Public vs Private Variables

- Within a class, we can declare class variables with certain access
- Public, private, protected
  - We won't really go into details for protected
- Public means any object of that class can directly access that variable
- Private means only the class itself can have direct access to that variable

Example

```
public class Car {
      public double maxMPH; // class variables should be private
      private String make;
      private String model;
      private int year;
      private String color;

      // constructor (used to initialize class variables)
      public Car(double mph, String make, String model, int year,
                 String color) {
            maxMPH = mph;
            this.make = make;
            this.model = model;
            this.year = year;
            this.color = color;
      }
}

public static void main(String args[]) {
      Car car = new Car(160, "Chevrolet", "Equinox", 2011, "black");
      System.out.println(car.maxMPH); // OK
      System.out.println(car.make); // ERROR: make is private
      System.out.println(car.model); // ERROR: model is private
      System.out.println(car.year); // ERROR: year is private
      System.out.println(car.color); // ERROR: color is private
}
```

- It is possible to make everything public (and it's also possible to write your entire program in the main method!)… but bad style to do so.
  - The object-oriented programming way of doing things is to hide certain functionality from consumers of the object and provide an interface for them to use in their programs.
  - For example, why would you care about buffers and bytes when trying to print something out to the console? If you did have to worry about those details, then nobody would use it!
  - Encapsulates the "details" of the class implementation.
  - Allow programmers to provide an API to use the class.
    - Helps prevent unintended side-effects (programmers can't "accidentally" mess something up within a class).

Accessor (getter) / Mutator (setter) Methods

- In order for a consumer of the class to modify private fields in the class, certain **public** methods are implemented that change the private variables.

```
// Access / getter methods
      public double getMaxMPH() {
            return maxMPH;
      }
      public String getMake() {
            return make;
      }
      public String getModel() {
            return model;
      }
      public int getYear() {
            return year;
      }
      public String getColor() {
            return color;
      }
// Mutator / setter methods
      public void setMaxMPH(int maxMPH) {
            this.maxMPH = maxMPH;
      }
      public void setMake(String make) {
            this.make = make;
      }
      public void setModel(String model) {
            this.model = model;
      }
      public void setYear(int year) {
            this.year = year;
      }
      public void setColor(String color) {
            this.color = color;
      }

Car car1 = new Car(140, "Chevrolet", "Equinox", 2011, "black");
Car car2 = new Car(120, "Pontiac", "GrandAM", 2001, "silver");
System.out.println(car1.getMaxMPH());
System.out.println(car1.getMake());
System.out.println(car1.getModel());
System.out.println(car1.getYear());
System.out.println(car1.getColor());
System.out.println(car2.getMaxMPH());
System.out.println(car2.getMake());
System.out.println(car2.getModel());
System.out.println(car2.getYear());
System.out.println(car2.getColor());
car2.setColor("pink");
System.out.println(car2.getColor());
```

The "this" keyword

- "this" refers to "this" class
- Used mainly to disambiguate variables within the class
- Without "this" in the example, Java thinks you're referring to parameter name instead of class name (due to scoping).

Static Class Variables

- Static class variables belong to the class instead of a specific instance of an object.
- Static variables are shared among ALL instances of the class.

Example

```
// Additional class variables in Car
      private static int staticCounter = 0;
      private int counter = 0;

// Update Car Constructor to increment the counters when constructed
      // constructor (used to initialize class variables)
      public Car(double mph, String make, String model, int year,
                 String color) {
            maxMPH = mph;
            this.make = make;
            this.model = model;
            this.year = year;
            this.color = color;
            staticCounter++;
            counter++;
      }

// Accessor methods for counters
      public int getStaticCounter() {
            return staticCounter;
      }

      public int getCounter() {
            return counter;
      }

// in main()
Car car1 = new Car(140, "Chevrolet", "Equinox", 2011, "black");
System.out.println(car1.getStaticCounter()); // 1
System.out.println(car1.getCounter()); // 1

Car car2 = new Car(120, "Pontiac", "GrandAM", 2001, "silver");
System.out.println(car2.getStaticCounter()); // 2
System.out.println(car2.getCounter()); // 1
```

Static Methods

- Methods that do not require an instance of an object (i.e. a constructed object) for it to be called.
- Does it make sense to call a method without an object instantiation?
    - Consider a method convertMilesToKilometers(double miles).
    - The functionality of this is not dependent on the state of the object.

```
// in Car.java
public static double convertMilesToKilometers(double miles) {
      return miles * 1.60934;
}

// in main()
System.out.println(Car.convertMilesToKilometers(1)); // 1.604
```

Method Overloading

- Combination between methods with the same name and parameter types is called the **method signature**.
- Within a class, we can have methods with the same name, but not the same signature
    - `public void setMaxMPH(int mph)`
    - `public void setMaxMPH(short mph)`
    - `public void setMaxMPH(byte mph)`
    - `public void setMaxMPH(long mph)`
    - `public void setMaxMPH(String mph)`

```
car1.setMaxMPH(1); // calls the int
car1.setMaxMPH(1.0); // calls the double

short x = 3;
car1.setMaxMPH(x); // calls the short
```

Overloading Constructors

- Constructors can be overloaded as well.
- A default constructor is a constructor without any parameters.
    - If no constructor is provided, Java makes one that sets the class variables to default values.
- If any constructor is defined, Java does not supply a default constructor and you must write one.

Pass by Value

- Pass by value is when the value of a variable is passed to a method, not the actual variable itself (i.e. memory location).
- Any changes to this variable will not remain changed when out of scope.

```
public static void addThree(int x) {
      x = x + 3;
      System.out.println(x); // 8
}
```

```
int x = 5;
addThree(x);
System.out.println("in main. x = " + x); // 5
```

Pass by Reference

- Pass by reference is when the variable is passed to a method. Any changes to this variable will remain changed when out of scope.

```
public static void main(String args[]) {
      Car car1 = new Car(140, "Chevrolet", "Equinox", 2011, "black");
      System.out.println(car1.getColor()); // black
      printColor(car1);
      System.out.println(car1.getColor()); // blue
}

public static void printColor(Car c) {
      c.setColor("blue");
      System.out.println(c.getColor()); // blue
}
```

- Seems like all Java objects are passed by reference, right?
    - WRONG!
    - Object **references** are passed by value…

```
public static void main(String args[]) {
      Car car1 = new Car(140, "Chevrolet", "Equinox", 2011, "black");
      System.out.println(car1.getColor()); // black
      printColor(car1);
      System.out.println(car1.getColor()); // still black
}

public static void printColor(Car c) {
      // new reference
      c = new Car(140, "Pontiac", "GrandAM", 2001, "silver");
      System.out.println(c.getColor()); // silver
}
```

Random Number Generation

- Creating a Random behavior / values in programming is really important.
- Java's API gives us the Random object (java.util.Random)

Example

```
Random generator = new Random();
int randomInt = generator.nextInt(4); // 0 -> 3
double randomDouble = generator.nextDouble(); // 0 -> 0.9999999
System.out.println(randomInt);
System.out.println(randomDouble);
int x = generator.nextInt(3) – 1; // returns [-1, 0, 1]
int y = generator.nextInt(6) + 5 // returns [5, 6, 7, 8, 9, 10]
```

Console Input

- A way for users to interact with their programs is by using the console for input.
    - Your program can prompt users for information and users can enter information into the program by typing it in the console.
    - There are MANY ways to do this, but we will use the **Scanner** object for this purpose.
    - The Scanner object can be used for several types of I/O interaction including files (more on that later in the quarter).

Example

```
Scanner s = new Scanner(System.in);
String t = s.nextLine();
System.out.println(t);
s.close(); // important so resources aren't wasted.

//////

Scanner s = new Scanner(System.in);
System.out.print("Enter your name: ");
String first = s.next(); // returns a value up to a newline or space
String last = s.next();

System.out.println(first);
System.out.println(last);

s.close();
```

- There are many other ways you can finagle with console input (i.e. set delimiters, etc.).
- For now, we will talk about the simplest scenarios.
- There are several ways to safeguard and check for user input being correct (Exception handling is the most common)
    - We'll get to that soon.