Static Methods

Avoiding Instantiation
Look at main()

See the static?
- public static void main(String[] args)

This means that the method can be run without instantiation.
- i.e., the main() method can be run without instantiating its class.
  - Good thing! Otherwise Catch-22.
    - Can’t run main without instantiating the class first.
    - Can’t instantiate the class without running main()!
Other Static Methods

- Lots of Java classes have static methods.
  - Math.java
    - e.g., Math.abs()

```java
public static void main(String[] args) {
    double d = Math.abs(-1.0);
}
```

Did not have to instantiate a Math variable! Just use the class name.

- Normally would say

```java
public static void main(String[] args) {
    Math m = new Math();
    double d = m.abs(-1.0);
}
```

Normally this is the correct way, but if a method is static, then this is poor form (and won’t always work).
Inside Math Class

- abs() method is static.

```java
public class Math {

    ...  
    public static double abs(double a) {
        ...  //can you fill this in? A good test! (Hint: use an “if-else”.)
    }
    ...
}
```
Can Make Our Methods Static

public class Insults {
    public static void printInsult() {
        System.out.println("You ski like Ned Flanders");
    }
}

public class Simpsons {
    public static void main(String[] args) {
        Insults.printInsult();
    }
}

No instantiation! Cool.
Make All Methods Static?

- Why not make all methods static?
- Need the variable names sometimes.
  - These variable names are called objects.

```java
Insults sneer = new Insults();
```

This is an **object** called sneer.
Can Have Multiple Objects

Consider a class called Car.

- Car myBeemer = new Car();
  myBeemer.setColor("blue");

  Car myToyota = new Car();
  myToyota.setColor("green");

Notice one Car class but multiple car objects!

- And they have different properties!
- They are unique objects.

String bmwColor = myBeemer.getColor();    //will be blue
String toyotaColor = myToyota.getColor(); //will be green
Static Can’t Have Multiple Values

Static methods apply to **all** of the objects.

- Suppose setColor() is static.
- Then setColor applies to **all** Car objects.

```java
Car myBeemer = new Car();
Car myToyota = new Car();
Car.setColor("blue");
String bmwColor = myBeemer.getColor();  //will be blue
String toyotaColor = myToyota.getColor();  //will be blue
```

- Makes **all** of the cars blue!
**Instance Variables: Or How an Object Stores Data**

**Instance (or “Object”) Variables**

```java
public class Car {
    private String color = null;

    public void setColor(String carColor) {
        color = carColor;
    }

    public String getColor() {
        return color;
    }
}
```

- Defined outside all methods.
- Each instantiation can store a different variable here.
- So if have “myBeemer”, can store one color. If have myToyota, can store another value.
- They each access their own color.
- We say that the variable “color” “belongs to the objects”, not to the class.
Static Class Variables

```java
public class Car()
{
    private static String color = null;

    public static void setColor(String carColor)
    {
        color = carColor;
    }

    public String getColor()
    {
        return color;
    }
}
```

Now all cars have this color. The static means that the variable “belongs to the class” (doesn’t belong to the objects).

The static method means it “belongs to the class”. So can use it by just calling the class. e.g., Car.setColor(“blue”).
Static Can’t Call Non-static

- **Warning!**
  - You can never call a non-static method from a static method without instantiation.
  - **MUST INSTANTIATE** to use non-static methods.
Wanna’ Learn More?

- These are *object*-oriented concepts.
  - Very, very cool.
  - Take OO Programming class for much more cool stuff like this!