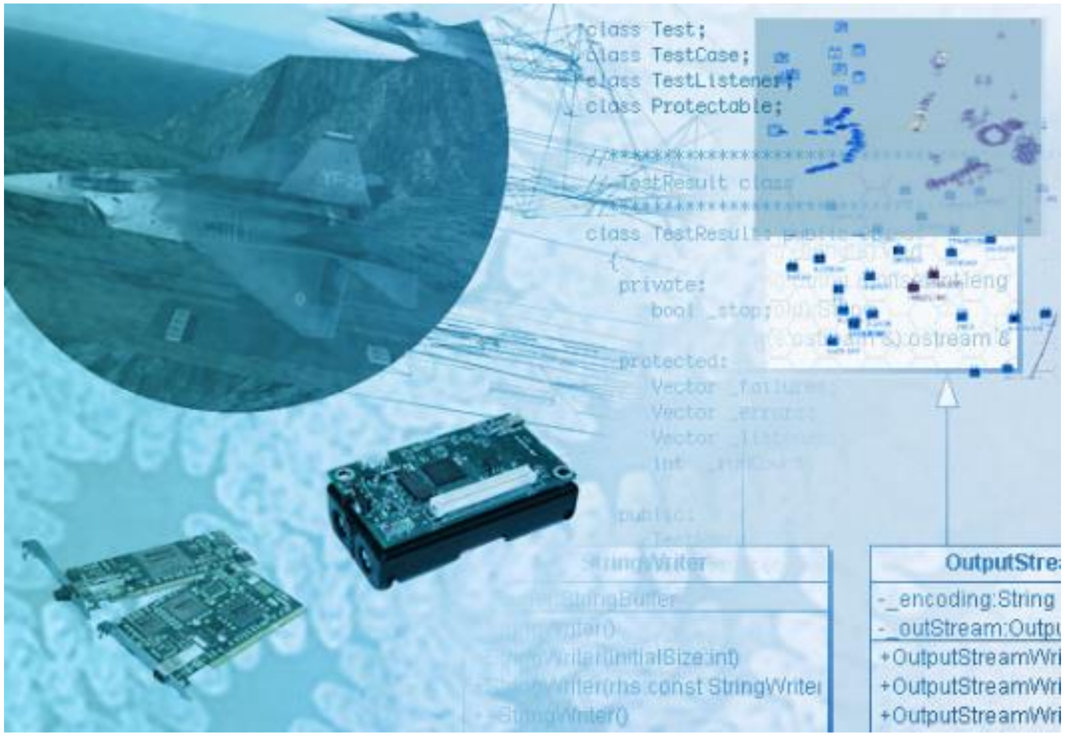
CSYE 6200

CONCEPTS OF OBJECT-ORIENTED DESIGN

FALL 2015 – WEEK 5



MARK G. MUNSON

ADMINISTRATION

- **Assignment #2b due Today**
  - Code and sample output in Blackboard

- **Assignment #3 will be due next week (October 11/12)**

- **Quiz (October 18/19)**
- **Mid-Term Exam on week 8 (October 25)**

THE LECTURE

- **Recap**
- **Diagraming with UML**
- **Packages – Organizing your code**
- **Errors and Error Handling**
  - Exceptions
  - Throw & Throws
  - Try-Catch
- **FileWriter Introduction**
- **Note: Interfaces and Abstract classes shifted to next week**

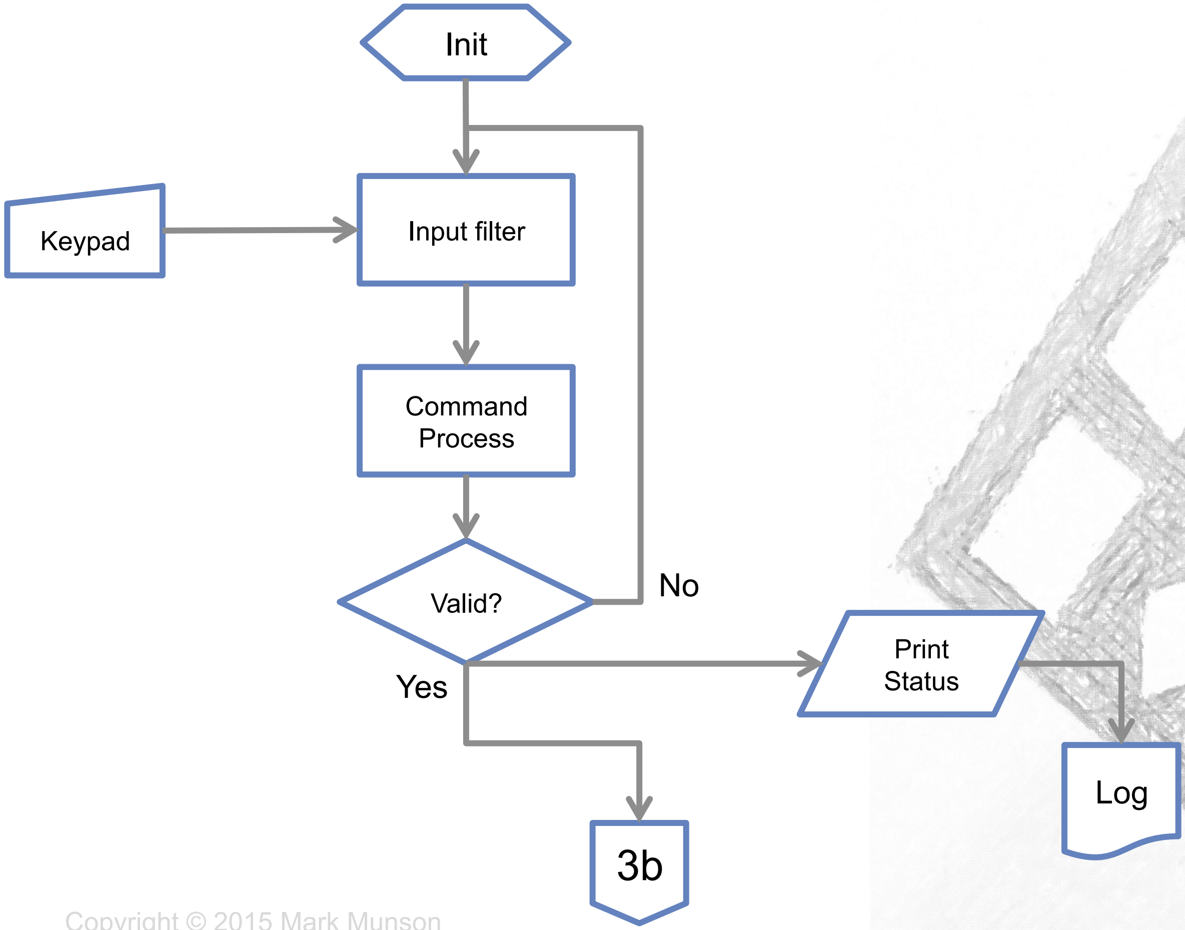
RECAP

DIAGRAMING WITH THE UNIFIED MODELING LANGUAGE

UML

ACTIVITY DIAGRAM

In the early days, program logic was drawn using a flowchart:

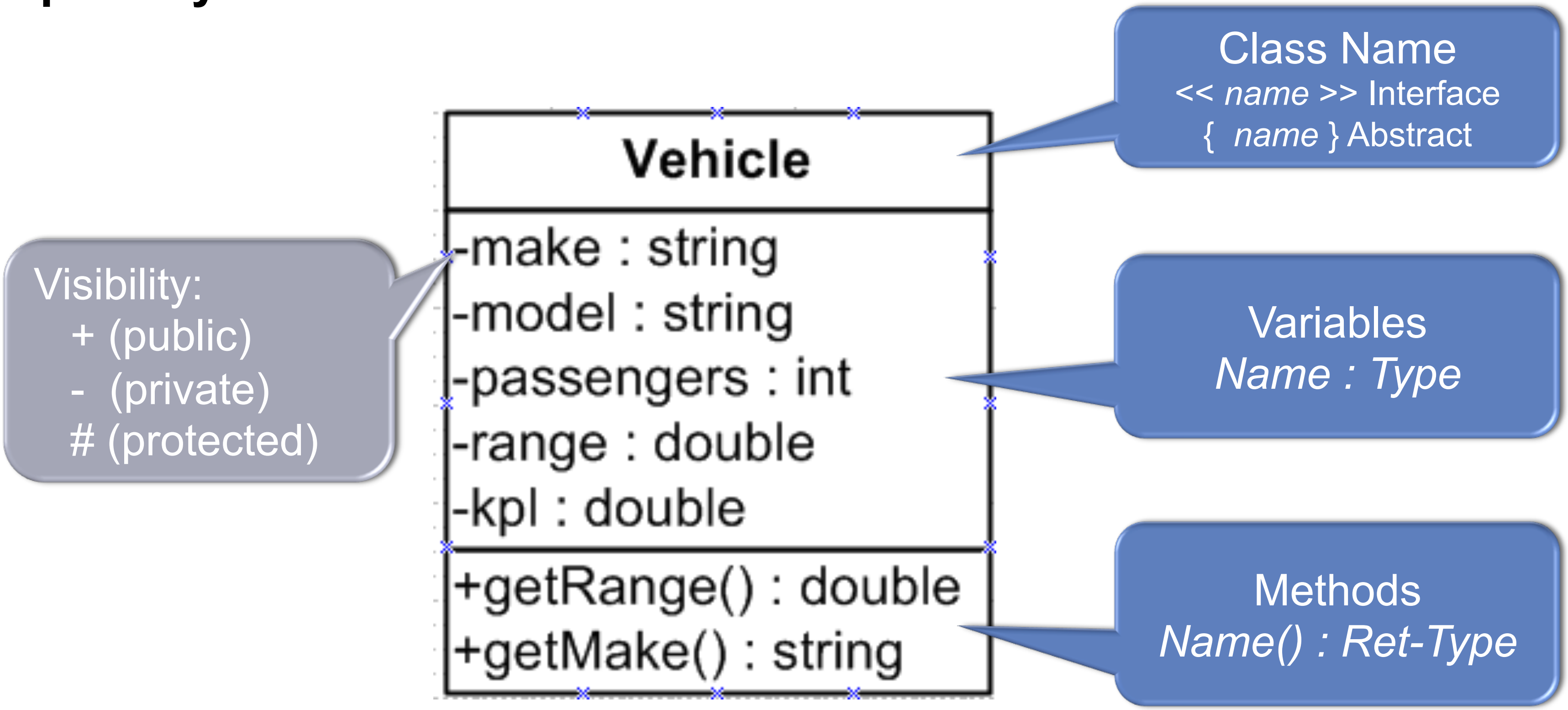


UML OVERVIEW

- **Reason for notation – Specifying, Documenting, and Visualizing**
- **Early 90's: Proliferation of standards (Booch, Jacobson, Rumbaugh, etc.)**
- **~1995: Consolidation into a single standard**
  - Unified Modeling Language (UML)
    - www.uml.org
  - Object Management Group (OMG)
    - www.omg.org
- **Multiple diagram types**
  - Activity diagrams
  - Use Cases (text and drawn)
  - Class diagram (content and relationships)
  - Sequence diagrams
  - Component diagrams

UML CLASS DIAGRAM

A class may be expressed as a UML diagram that shows the primary variables and method calls



Visibility:
+ (public)
- (private)
# (protected)

Vehicle

-make : string
-model : string
-passengers : int
-range : double
-kpl : double

+getRange() : double
+getMake() : string

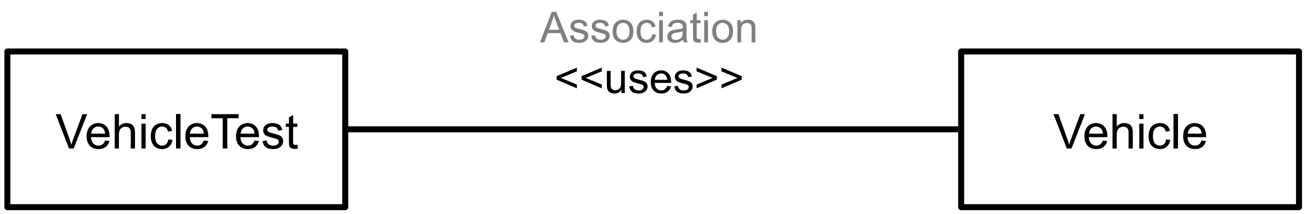
Class Name
<< *name* >> Interface
{ *name* } Abstract

Variables
Name : Type

Methods
Name() : Ret-Type

UML CLASS DIAGRAM

- Class diagrams may be drawn to show the relationships between classes (a static view)
- Inner class detail is often omitted to stress the class interactions and dependencies.

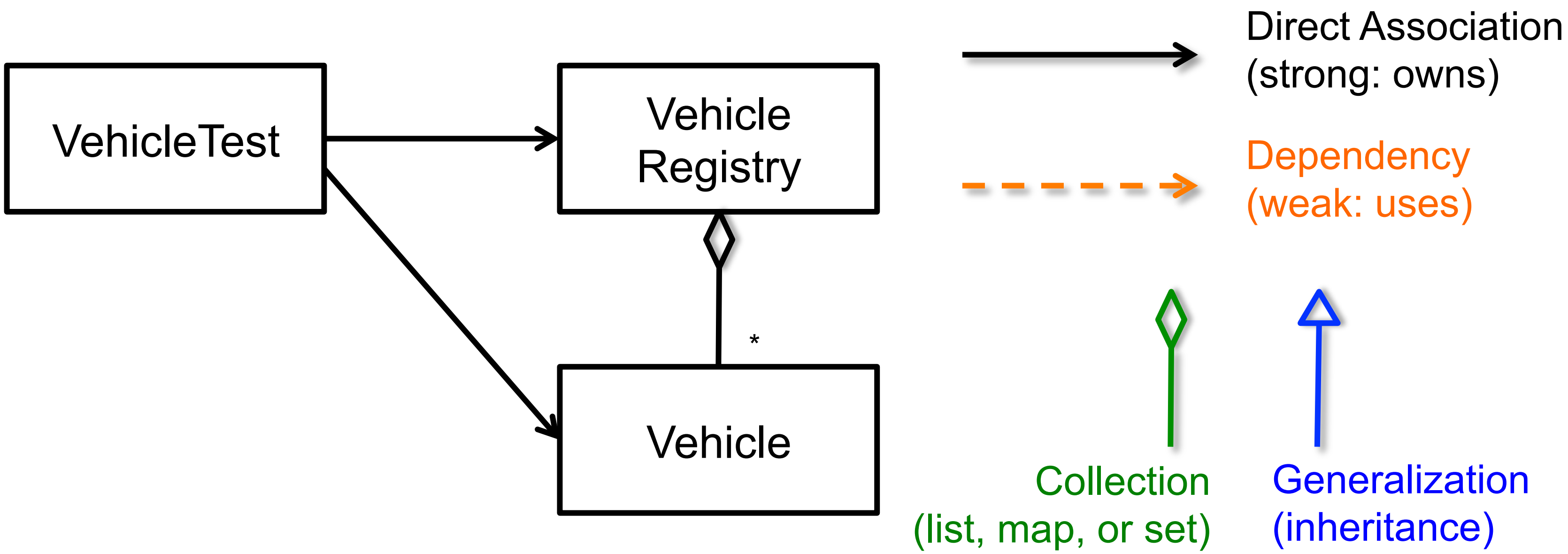


RELATIONSHIPS

- **Association**
  - Aggregation – A class owns other objects as instance variables
  - This relationship is referred to as “Has-A”
- **Generalization**
  - Inheritance – A class inherits variables/methods from a parent class
  - Invoked in Java using the **`extends`** or **`implements`** keywords
  - This relationship is referred to as “Is-A”

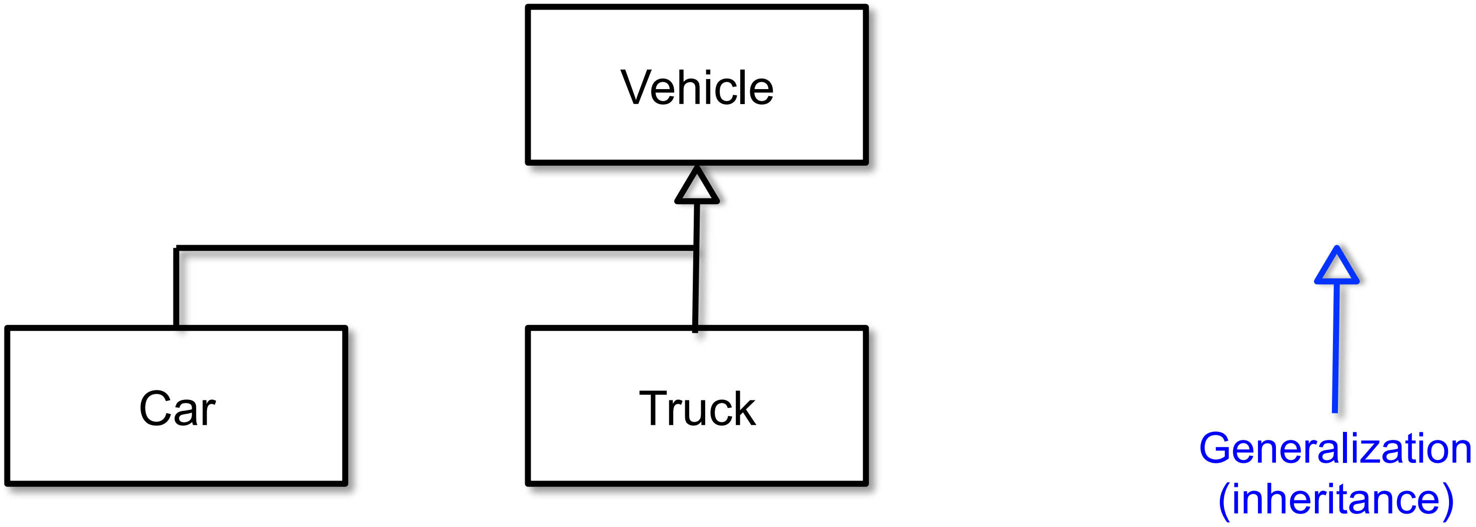
UML CLASS DIAGRAM

- **For assignment 2b, our class structure could be drawn as:**


VehicleTest
Vehicle Registry
Vehicle
*
Direct Association (strong: owns)
Dependency (weak: uses)
Collection (list, map, or set)
Generalization (inheritance)

UML CLASS DIAGRAM

- **When inheriting, each inherited class “Is-A” instance of the parent class**



Vehicle

Car

Truck

Generalization
(inheritance)

UML DIAGRAM DEMO

ORGANIZING YOUR CLASSES

JAVA PACKAGES

PACKAGES

- **To organize and group related software, each class is placed in a `package`.**
  - All classes belong to a package
  - If no package is specified, then the default (global) package is used
  - Java uses the filesystem to manage packages

```
project/src/
            assign2/
                    Vehicle.java
                    VehicleRegistry.java
                    VehicleTest.java
```

PACKAGES

```
project/src/
            assign2/
                    Vehicle.java
                    VehicleRegistry.java
                    VehicleTest.java
```

Package ‘assign2’

- **The `package` statement is placed at the start of each .java file:**

```
package assign2;

class Vehicle {
   public int passengers;
   private double kpl;
…
   public double getKpl() {   // A “getter” method
      return kpl;
   }
…
}
```

PACKAGES (CONT.)

- **After compiling with `javac`, any java source files with a package definition of ‘assign2’ will have its `.class` file placed in a corresponding subdirectory**

```
project/src/
             assign2/
                      Vehicle.java
                      Vehicle.class
                      VehicleRegistry.java
                      VehicleRegistry.class
                      VehicleTest.java
                      VehicleTest.class
```

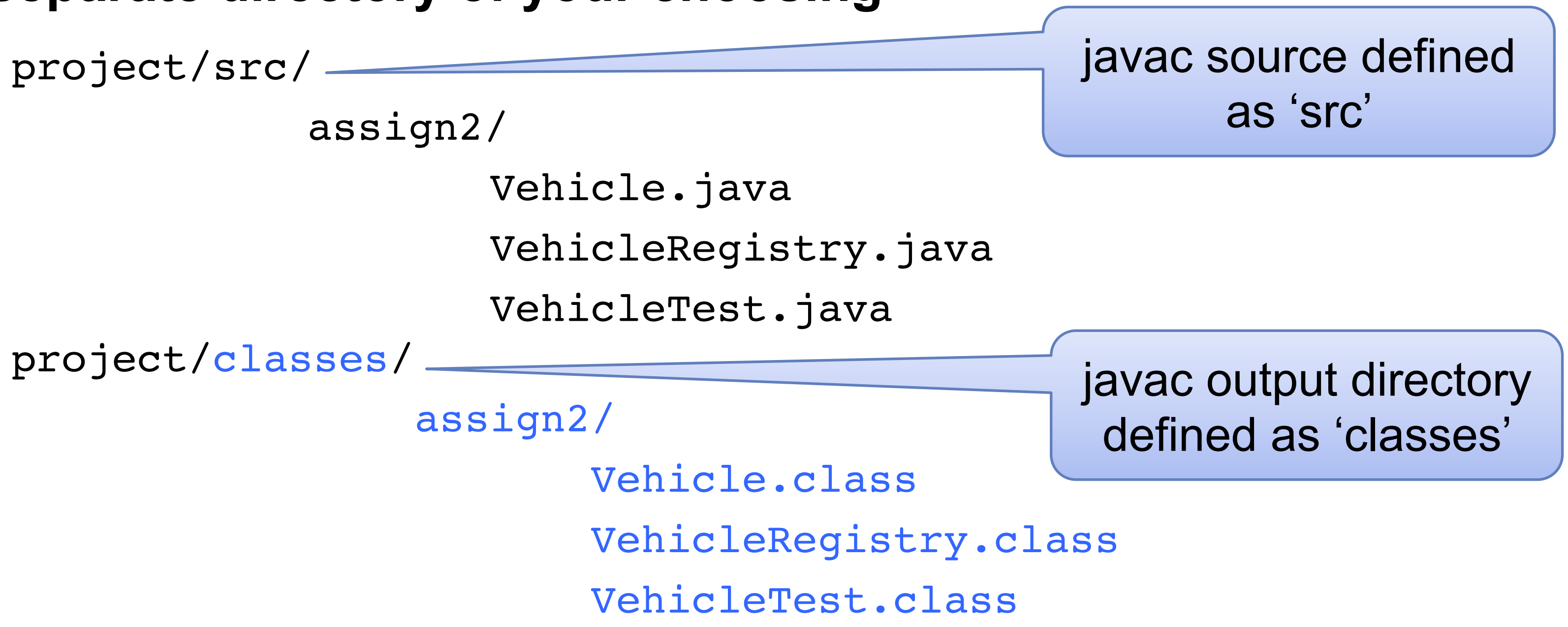
Run `javac` from here

- **To run with a package, just use the full *package.Classname* as the target entry point**

```
java assign2.VehicleTest
```

PACKAGES (CONT.)

- **Comingling your source and compiled .class files is inconvenient, so most IDE's will place the .class files in a separate directory of your choosing**



```
project/src/
            assign2/
                    Vehicle.java
                    VehicleRegistry.java
                    VehicleTest.java
project/classes/
                assign2/
                        Vehicle.class
                        VehicleRegistry.class
                        VehicleTest.class
```

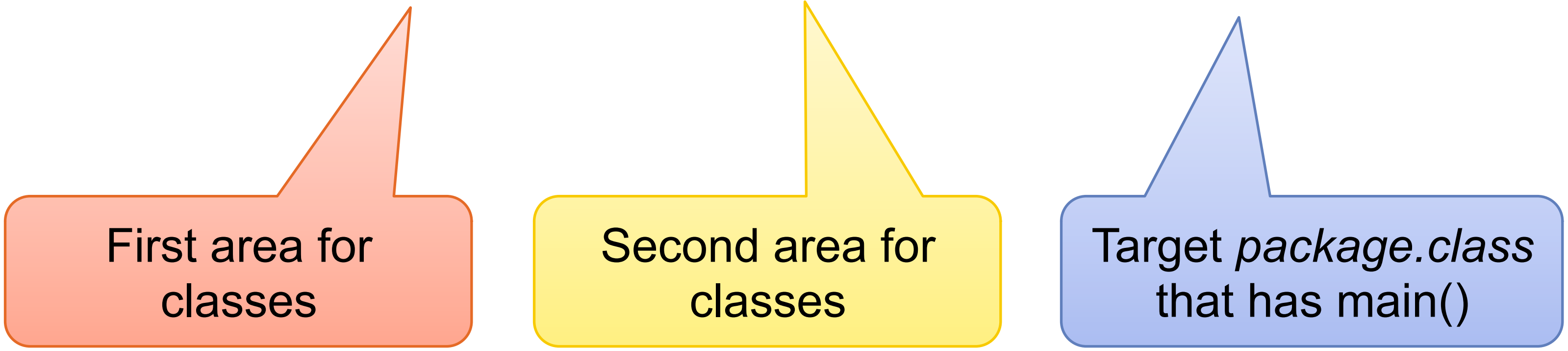
javac source defined as 'src'

javac output directory defined as 'classes'

CLASSPATH

- Java uses the CLASSPATH environment variable to locate where .class files reside
- The CLASSPATH variable may be set as part of the java command

```
> java -cp “/proj1/classes;/proj2/classes” assign1.Vehicle
```



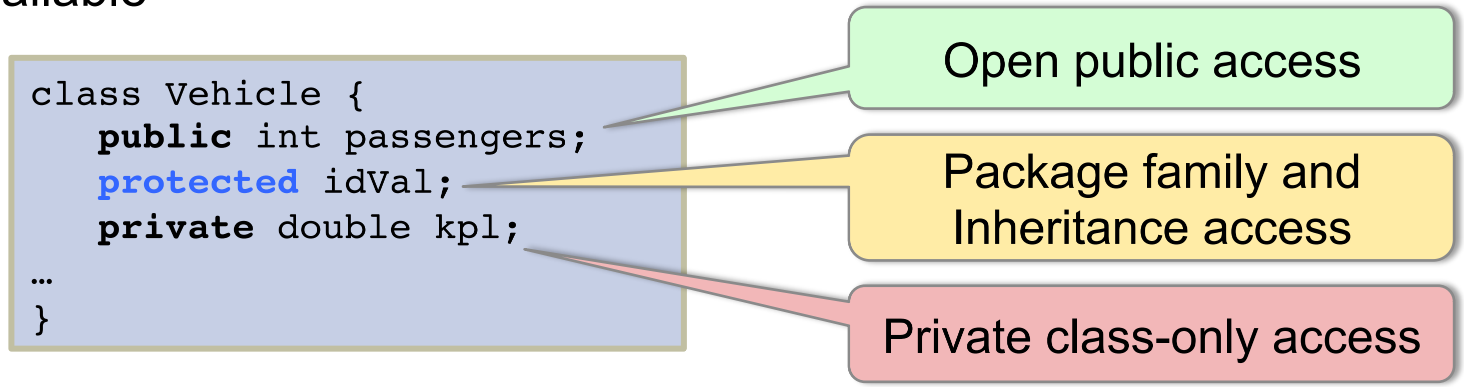
First area for classes

Second area for classes

Target *package.class* that has main()

PROTECTED

- **With Java, membership in a package has special benefits**
  - Each class in a package is generally aware of the others, and doesn't need to search to find them (no 'import' needed)
  - In addition to public/private, a third option, **`protected`**, is available



```
class Vehicle {
   public int passengers;
   protected idVal;
   private double kpl;
…
}
```

Open public access

Package family and Inheritance access

Private class-only access

- **In Java, both inherited class and others classes in a package have permission to access `protected` variables and methods.**

IMPORT (CONT.)

- The `import` statement has the form

```
     import package.classname;
```

- Each class that is used from another package must be defined on an import statement
- If most or all classes from another package need to be imported, you may use an asterisk for the class name

```
     import package.*;
```

VERSION CONTROL

VERSION CONTROL

- **Git – Distributed version control**
  - **Offline support:** Each developer has their own repository
    - A local commit may be made, even if internet access isn't available
  - **Atomic:** Commits are handled as full transactions for an entire development tree
  - **Flexible:** Workflow support allows you to use the tool in the way that you want
  - **Clean:** Git only creates a single hidden folder (.git), instead of creating lots of hidden artifacts
  - **Branches:** Code branches are lightweight, instead of cloning the entire codebase

GIT
CREATING A LOCAL REPOSITORY

Create a build area (i.e. /proj/CSYE6200)

```
>mkdir CSYE6200
```

Change directories into your build area

```
>cd CSYE6200
```

Create a bare repository

```
> git init —bare
```

Mark files for addition

```
> touch src
> git add src
```

Commit changes to your local repository

```
> git commit —m “Initial source commit”
```

GIT
REMOTE REPOSITORY

Add a remote connection

```
> git remote add origin git@localhost:GitRepos/CSYE6200
```

Commit your changes

```
> git commit -a –m “your commit comment”
```

Push your changes to the remote repository

```
> git push origin master
```

Pull changes by others from the remote repository

```
> git pull origin master
```

SCOPE { }

SCOPE { } REVISITED

- Although presented as a collection of statements, scope ({ … }) carries special meaning with regard to variable visibility
- Any time you cross into a new scope, you are effectively creating a new variable space

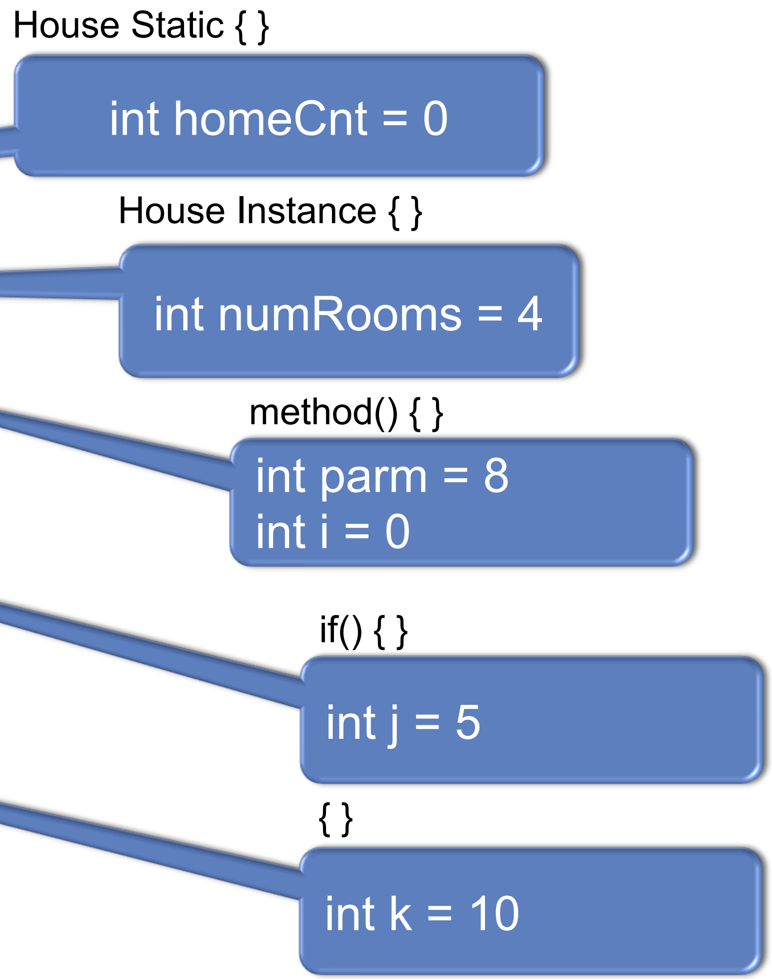
```
class { … }
  void method ( )  {  … }
    if ( ) { … }
    { … }
```

- Variables created within a scope, go away when the scope ends

SCOPE { } REVISITED

- Any time you cross into a new scope ({ … }), you are creating a new variable space

```
class House {
   static int homeCnt = 0;
   int numRooms = 4;
   void method (int parm )  {
      int i = 0;
      if (true) {
         int j = 5;
      }
      {
         int k = 10;
      }
      j = 5; // ERROR
      k = 9; // ERROR
   }  // method
} // class
```



House Static { }

int homeCnt = 0

House Instance { }

int numRooms = 4

method() { }

int parm = 8
int i = 0

if() { }

int j = 5

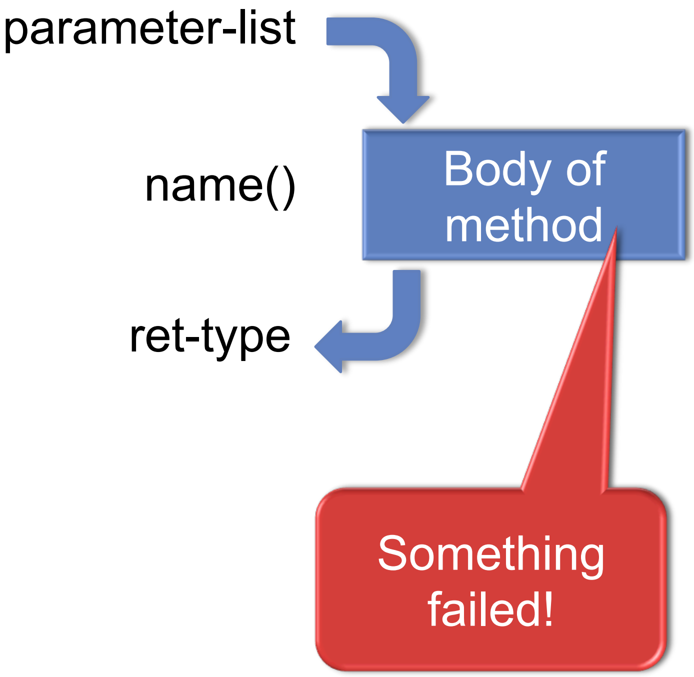
{ }

int k = 10

EXCEPTIONS

DEALING WITH ERRORS

- **Error handling has always been a vexing problem**
- **Early attempts to deal with it**
  - Global error variable – requires that you check it often
  - Use ret-type to flag an error – then check a global value to find the type
  - Implement an error method call that checks for a recorded error
- **With Java, there is a better way…**



ERRORS INTO EXCEPTIONS

- In Java each instance of an error is converted into a class called an Exception

```
Exception ex01 = new Exception();
```

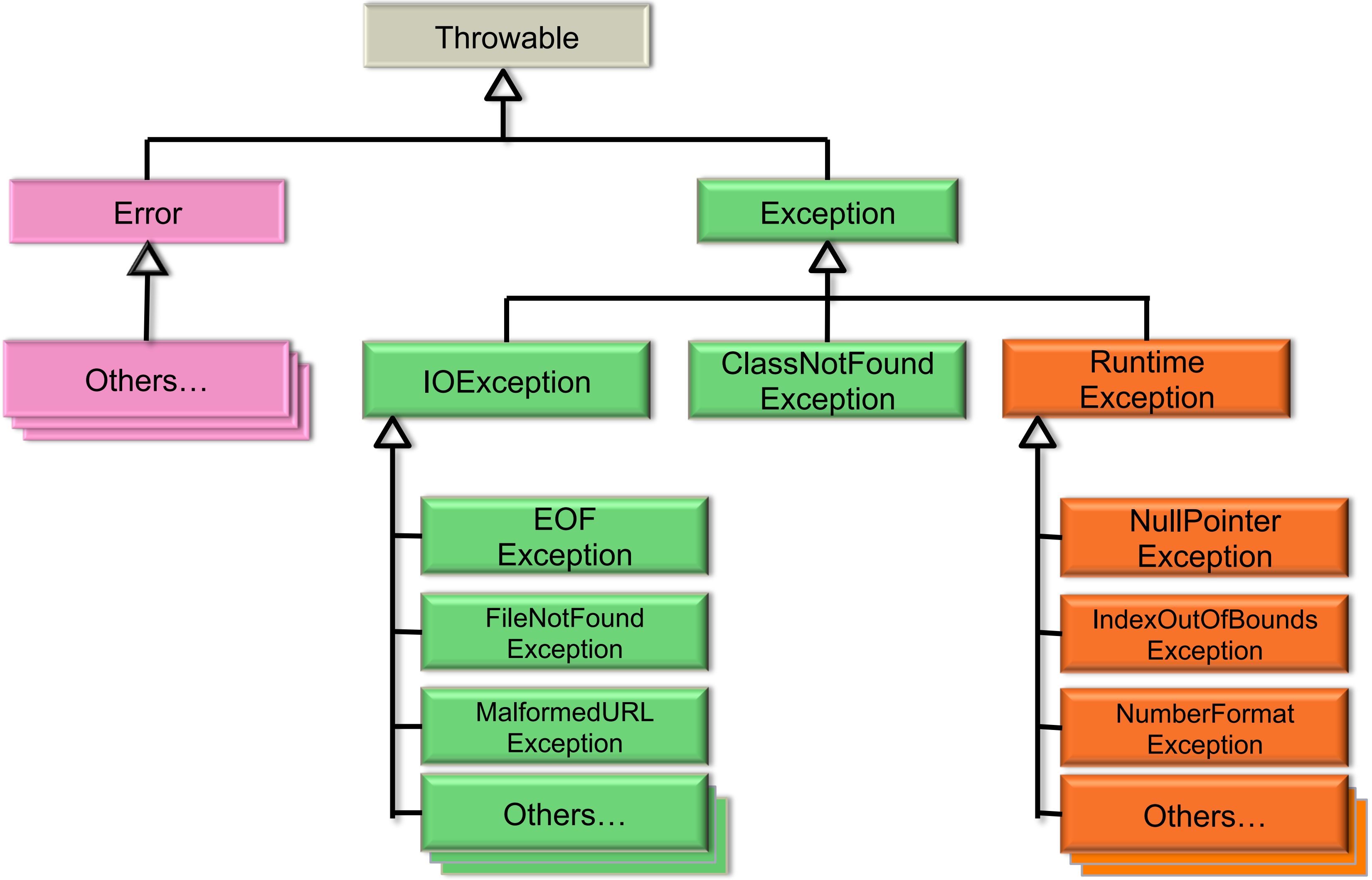
- Exceptions secretly record where you are, and how you arrived there
- Exceptions are organized into a hierarchy of classes, so you can pick one that describes the type of error

```
IOException ex01 = new IOException();
```

- Each Exception allows for a detailed error message

```
IOException ex1 =
  new IOException(“Houston, we have a problem”);
```

EXCEPTION HIERARCHY



THROWING AN EXCEPTION

Within a method, an Exception is cast using the throw statement

Any methods that aren't handled locally, are flagged using the throws statement

```
private void setName(String name)
  throws IllegalArgumentException {
   if (name.length() == 0)
     throw new IllegalArgumentException("No name");
   …
```

TRY-CATCH

To handle an exception yourself, or to handle one thrown by a routine you called, implement the try-catch block

```
try {
    … statement(s) under test
}
catch (exception-class varName) {
    … actions to take
}
```

TRY-CATCH EXAMPLE

```
public void testExcep() {
  try {
     // Let’s make an error (Exception) and throw it
     throw new Exception(“This is an error!”);
  }
  catch (Exception ex) {
     ex.printStackTrace(); // just print the trace
  }
  finally {
    System.err.println(“We caught an error… finally”);
  }
}
```

TRY-CATCH-FINALLY

To handle an exception yourself, or to handle one thrown by a routine you called, implement the try-catch block

```
try {
     … statement(s) under test
}
catch (exception-type1 varName) {
     … actions to take if type1 error is caught
}
catch (exception-type2 varName) {
     … actions to take if type2 error is caught
}
finally {
     … final actions to take if any error is received
}
```

FILE I/O INTRO

FILEWRITER

A simple way to write files to disk is to create a `FileWriter` instance

```
import java.io.*;
…
public writeDataToFile(String filename) {
   FileWriter fw;
   try {
      fw = new FileWriter(filename);
      fw.write(“first line of text”);
      fw.close();
   } catch(IOException ex) {
      System.err.println(“IO ERROR received: “ + ex.getMessage());
      ex.printStackTrace();
   }
}
```

Imports FileWriter and IOException classes

FILEWRITER (CONT.)

An open FileWriter may be passed to a method()

```
…
public void writeVehicleData(FileWriter fw, Vehicle veh)
    throws IOException {
    fw.write(“Line of text”);
    fw.write(veh.getMake());
    …
    }
```

FILEWRITER DEMO

FILEREADER

A simple way to read the contents of a file is to create a `FileReader` instance

```
import java.io.*;
…
public void readDataFromFile(String filename) {
   try {
      FileReader fr = new FileReader(filename);
      BufferedReader in = new BufferedReader(fr);
      String str;
      while ((str = in.readLine()) != null) {
         System.out.println("> " + str);
      }
      in.close();
   } catch (FileNotFoundException  | IOException e) {
      e.printStackTrace();
   }
}
```

Imports FileReader and IOException classes

NEXT WEEK / ASSIGNMENT #3

JABG: Read

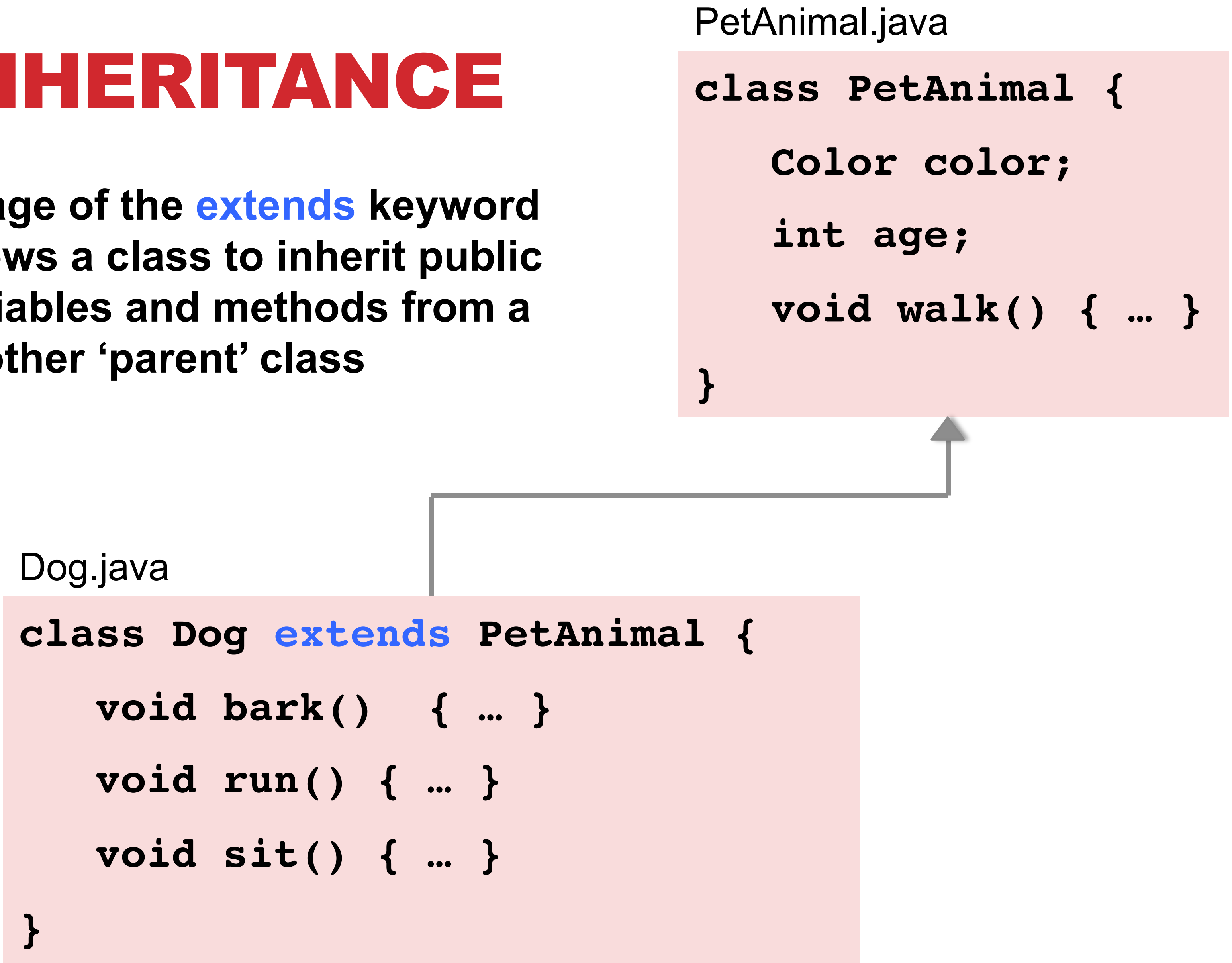
- Ch. 7 Abstract classes p. 259-262
- Ch. 8 Interfaces p. 278-298
- Ch. 10 Using I/O

- **Assignment: Due Feb. Oct 11th/12th, 6:00 pm (prior to class) – 20 pts.**
  - Create a TruckVehicle class, and use inheritance to extend from your Vehicle class
    - Add member variables to this class that track the height, width, and length of the truck cargo bed (select and document your units of measure).
    - Add a new public method in TruckVehicle that calculates the cargo area
    - Add an attractive print routine that leverages the Vehicle print routine from 2b, but adds on information about the cargo area (hint: use ‘super’ to access parent class methods with the same method name)
  - Create a RegistryIO class: This class will permit vehicle registry information to be stored (and possibly retrieved) from disk
    - Add public methods to load() and save() a VehicleRegistry. For both methods, pass in a VehicleRegistry and a filename as parameters. Implement the save() method functionality that writes data to a file.
    - Create a private ‘save’ method for writing a single vehicle to an open file
    - For all IO operations, use a try-catch() block to capture and print appropriate error messages
  - BONUS (+5 pts) Implement the load() method functionality and read the contents written by your save() method back into a new Vehicle class.
  - Upload your .java files to Blackboard. You should zip up the entire `src` directory (use archive folder on OSX). Include a copy of your program’s output captured in a text file.

EXTRAS

INHERITANCE

Usage of the extends keyword allows a class to inherit public variables and methods from a another ‘parent’ class



PetAnimal.java

```
class PetAnimal {
    Color color;
    int age;
    void walk() { … }
}
```

Dog.java

```
class Dog extends PetAnimal {
    void bark()  { … }
    void run() { … }
    void sit() { … }
}
```