No Experience Necessary!

Email fsae@brown.edu with questions or for more information
Review: Instance Variable Initialization

- Split up declaration and assignment of instance variable:
  - **declare** instance variable at the top of the class, to notify Java
  - **initialize** the instance variable by **assigning** a value to it in the constructor
  - Purpose of constructor is to initialize all instance variables so the instance has a valid initial “state” at its “birth”
  - **state** is the set of all values for all properties—we don’t consider local variables to be properties, they are “temporaries”

```java
public class PetShop {
    private DogGroomer _groomer;

    /* This is the constructor! */
    public PetShop() {
        _groomer = new DogGroomer();
        Dog dogE = new Dog();
        _groomer.groom(dogE);
    }
}
```
Lecture 5

Inheritance
Inheritance Overview
Inheritance is a hallmark of OOP – a powerful tool for modeling & code re-use

• Inheritance diagrams
  o Example: animals

• Inheriting methods and instance variables
  o Example: cell phones

• Overriding methods
  o Example: CS15 students

• Abstract methods and classes
  o Example: vehicles

• Method resolution
Car Troubles

- Andy, the HTAs, and the UTAs all need transportation to Nice Slice (really?!?)
- Need to model three different types of vehicle to suit everyone’s needs:
  - Andy will be getting there in a hurry in a Convertible
  - The HTAs will drive in a reliable old CS15Mobile
  - The UTAs need to pile into a Van large enough to fit them all (need a bus, but we take artistic license…)

Andrew van Dam © 2015 09/24/15
Similarities and Differences

• We’ll create three classes: Convertible, CS15Mobile, and Van

• Similarities:
  o properties: engine, doors, steering wheel, driver,…
  o capabilities: start engine, move, turn,…

• Differences among these classes:
  o Convertible: moves fast, 2 doors, holds 2 people
  o CS15Mobile: moves moderately, 4 doors, holds 5 people
  o Van: moves slowly, 4 doors (2 sliding), holds many people
Inheritance (1/5)

- In OOP, groups of related classes are modeled with an inheritance hierarchy.
- Inheritance models an “is a” relationship:
  - convertible “is a” vehicle
  - golden retriever breed “is a” dog
  - dog “is a” mammal
Inheritance (2/5)

- Each box in an inheritance diagram represents a class.
- A Mammal “is an” Animal, and a Moose “is a” Mammal.
- Transitivity: A Moose “is an” Animal.
- We say Mammal inherits from Animal, and Moose inherits from Mammal.
- In biology, would call this a taxonomy.
Inheritance (3/5)

- A **superclass** is a class that is inherited from; also called parent
- A **subclass** is a (child) class that inherits/derives from another class
- **Animal** is superclass of **Reptile, Mammal, and Fish**
- **Reptile, Mammal, and Fish** are subclasses of **Animal**
Inheritance (4/5)

- Class can be both a superclass and subclass at the same time--Mammal is a superclass of Cat, and a subclass of Animal
- In Java, each class can only inherit from a single superclass
  - Some other languages, like C++, allow for multiple inheritance, but it is easy to mess up
Inheritance (5/5)

- All classes are ultimately descended from Java’s `Object` class, which implements behavior common to all classes
- Every class is implicitly a subclass of `Object`
- `Animal` and any other highest-level superclass you define actually inherits from `Object`
- `Object` provides very few methods – not particularly useful
Inheritance: Motivation (1/2)

- A subclass inherits properties and capabilities from its superclass
- Since Animal has public methods eat and sleep, all of its subclasses (Reptile, Mammal, and Fish) automatically inherit these methods
- Same goes for all of their subclasses, and all of their subclasses’ subclasses, etc.
- The big win: subclass designer doesn’t have to recode an inherited method - it’s a freebie! Only implement what’s different...
Inheritance: Motivation (2/2)

- If we want to define a new class, Alligator, can derive it from an existing class that already contains a lot of the code needed for Alligator: code re-use is great!
- Can make Alligator a subclass of Reptile, and it will automatically inherit:
  - methods common to all Animals, like eat and sleep
  - methods common to all Reptiles, like layEggs
Inheritance: Superclasses and Subclasses

● A superclass **factors out commonalities** among its subclasses
  o describes everything that all subclasses have in common
  o *Animal* defines things common to all *Animals*

● A subclass **differentiates/specializes** its superclass by:
  o **adding** new methods: the *Fish* class should define *swim*. Not all *Animals* can *swim*, but all *Fish* can
  o **overriding** inherited methods: a *Bear* class might override its inherited *sleep* method so that it hibernates rather than sleeping as most other *Animals* do
  o **defining** “abstract” methods that the superclass declares but does not define (more on this later!)
Example 1: Cell Phones

- Let’s look at an example – **CellPhones** and **SmartPhones**
- **CellPhone** is the **superclass**
- **SmartPhone** is the **subclass**
Inheriting Methods and Instance Variables (1/8)

- Each Java class has methods and instance variables that can be either public or private
  - in CS15, instance variables are usually private and methods are usually public

```java
public class CellPhone{
    private Battery _battery;
    public CellPhone(){
        _battery = new Battery();
    }
    public void call(int numberToCall){
        // implementation elided
    }
}
```

_battery is private — can only be accessed from within CellPhone class

The call method is public — it can be accessed from anywhere
Inheriting Methods and Instance Variables (2/8)

- All **public** methods and instance variables of a superclass are inherited by subclasses.
- Since the `CellPhone` class has the **public** method `call`, all subclasses (like `SmartPhone`) automatically inherit this method.
  - A child knows how to do everything its parent does (and more)

```java
public class CellPhone{
    private Battery _battery;

    public CellPhone(){
        _battery = new Battery();
    }

    public void call(int numberToCall){
        // implementation elided
    }
}
```

- The `call` method is **public** -- it can be accessed from anywhere.
- `_battery` is **private** -- can only be accessed from within `CellPhone` class.
Inheriting Methods and Instance Variables (3/8)

- **SmartPhone** doesn’t have to redefine the `call` method (it *may optionally* redefine it-- this is called overriding. Stay tuned)

- If we instantiate a `SmartPhone`, we can tell it to `call` because it is a `CellPhone`
Inheriting Methods and Instance Variables (4/8)

- **SmartPhone** may define its own methods and variables, both public and private.
- Could define public methods `connectToWifi` and `openBrowser`, and private instance variable `_gps`.
- Now **SmartPhone** has properties and capabilities that not all **CellPhone**s have.

```cpp
private:
    private Battery _battery
    ...

public:
    public void call(int numberToCall)
    ...
```

```cpp
private:
    private GPSReceiver _gps
    ...

public:
    public void call(int numberToCall)
    public void connectToWifi()
    public void openBrowser()
    ...
```
Inheriting Methods and Instance Variables (5/8)

- What about `CellPhone`’s private instance variable `_battery`? Is it inherited by `SmartPhone`?
- A `SmartPhone` should have a battery, like any other `CellPhone`!
- But `_battery` is private-- it can only be accessed from within the `CellPhone` class
Inheriting Methods and Instance Variables (6/8)

- **private** variables or methods of a superclass are **not directly inherited** by its subclasses; superclass protects them from even its subclasses
- Subclass cannot directly access any of **CellPhone**’s **private** variables from **SmartPhone** class
- **SmartPhone** is completely unaware that a variable named **_battery** exists
Inheriting Methods and Instance Variables (7/8)

- But that’s not the whole story...
- Every instance of a subclass is also an instance of its superclass-- every instance of `SmartPhone` is also a `CellPhone`
- Therefore a variable named `_battery` does exist within every `SmartPhone` instance, but is *not directly accessible* by `SmartPhone` methods!!!

```cpp
private:
    private Battery _battery
    ...

public:
    public void call(int numberToCall)
    ...
```

```cpp
private:
    private GPSReceiver _gps
    [private Battery _battery;
    ...

public:
    public void call(int numberToCall)
    public void connectToWifi()
    public void openBrowser()
    ...
```
Inheriting Methods and Instance Variables (8/8)

- What if we want to allow subclasses access to the superclass’s instance variables?
  - can give *indirect* access by defining **public accessor** and **mutator methods** for **private** instance variables
  - we’ll see how to do this later in the lecture...

```cpp
private:
    private Battery _battery
    ...

public:
    public void call(int numberToCall)
    ...
```

```cpp
private:
    private GPSReceiver _gps
    [private Battery _battery;]
    ...

public:
    public void call(int numberToCall)
    public void connectToWifi()
    public void openBrowser()
    ...
```
Inheritance Summary

• A subclass inherits all **public** methods and variables of its superclass
  - if *Animal* has public methods *eat* and *sleep*, then *Reptiles*, *Mammals*, and *Fish* automatically have these methods too

• A subclass does not directly inherit the **private** variables* of its superclass
  - cannot directly access its superclass’s private variables by name (parent’s implementation is a black box even to a child!)
  - superclass can choose to define public accessor and mutator methods to set and get values of private instance variables – these **public** methods would be inherited by all subclasses, and be available to any other object also

* same goes for **private methods** which you’ll learn about soon!
Example 2: Brown Students

- Let’s say we’ve defined a class *BrownStudent*
- We will create a more specific subclass, *BrownCSStudent*, to model a particular type of *BrownStudent*
- Then we’ll define *CS15Student* as a subclass of *BrownCSStudent*
- Each class in our hierarchy will inherit from and further specialize its superclass

We’re omitting other types of *BrownStudent* and *BrownCSStudent* for now
Inheriting and Overriding **public** Methods

- In this example, we’ll look at the inheritance of **public** methods
- This is how the typical **BrownStudent** implements the method **study**
- When told to **study**, a **BrownStudent** will go to the SciLi, open a book, read, check Facebook, and then take a nap

```java
public class BrownStudent {
  /* Constructor elided */
  public void study() {
    this.goToSciLi();
    this.openBook();
    this.read();
    this.checkFacebook();
    this.takeNap();
  }
  /* other methods elided */
}
```
public class BrownCSStudent {

- We’re going to define the BrownCSStudent class as a subclass of BrownStudent
- How do we specify that we want BrownStudent to be our superclass?
}
public class BrownCSStudent extends BrownStudent {

- Specify superclass in class declaration with the `extends` keyword
- The `extends` keyword means “is a subclass of”
- Whenever you create a class that inherits from a superclass, must include “`extends <superclass name>`” in class declaration
  - `extends` is just another way to say “inherits from”
As it stands now, must a BrownCSStudent have any capabilities (other than being constructed)?

- **Yes.** Since it extends BrownStudent, BrownCSStudent automatically inherits all its public methods (and we don’t have to redefine them!!!)

- We can call study on a BrownCSStudent, and she will study exactly the same way a BrownStudent does.
public class BrownCSStudent extends BrownStudent {

    public BrownCSStudent() {
        // constructor code goes here!
    }

    // Doesn’t really make sense for a BrownCSStudent to study the same way all BrownStudents do...

    BrownCSStudents study by going to the Sunlab, logging in, etc.-- typically don’t go to the SciLi and read!

    BrownCSStudent class should override its superclass’s study method

    Overriding a superclass’s method means redefining it in the subclass
public class BrownCSStudent extends BrownStudent {

    public BrownCSStudent() {
        // constructor code goes here!
    }

    @Override
    public void study() {
        this.goToSunlab();
        this.logIn();
        this.reviewLectureSlides();
    }
}

- We include `@Override` right before we declare the method that we mean to override
- `@Override` is an annotation—signals to compiler (and to anyone reading your code) that you’re overriding a method of the superclass
- Annotations, like comments, have no effect on how your code behaves at runtime
public class BrownCSStudent extends BrownStudent {

    public BrownCSStudent() {
        // constructor code goes here!
    }

    @Override
    public void study() {
        this.goToSunlab();
        this.logIn();
        this.reviewLectureSlides();
    }
}

- Here’s where we re-declare method we want to override
- Be careful-- method signature (name of method, and list of parameters) must match that of the superclass’s method exactly!
  - Or else Java will create a new, additional method instead of overriding
- study() is the method signature, indicating that name of method is study and it takes in no parameters
public class BrownCSStudent extends BrownStudent {
    public BrownCSStudent() {
        // constructor code goes here!
    }

    @Override
    public void study() {
        this.goToSunlab();
        this.logIn();
        this.reviewLectureSlides();
    }
}

- Using @Override before each overridden method is not mandatory, but it’s good practice!
- Helps catch errors at compile-time
- If you include @Override but make a typo in the method signature, compiler will warn you that it can’t find matching method in the superclass
public class BrownCSStudent extends BrownStudent {

    public BrownCSStudent() {
        // constructor code goes here!
    }

    @Override
    public void study() {
        this.goToSunlab();
        this.logIn();
        this.reviewLectureSlides();
    }
}

- Fill in body of method with whatever we want a BrownCSStudent to do when she is told to study
- In this case, we’re fully overriding the method
- When a BrownCSStudent is told to study, she will execute this code instead of the code in her superclass’s study method (Java does this automagically - stay tuned)
public class BrownCSStudent extends BrownStudent {

    public BrownCSStudent() {
        // constructor code goes here!
    }

    @Override
    public void study() {
        this.goToSunlab();
        this.logIn();
        this.reviewLectureSlides();
    }

    public void code() {
        // code to code code goes here
    }
}

- Can define additional methods in BrownCSStudent class as well -- methods specific to BrownCSStudents, but not to all BrownStudents
- We’ve defined code method so all instances of BrownCSStudent (or any class that inherits from BrownCSStudent) can code
Now we’ll define even lower-level class in inheritance hierarchy: `CS15Student`, which extends `BrownCSStudent`.

What methods does a `CS15Student` already have?

- `study` (originally defined in `BrownStudent` and overridden in `BrownCSStudent`)
- `code` (defined in `BrownCSStudent`)

```java
class CS15Student extends BrownCSStudent {
    /* Constructor elided */
}
```
• How does a **CS15Student** study?

• Kind of like **BrownCSStudents** study: go to the Sunlab, log in, review lecture slides…

• But a **CS15Student** should also **codeTetris**!

• To make **CS15Students** study the way **BrownCSStudents** do but then execute additional steps, need to **partially override** **study** method - partially accept your inheritance…

```java
public class CS15Student extends BrownCSStudent {
    /* Constructor elided */
}
```
• Just like previous example, use `@Override` to tell compiler we’re about to override a method

• Declare the `study` method, making sure that method signature matches that of superclass’s `study` method
• When a **CS15Student** studies, she first does what every **BrownCSStudent** does: goes to the sunlab, logs in, reviews lecture slides

• First thing to do in **CS15Student**’s **study** method is say “study as if I were just a **BrownCSStudent**, and nothing more”

• Keyword **super** used to invoke overridden method from parent: in this case, **study** as implemented in parent **BrownCSStudent**

```java
public class CS15Student extends BrownCSStudent {

    /* Constructor elided */

    @Override
    public void study() {
        super.study(); //super = parent
        this.codeTetris();
    }

    public void codeTetris() {
        // implementation elided
    }
}
```
After doing everything a BrownCSStudent does to study, the CS15Student needs to code Tetris!

In this example, the CS15Student **partially overrides** the BrownCSStudent’s study method: it studies the way its superclass does, then does something specialized.

```java
public class CS15Student extends BrownCSStudent {
    /* Constructor elided */

    @Override
    public void study() {
        super.study();
        this.codeTetris();
    }

    public void codeTetris() {
        // implementation elided
    }
}
```
• If we think our `CS15Student` should do a little extra studying, we can call `super.study()` multiple times

• Can also call methods of the superclass that aren’t the one we’re overriding: like `BrownCSStudent`’s `code` method (this example is rather contrived)
Example 3: Vehicles

- Let us return to our “Vehicle Example” from earlier
- Convertible, CS15Mobile, and Van are all subclasses of Vehicle
- Vehicle is their superclass
Indirectly Accessing **private** Instance Variables

```java
public class Vehicle {
    private Human _driver;
    public Vehicle() {
        _driver = new Human();
    }
    /* Other methods elided */
}
```

- Remember from earlier **SmartPhone** example that **private** variables are not directly inherited by subclasses
- Consider **Vehicle**'s **private** instance variable, **_driver**
- A subclass of **Vehicle**, like **Convertible**, cannot access **_driver** by name, has no knowledge of it
Defining Accessors and Mutators in Superclass

```java
public class Vehicle {

    private Human _driver;

    public Vehicle() {
        _driver = new Human();
    }

    public void move() {
        // implementation elided
    }

    public void setDriver(Human driver) {
        _driver = driver;
    }

    public Human getDriver() {
        return _driver;
    }
}
```

- If `Vehicle` does want its subclasses (and other classes) to be able to access and change the value of `_driver`, it can define public accessor and mutator methods.

- Important to consider these design decisions in your own programs— which properties will need to be accessible to other classes?
  - don’t always need both `set` and `get`
public class Convertible extends Vehicle {

    public Convertible() {
        // Constructor code
    }

    public void dragRace() {
        this.getDriver().stepOnIt();
    }
}

- Since subclasses automatically inherit these public accessor and mutator methods, `Convertible` can get a reference to `_driver` by calling `this.getDriver()`

- Note that using “double dot” we’ve chained two methods together
  - first, `getDriver` is called, and returns the driver
  - next, `stepOnIt` is called on that driver
Making Sure Superclass’s Instance Variables are Initialized

```java
public class Convertible extends Vehicle {

    private ConvertibleTop _top;

    public Convertible() {
        // how does _driver get initialized?
        _top = new ConvertibleTop();
    }

    public void dragRace() {
        this.getDriver().stepOnIt();
    }
}
```

- **Convertible** may declare its own instance variables, which it initializes in its constructor.
- Wait… when we instantiate a **Convertible**, how do superclass **Vehicle**’s private instance variables get initialized?
super(): Invoking Superclass’s Constructor

- Vehicle’s instance variables (like _driver) are initialized in Vehicle’s constructor
- To make sure that _driver is initialized whenever we instantiate a Convertible, we need to call the superclass’s constructor
- The syntax for doing this is “super()”
- Here super() is parent’s constructor; before it was the parent itself (verb vs. noun)
**super(): Invoking Superclass’s Constructor**

```
public class Convertible extends Vehicle {
    private ConvertibleTop _top;

    public Convertible() {
        super();
        _top = new ConvertibleTop();
    }

    public void dragRace() {
        this.getDriver().stepOnIt();
    }
}
```

- We call `super` from the subclass’s constructor to make sure the superclass’s instance variables are initialized properly.
- Can only make this call *once*, and it must be the **very first line** in the subclass’s constructor.
  - This rule doesn’t apply for calling any of superclass’s other methods.
super() : Invoking Superclass’s Constructor

public class Vehicle {

    private Human _driver;

    public Vehicle(Human driver) {
        _driver = driver;
    }

    public void setDriver(Human driver) {
        _driver = driver;
    }

    public Human getDriver() {
        return _driver;
    }
}

- What if the superclass’s constructor takes in a parameter?
- We’ve modified Vehicle’s constructor to take in a Human as a parameter, rather than instantiating one itself.
- After all, doesn’t really make sense for a car to create its driver— we should be able to create a Vehicle for a pre-existing Human to drive.
- How do we invoke this constructor correctly from the subclass?
super(): Invoking Superclass’s Constructor

public class Convertible extends Vehicle {
    private ConvertibleTop _top;

    public Convertible(Human driver) {
        super(driver);
        _top = new ConvertibleTop();
    }

    public void dragRace() {
        this.getDriver().stepOnIt();
    }
}
What if we don’t call super()?

public class Convertible extends Vehicle {
    private ConvertibleTop _top;

    public Convertible(Human driver) {
        // oops! We forgot to call super
        _top = new ConvertibleTop();
    }

    public void dragRace()
    {
        this.getDriver().stepOnIt();
    }
}

- What if we forget to call super()?
- If you don’t explicitly call super() first thing in your constructor, Java automatically calls it for you, passing in no arguments.
- But if superclass’s constructor requires a parameter, you’ll get an error!
- In this case, we get a compiler error saying that there is no constructor “public Vehicle()” since it was declared with a parameter.
super: The Rules

public class Convertible extends Vehicle {

    private ConvertibleTop _top;

    public Convertible(Human driver) {
        super(driver);
        _top = new ConvertibleTop();
    }

    public void dragRace() {
        this.getDriver().stepOnIt();
    }
}

- When calling superclass’s constructor in subclass’s constructor:
  - can only call super() once!
  - must be first

- When calling any other methods of superclass (for example, when partially overriding methods) these rules don’t apply
  - call super.<method>() however many times you want, from wherever you want
abstract Methods and abstract Classes

- Every Vehicle should know how to move
- Makes sense to declare a move method in the Vehicle class
- But how would we fill it in? Vehicles are varied enough that each subclass would need to fully override the method--airplanes and bicycles are both vehicles, but move completely differently
**abstract Methods and abstract Classes**

- We declare a method **abstract** in a superclass when the subclasses can’t really re-use any implementation it might provide.

- In this case, we know that all **Vehicles** should **move**, but each subclass will move very differently.

- **abstract** method is declared in superclass, but not defined-- up to subclasses farther down hierarchy to provide their own implementations.
abstract Methods and abstract Classes

public abstract class Vehicle {
    private Human _driver;

    public Vehicle(Human driver) {
        _driver = driver;
    }

    public abstract void move();

    /* Other methods elided */
}

- Here, we’ve modified Vehicle to make it an abstract class: a class with at least one abstract method.
- We declare both Vehicle and its move method abstract: if one of a class’s methods is abstract, class itself must also be declared abstract.
- An abstract method is only declared by the superclass, not implemented—use semicolon after declaration instead of curly braces.
public class Convertible extends Vehicle {
    /* Other methods elided */

    @Override
    public void move() {
        // code to move super fast!
    }
}

public class Van extends Vehicle {
    /* Other methods elided */

    @Override
    public void move() {
        // code to putter along slowly
    }
}

public class CS15Mobile extends Vehicle {
    /* Other methods elided */

    @Override
    public void move() {
        // code to move at moderate speed
    }
}

● All concrete subclasses of Vehicle override by providing a concrete implementation for Vehicle’s abstract move() method

● As usual, method signature must match the one that Vehicle declared
**abstract Methods and abstract Classes**

- **abstract** classes cannot be instantiated!
  
  - this makes sense—shouldn’t be able to just instantiate a 
    generic `Vehicle`, since it has no code to `move()`
  
  - instead, provide implementation of `move()` in concrete 
    subclass, and instantiate subclass

- Subclass at any level in inheritance hierarchy can make 
  **abstract** method concrete by providing implementation

- Even though **abstract** class can’t be instantiated, its 
  constructor **must** still be invoked via `super()` by a subclass 
  (thus, it can still initialize its own instance variables)
Under the Hood: Method Resolution

- When we call `move` on some instance of `Van`, or `study` on some instance of `CS15Student`, how does Java know which version of the method to call?
- Starts by looking at the instance’s class, regardless of where class is in the inheritance hierarchy
  - If method is defined in the instance’s class, Java calls it
  - Otherwise, it checks the superclass
    - If method is explicitly defined in superclass, Java calls it
    - Otherwise, checks the superclass up one level…etc.
Under the Hood: Method Resolution

- Essentially, Java “walks up the class inheritance tree” from subclass to superclass until it either:
  - finds the method, and calls it
  - doesn’t find the method, and generates a compile-time error. You can’t send a message for which there is no method!
Under the Hood: Method Resolution

- This process is called method resolution
  - When we call `study()` on a `CS15Student` instance, `CS15Student`'s implementation of `study()` is called
  - When we call `code()` on a `CS15Student`, `BrownCSStudent`'s implementation of `code()` is called
That’s Inheritance!

- Things you might find in an inheritance hierarchy:
  - superclass specifies a method, subclasses inherit it
  - superclass defines general methods common to all subclasses, each subclass adds its own specialized methods
  - superclass declares an abstract method, each subclass must implement it – until a sub...subclass does, it can’t be instantiated
  - superclass specifies a method, and subclasses choose to completely override it
  - superclass specifies a method, and subclasses choose to partially override it, thus making use of parent’s code but augmenting it
Announcements

• Homework 2 and LiteBrite are released today

• HW2 is due on Sunday 2PM
  o Remember: homeworks only have one handin time

• LiteBrite early hand-in is **Tuesday 09/29**, on-time handin is **Thursday 10/1**, late handin is **Saturday 10/3**

• There will be a LiteBrite Help Session **Sunday 6-8pm** in **Salomon 001**. We strongly encourage you to attend

• **START EARLY, START TODAY, START YESTERDAY!!!!!**

• **PLEASE FILL OUT THE INITIAL SURVEY**