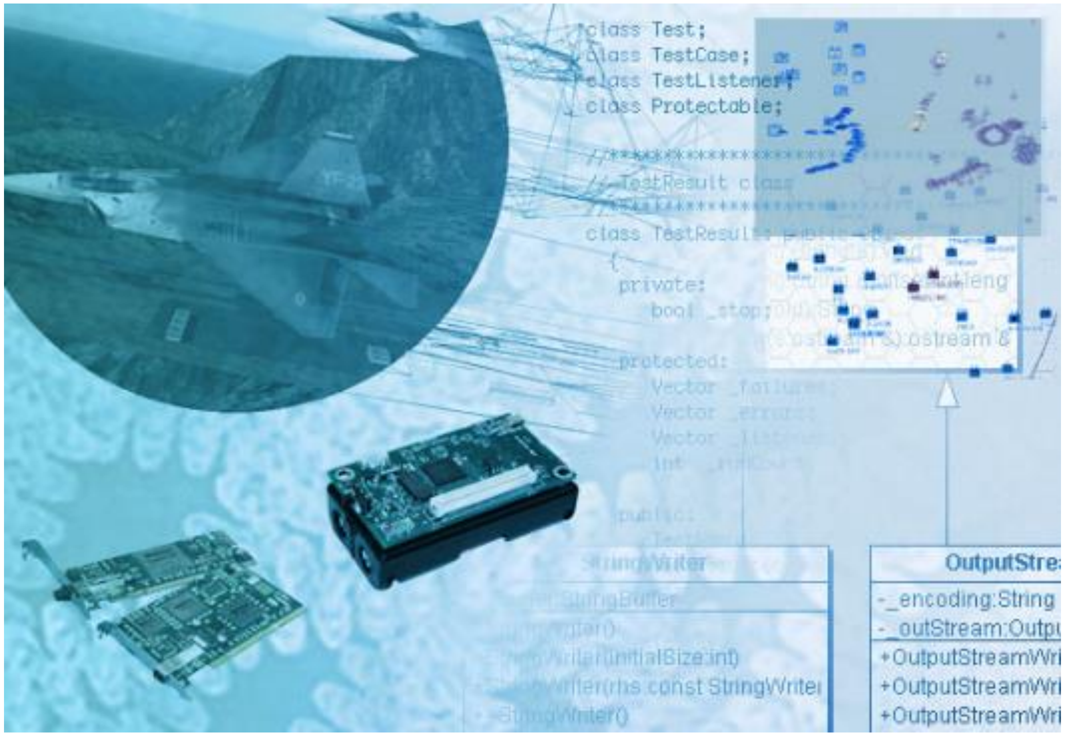
CSYE 6200

CONCEPTS OF OBJECT-ORIENTED DESIGN

FALL 2016 – WEEK 4



MARK G. MUNSON

ADMINISTRATION

- **Assignment #2 is due:**
  - October 4 (Tues. section) / 5 (Wed. section), 6:00 pm (prior to class)
  - Code and sample output in Blackboard
  - NOTE: Assignment #2 has some additions, we'll see them at the end of the lecture

- **TA's are on Blackboard and are available for help**

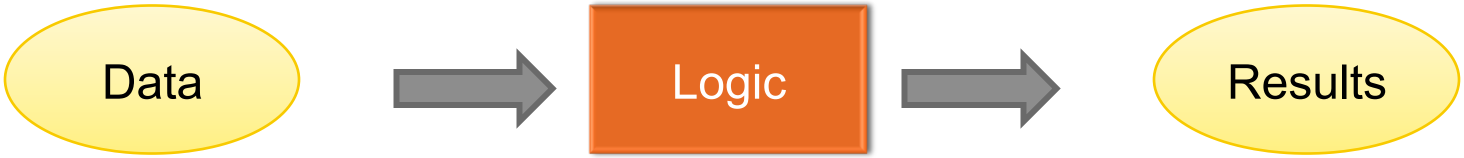
THE LECTURE

- **Recap**
- **Object Oriented**
  - Classes
  - Inheritance
  - Public/Private
  - Encapsulation
  - Polymorphism
- **Static**
- **ArrayList**
- **HashMap**

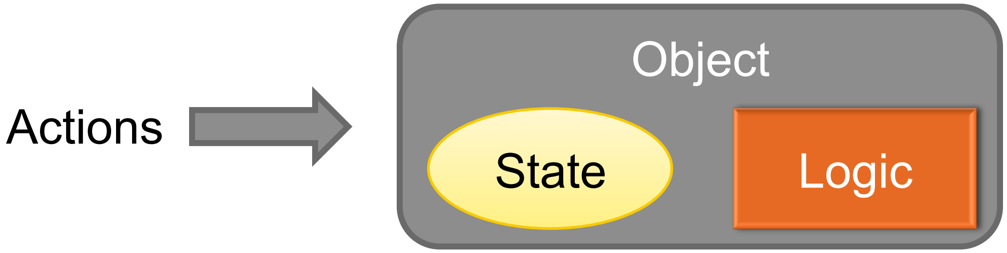
OBJECT-ORIENTED

OBJECT ORIENTED

Historical model: Data is fed into actions for processing (logic) to produce/output results



The Object Oriented model changes that view, and instead says that there are objects that need manipulation.



An Object is an entity that has state and performs actions or has behavior

OO: OBJECT

An object is an entity that has state and performs actions or has behavior

State

Has address

Has rooms

Has windows

Has plumbing

Actions

Can open/close

Can heat

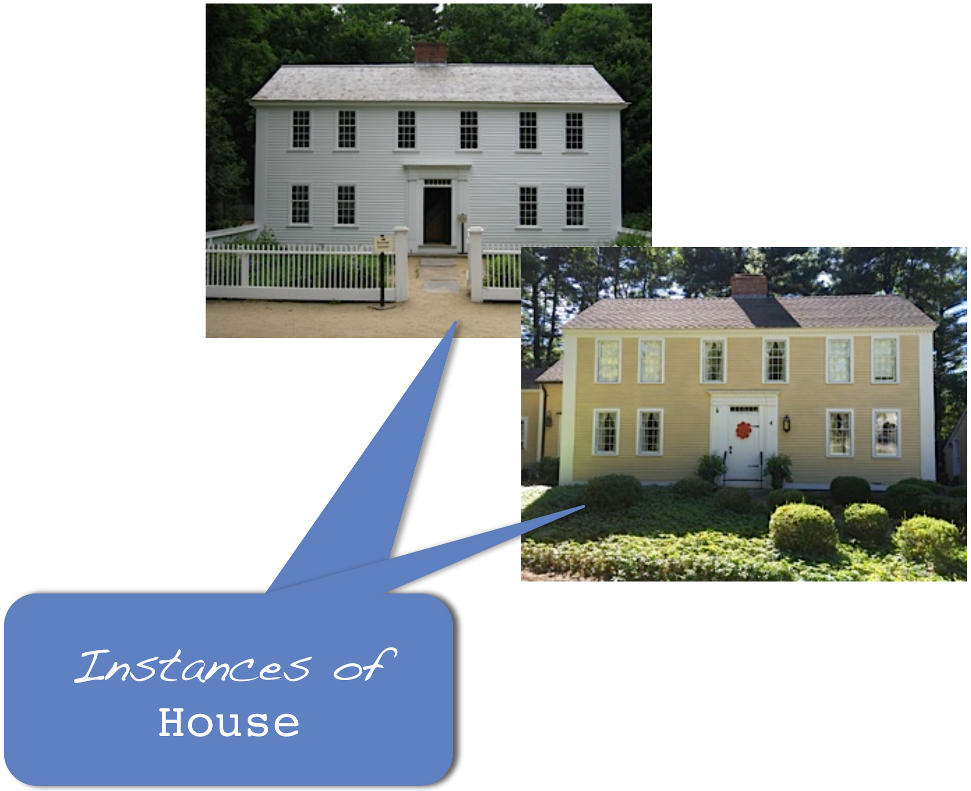
Can cool

OO: CLASS

A class is a set of attributes or behavior that is common to a group of objects

```
class House
```

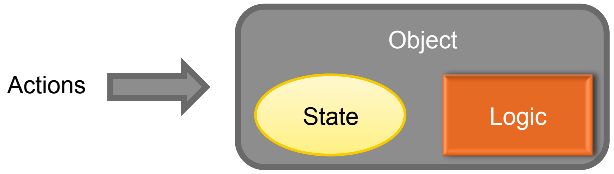
Has address
Has rooms
Has windows
Can open/close
Can heat
Can cool



Instances of
House

OBJECT ORIENTED

- An object is an entity that has state and performs actions or has behavior



- A class is a set of attributes or behavior that is common to a group of objects

`class House`

Has address
Has rooms
Has windows
Can open/close
Can heat

- An instance is an individual object that belongs to a class

OBJECT ORIENTED

Objects, classes, and instances:

An Object

```
House houseInst1 = new House();

House houseInst2 = new House();
```

Each instance is an object of class House

Class House is a blueprint for making instances

```
class House {

    Address address;

    int rooms;

    int size;

    void open();

}
```

CLASS FORM

- **In Java, a class may be specified using the following form:**

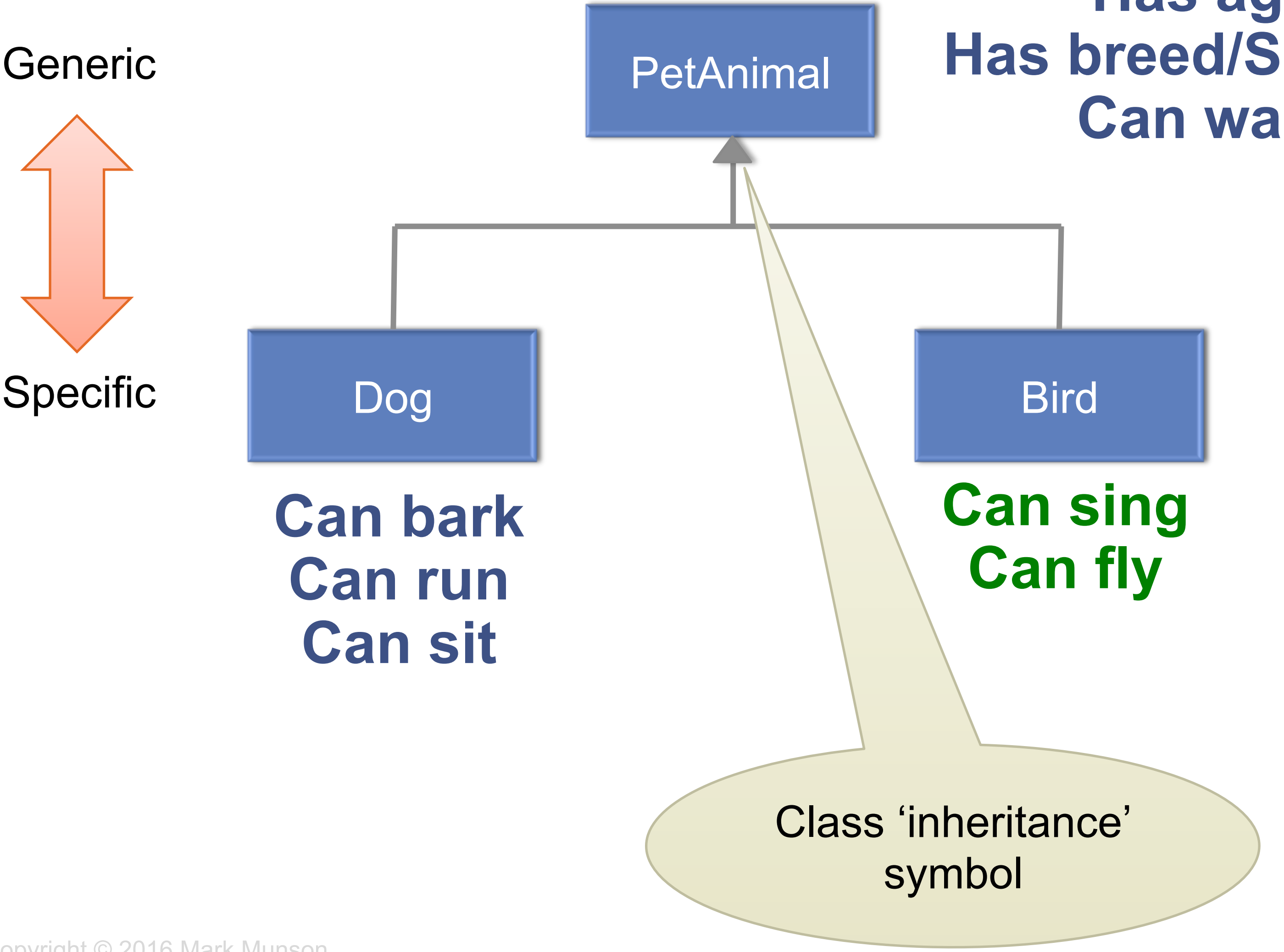
```
class classname {
 // instance variables
 type var1;
 type var2;

 // declare methods
 ret-type method1(parameters) {
   // body of method
 }
 ret-type method2(parameters) {
  // body of method
 }
}
```

OBJECT ORIENTED

- **What does it mean to be Object Oriented?**
  - **Inheritance**
  - Encapsulation
  - Polymorphism

OO INHERITANCE



Generic

Specific

PetAnimal

Has color
Has age
Has breed/Species
Can walk

Dog

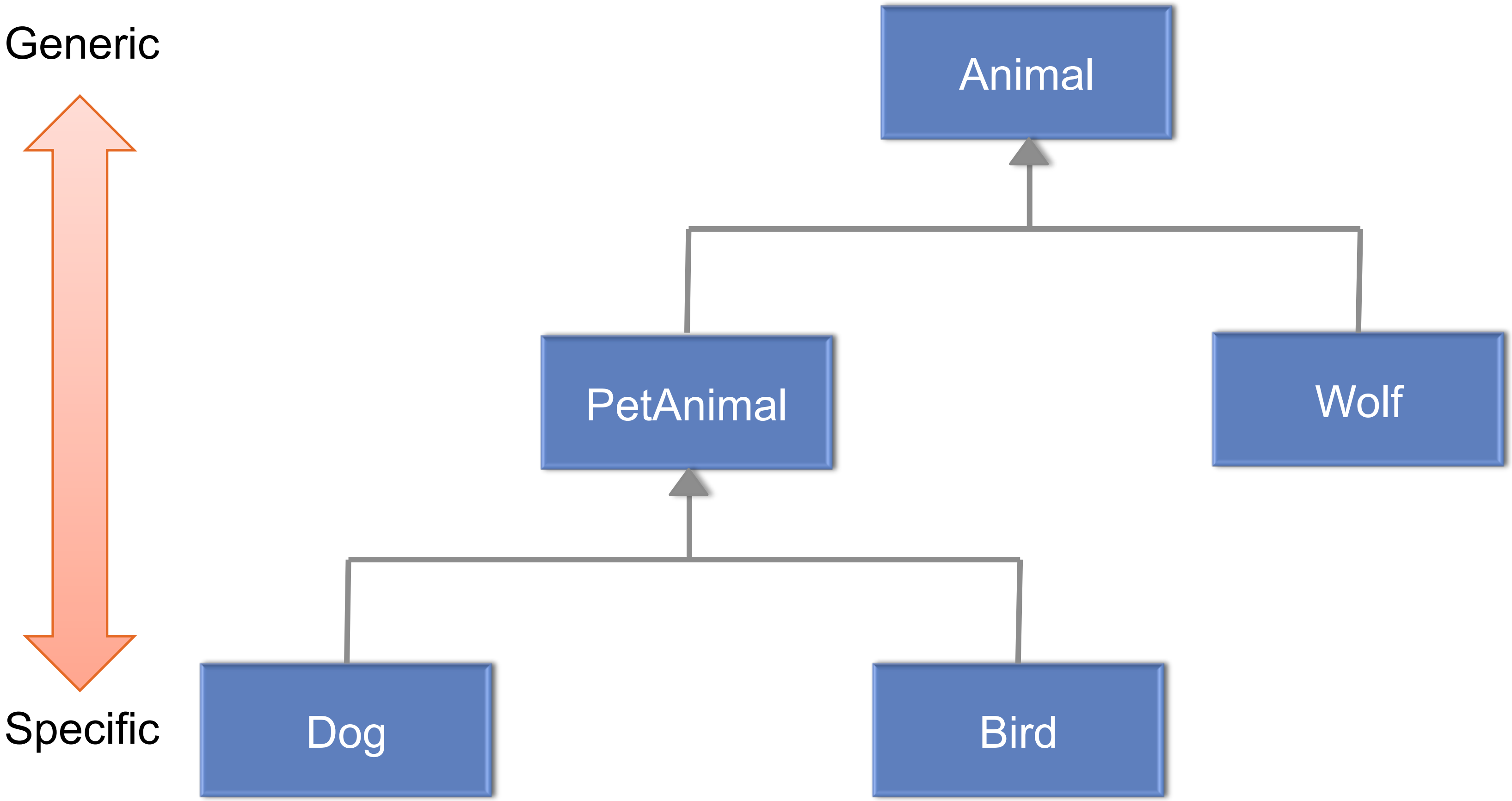
Can bark
Can run
Can sit

Bird

Can sing
Can fly

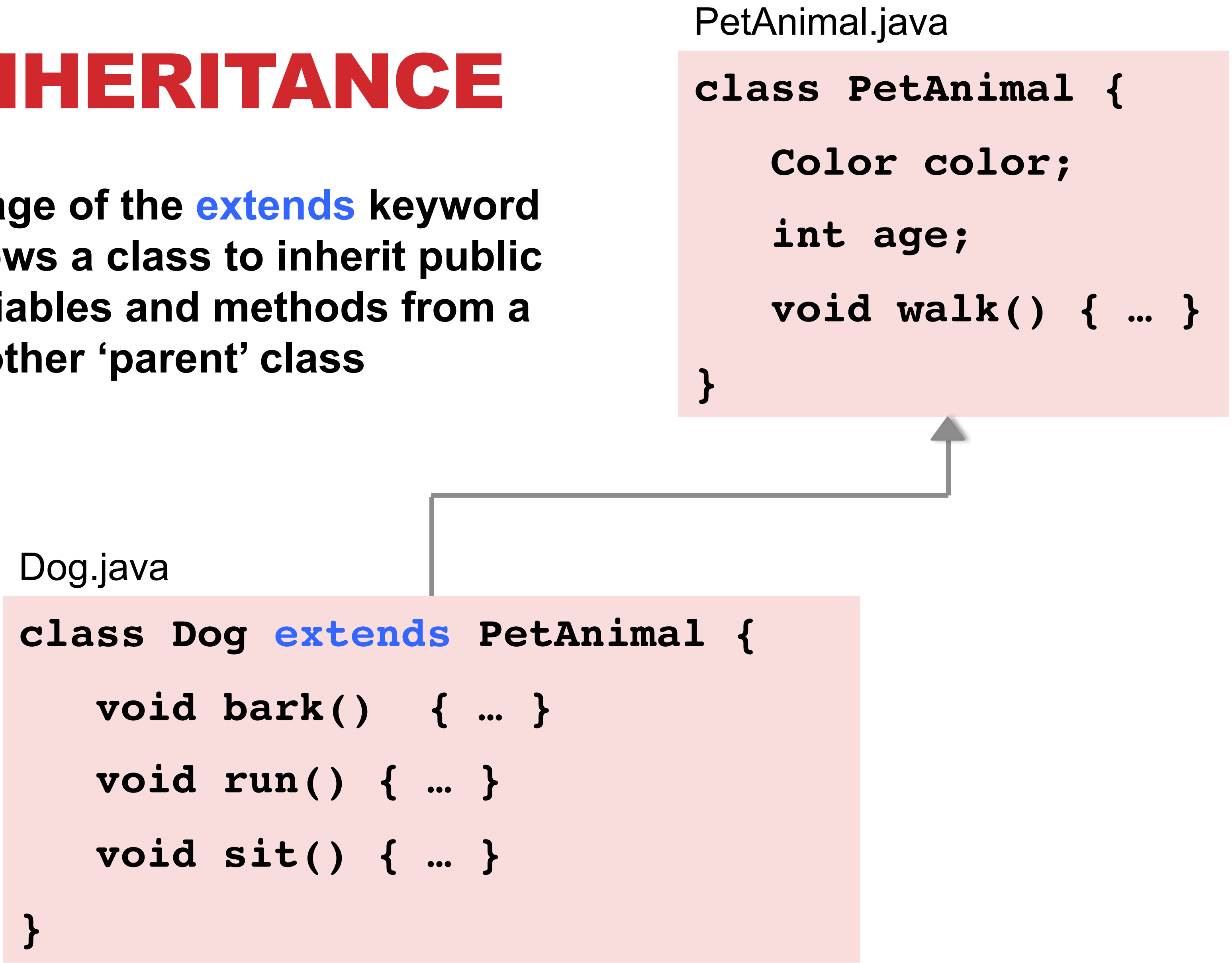
Class ‘inheritance’ symbol

OO INHERITANCE



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() { … }
}
```

OBJECT ORIENTED

- **What does it mean to be Object Oriented?**
  - Inheritance
  - **Encapsulation**
  - Polymorphism

ACCESS MODIFIERS

PUBLIC / PRIVATE

PUBLIC / PRIVATE VARIABLES

Variables that are marked public may be accessed from outside the class.

```
class Vehicle {
   public int passengers;
   private int fuelCap;
   private double kpl;
   …
```

```
class VehicleTest {

   public static void main(String args[]) {

      Vehicle minivan = new Vehicle();

      minivan.passengers = 7; // public – OK

      minivan.fuelCap = 40; // ILLEGAL CALL

   }

}
```

PUBLIC / PRIVATE VARIABLES

- Class instance variables that are marked private can only be accessed from methods inside their class.
- Accessor methods may be used to set or expose private variables.

```
class Vehicle {
   public int passengers;
   private int fuelCap;
   private double kpl;

   public Vehicle(int passengers, int fuelCap, double kpl) {
      this.passengers = passengers;
      this.fuelCap = fuelCap;
      this.kpl = kpl;
   }

   public double getKpl() {  // A “getter” method
      return kpl;
   }
}
```

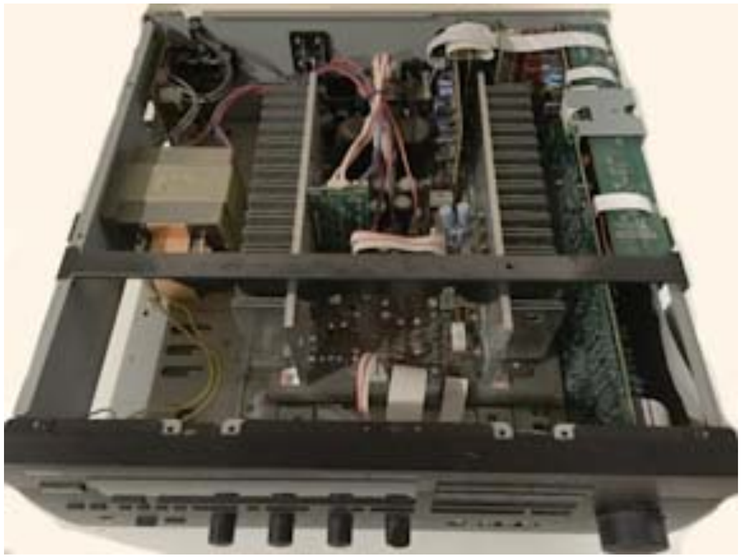
PUBLIC / PRIVATE METHODS

- **Methods marked as private may only be accessed by internal calls**
- **Public methods may be called by external classes**

```
class Vehicle {
   public int passengers;
   private int fuelCap;
   private double kpl;
…
   private double testGGECalc(int passengers, double kpl) {
      return(kpl / 1.39); // E85 to gasoline GGE (1.3900)
   }
   public double getKpl() {  // A “getter” method
      return kpl;
   }
   public double runTests() {  // A “getter” method
      return (testGGECalc(9,40));
   }
}
```

ENCAPSULATION

- The Public/Private keywords allow your code to hide internal working details while leaving important methods and variables exposed for public access.
- This technique of hiding internal object complexity is referred to as encapsulation.



Fully exposed

‘Encapsulated’

OBJECT ORIENTED

- **What does it mean to be Object Oriented?**
  - Inheritance
  - Encapsulation
  - **Polymorphism**

OVERLOADING METHODS

In Java, methods may share the same name, as long as the input parameter definitions are different

Procedural style:

```
void moveToInt(int x, int y) {…

void moveToDbl(double x, double y) {…

void moveToLoc(Location loc) {…
```

Java method overloading:

```
void moveTo(int x, int y) {…

void moveTo(double x, double y) {…

void moveTo(Location loc) {…
```

OVERLOADING CONSTRUCTORS

In Java, class constructors may be overloaded, just like regular methods:

```
class Location {
  private double x = 0.0;
  private double y = 0.0;

  public Location(int x, int y) {…

  public Location(double x, double y) {…

  public Location(Vector2d location) {…
```

POLYMORPHISM

- The ability for multiple with different parameter signatures to share the same name simplifies coding and make programming easier.
- This feature is called polymorphism, which roughly translates to ‘many forms/shapes’.

STATIC

STATIC

- **Static is one of the more difficult keywords to understand, because it's used in many different contexts**
  - Static member variables
  - Static methods
  - Static blocks
  - Static classes

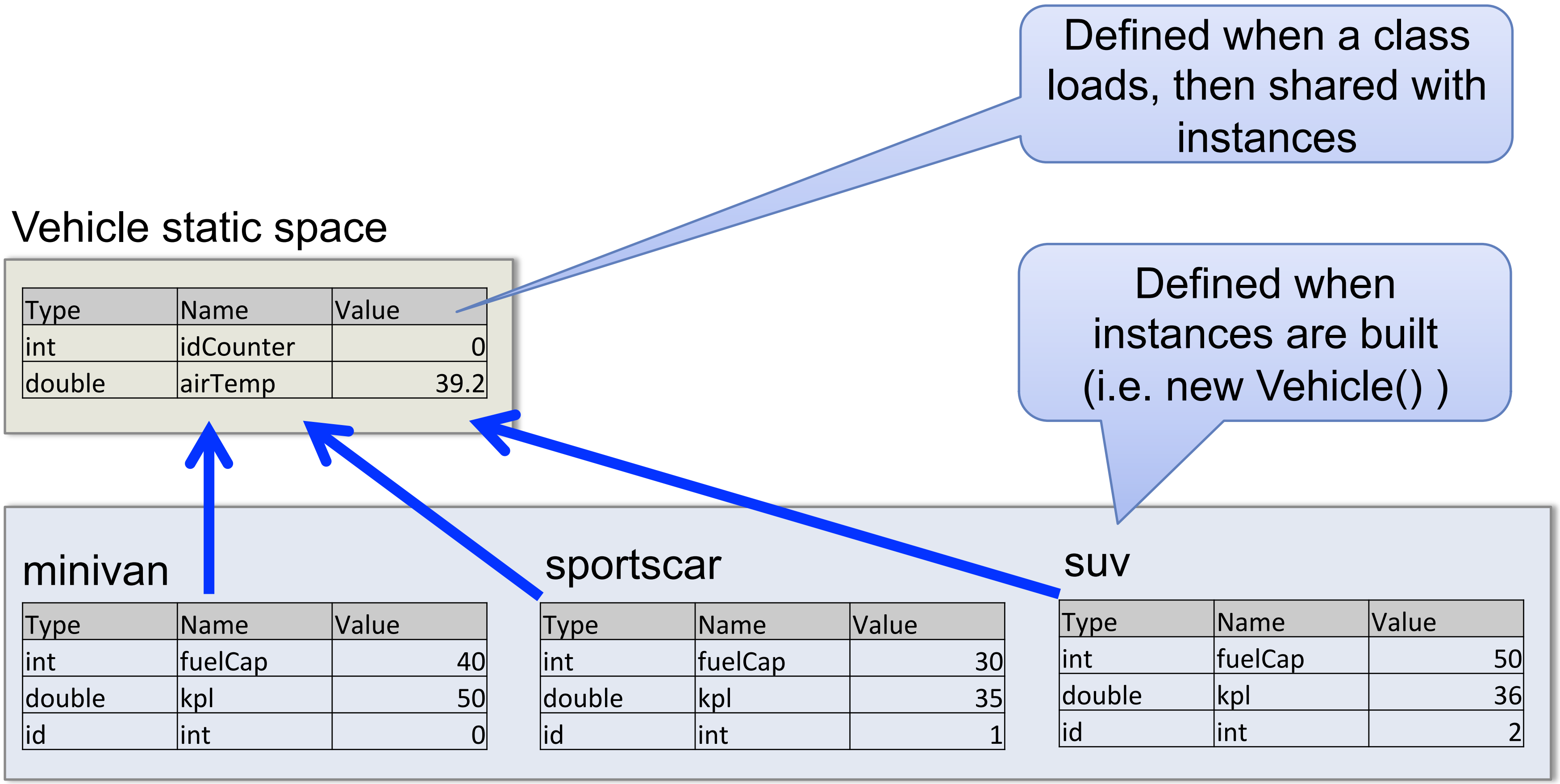
STATIC VARIABLES

- If a class variable is marked as static, it be shared by all instances of the class
- Altering a static variable in one class changes it for all other class instances

```
class Vehicle {
   static int idCounter = 0;
   static double airTempC = 39.2;
   private int fuelCap;
   private double kpl;
   private int id;
…

   private Vehicle() {
      id = idCounter++; // record the current value, then increment
   }
…
}
```

STATIC VARIABLES



Defined when a class loads, then shared with instances

Vehicle static space

Type	Name	Value
int	idCounter	0
double	airTemp	39.2

Defined when instances are built (i.e. new Vehicle())

minivan

Type	Name	Value
int	fuelCap	40
double	kpl	50
id	int	0

sportscar

Type	Name	Value
int	fuelCap	30
double	kpl	35
id	int	1

suv

Type	Name	Value
int	fuelCap	50
double	kpl	36
id	int	2

STATIC CODE BLOCKS

- If a block of code is marked as static, it will execute upon loading
- Static blocks can only use static class variables

```
class Vehicle {
   static int networkPort;
   private int fuelCap;
   private double kpl;
…
   static {
      networkPort = NetworkLoader.getVehPort();
   }

   private double loadFromServer() {
      connectServer(networkPort);
   }

   public double runTests() {…
}
```

STATIC METHODS

- A static method may be called without the need for creating an instance variable.
- A static method cannot use class instance variables, and can only reference static class variables

```
class Vehicle {
   static int idCounter = 0;
   static double airTempC = 39.2;
   private int fuelCap;
…
   static double getDrag(double Cd, double SArea, double Velo )
      return(Air.Pressure / (Air.Rspec * airTempC )); // Drag
   }
…
}
```

- To call a static method without an instance, use the name of the class:

```
 Double drag = Vehicle.getDrag(cd, area, v);
```

STATIC CLASSES

- Java permits you to define a class within a class (i.e. an inner-class). These are usually used as holders for user interface (UI) call-back functions.
- If you define the inner class to be static, you can reference its constructor without having to create an instance of the parent class first.
- OK, these are just plain confusing… let’s skip it.

WRAPPER CLASSES

- **Each of the primitive data types may be wrapped to create an Object.**
- **The wrapper classes are:**
  - Double, Float
  - Byte, Short, Integer, Long
  - Character
- **Each wrapper class can hold a single primitive value**
- **Static convenience methods allow for parsing, conversion, and manipulation of the primitive type**
  - Convert string “true” or “false” into a boolean
  - Convert a string number “3.145” into a double

CHARACTER STATIC METHODS

- **The Character wrapper class has many useful methods**
  - static boolean isDigit(char character)
  - static boolean isISOControl(char character)
  - static boolean isLetter(char character)
  - static boolean isLetterorDigit(char character)
  - static char toLowerCase()
  - static char toUpperCase()

```
char numC = ‘9’;
boolean numChk = Character.isDigit(numC);
```

STRING METHODS

- **The String class has many useful methods**
  - Char charAt(int index)
  - String toLowerCase() – convert all upper-case to lower
  - String toUpperCase() – convert all lower-case to upper
  - String trim() – remove ‘white space’ from before and after
  - String replace(char oldchar, char newchar)

- **Useful static methods**
  - void format(String format, Object… args)
  - String valueOf(double d) – convert a number to a string
  - String valueOf(int i) – convert a number to a string

ARRAYLIST

ARRAYLIST

- **ArrayList provides a list of objects (i.e. Elements), which may be accessed using the List interface**

```
ArrayList vehicleList = new ArrayList(16);
```

- **Typical ArrayList operations**
  - boolean isEmpty() – true if empty
  - int size() – the number of entries in the list
  - add(Object o)
  - Object get(int index)
  - int indexOf(Object o)
  - Object remove(int index)
  - boolean void remove(Object o)
  - void clear()

```
vehicleList.add(new Vehicle(“Ford”,”Mustang”));
```

ARRAYLIST (CONT.)

- ArrayList may be typed to ensure that all Elements are of the specified type

```
ArrayList<Vehicle> carList = new ArrayList<Vehicle>();
```

- A normal for loop may be used to access each element

```
for (int i = 0; i < carList.size(); i++) {
   Vehicle car = carList.get(i);
   // do something
   }
```

- A typed list may be iterated over within a for loop:

```
for (Vehicle car : carList ) { // do something }
```

HASHMAP

HASHMAP

- **HashMap provides a list of objects (i.e. Elements), which may be accessed using the Map interface**

```
HashMap vehicleMap = new HashMap();
```

- **Typical HashMap operations**
  - boolean isEmpty() – true if empty
  - int size() – the number of entries in the map
  - Object put(Object key, Object value)
  - Object get(Object key)
  - Object remove(Object key)
  - void remove(Object key Object value)
  - void clear()
- **Type identifiers may be added to improve coding:**

```
HashMap<String, Vehicle> vehicleMap = new HashMap<String,
Vehicle)();
```

```
vehicleMap.put(vehicle.getName(), vehicle);
```

NEXT WEEK / ASSIGNMENT #2B

- **JABG: Read Ch. 8 & 9 (Packages, Interfaces, and Exception Handling)**
- **Assignment 2a/b: Due October 4 (Tues. section) / 5 (Wed. section), 6:00 pm (prior to class)**
  - Write a Vehicle.java class and a VehicleTest.java class
  - Use the sample starter code (CSYE6200Assign2.zip) which will be uploaded to the course material site. Please fill in your name and NUID number.
  - To the Vehicle class
    - add Strings for both the make and model (i.e. make: Volvo, model: S80)
    - Add a constructor that sets the make and model, along with the other instance variables
    - Add a model year
    - Add a license plate string (i.e. “VBG 984”)
    - Add a method to calculate the vehicle range
    - Add a method to print an attractive display of the vehicle data including the range
  - In the VehicleTest program, use the ‘new’ operation with your Vehicle constructor to generate two instances of different vehicle objects.
    - Place all test code (i.e. from ‘main’) and place into a run() method inside VehicleTest. Call run() from main().
  - Write a VehicleRegistry.java class
    - Add a private ArrayList for holding Vehicles, and add public convenience methods that supports adding, getting, and removing vehicles.
    - Create a public method that loops through all vehicles in your ArrayList, and prints the contents (add an attractive header).
    - Add a private HashMap that stores Vehicles by license number. Add a method to retrieve by license plate string.
  - Submit your source code Blackboard as .java files. Include a copy of your program’s output captured in a text file.