Why couldn’t the programmer dance to the song?

Because he didn’t get the…
algo-rhythm…

Module 1-09

Classes and Objects (Part 2)
Encapsulation
Static Members
Garbage Collection

Objectives

- **Define encapsulation**, give a good example of it, how it is implemented, and describe why it is used
- Define **"loosely coupled"** and explain the characteristics of a loosely coupled system
- Define and use **static methods** and be able to describe what they are for
- Explain access modifiers and when you would use private vs. public
- Define and use overloading in an OOP language
- Describe **constant variables**, how to create them, and their use

Principles of Object-Oriented Programming (OOP)

- **Encapsulation -** the concept of hiding values or state of data within a class, limiting the points of access
- **Inheritance -** the practice of creating a hierarchy for classes in which descendants obtain the attributes and behaviors from other classes
- **Polymorphism -** the ability for our code to take on different forms
- **(Abstraction) –** extension of encapsulation. We can't build a car from scratch, but we know how to use (drive) it.

Encapsulation!

Encapsulation

Encapsulation & Data Hiding

- **Encapsulation** is the process of combining related data members and methods into a single unit.
  - In Java, encapsulation and data hiding are achieved by putting all related data members and methods in a class.

- **Data hiding** is the process of obscuring the internal representation of an object to the outside world.
  - In Java, data hiding is achieved by setting all members to private and providing getters and setters for said members.

Encapsulation

Rule: instance variables (properties, data members) are private and methods are public

```
public class Car {
        private int year;

        public void setYear(int year) {
            this.year = year;
        }
        public int getYear() {
            return year;
        }
}
```

private can only be accessed or used inside the Car class

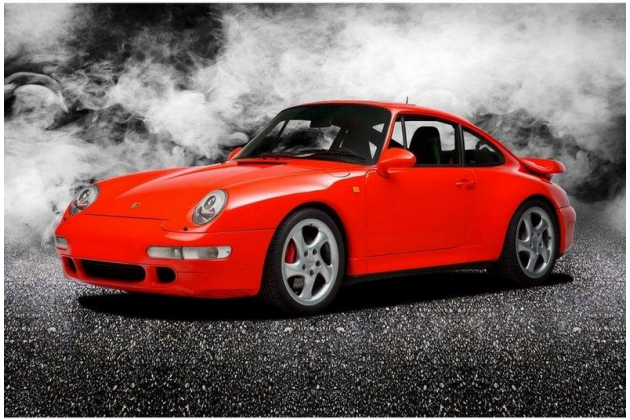
public means this method can be called outside of this class

Goal of Encapsulation

- Makes code extendable
- Made code maintainable
- Promotes “loose coupling”
  - Each of its components has or makes use of little or no knowledge of the definitions of other separate components

Loose vs. tight coupling

- Loose coupling
  - Changing hats does not require any change to the body
  - Changing shirts does not require any change to body
  - Changing color of car does not require a redesign of the car itself



Loose vs. tight coupling

- Tight coupling
  - Skin is tightly coupled to body

Composition

- Made up of, or composed of
- A herd is made up of many elephants
- A deck of cards is made up of many cards
- A pod of Aliens

Static

Definition of Static in Java

If a method or data member is marked as static, it means **there is exactly one** copy of the method, or one copy of the data member shared across all objects of the class.

One way to think about this, is that the static member is a unique property of the “blueprint” that is the same for all objects created from that blueprint.

FordCar class might have a static data member logo. All FordCar objects will share the same static data member.

The non-static methods and data members we have defined so far are often called Instance members or Instance methods.

Static Members: Declaration

Static members and methods are declared by adding the keyword static.

```
public class Car {
		public static String carBrand = "Ford";

		public static void honkHorn() {
			System.out.println("beeep?");
		}
...
}
```

Static: Calling

Assuming we have the static member declarations from the previous slide, this is how you call them from a different class. Note that we should use the class name (Car) as opposed to the name of an instance of a car (thisCar).

```
public class Garage {

		public static void main(String args[]) {

				System.out.println(Car.carBrand); // Correct way to refer to a static member.
				Car.honkHorn(); // Correct call to a static method.

				Car thisCar = new Car("Red", 2);
				System.out.println(thisCar.brand); // Not a typical way to call a static member.
				thisCar.honkHorn() // Not a typical way to call a static member

		}
}
```

Static: Assignment

Public Static data members can be reassigned to new values.

```
public class Garage {

        public static void main(String args[]) {
        Car.carBrand = “GM”;
        }
}
```

Static: Constants

Constants are variables that cannot change. The closest thing to a constant in Java is declaring a data member with **static final**.

```
public class Car {
		public static final String CAR_BRAND = "Ford";
...
}
```

Attempts to change the value of this data member will result in an error. This, for example is invalid:

```
public class CarDealership {

		public static void main(String args[]) {

				Car.CAR_BRAND = "GM";
		}
}
```

Static: Rules

There are some rules to observe when using static methods or data members:

- **Static** variables can be accessed by **Instance** methods.
- **Static** methods can be accessed by **Instance** methods.

The opposite of the above is not true:

- **Static** methods cannot access **Instance** data members.
- **Static** methods cannot call **Instance** methods.

Static: Rules

```
String someInstanceVariable;

public static void someStaticMethod() {
        System.out.printlnString (someInstanceVariable);
        someInstanceMethod();
}

public void someInstanceMethod() {

}
```

This is an instance (non-static data member)

We are inside a static method, but we are referencing an instance member, which is not allowed

We are inside a static method, but we are calling an instance method, which is not allowed.

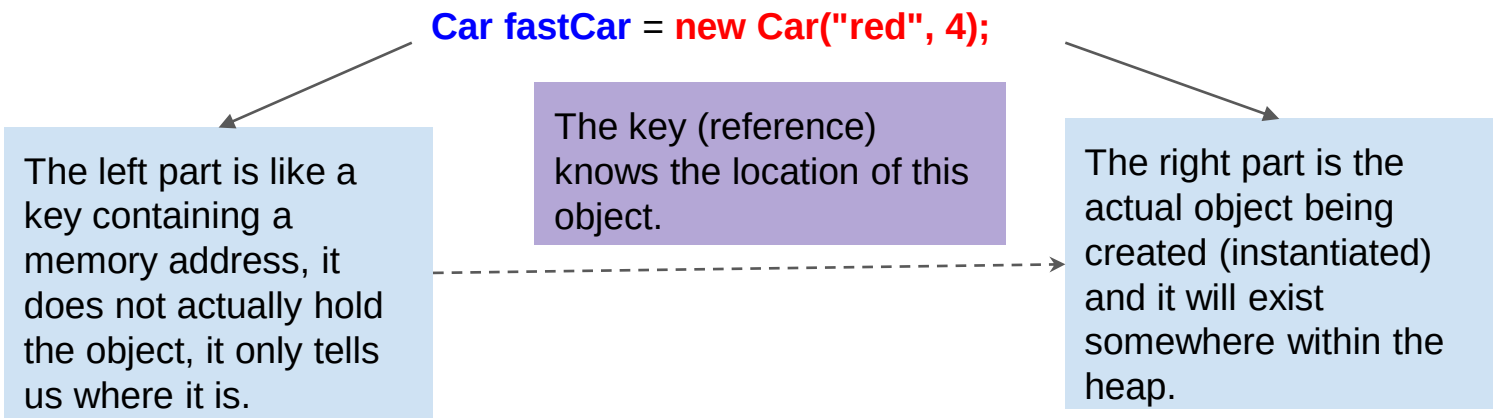
You have encountered this issue before - recall that any method directly called by public static void main had to also be a static.

Garbage Collection

Java Memory Management

Memory management in Java is for the most part an automated process. A hidden process known as “**Garbage Collection**” in the JVM automatically scoops up and destroys objects no longer in use.

To understand this process better, recall the key and locker analogy:



Java Memory Management

Consider the following example:

```
Car fastCar = new Car("red", 4);
Car fastCar2 = new Car("red", 4);

if (fastCar == fastCar2) {
        System.out.println("They are the same car");
}
else {
        System.out.println("Not the same car.");
}

Car fastCar3 = fastCar;
if (fastCar == fastCar3) {
        System.out.println("They are the same car");
}
else {
        System.out.println("Not the same car.");
}
```

These are separate instantiations, each taking up a different part of memory.

Because fastCar and fastCar2 point at different things in the heap, the else will execute.

We have now set fastCar3 and fastCar to point at the same location in memory, they are now therefore referring to the same thing! The program will print “They are the same car.

Java Memory Management

```
Car fastCar = new Car("red", 4);
Car fastCar2 = new Car("blue", 4);
Car fastCar3 = fastCar;
fastCar = null;
```

At this point in time, there are two references pointing to fastCar.

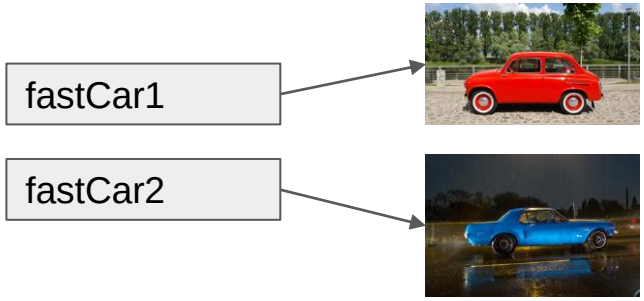
Here, the first reference has been set to null, meaning it’s not pointing at anything anymore.

The red car we instantiated on the first line can still be accessed via fastCar3! But what if fastCar3 also became null?

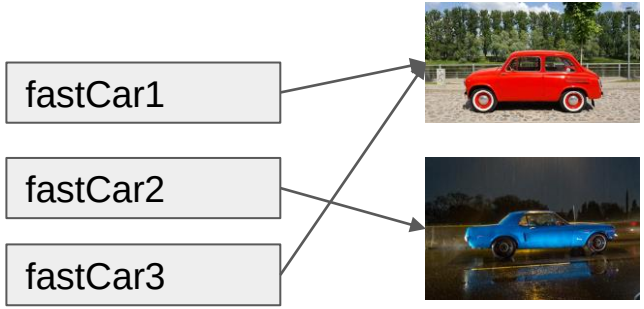
Java Memory Management

Here is a more visual representation of the previous sequence of events:

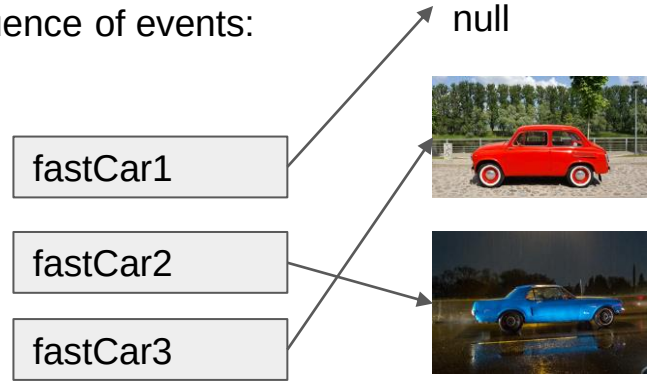
1)



2)



3)



4)