Module 1-10

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

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.

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.

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: Declaration

Static members are properties 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 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 valid way to call a static member.
        thisCar.honkHorn() // Not a valid way to call a static method.

    }
}
```

Static: Assignment

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 carBrand = "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.carBrand = "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.

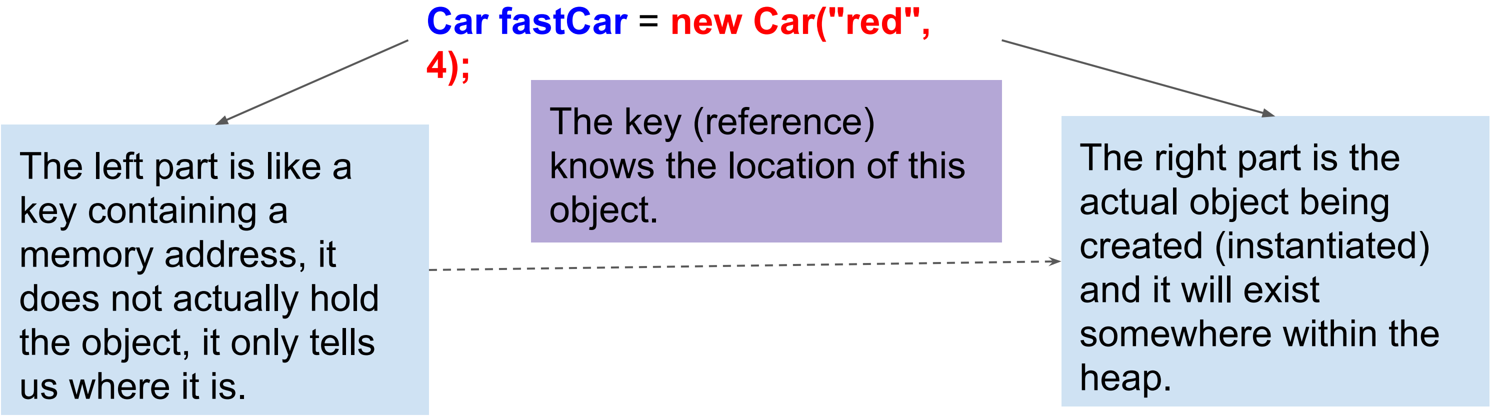
You have encountered this issue before - recall that any method directly called by public static void main had to also be a static.

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:



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

The left part is like a key containing a memory address, it does not actually hold the object, it only tells us where it is.

The key (reference) knows the location of this object.

The right part is the actual object being created (instantiated) and it will exist somewhere within the heap.

Java Memory Management

Consider the following example:

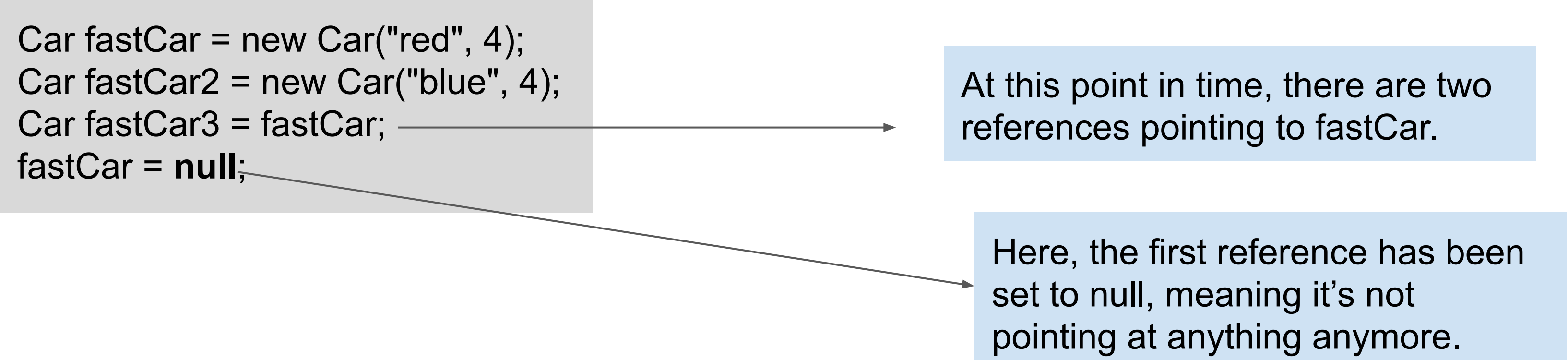
```
Car fastCar = new Car("red", 4);
Car fastCar2 = new Car("red", 4);

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;
```

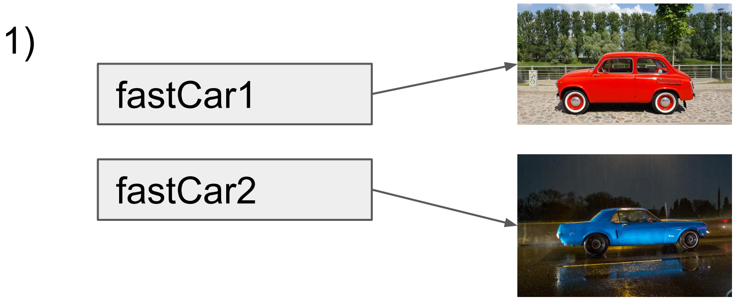
At this point in time, there are two references pointing to fastCar.

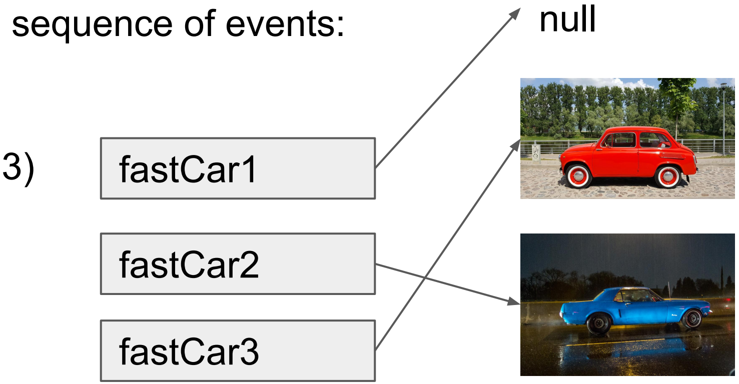
Here, the first reference has been set to null, meaning it’s not pointing at anything anymore.

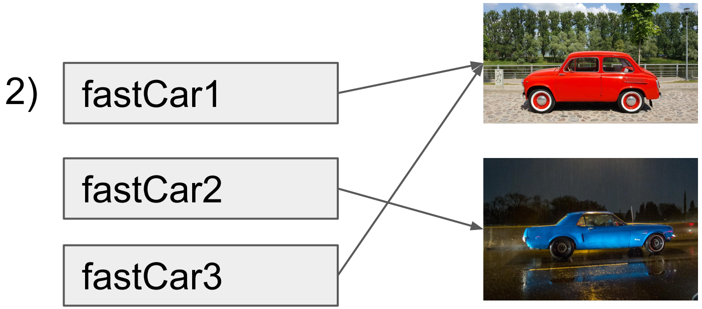
The red car we instantiated on the first line can still be accessed via fastCar3! But what if fastCar3 also became null?

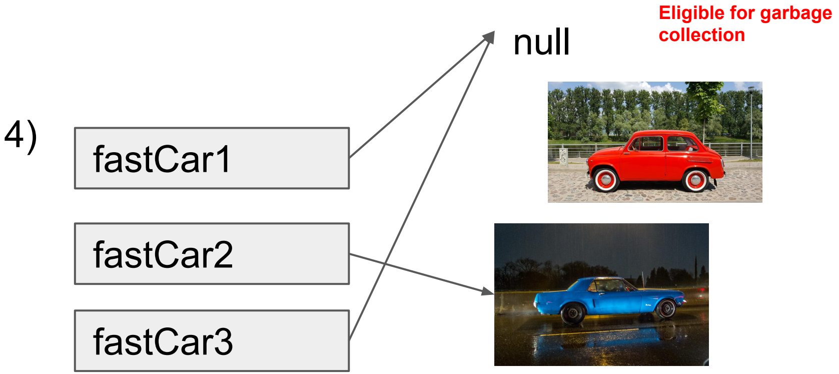
Java Memory Management

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


1)
fastCar1
fastCar2


null
3)
fastCar1
fastCar2
fastCar3


2)
fastCar1
fastCar2
fastCar3


Eligible for garbage collection
null
4)
fastCar1
fastCar2
fastCar3