# Static Members & Methods

CMSC 202

# What Does “static” Mean?

- Instance variables, constants, and methods may all be labeled as `static`.
- In this context, static means that the variable, constant, or method belongs to the class.
- It is not necessary to instantiate an object to access a static variable, constant or method.

# Static Constants

- A ***static constant*** may either be public or private.
  - The value of a static defined constant cannot be altered. Therefore it is safe to make it `public.` Making it public allows client programmers to use it.
  - A `private` constant can only be used within the class definition.
  - The declaration for a static defined constant must include the modifier `final`, which indicates that its value cannot be changed.

```
public static final int INVENTED = 1769;

public static final String INVENTOR = "Nicolas-Joseph Cugnot";
```

- Static constants belong to the class as a whole, not to each object, so there is only one copy of a static constant. It is available to the client programmer (if it's public) and to all objects of the class.

- When referring to such a defined constant outside its class, use the name of its class in place of a calling object.

```
int year = Car.INVENTED;
String inventor = Car.INVENTOR;
```

# Static Variables

- A ***static variable*** belongs to the class as a whole, not just to one object.
- There is only one copy of a static variable per class.
- All the member functions of the class can read and change a static variable.
- A static variable is declared with the addition of the modifier `static`.

```
    private static int myStaticVariable;
```

- Static variables can be declared and initialized at the same time.

```
    private static int myStaticVariable = 0;
```

# Static Variables vs. Instance Variables

- Instance variables are local to the instance in which they are created. Notice the results of a mutator modifying the value contained.

```
private static int numWheels = 4;
public int getNumWheels(){
   return numWheels;
}
public void setNumWheels(int nWheels){
   numWheels = nWheels;
}
public static void main(String args[]){
   Car defaultCar = new Car();
   Car chevy = new Car("9431a",2000,"Chevy","Cavalier");
   Car dodge = new Car("8888","Orange","Dodge","Viper", 5,400,2,1996);
   System.out.printf("NumWheels: chevy %d dodge %d default %d%n", chevy.getNumWheels(),
           dodge.getNumWheels(), defaultCar.getNumWheels());
   dodge.setNumWheels(-2);
   System.out.printf("NumWheels: chevy %d dodge %d default %d%n", chevy.getNumWheels(),
           dodge.getNumWheels(), defaultCar.getNumWheels());
   chevy.setNumWheels(5);
   System.out.printf("NumWheels: chevy %d dodge %d default %d%n", chevy.getNumWheels(),
          dodge.getNumWheels(), defaultCar.getNumWheels());
}
```

static variables can be changed!!!

```
NumWheels: chevy 4 dodge 4 default 4
NumWheels: chevy -2 dodge -2 default -2
NumWheels: chevy 5 dodge 5 default 5
```

# Static Methods

So far,

- class methods required a calling object in order to be invoked.
  - These are sometimes known as ***non-static methods***.

```
Car myCaddy = new Car("82978",2011,"Cadillac","Escalade");
System.out.println("My Caddy "+ ((myCaddy.hasSpoiler())? "a spoiler" :
                                 "no spoiler"));
```

***Static methods***:

- Still belong to a class, but need no calling object, and often provide some sort of utility function.
- Static methods are called on the class name (as opposed to an instance name)

```
public static Car[] findAntiques(Car[] cars) { /* ... */ }

Car[] antiques = Car.findAntiques(cars);
for(Car c: antiques) {
    System.out.println(c);
}
```

Use the class name to call the static function.

# Rules for Static Methods

- Static methods have no calling/host object (they have no `this`).

- Therefore, static methods cannot:
  - Refer to any instance variables of the class
  - Invoke any method that has an implicit or explicit `this` for a calling object

- Static methods may invoke other static methods or refer to static variables and constants.

- A class definition may contain both static methods and non-static methods.

# Static Temperature Converting Examples

```
public class Temperature {

  public static double convertFahrenheitToCelsius(double degreesF){
    return 5.0/9.0 * (degreesF - 32);
  }

  public static double convertFahrenheitToKelvin(double degreesF){
    return (degreesF + 459.67) * (5.0/9.0);
  }

  public static void main(String[] args){
    double degreesF = 100;

    // since we have 2 static methods, no instances
    // of the TemperatureConverter class are required
    System.out.printf("%f degrees Fahrenheit%n", degreesF);
    System.out.printf(" is %f Celsius%n",
                      Temperature.convertFahrenheitToCelsius(degreesF));

    System.out.printf("is %f Kelvin%n",
                      Temperature.convertFahrentoKelvin(degreesF));
  }
}
```

# main is a Static Method

Let us take note that the method signature of main( ) is

```
public static void main(String [] args)
```

Being static has two effects:

- main can be executed without an object.
- “Helper” methods called by main must also be static.

# Any Class Can Have a main( )

- Every class can have a public static method name main( ).

- Java will execute main in whichever class is specified on the command line.

```
java <className>
```

- A convenient way to write test code for your class.

# Static Review

- Given the skeleton class definition below

```
public class C
{
  public int a = 0;
  public static int b = 1;

  public void f( ) {…}
  public static void g( ) {…}
}
```

- Can body of f( ) refer to a?
- Can body of f( ) refer to b?
- Can body of g( ) refer to a?
- Can body of g( ) refer to b?
- Can f( ) call g( )?
- Can g( ) call f( )?
  - For each, explain why or why not.

# The `Math` Class (Static Class)

- The Math class provides a number of standard mathematical methods.
  - All of its methods and data are static.
    - They are invoked with the class name `Math` instead of a calling object.
  - The `Math` class has two predefined constants, `E` (*e*, the base of the natural logarithm system) and `PI` (π, 3.1415 . . .).

```
area = Math.PI * radius * radius;
```

# Wrapper Classes

- ***Wrapper classes***
  - Provide a class type corresponding to each of the primitive types
  - Makes it possible to have class types that behave somewhat like primitive types
  - The wrapper classes for the primitive types:

**`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, and **`char`**

are (in order)

**`Byte`**, **`Short`**, **`Integer`**, **`Long`**, **`Float`**, **`Double`**, and **`Character`**

  - Wrapper classes also contain useful
    - predefined constants
    - static methods

# Constants and Static Methods in Wrapper Classes

- Wrapper classes include constants that provide the largest and smallest values for any of the primitive number types.
  - **Integer.MAX_VALUE**, **Integer.MIN_VALUE**, **Double.MAX_VALUE**, **Double.MIN_VALUE**, etc.

- The **Boolean** class has names for two constants of type **Boolean.**
  - **Boolean.TRUE** corresponds to **true**
  - **Boolean.FALSE** corresponds to **false**

  of the primitive type **boolean.**

# Constants and Static Methods in Wrapper Classes

- Some static methods convert a correctly formed string representation of a number to the number of a given type.
  - The methods `Integer.parseInt()`, `Long.parseLong()`, `Float.parseFloat()`, and `Double.parseDouble()`

    do this for the primitive types (in order) `int`, `long`, `float`, and `double`.
- Static methods convert from a numeric value to a string representation of the value.
  - For example, the expression

    ```
    Double.toString(123.99);
    ```

    returns the string value `"123.99"`
- The `Character` class contains a number of static methods that are useful for string processing.

# Wrappers and Command Line Arguments

- Command line arguments are passed to main via its parameter conventionally named args.

```
public static void main (String[ ] args)
```

- For example, if we execute our program as

```
java proj1.Car Shelby Cobra 1967
```

then args[0] = “Shelby”, args[1] = “Cobra”, and args[2] = “1967”.

- We can use the static method **Integer.parseInt( )** to change the argument “1967” to an integer variable via

```
int year = Integer.parseInt(args[2]);
```

  - Each Wrapper Class has the ability to parse its primitive type from a string

# Boxing

- ***Boxing***: The process of converting from a value of a primitive type to an object of its wrapper class.
  - Create an object of the corresponding wrapper class using the primitive value as an argument
  - The new object will contain an instance variable that stores a copy of the primitive value.

```
Integer integerObject = new Integer(5);
```

  - Unlike most other classes, a wrapper class does not have a no-argument constructor.
  - The value inside a Wrapper class is ***immutable***.

# Unboxing

- ***Unboxing***: The process of converting from an object of a wrapper class to the corresponding value of a primitive type.
  - The methods for converting an object from the wrapper classes

    **`Byte`**, **`Short`**, **`Integer`**, **`Long`**, **`Float`**, **`Double`**, and **`Character`**

    to their corresponding primitive type are (in order)

    **`byteValue`**, **`shortValue`**, **`intValue`**, **`longValue`**, **`floatValue`**, **`doubleValue`**, and **`charValue`**.

  - None of these methods take an argument.

```
int i = integerObject.intValue();
```

# Automatic Boxing and Unboxing

Starting with version 5.0, Java can automatically do boxing and unboxing for you.

- Boxing:

```
Integer integerObject = 5;
```

rather than:

```
Integer integerObject = new Integer(5);
```

- Unboxing:

```
int i = integerObject;
```

rather than:

```
int i = integerObject.intValue();
```