Extending Existing Classes

Using Inheritance

Example: Traffic Simulation

- Various vehicles move around and we can model the interactions and look for "problem" situations
- For my simulation I need
  - Cars, Trucks, Motorcycles, Vans, SUVs, Mopeds, etc.
  - Notice similarities among these
- These are all Vehicles
- Vehicle is something that
  - it has a motor that consumes fuel
  - moves north, east, south, or west (if it has fuel)
  - They have weight, number of wheels, etc

Vehicle Class

- Base class for my simulation
  - "Various types of vehicles…"
  - but different types behave differently…
- Option A (not good)
  - Vehicle has a type attribute
  - Each method in Vehicle has an "IF" statement that changes actions depending on the type
  - Lots of code
  - What about adding a new Vehicle type? (e.g. golf cart)

Vehicle Class - Use Inheritance

- The similar, but different pattern
- The "IS-A" relationship
  - a Car is a Vehicle…with 4 wheels, and medium weight
  - a Truck is a Vehicle…with 6+ wheels, and heavy weight
  - a Motorcycle is a Vehicle…with 2 wheels, and light weight
- Use inheritance
  - to allow multiple classes to 'share' some attributes
  - but allows each class to differentiate itself as needed

Object Class

- Every class inherits from Object *(every class **is-a** Object)*
  - You don't have to explicitly declare it
- It is built-in to Java
- "Cosmic Superclass"
  - Everything is an object (eventually)
- Creating our own Classes (Card or Dice)
- Creating our own sub-classes

GDie

- GDie is a special type of Die
  - 6 - sided green or red graphical view of a Die
- GDie inherits…
  - All the basic functionality of a Die
- GDie must add… (*i.e. "extend"*)
  - All the graphical functionality

GDie

```
public class GDie extends Die {
        …
}
```

- Constructors
  - public GDie (int s, int v, boolean down){
    - Since a GDie is-a Die…
      - The Die constructor is called first (implicitly)…
      - …or explicitly. *But must be called as first line in constructor!*
- Use the keyword **super**
  - To access/call the superclass's constructor explicitly

FilledRect example

- FilledRect is a special type of Grect
  - It always has a fill color and is filled
- Constructor
  - First must create a Grect – in order to fill it

```
public FilledRect (double x, double y, double w, double h, Color c){
        super(x,y,w,h);
        setFilled(true);
        setColor ( c );
}
public FilledRect (double x, double y, double w, double h){
        super (x,y,w,h);
        setFilled (true);
        setColor (DEFAULT_COLOR);
}
public FilledRect (double x, double y, double w, double h){
        this (x,y,w,h,DEFAULT_COLOR);
}
```

```
public FilledRect ( ) {
    this (10,20,30,40,DEFAULT_COLOR);
}
```

What about a DEFAULT CONSTRUCTOR?

Inherited Constructors

- Every constructor in hierarchy will be called in order
  - For deeper inheritance hierarchies, this could be many calls
  - Will call default constructor implicitly (i.e. if you don't)
  - **Explicit** calls require: super(…); statement
- Default constructors are allowed
  - Java will create an empty default constructor if needed
  - But not if any other constructor exists
- ***If*** there is no default constructor
  - Subclasses *MUST* use super(…) with parameters

Inherited Methods

- What if an object calls a method that is defined in that class AND in superclasses?
  - Rule: execute the one that is closest in the hierarchy
- So, a method called on a FilledRect object will…
  - Look in FilledRect for method with same name and parameter structure
  - Then, in GRect
  - Then, in GObject
  - Then, in Object

Summary

- Inheritance
  - One class "inherits" the properties of another
  - Description includes the "IS A" relationship
    - *(if not, then probably don't use inheritance)*
  - Super and Sub-class
    - ancestry.com
    - sub classes add/change attributes or methods
    - reduces redundant code (eliminating re-writes!)

Cannon Shot