CSCI 201: Data Structures

Fall 2024

Lecture 2R: Objects

Goals for today:

- Start styling our codes nicely.
- Create objects using the `new` keyword.
- Define **constructors** which are called when creating objects.
- Reference the current object using the `this` keyword.
- Decide whether member variables (fields) or methods should be declared `public` or `private`.
- Use the `.` "dot" operator to access member variables (fields) or call methods.
- Write **setter** and **getter** methods.
- Make some member variables (fields) **mutable** or **immutable**.
- Reinforce the decision about whether to make a method `static`.

Let's set up a few style guidelines.

- Opening `{` ends first line of block (`if`, `for`, class, method).
- Indent every line inside block.
- Closing `}` should be on separate line, "undoing" indentation of block.
- Use `CamelCase` for class names, `drinkingCamelCase` for variables/methods, lowercase for single words.
- Use spaces between keywords and other control characters (`( )`, `{ }`) and operators (`&&`, `<`, etc.).
- No space between semi-colon and **previous** character (`i = 0;`, not `i = 0 ;`).
- Use a space after semi-colon and next statement (in definition of `for`-loop).
- One line, one statement (unless comma-separated, e.g. `int i = 0, j = 0`).

```
 1  int x = 5, y = 20;
 2  String my_name = "Mike Wazowski"; int age = 20;
 3  for(int i = 0; i<values.length;i++){
 4    if(condition) {
 5
 6    }
 7    else
 8    {
 9
10    }
11  }
```

```
 1  int x = 5, y = 20;
 2  String myName = "Mike Wazowski";
 3  int age = 20;
 4  for (int i = 0; i < values.length; i++) {
 5    if (condition) {
 6
 7    } else {
 8
 9    }
10  }
```

3

`Java` is object-oriented.

- A language is **object-oriented** if programs in that language are organized by the specification and use of **objects**.
- An object consists of (1) some **internal data items** along with (2) **operations** that can be performed on that data.
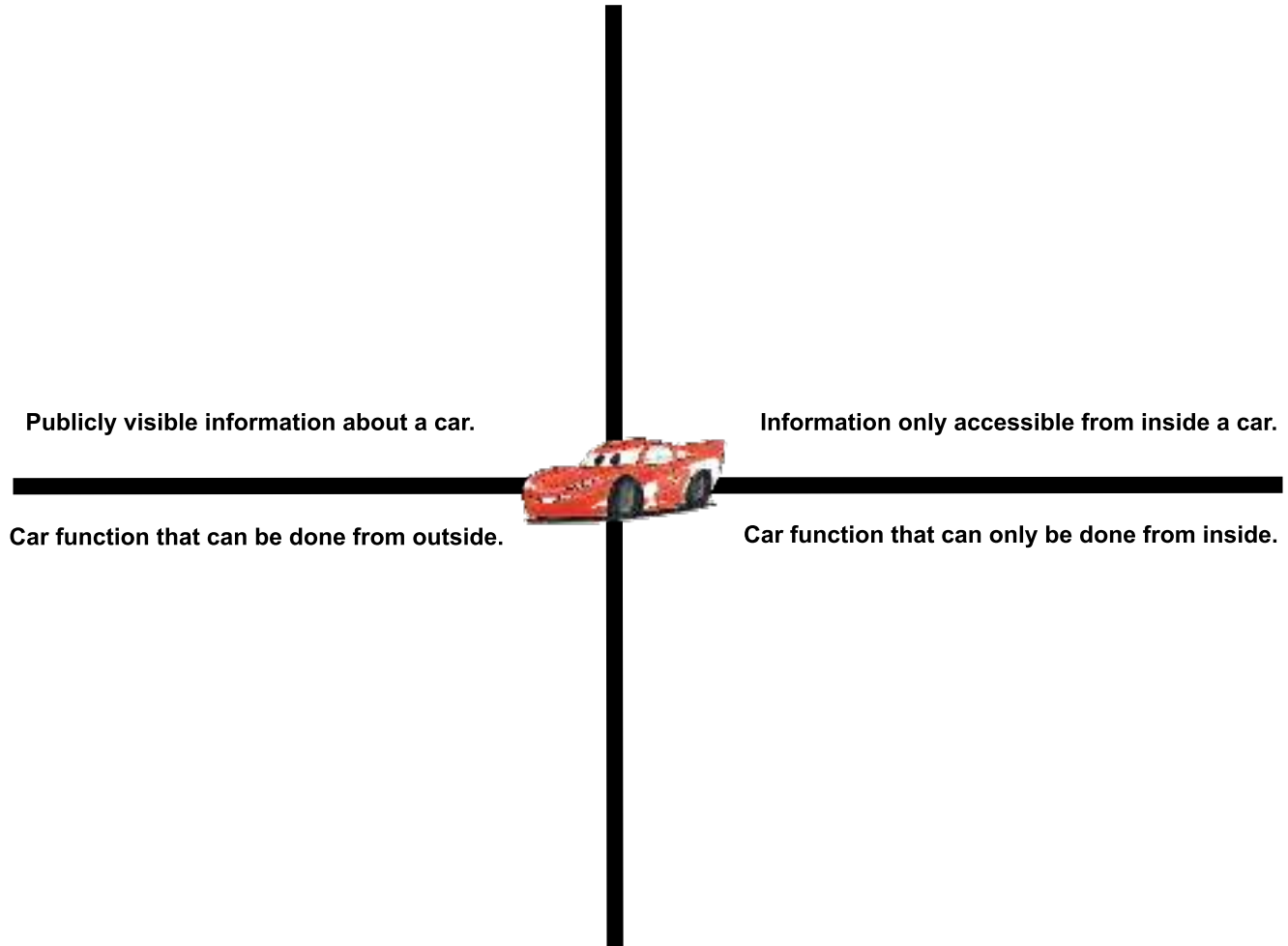
Let's design and create a car!


```
1 class Car {
2    ...
3 }
```

```
1 Car car = new Car("Subaru", 2019);
```

In groups of 3: think about the following questions.

- What information about a car is publicly visible?

- What information about a car can only be accessed if you're inside the car?

- What is a function of a car that can be done from outside the car?

- What is a function of a car that can only be done from inside the car?

- What is something about a car that can change?

- What is something about a car that cannot change?

Publicly visible information about a car.

Information only accessible from inside a car.

Car function that can be done from outside.

Car function that can only be done from inside.

7

Anatomy of a `Java` class definition.

```java
1  // in a file called CarExample.java
2  class Car {
3
4    // member variables (fields)
5    String make;
6    int year;
7
8    // methods
9    Car(String make, int year) { // constructor
10     this.make = make;
11     this.year = year;
12   }
13
14   void drive() {
15     System.out.println("Starting the " + make + "...");
16     System.out.println("Vroom vroom.");
17   }
18 }
19
20 public class CarExample {
21   public static void main(String[] args) {
22     Car car = new Car("Subaru", 2019);
23     car.drive();
24   }
25 }
```

Use *access modifiers* to control what is visible and what is not.

- `public`: can be accessed by code *outside* of the class (also inside).
- `private`: can only be accessed by code *inside* the class.

Why do we use these? readability, correctness.

Anatomy of a `Java` class with *access modifiers*.

```java
1  // in a file called CarExample.java
2  class Car {
3
4    // member variables (fields)
5    public String make;
6    public int year;
7    private int gear;
8
9    // methods
10   Car(String make, int year) { // constructor
11     this.make = make;
12     this.year = year;
13   }
14
15   public void drive() {
16     System.out.println("Starting the " + make + "...");
17     setGear(1); // put the car in first gear
18     System.out.println("Vroom vroom.");
19     honk();
20   }
21
22   private void setGear(int gear) {
23     this.gear = gear;
24   }
25
26   private void honk() {
27     System.out.println("beep beep");
28   }
29 }
30
31 public class CarExample {
32   public static void main(String[] args) {
33     Car car = new Car("Honda", 2019);
34     car.drive();
35   }
36 }
```

What's with the `this`?

```
 1  String make; // fields
 2  int year;
 3  int gear;
 4
 5  Car(String make, int year) { // constructor
 6     this.make = make;
 7     this.year = year;
 8  }
 9
10  private void setGear(int gear) {
11     this.gear = gear;
12  }
```

Primary uses:
- Avoid ambiguity in fields and parameters (notice `make`, `year` and `gear` are **parameters** and **fields**).
- Pass a reference to this object to some other function. For example, imagine we keep a reference to an instance of a `Garage` class called `garage`. Perhaps we need to call `garage.changeOil(this);`

11

Arrays can be used to hold many **Car** objects.

Items are initially null: we need to use new to actually create Car objects.

```
1  int nCars = 5;
2  Car[] cars = new Car[nCars];
3  cars[0] = new Car("Subaru", 2019);
4  cars[1] = new Car("Honda", 2021);
5  cars[2] = new Car("Ford", 2024);
6  // cars[3] and cars[4] are still null
7  System.out.println("Car 3 make = " + cars[3].make); // Exception!!
```

```
Exception in thread "main" java.lang.NullPointerException: Cannot read field
    "make" because "cars[3]" is null at CarExample.main(CarExample.java:7)
```

What will be the value of **same**?
Go to slido.com (event #2743882).

```
1 cars[0] = new Car("Honda", 2019);
2 cars[1] = new Car("Honda", 2019);
3 boolean same = (cars[0] == cars[1]);
```

cs201-lecture02R

There are no active polls
at the moment

A note about *mutability* and *immutability*.

Think about whether you can "mutate" the object.

- **mutable:** object fields can be modified after creation.
- **immutable:** objects fields cannot be modified after creation.

```
1  String s = "Hello";
2  s += " World"; // append
```

```
1  // essentially the same as
2  String sOld = "Hello";
3  String sNew = new String(sOld + " World");
```

Strings are actually **immutable**.
We end up creating a new String when appending.

Standard way to expose what can be set and not set, and what can be retrieved: *setter* and *getter* methods.

Recall what we did in our `Car` class.

```
1 private void setGear(int gear) {
2    this.gear = gear;
3 }
```

```
1 public void setLocked(boolean value) {
2    locked = value;
3 }
```

```
1 public int getSpeed() {
2    return speed;
3 }
```

Remember that **`static`** keyword means it doesn't require an *instance* (object) - it's part of the class definition.

```
1  class Car {
2    public static int numberOfWheels() {
3      return 4; // we don't need to create a car to ask this question
4    }
5    ...
6  }
7
8  // somewhere else in the code
9  System.out.println("The car design has " + Car.numberOfWheels() + "wheels");
```

BUT! Really this should be a *constant*, defined using `static final`

```
1  class Car {
2    public static final int NUMBER_OF_WHEELS = 4; // convention to use SNAKE_CASE for constan
3    ...
4  }
5
6  // somewhere else in the code
7  System.out.println("The car design has " + Car.NUMBER_OF_WHEELS + "wheels");
```

Ask yourself: can I use the blueprint/design or do I need a built object?

Rest of class: do something similar for a `Student`.

Things to think about:

- First, think about what a `Student` object should store: name? array of classes? age? dorm? favorite cereal? study spots? mailbox?
- What kinds of functions should a `Student` be able to do?
- Which fields and methods should be `public`?
- Which fields and methods should be `private`?
- Which fields should be modifiable via a **setter** or retrievable from a **getter**?

Intentionally open-ended!
Brainstorming and designing is part of developing code!

See you tomorrow!

- Homework 1 extended to tomorrow night (9/20) at 11:59pm.
- I'm sorry about not having office hours this week, but I'll post them this weekend.
- Please visit Noah (`go/noah`) or Smith (`go/smith`) for office hours or see the course assistants tonight from 4 - 8pm (schedule at `go/cshelp`).
- Submit exit ticket 2R today.
- For more information about access modifiers, see Oracle's documentation: https://docs.oracle.com/javase/tutorial/java/javaOO/variables.html