Classes & Objects

CMSC 202

Programming & Abstraction

- All programming languages provide some form of ***abstraction***.
  - Also called ***information hiding***
  - Separating how one uses a program and how the program has been implemented
- Procedural Programming
  - Data Abstraction – using data structures
  - Control Abstraction – using functions
- Object Oriented Languages
  - Data and Control Abstraction – uses classes

Procedural vs. Object Oriented

Procedural

- Calculate the area of a circle given the specified radius
- Sort this class list given an array of students
- Calculate the student’s GPA given a list of courses

Object Oriented

- Circle, what’s your radius?
- Class list, sort your students
- Transcript, what’s the student’s GPA?

What is a Class?

- From the Dictionary
  - A kind or category
  - A set, collection, group, or configuration containing members regarded as ***having certain attributes or traits in common***
- From an Object Oriented Perspective
  - A group of objects with ***similar properties, common behavior, common relationships with other objects, and common semantics***
  - We use classes for ***abstraction*** purposes.

Classes

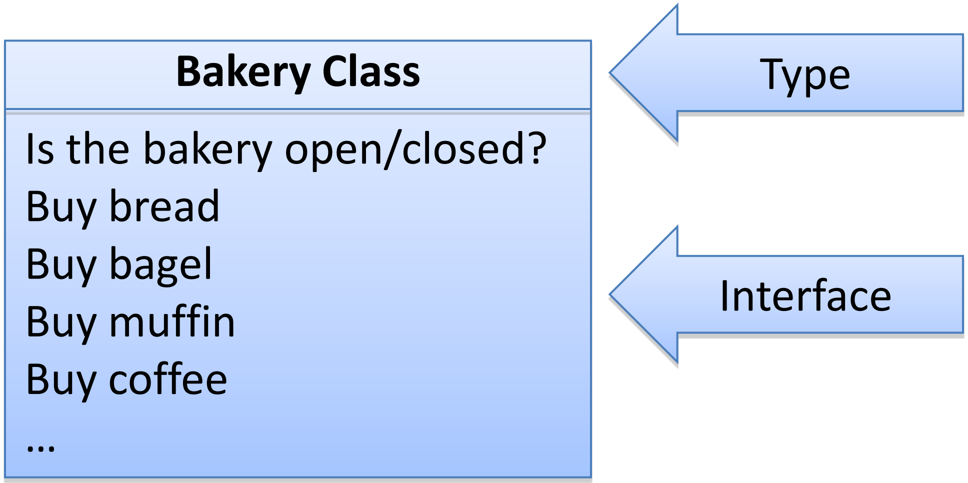
- Classes are “blueprints” for creating a group of objects.
  - A bird class to create bird objects
  - A car class to create car objects
  - A shoe class to create shoe objects
- The blueprint defines
  - The class’s state/attributes as variables
  - The class’s behavior as methods

Class or Object?

- Variables of class types may be created just like variables of built-in types.
  - Using a set of blueprints you could create a bakery.
- You can create as many instances of the class type as you like.
  - There is more than one bakery in Baltimore.
- The challenge is to define classes and create objects that satisfy the problem.
  - Do we need an Oven class?

Class Interface

- The requests you can make of an object are determined by its ***interface***.
- Do we need to know how bagels are made in order to buy one?
  - All we actually need to know is which bakery to go to and what action we want to perform.



Bakery Class

Is the bakery open/closed?
Buy bread
Buy bagel
Buy muffin
Buy coffee
…

Type

Interface

Implementation

- Code and ***hidden data*** in the class that satisfies requests make up the class's ***implementation***.
  - What’s hidden in a bakery?
- Every request made of an object must have an associated method that will be called.
- In OO-speak we say that you are ***sending a message*** to the object, which responds to the message by executing the appropriate code.

Class Definitions

- We’ve already seen...
  - How to use classes and the objects created from them...

```
Scanner input = new Scanner(System.in);
```

  - How to invoke their methods using the dot notation…

```
int num = input.nextInt();
```

- Let us add to what we already know …

Class Definition

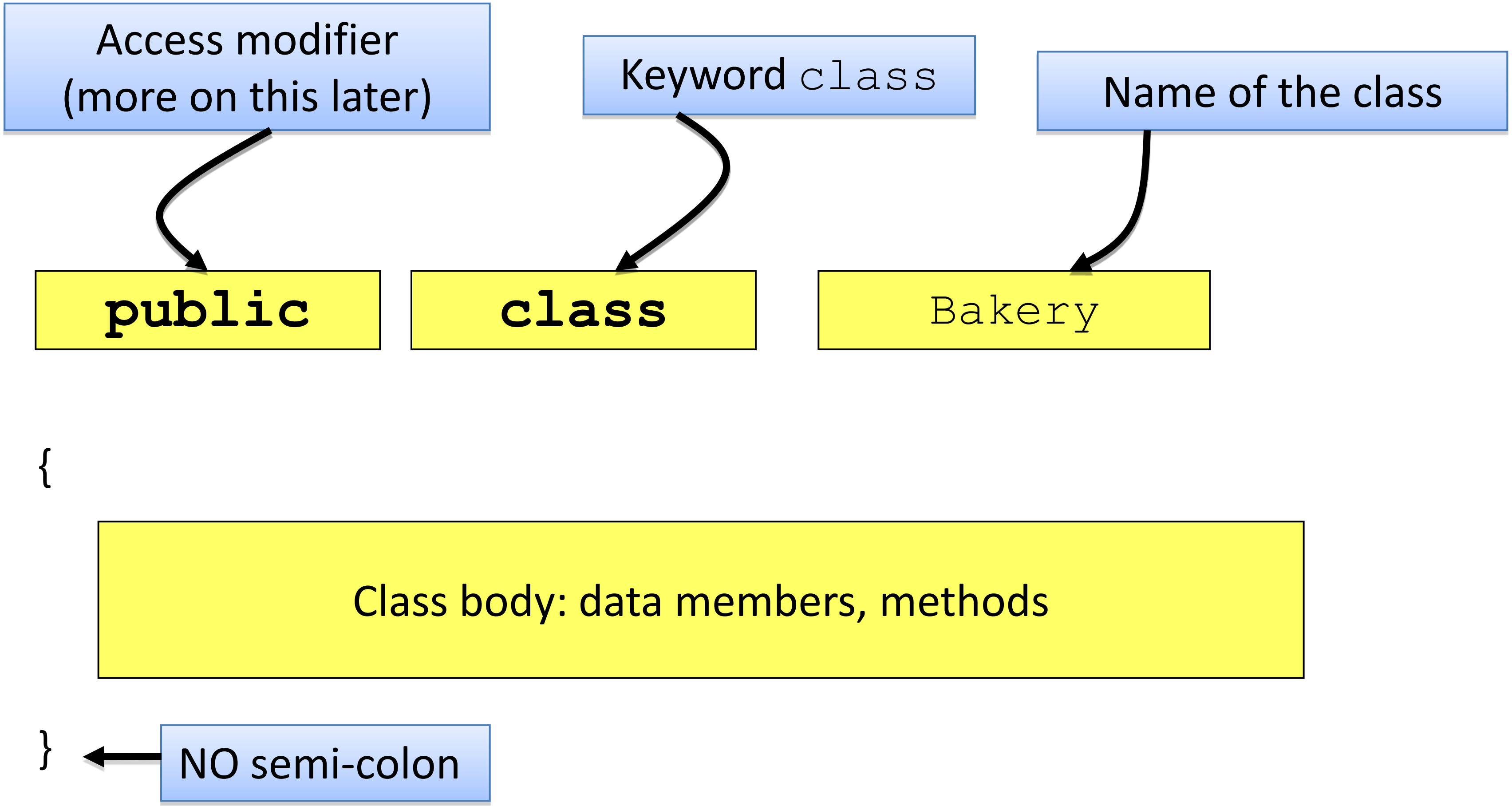
- A ***class definition*** defines the class blueprint.
  - The behaviors/services/actions/operations of a class are implemented ***methods***.
    - Also known as ***member functions***
  - The state of the class is stored in its ***data members***.
    - Also known as ***fields***, ***attributes***, or ***instance variables***

- A challenging aspect of OOP is determining what classes get modeled and at what level of detail.
  - This answer will vary based on the problem at hand.

Objects

- Remember an ***object*** is a particular ***instance*** of an a ***class***.
- As such, all objects have…
  - ***Data Members***
    - The variable types and names (same across all instances)
    - The members of each object can hold different values (unique to that instance).
    - The ***state*** of an object is defined by these values.
  - ***Methods***
    - The tasks that the object can perform (same across all instances)

Anatomy of a Java Class



Access modifier (more on this later)

Keyword `class`

Name of the class

```
public class Bakery
{
```

Class body: data members, methods

```
}
```

NO semi-colon

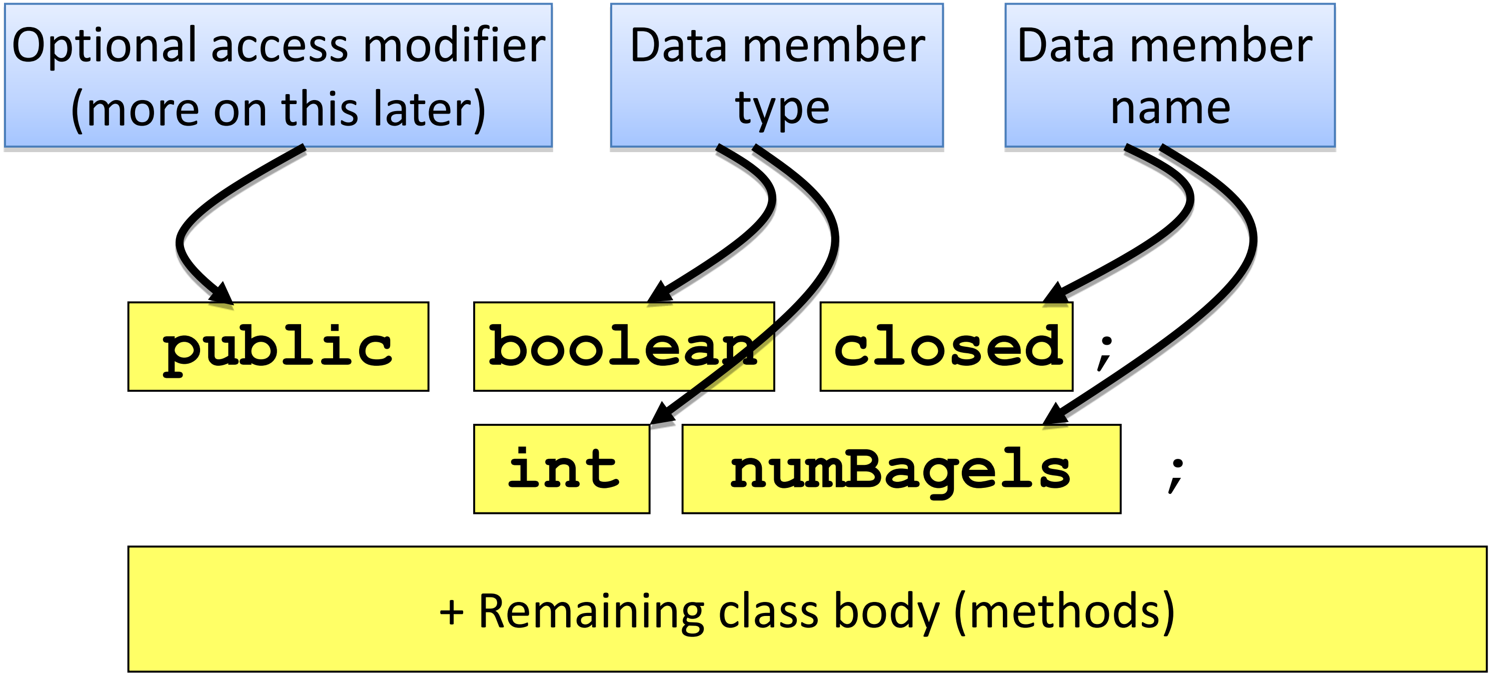
Data Members

- Objects store their individual states in "non-static fields" known as ***data members***.
- Primitive types or reference types
- Accessible by all methods of the class
  - Thus the members are said to have ***class scope***.
- Members are referenced using the ***dot operator***…

```
numItems = array.length;
```

Anatomy of Class Data Members

```
public class Bakery
{
```



Optional access modifier (more on this later)

Data member type

Data member name

```
public boolean closed ;
int numBagels ;
```

+ Remaining class body (methods)

```
}
```

Car Example

- What characteristics (data members) are necessary to store the state for a Car?

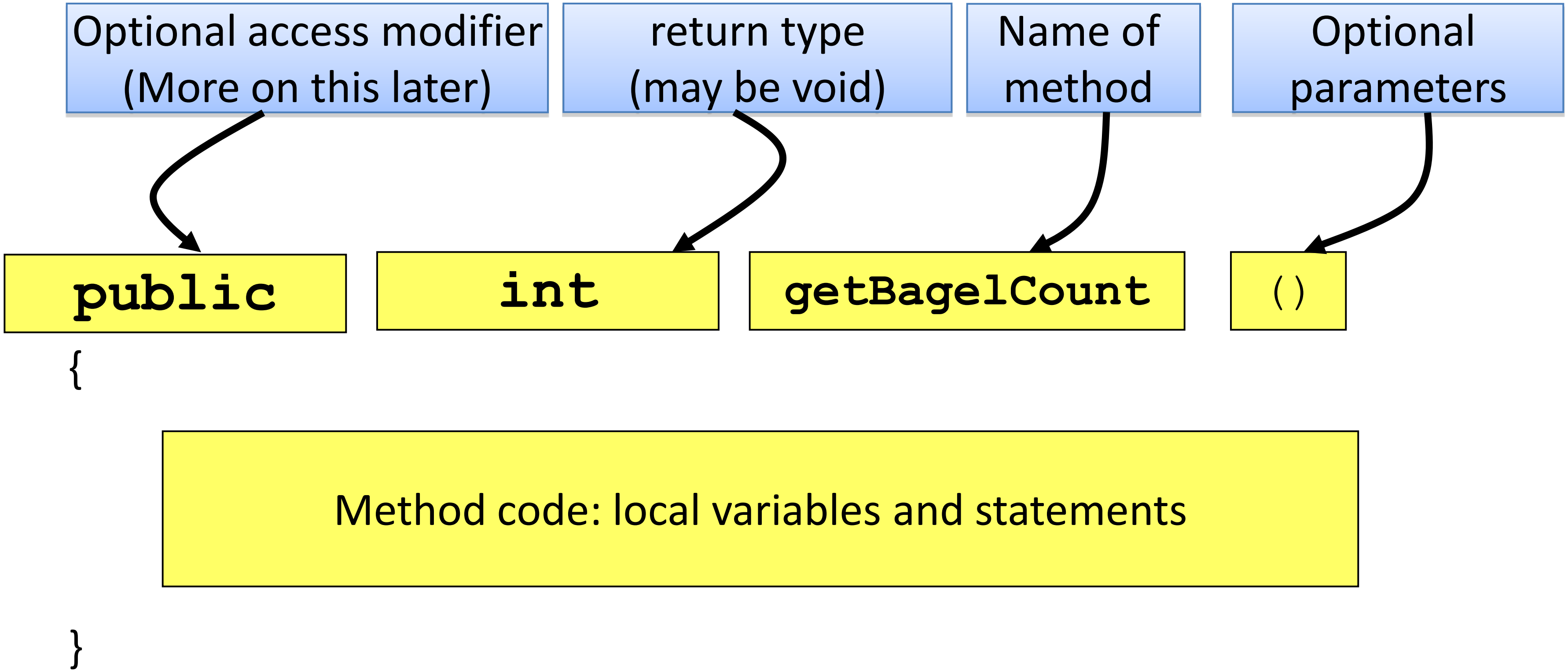
```
public class Car {

    int horsepower;
    int numDoors;
    int year;

    String vin;
    String color;
    String model;
    String make;

    // ...
}
```

Methods

- Objects are sent messages which in turn call ***methods***.
- Methods may be passed ***arguments*** and may ***return*** something as well.
- Methods are available to all instances of the class.
- Like data members, methods are also referenced using the ***dot operator*** …

```
System.out.println(name.charAt(0));
```

Anatomy of a Method



Optional access modifier (More on this later)

return type (may be void)

Name of method

Optional parameters

```
public int getBagelCount ()
{
    Method code: local variables and statements
}
```

Car Example

- What services/behaviors might be appropriate for a Car?

```java
public class Car {
    // ...
    void unlockDoors() { /* ... */ }
    void changeColor(String color) { /* ... */ }
    void changeGear(char gear) { /* ... */ }
    boolean isParkingBrakeEngaged() { /* ... */ }
    void engageParkingBrake() { /* ... */ }
    void disengageParkingBrake() { /* ... */ }
    void depressAccelerator(float percentage) { /* ... */ }
    void depressBrake(float percentage) { /* ... */ }
    // ...
}
```

Creating a Car

- The following defines a reference variable of type Car.
  - However there is no Car object yet!

```
Car myCar;
```

- The statement myCar = new Car( ) creates a “new” Car object and associates it with the variable “myCar”.
  - Now “myCar” refers to a Car object.

```
myCar = new Car();
```

- For convenience, these statements can be (and are typically) combined.

```
Car myCar = new Car();
```

Car Example

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

    Car myCar = new Car();
    myCar.vin = "123567890ABCDEF";
    myCar.numLiters = 2;
    myCar.horsepower = 195;
    myCar.year = 2008;
    myCar.changeColor("Black");

    System.out.println("Car is colored: " + myCar.color);
    System.out.println("Car is " + (2011 - myCar.year) +
                        " years old");

}
```

Painting the Car

- We can change the state of any Car through services defined in the class definition.

```
public void changeColor(String color){
   color = color;
}
```

Which color are we referring to?

- The compiler assumes that all uses of color refer to the ***method parameter*** and hence ***this code has no effect***.

```
// change car color
myCar.changeColor("Blue");
System.out.println(myCar.color);
```

The Calling Object

- Within a method, a variable is reconciled in a specific order.
  1. The parameter list is checked for a variable with that name.
  2. The class's members are checked to see if there's a match.

- What we're really looking for is something to refer to the calling object…

```
public void setColor(String color) {
    "calling object".color = color;
}
```

- In Java, the reserved word ***this*** represents the calling object.
  - It is sometimes necessary to identify the calling object.
  - It is also a matter of style.

```
public void setColor(String color) {
    this.color = color;
}
```

Printing an Object

- If you print you class by passing it to System.out.println(), you’ll get some cryptic looking output like so …

```
Car@54fc9944
```

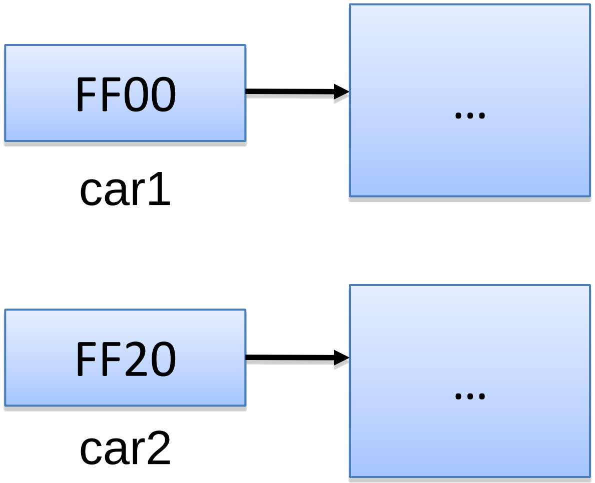
- The print methods will utilize a method called toString() to format the output if you’ve implemented it.
- It’s usually a good idea to implement this method so you can easily see the state of your objects.

```
public String toString() {
        String state = "";
        state += "make: " + make;
        state += " model: " + model;
        // ...
        return state;
}
```

Object Equality

- ***Reference*** variables ***cannot*** be tested for equality using the ***== operator***.
- Testing two reference types for equality will resulting in comparing the underlying addresses.

```
public static void main(String[] args){
    Car car1 = new Car();
    Car car2 = new Car();
    // customize both cars
    if(car1 == car2){
        System.out.println("Same Car");
    } else{
        System.out.println("Different Cars");
    }
}
```



.equals()

- To actually ***compare the state*** of two objects we must implement a ***.equals()*** method.

```
public boolean equals(Car otherCar){
   if(horsepower != otherCar.horsepower){
      return false;
   }
   if(!make.equals(otherCar.make)){
      return false;
   }
   // ... compare necessary members ...
   // otherwise, if all equal return true
   return true;
}
```

Notes:

- Returns a boolean
- Compares only Cars as implemented
- Definition of what constitutes “equals” may vary class to class

Class & Method Documentation

- Class and method level documentation is intended for the consumer of the class – it serves to help the user…
  - Determine if the class is useful/applicable to their problem
  - Find the appropriate method(s) and use them correctly
- Class comments
  - High level documentation as to what the class represents and does
- Method comments — important to explain…
  - What the method does
  - What the method takes as arguments
  - What it returns
  - Pre-conditions and Post-conditions

Pre-conditions & Post-conditions

Pre and post-conditions are important to document in the method comments.

- Pre-conditions
  - All assumptions made about functional parameters and the state of the calling object.
  - For example: The parameter mileage is non-negative.
- Post-conditions
  - All assumptions a user can make after method execution.
  - For example: The car will have a new paint color.

Javadocs

- Java provides API documentation (known as javadocs) for the built-in class library.
- The documentation for each class contains class and method-level documentation.
- Found online (e.g. String, Math, Scanner)
- These documents are created using the javadoc tool.
    - Required for CMSC 202 project documentation
    - Demonstrated in Lab 1

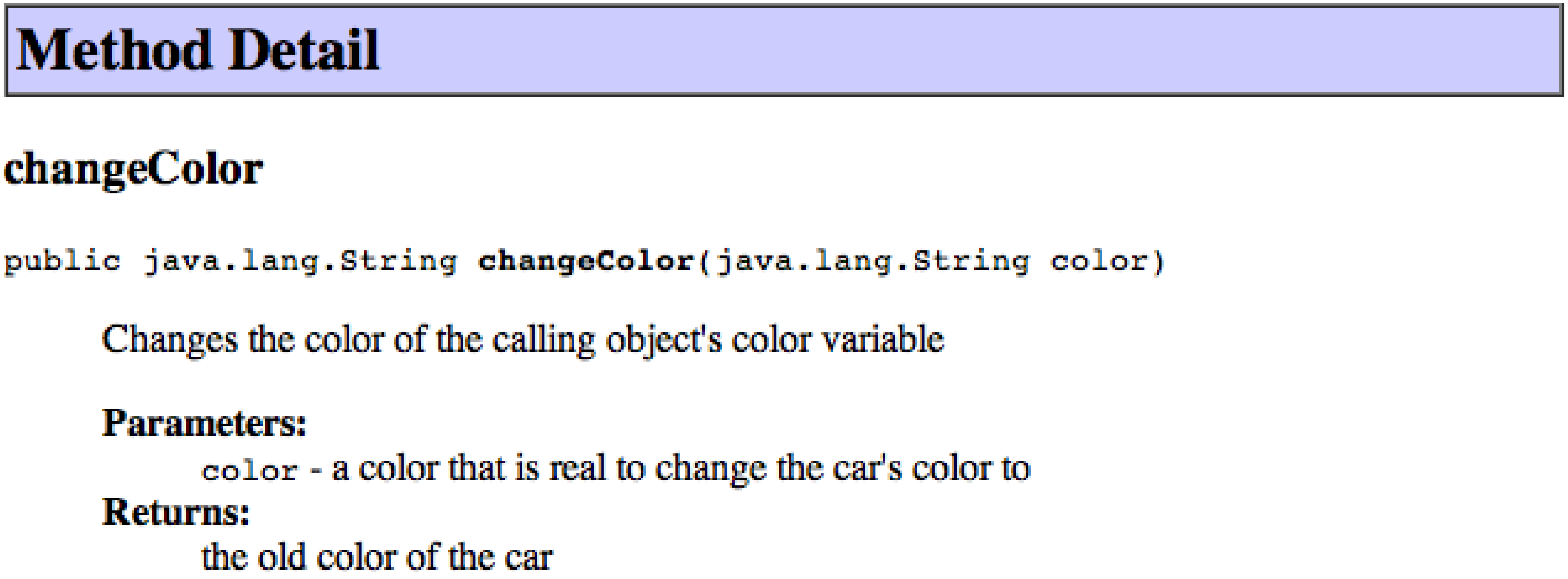
Javadoc Format

- Free-form text to describe method
- @param tag to identity and describe parameters
  - You should have a @param tag for each argument.
- @return tag to detail what is returned when called

```
/**
 * <description of what the method does>
 *
 * @param arg1 <description of arg1>
 * @param arg2 <description of arg2>
 * @return <description of what's returned>
 */
<return type> methodName(<type 1> arg1, <type 2> arg2) {
   // method body
}
```

Example Javadoc

```java
/**
 * Changes the color of the calling object's color variable
 *
 * @param color - a color that is real to change the car's color to
 * @return the old color of the car
 */
public String changeColor(String color){
    String old = this.color;
    this.color = color;
    return old;
}
```



Method Detail

changeColor

```
public java.lang.String changeColor(java.lang.String color)
```

Changes the color of the calling object's color variable

Parameters:

`color` - a color that is real to change the car's color to

Returns:

the old color of the car