
`  
C. `public class Rectangle implements Colorable {
    //constructor elided
    public Color getColor() {
        return Color.PURPLE;
    }
}
`  
D. `@Override
    public Color getColor() {
    return Color.PURPLE;
    }
`  
E. `public class Rectangle implements Colorable {
    //constructor elided
    public Color getColor() {
        return Color.PURPLE;
    }
}
```

Implementing an Interface (1/6)

Let's modify `Car` to implement `Transporter`

```java
public class Car implements Transporter {
    public Car() {
        // constructor
    }
    public void drive() {
        // code for driving the car
    }
}
```

- Let's modify `Car` to implement `Transporter`
- Declare that `Car"acts-as"Transporter`
- Add implements `Transporter` to class declaration
- Promises compiler that `Car` will define all methods in `Transporter` interface (i.e., `move()`)
- Will this code compile?

Implementing an Interface (2/6)

```java
public class Car implements Transporter {
    public Car() {
        // constructor
    }
    public void drive() {
        // code for driving the car
    }
}
```

"Error: Car does not override method move() in Transporter"

- Will this code compile?
- Never implemented `move()` and `drive()` - doesn’t suffice. Compiler will complain accordingly.

Implementing an Interface (3/6)

```java
public class Car implements Transporter {
    public Car() {
        // constructor
    }
    public void drive() {
        // code for driving the car
    }
    @Override
    public void move() {
        this.drive();
    }
}
```

- Next: honor contract by defining a `move()` method
- Method `signature` (name and number/type of arguments) must match how its declared in interface
Implementing an Interface (4/6)

What does `@Override` mean?

- Include `@Override` right above the method signature
- `@Override` is an annotation—a signal to the compiler (and to anyone reading your code)
  - allows compiler to enforce that interface actually has method declared
  - more explanation of `@Override` in next lecture
- Annotations, like comments, have no effect on how code behaves at runtime

```java
public class Car implements Transporter {
    public Car() {
        // constructor
    }
    public void drive() {
        //code for driving car
    }
    @Override
    public void move() {
        this.drive();
    }
}
```

Implementing an Interface (5/6)

- Defining interface method is like defining any other method
- Definition can be as complex or as simple as it needs to be
- Ex.: Let’s modify `Car`’s `move` method to include braking
- What will instance of `Car` do if `move()` gets called on it?

```java
public class Car implements Transporter {
    //previous code elided
    public void drive() {
        //code elided
    }
    @Override
    public void move() {
        this.drive();
        this.brake();
        this.drive();
    }
    public void brake() { //code elided}
}
```

Implementing an Interface (6/6)

- As with signing multiple contracts, classes can implement multiple interfaces
  - “I signed my rent agreement, so I’m a renter, but I also signed my employment contract, so I’m an employee. I’m the same person.”
  - what if I wanted `Car` to change color as well?
  - create a `Colorable` interface
  - add that interface to `Car`’s class declaration
- Implementing class must define every single method in each of its every interfaces

```java
public interface Colorable {
    public void setColor(Color c);
    public Color getColor();
}
public class Car implements Transporter, Colorable {
    public Car() { //body elided }
    public void drive(){ //body elided }
    public void move(){ //body elided }
    @Override
    public void setColor(Color c){ //body elided }
    public Color getColor(){ //body elided  }
}
```

Modeling Similarities While Ensuring Consistency

- Interfaces are formal contracts and ensure consistency
  - compiler will check to ensure all methods declared in interface are defined
  - Can trust that any object from class that implements `Transporter` can `move()`
  - Will know how 2 classes are related if both implement `Transporter`

```
public interface Clickable {
    public void click();
}
```

Clicker Question

Given the following interface:

```java
public interface Clickable {
    public void click();
}
```

Which of the following would work as an implementation of the Clickable interface? (don’t worry about what `changeXPosition` does)

A. `@Override
    public void click() {
        this.changeXPosition(100.0);
    }
    @Override
    public void click(double xPosition) {`  
B. `@Override
    public void clickIt() {
        this.changeXPosition(100.0);
    }
    @Override
    public double click() {`
C. `@Override
    public void clickIt() {
        this.changeXPosition(100.0);
    }
    @Override
    public void click() {
        return this.changeXPosition(100.0);
    }
```

Back to the CIT Race

- Let’s make transportation classes use an interface

```java
public class Car implements Transporter{
    public Car() { //previous code elided
        //code elided
    }
    @Override
    public void drive() {
        //code elided
    }
    @Override
    public void move() {
        this.drive();
    }
}
```

```java
public class Bike implements Transporter{
    public Bike() { //previous code elided
        //code elided
    }
    @Override
    public void pedal() {
        //code elided
    }
    @Override
    public void move() {
        this.pedal();
    }
    //more methods elided
}
```
Leveraging Interfaces

- Given that there's guarantee anything that implements Transporter knows how to move, how can it be leveraged to create single useTransportation() method?

```java
Racer
useCar(Car car)
useBike(Bike bike)
useHoverBoard(HoverBoard hoverboard)
useScooter(Scooter scooter)
useMotorcycle(Motorcycle motorcycle)
usePogoStick(PogoStick pogo)
```

Introducing Polymorphism

- Poly = many, morph = forms
- A way of coding generically
  - way of referencing many related objects as one generic type
    - cars and bikes can both move() — refer to them as Transporter objects
    - phones and Teslas can both getCharged() — refer to them as Chargeable objects, i.e., objects that implement Chargeable interface
    - cars and boomboxes can both playRadio() — refer to them as Radioplayer objects
- How do we write one generic useTransportation() method?

```java
public class Racer {
    //previous code elided
    public void useTransportation(Transporter transportation) {
        transportation.move();
    }
}
```

What would this look like in code?

```java
public class Racer {
    //previous code elided
    public void useTransportation(Transporter transportation) {
        transportation.move();
    }
}
```

Let's break this down.

1. Actual vs. Declared Type
2. Method resolution

Actual vs. Declared Type (1/2)

- Consider following piece of code:
  ```java
  Transporter dansCar = new Car();
  ```
- ...is that legal?
  - doesn't Java do strict type checking? (type on LHS = type on RHS)
  - how can instances of Car get stored in Transporter variable?

Actual vs. Declared Type (2/2)

- Can treat Car/Bike object as Transporter objects
- Can be the actual type
  - Java will look in this class for the definition of the method
- Transporter is declared type
  - Java will limit caller so it can only call methods on instances that are declared as Transporter objects
- If Car defines playRadio() method, is transportation.playRadio() correct?
Determining the Declared Type

- What methods do Car and Bike have in common?
  - `move()`
- How do we know that?
  - They implement `Transporter`.
    - Guarantees they have `move()` method.
- Think of `Transporter` like the "lowest common denominator".
  - It's what all transportation classes will have in common.

Is this legal?

- `Transporter sophiasBike = new Bike();` ✓
- `Transporter sophiasCar = new Car();` ✓
- `Transporter sophiasRadio = new Radio();` ✗

Motivations for Polymorphism

- Many different kinds of transportation but only care about their shared capability.
  - I.e. how they move.
- Polymorphism let programmers sacrifice specificity for generality.
  - Treat any number of classes as their lowest common denominator.
  - Limited to methods declared in that denominator.
- For this program, that sacrifice is ok.
  - Racer doesn't care if instance of Car can `playRadio()` or if instance of Bike can `dropKickstand()`.
  - Only method Racer wants to call is `move()`.

Polymorphism in Parameters

- What are implications of this method declaration?
  ```java
  public void useTransportation(Transporter transportation) {
      //code elided
  }
  ```
  - `useTransportation` will accept any object that implements `Transporter`.
  - `useTransportation` can only call methods declared in `Transporter`.

Is this legal?

- `Transporter sophiasBike = new Bike();` ✓
- `Transporter sophiasCar = new Car();` ✓
- `Transporter sophiasRadio = new Radio();` ✗

```java
public class Racer {
    //previous code elided
    public void useTransportation(Transporter transportation) {
        transportation.move();
    }
}
```

Why `move()`? (1/2)

- Why call `move()`?
- What `move()` method gets executed?
Why move()? (2/2)

- Only have access to `Transporter` object
  - cannot call `transportation.drive()` or `transportation.pedal()`
    - that’s okay, because all that’s needed is `move()`
  - limited to the methods declared in `Transporter`

Method Resolution: Which `move()` is executed?

- Consider this line of code in `Race` class:
  ```java
  _sophia.useTransportation(new Bike());
  ```
- Remember what `useTransportation` method looked like
  ```java
  public void useTransportation(Transporter transportation) {
    transportation.move();
  }
  ```
  What is “actual type” of `transportation` in this method invocation?

Method Resolution (1/4)

- `Bike` is actual type
  - `Racer` was handed instance of `Bike` `new Bike()` is argument
  - `Transporter` is declared type
  - `Racer` treats `Bike` object as `Transporter` object
  - So… what happens in `transportation.move()`?
    - `move()` method gets used?

Method Resolution (2/4)

- `_Sophia` is a `Racer`
- `Bike`’s `move()` method gets used
- Why?
  - `Bike` is actual type
  - Java will execute methods defined in `Bike` class
  - `Transporter` is declared type
  - Java limits methods that can be called to those declared in `Transporter` interface

Method Resolution (3/4)

- What if `_sophia` received instance of `Car`?
  - `Car`!
    ```java
    public class Race {
      private Racer _sophia;
      //previous code elided
      public void startRace() {
        _sophia.useTransportation(new Car());
      }
    }
    ```

Method Resolution (4/4)

- This method resolution is example of dynamic binding, which is when actual method implementation used is not determined until runtime
  - contrast with static binding, in which method gets resolved at compile time
  - `move()` method is bound dynamically – Java does not know which `move()` method to use until program runs
  - same “`transport.move()`” line of code could be executed indefinite number of times with different method resolution each time
Clicker Question
Given the following class:

```java
class Laptop implements Typeable, Clickable {
    public void type() {
        // code elided
    }
    public void click() {
        // code elided
    }
}
```
Given that typeable has declared the type method and clickable has declared the click method, which of the following calls is valid?

A. `Typeable macBook = new Typeable();
macBook.type();`
B. `Clickable macBook = new Clickable();
macBook.type();`
C. `Typable macBook = new Laptop();
macBook.click();`
D. `Clickable macBook = new Laptop();
macBook.click();`

Why does that work? (1/2)
- **Declared type and actual type work together**
  - **declared type** keeps things generic
    - can reference a lot of objects using one generic type
  - **actual type** ensures specificity
    - when defining implementing class, methods can get implemented in any way

Why does that work? (2/2)
- **Declared type and actual type work together**
  - **declared type** keeps things generic
    - can reference a lot of objects using one generic type
  - **actual type** ensures specificity
    - when defining implementing class, methods can get implemented in any way

When to use polymorphism?
- **Using only functionality declared in interface or specialized functionality from implementing class?**
  - if only using functionality from the interface → polymorphism!
  - if need specialized methods from implementing class, don't use polymorphism
- **If defining goOnScenicDrive()**
  - Want to put `topDown()` on `Convertible`, but not every `Car` can put top down
    - Don't use polymorphism, every `Car` can't `goOnScenicDrive()` i.e., can't code generically

Why use interfaces?
- **Contractual enforcement**
  - will guarantee that class has certain capabilities
  - Example: `Car` implements `Transporter`, therefore it must know how to move()
- **Polymorphism**
  - Can have implementation-agnostic classes and methods
    - know that these capability exists, don't care how they're implemented
    - allows for more generic programming
    - `useTransportation()` can take in any `Transporter` object
    - can easily extend this program to use any form of transportation, with minimal changes to existing code
    - an extremely powerful tool for extensible programming

Why is this important?
- **With 2 modes of transportation!**
- **Old Design:**
  - need more classes → more specialized methods (useRollerblades(), useBike(), etc)
- **New Design:**
  - as long as the new classes implement `Transporter`, `Racer` doesn't care what transportation it has been given
  - don't need to change `Racer`!
    - less work for you!
    - just add more transportation classes that implement `Transporter`
The Program

```java
public class Race {
    private Racer _dan, _sophia;
    public Race() {
        _dan = new Racer();
        _sophia = new Racer();
    }
    public void startRace() {
        _dan.useTransportation(new Car());
        _sophia.useTransportation(new Bike());
    }
}
```

```java
public class App {
    public App() {
        Race r = new Race();
        r.startRace();
    }
}
```

```java
public class Racer {
    public Racer() {};
    public void useTransportation(Transporter transport) {
        transport.move();
    }
}
```

```java
public class Car implements Transporter {
    public Car() {};
    public void drive() {
        //code elided
    }
    public void move() {
        this.drive();
    }
}
```

```java
public class Bike implements Transporter {
    public Bike() {};
    public void pedal() {
        //code elided
    }
    public void move() {
        this.pedal();
    }
}
```

```java
public interface Transporter {
    public void move();
}
```

In Summary

- Interfaces are contracts
  - force classes to define certain methods
- Polymorphism allows for extremely generic code
  - treats multiple classes as their "generic type" while still allowing specific method implementations to be executed
- Polymorphism + Interfaces
  - generic coding
- Why is it helpful?
  - want you to be the laziest (but cleanest) programmer you can be

Announcements

- HW2 and LiteBrite are released today
- HW2 is due on Sunday (9/25) 2PM
  - Remember: homeworks only have one handin time
- LiteBrite early handin is Tuesday 09/27, on-time handin is Thursday 9/29, late handin is Saturday 10/1
  - LiteBrite help slides will be released on Friday
- Review sessions are held every Thursday from 7:30-9pm and Sunday from 12:00-1:30pm in MacMillan 115
- START EARLY, START TODAY, START YESTERDAY!!!!!
- PLEASE FILL OUT THE INITIAL SURVEY