## Week 8: Topics

- `static` Keyword in Java

## `static` Keyword in Java

- The `static` keyword is used in Java mainly for memory management
  - Used with variable and methods
- It is used for a *constant* variable or a method that is the *same* for every instance of a class
- On the other hand, every object has its own copy of **all** the *instance* variable of a class

## A quick tutorial:

```
public class Counter
{
    int count = 0;

    Counter()
    {
        count++;
        System.out.println(count);
    }

    public static void main(String[] args)
    {
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        Counter c3 = new Counter();
    }
}
```

In this example, we have created an *instance* variable named `count` which is incremented in the constructor.

Each object will have its own copy of the instance variable.

What is the value of the variable `count` in each object?

## `static` Variables in Java

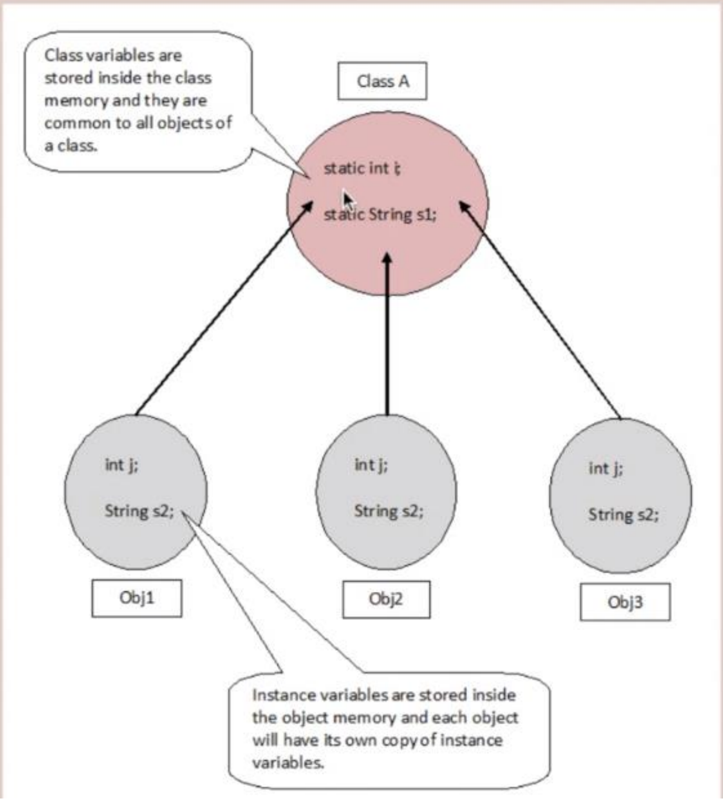
- In certain cases, only one copy of a particular variable should be *shared* by all objects of a class
- A `static` field called a **class variable** is used in such cases

## Class vs Instance Variables

```
public class A
{
    // Class variable
    static int i;

    // Instance variable
    int j;

    // Class variable
    static String s1;

    // Instance variable
    String s2;
}
```

Class variables are common to all objects of a class

## Class vs Instance Variables (cont.)


Class variables are stored inside the class memory and they are common to all objects of a class.
Class A
static int i;
static String s1;
int j;
String s2;
int j;
String s2;
int j;
String s2;
Obj1
Obj2
Obj3
Instance variables are stored inside the object memory and each object will have its own copy of instance variables.

## Stack vs Heap Memory

- Heap memory is used by all the parts of the application
  - Whereas Stack memory is used only by one thread of execution
- Whenever an object is created, it's always stored in the Heap space and Stack memory contains the reference to it
  - Stack memory only contains local primitive variables and reference variables to objects in heap space
- Objects stored in the Heap are globally accessible
  - Whereas Stack memory cannot be accessed by other threads
- Memory management in Stack is done in LIFO manner
  - Whereas it is more complex in Heap memory

## Motivating `static`

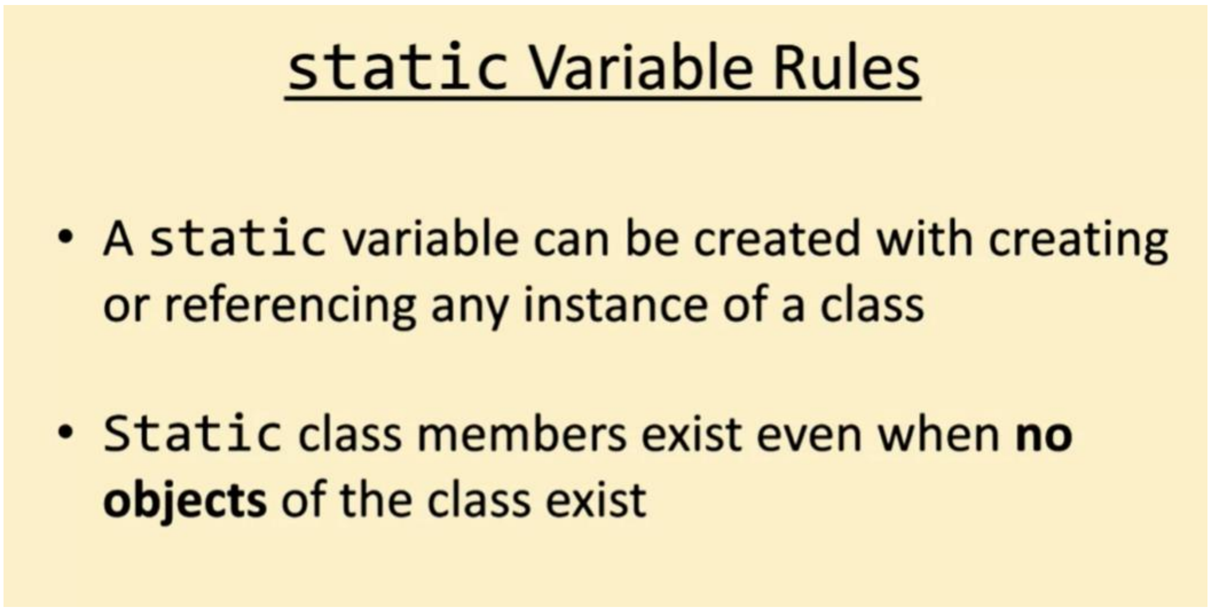
- Suppose we want to store a record of all employees of a company
  - In this case the "employee id" is unique but the "company name" is common for all
- When we create a `static` variable for the company name only one copy of `static companyName` is created
  - Makes the program more efficient
    - i.e., saves memory

## Problem:

Create a class `EmployeeRecord` which stores a record `(employeeId, name, firstname, lastname, companyName)` of all employees in a company.

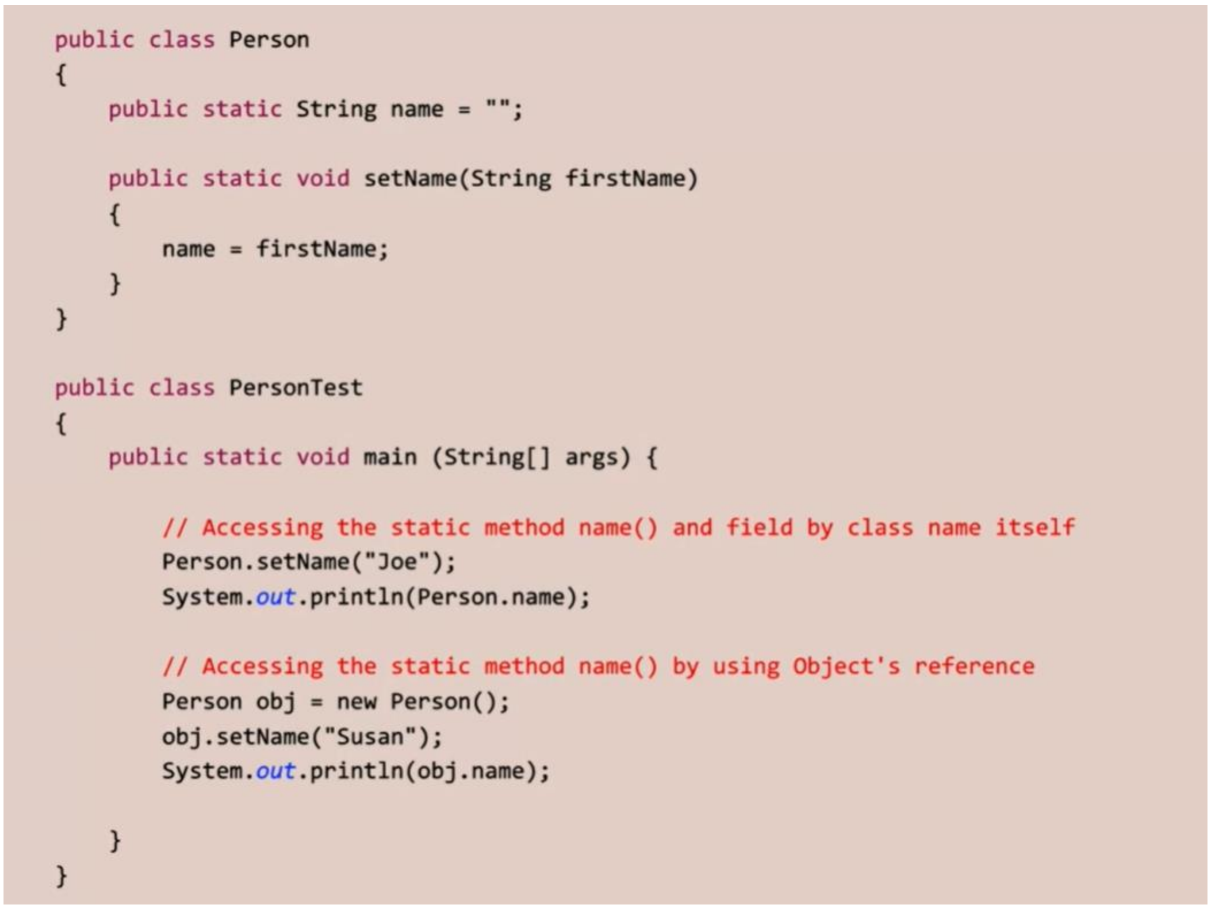
## EmployeeRecord:

```
public class EmployeeRecord
{
    private int employeeId;
    private String firstName;
    private String lastName;
    public static String companyName;
}
```



## `static` Variable Rules

- A `static` variable can be created with creating or referencing any instance of a class
- `Static` class members exist even when **no objects** of the class exist



```java
public class Person
{
    public static String name = "";

    public static void setName(String firstName)
    {
        name = firstName;
    }
}

public class PersonTest
{
    public static void main (String[] args) {

        // Accessing the static method name() and field by class name itself
        Person.setName("Joe");
        System.out.println(Person.name);

        // Accessing the static method name() by using Object's reference
        Person obj = new Person();
        obj.setName("Susan");
        System.out.println(obj.name);

    }
}
```

## static Variable Rules

- A static variable can be created with creating or referencing any instance of a class
- Static class members exist even when **no objects** of the class exist

```
public class Person
{
    public static String name = "";

    public static void setName(String firstName)
    {
        name = firstName;
    }
}

public class PersonTest
{
    public static void main (String[] args) {

        // Accessing the static method name() and field by class name itself
        Person.setName("Joe");
        System.out.println(Person.name);

        // Accessing the static method name() by using Object's reference
        Person obj = new Person();
        obj.setName("Susan");
        System.out.println(obj.name);

    }
}
```

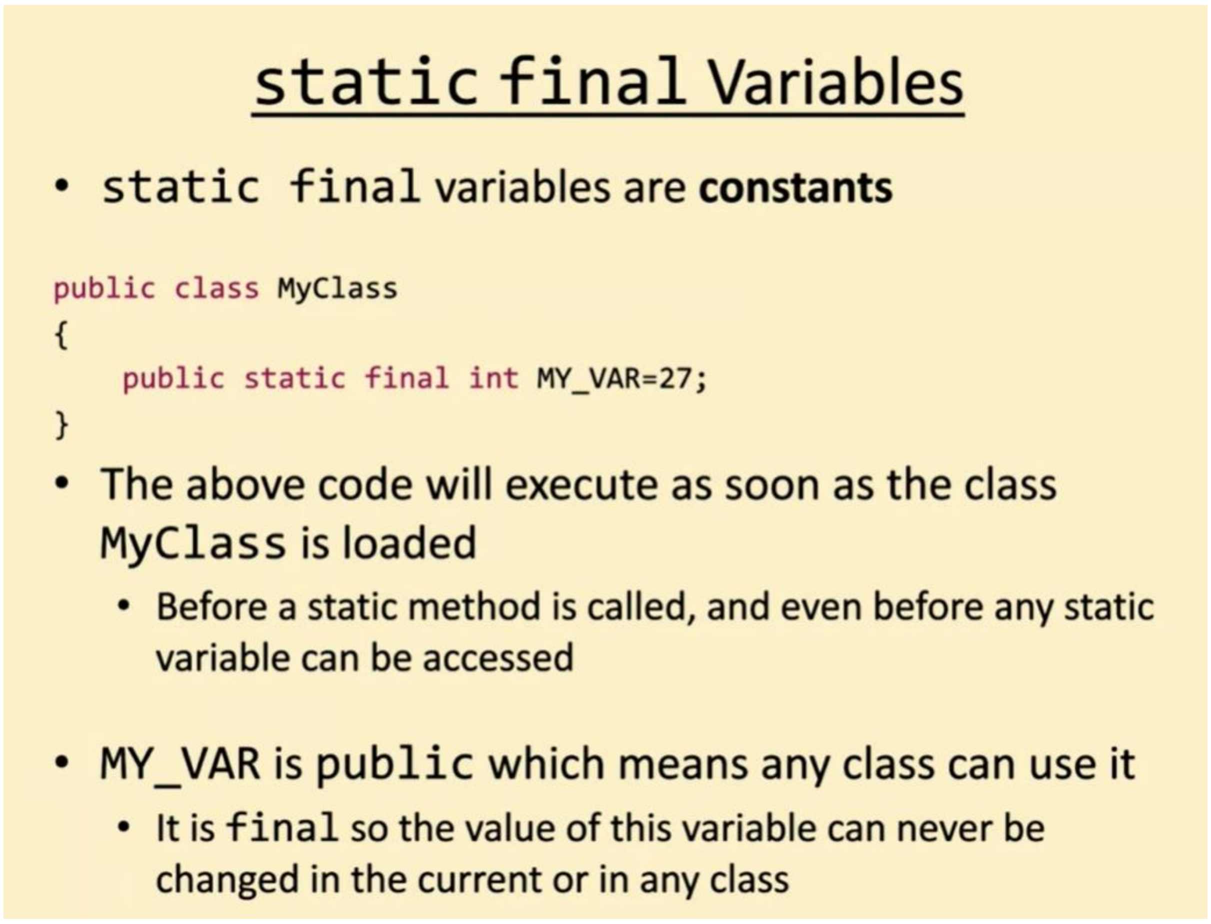
## static Variable Rules

- If there are multiple instances of a class
  - A `static` variable of the class will be shared by all instances of that class
- This will result in only *one* copy

## A quick tutorial:

```
public class Counter
{
    static int count = 0;

    Counter()
    {
        count++;
        System.out.println(count);
    }

    public static void main(String[] args)
    {
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        Counter c3 = new Counter();
    }
}
```

What is the output?

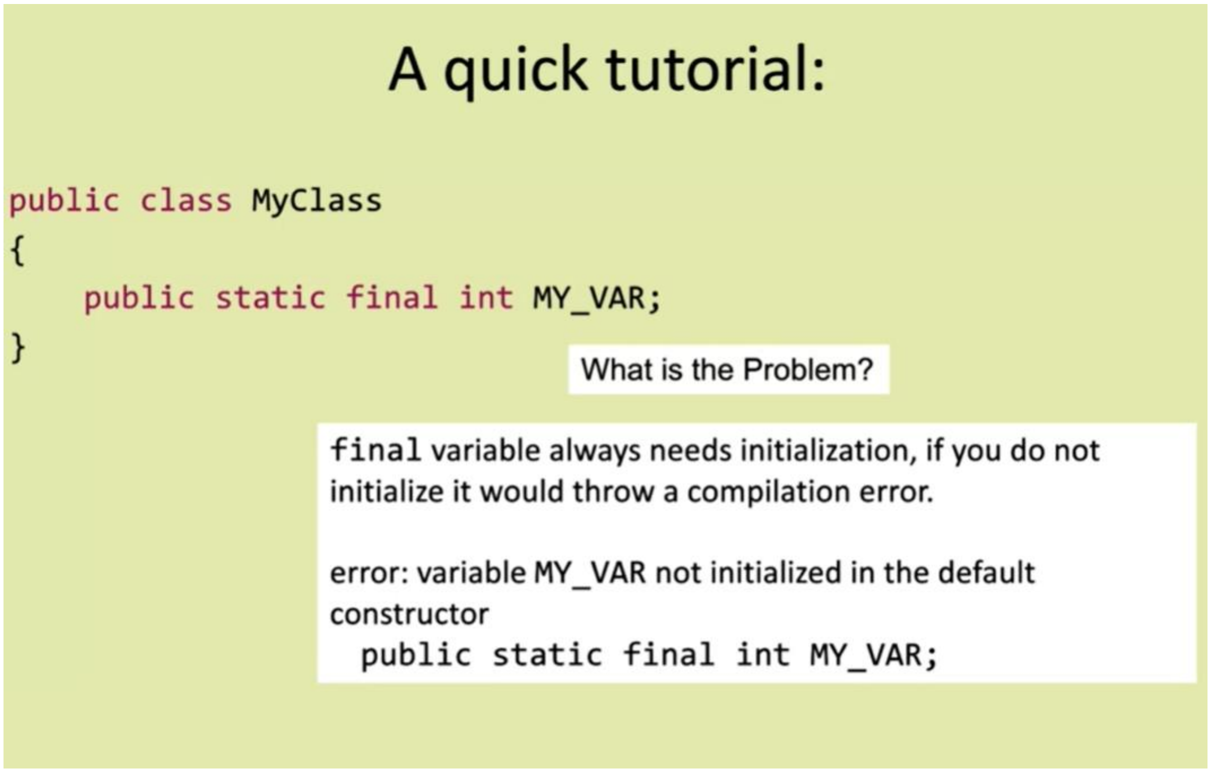


## static final Variables

- `static final` variables are **constants**

```
public class MyClass
{
    public static final int MY_VAR=27;
}
```

- The above code will execute as soon as the class `MyClass` is loaded
  - Before a static method is called, and even before any static variable can be accessed
- MY_VAR is `public` which means any class can use it
  - It is `final` so the value of this variable can never be changed in the current or in any class



## A quick tutorial:

```
public class MyClass
{
    public static final int MY_VAR;
}
```

What is the Problem?

`final` variable always needs initialization, if you do not initialize it would throw a compilation error.

error: variable MY_VAR not initialized in the default constructor

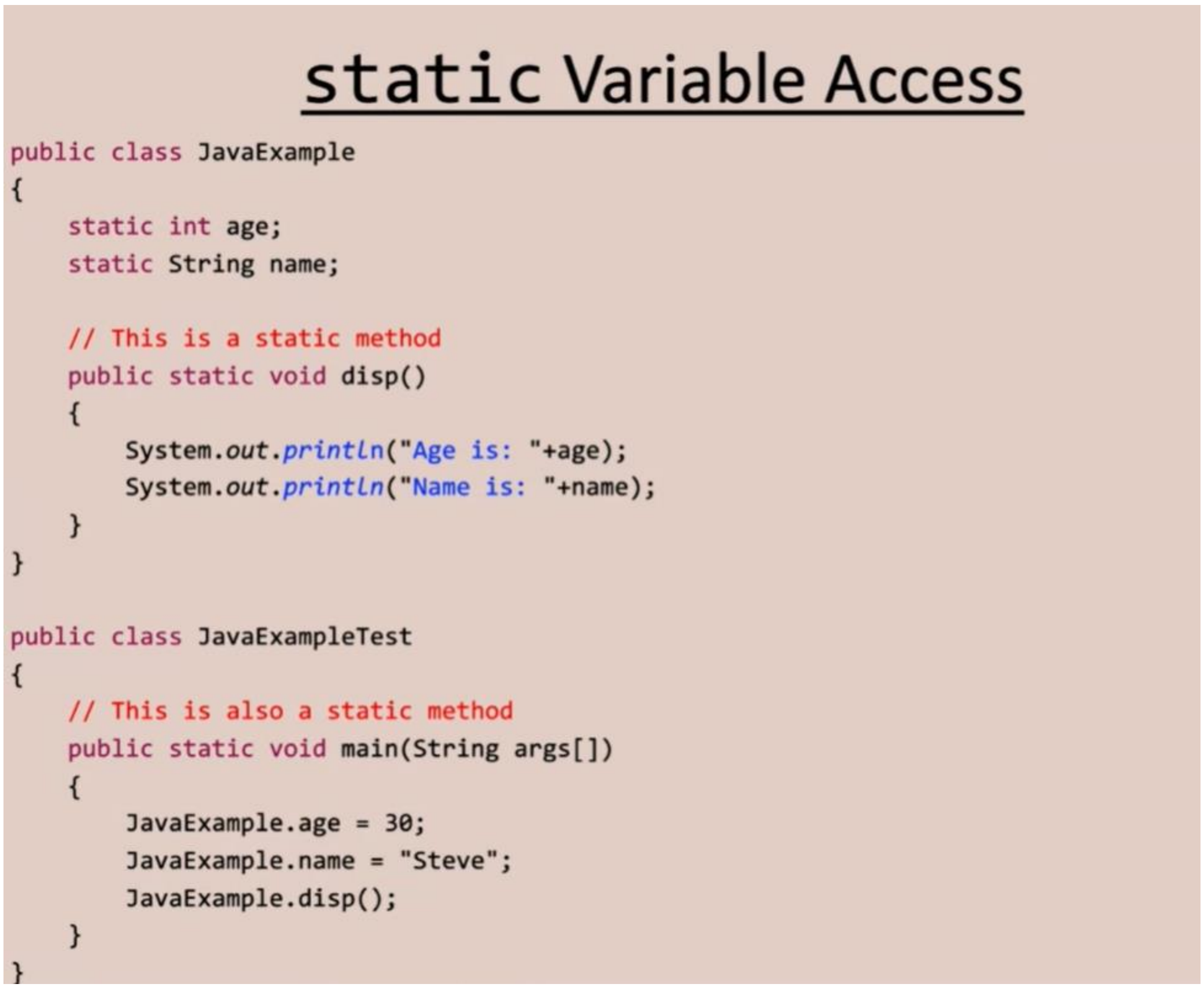
```
  public static final int MY_VAR;
```

## `static` Methods in Java

- A static method *belongs to the class* rather than an object of the class
- A `static` method can be invoked without creating an instance/object of a class
  - e.g. `public static void main (String[] args)`

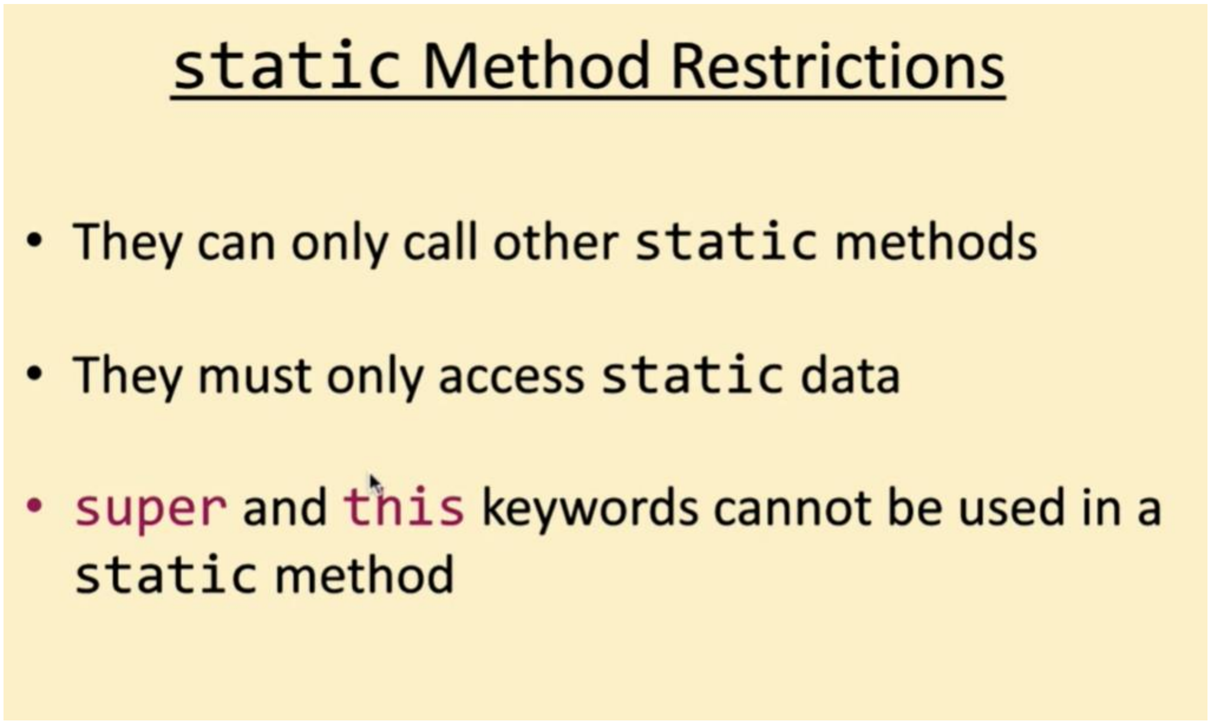
## `static` Methods in Java (cont.)

- A `static` method can access a `static` variable and change the value of it
  - `<<ClassName>>.<<VariableName>>`
- A `static` method can be directly called by using the class name
  - `<<ClassName>>.<<MethodName>>`



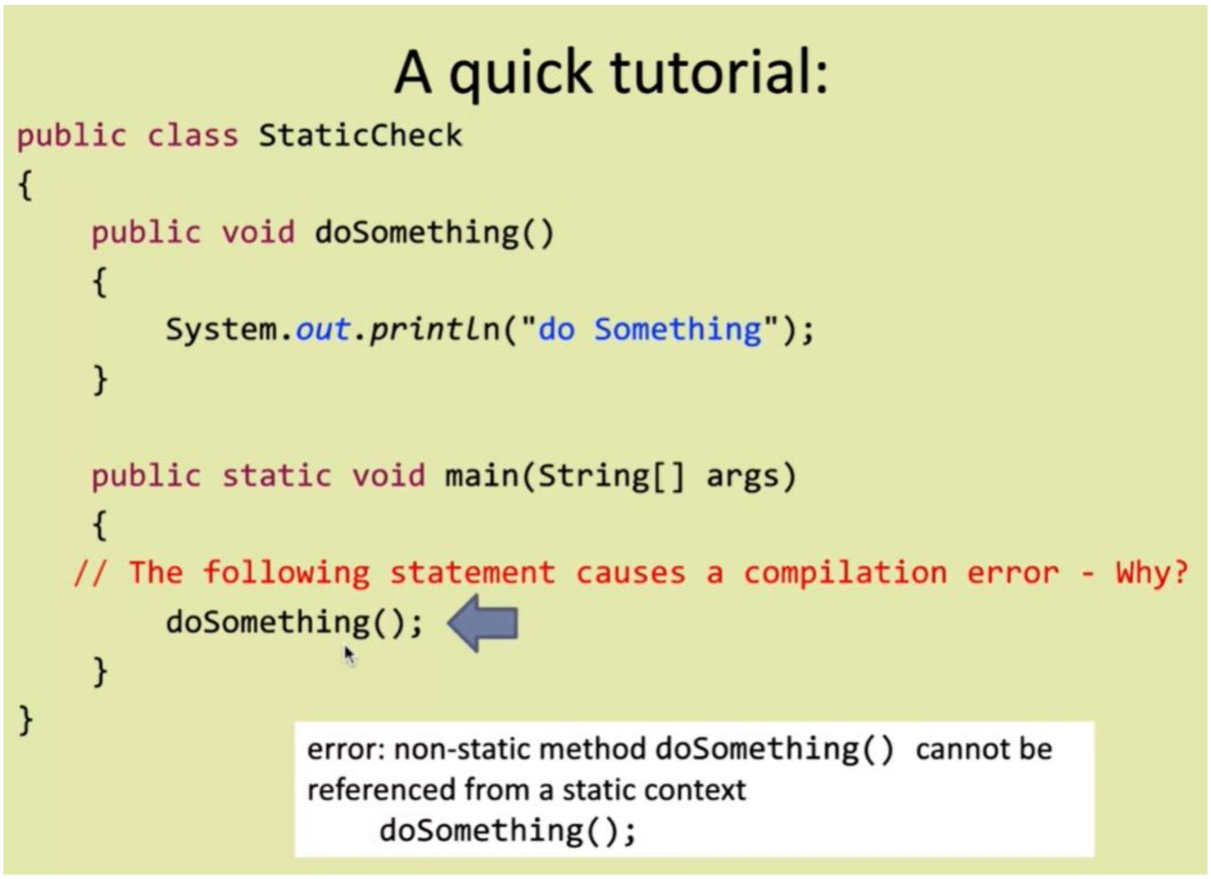
## static Variable Access

```
public class JavaExample
{
    static int age;
    static String name;

    // This is a static method
    public static void disp()
    {
        System.out.println("Age is: "+age);
        System.out.println("Name is: "+name);
    }
}

public class JavaExampleTest
{
    // This is also a static method
    public static void main(String args[])
    {
        JavaExample.age = 30;
        JavaExample.name = "Steve";
        JavaExample.disp();
    }
}
```



## static Method Restrictions

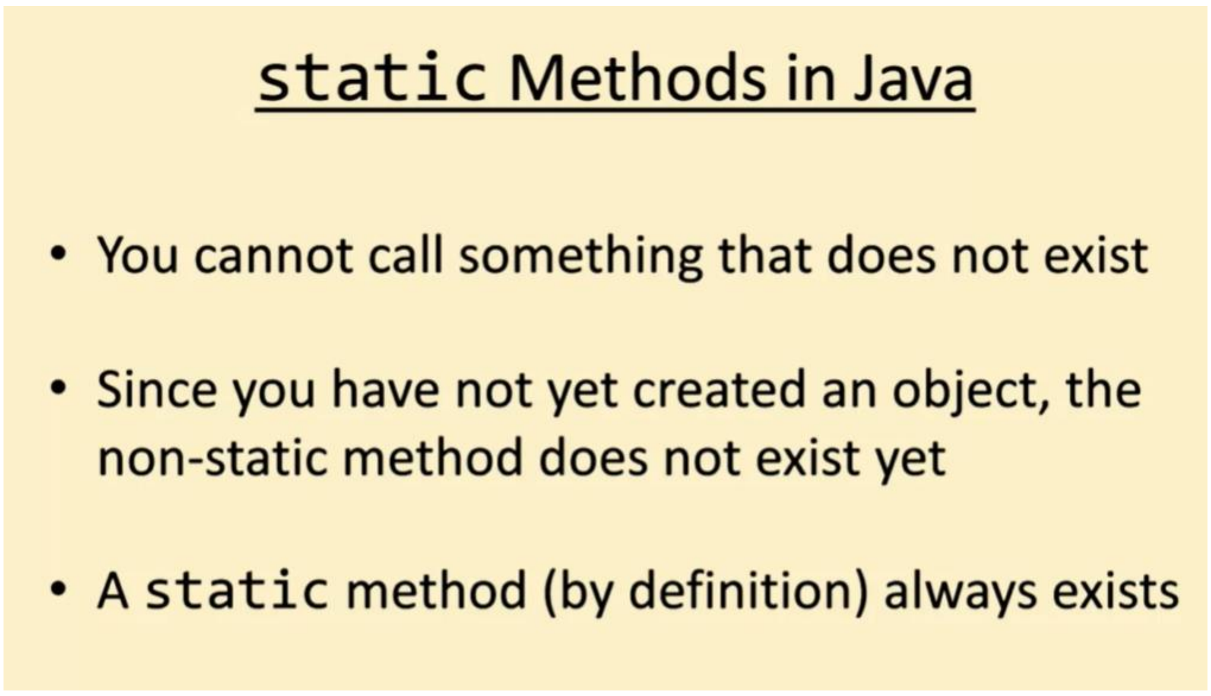
- They can only call other `static` methods
- They must only access `static` data
- `super` and `this` keywords cannot be used in a `static` method



## A quick tutorial:

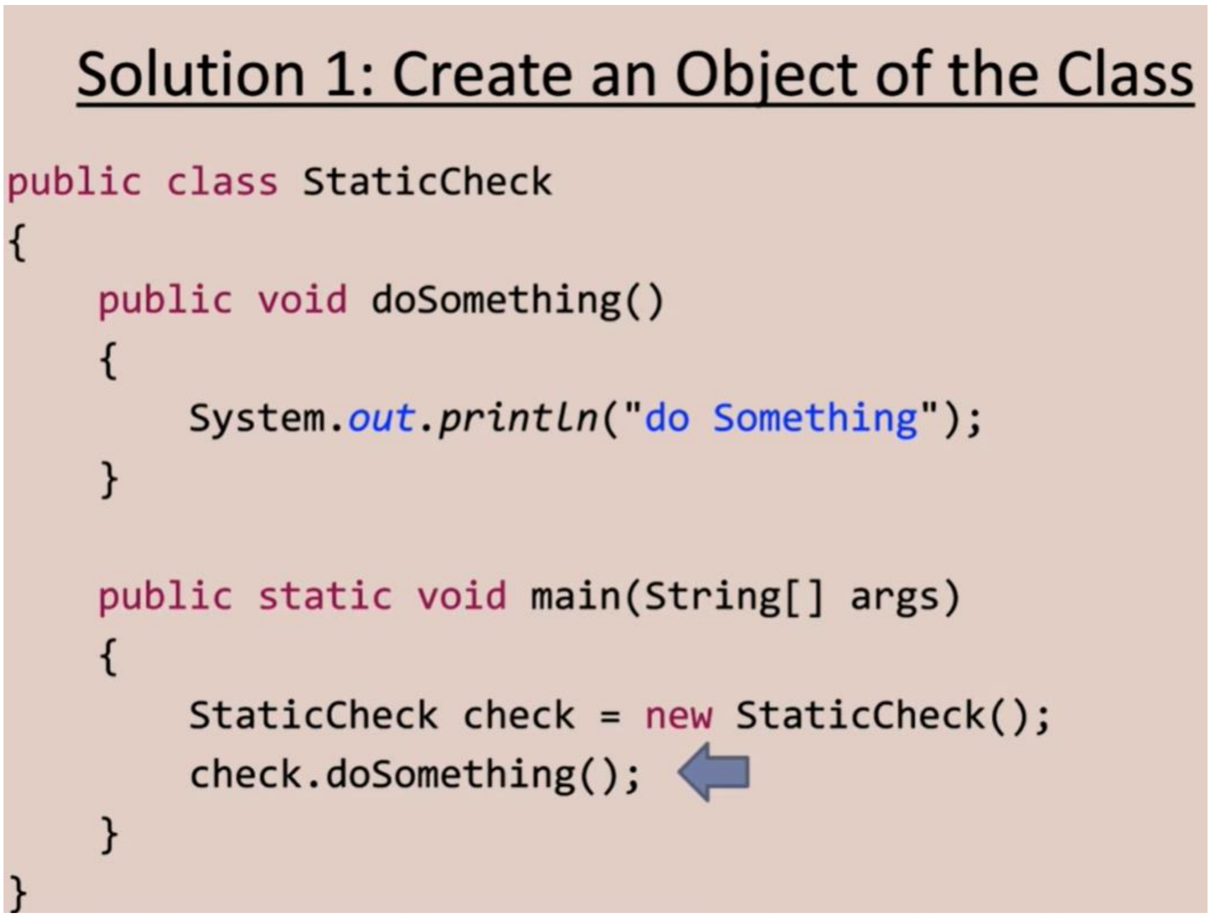
```java
public class StaticCheck
{
    public void doSomething()
    {
        System.out.println("do Something");
    }

    public static void main(String[] args)
    {
   // The following statement causes a compilation error - Why?
        doSomething();
    }
}
```

```
error: non-static method doSomething()  cannot be
referenced from a static context
     doSomething();
```



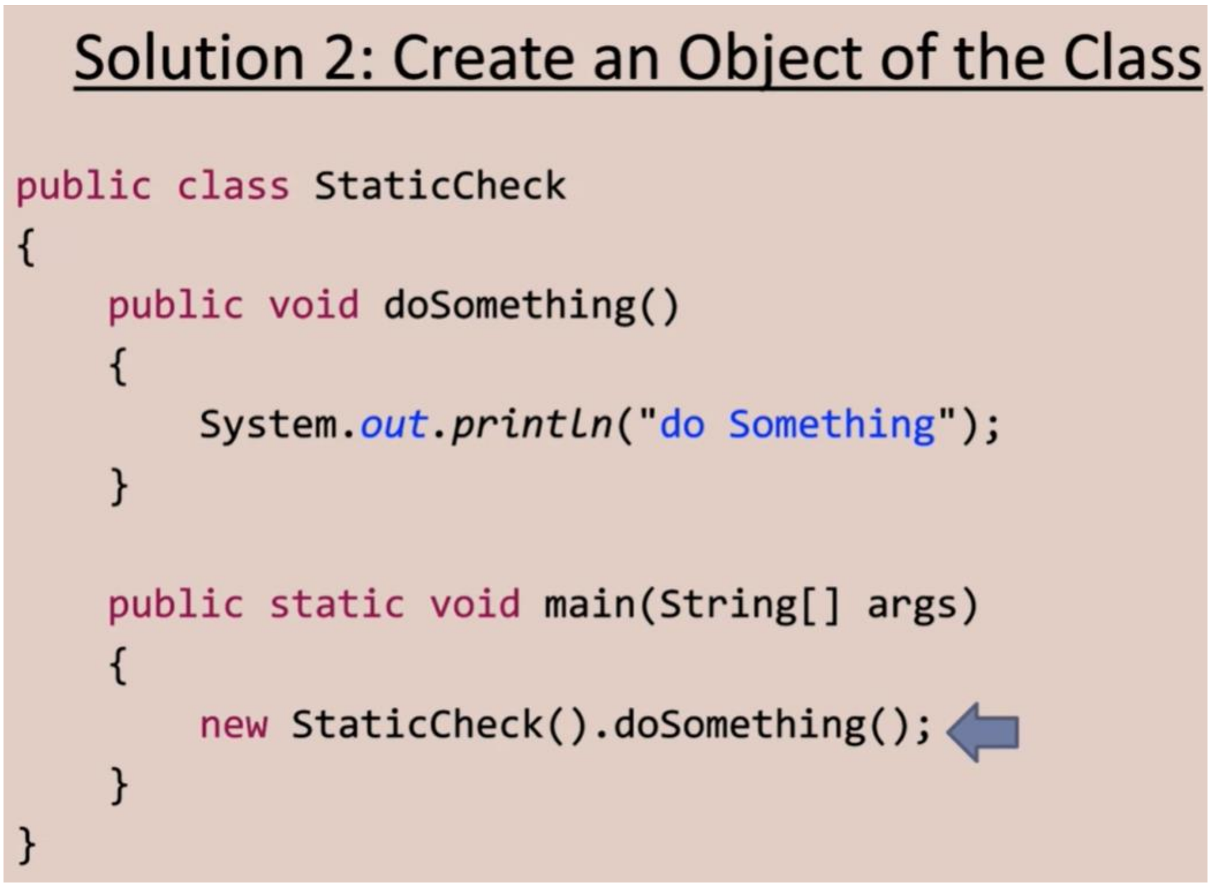
## static Methods in Java

- You cannot call something that does not exist
- Since you have not yet created an object, the non-static method does not exist yet
- A static method (by definition) always exists



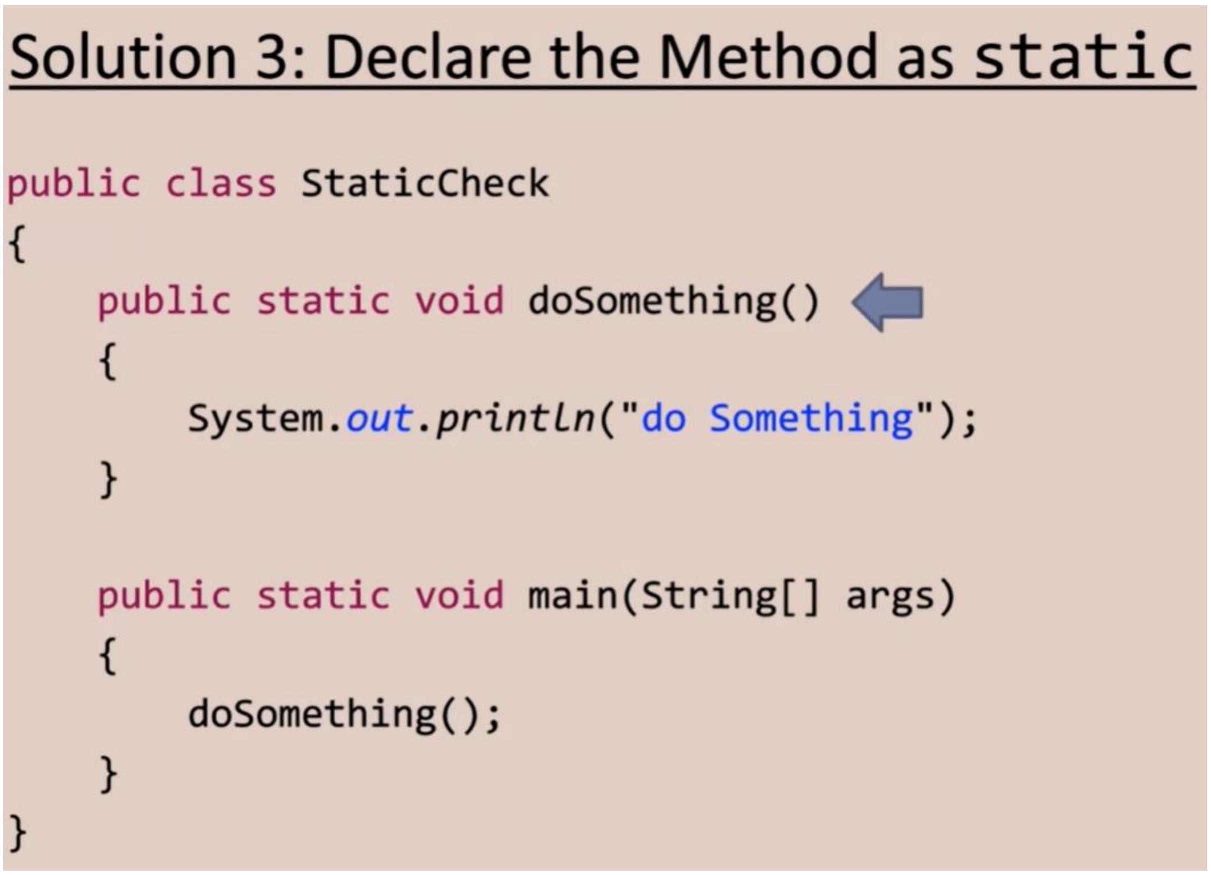
## Solution 1: Create an Object of the Class

```java
public class StaticCheck
{
    public void doSomething()
    {
        System.out.println("do Something");
    }

    public static void main(String[] args)
    {
        StaticCheck check = new StaticCheck();
        check.doSomething();
    }
}
```



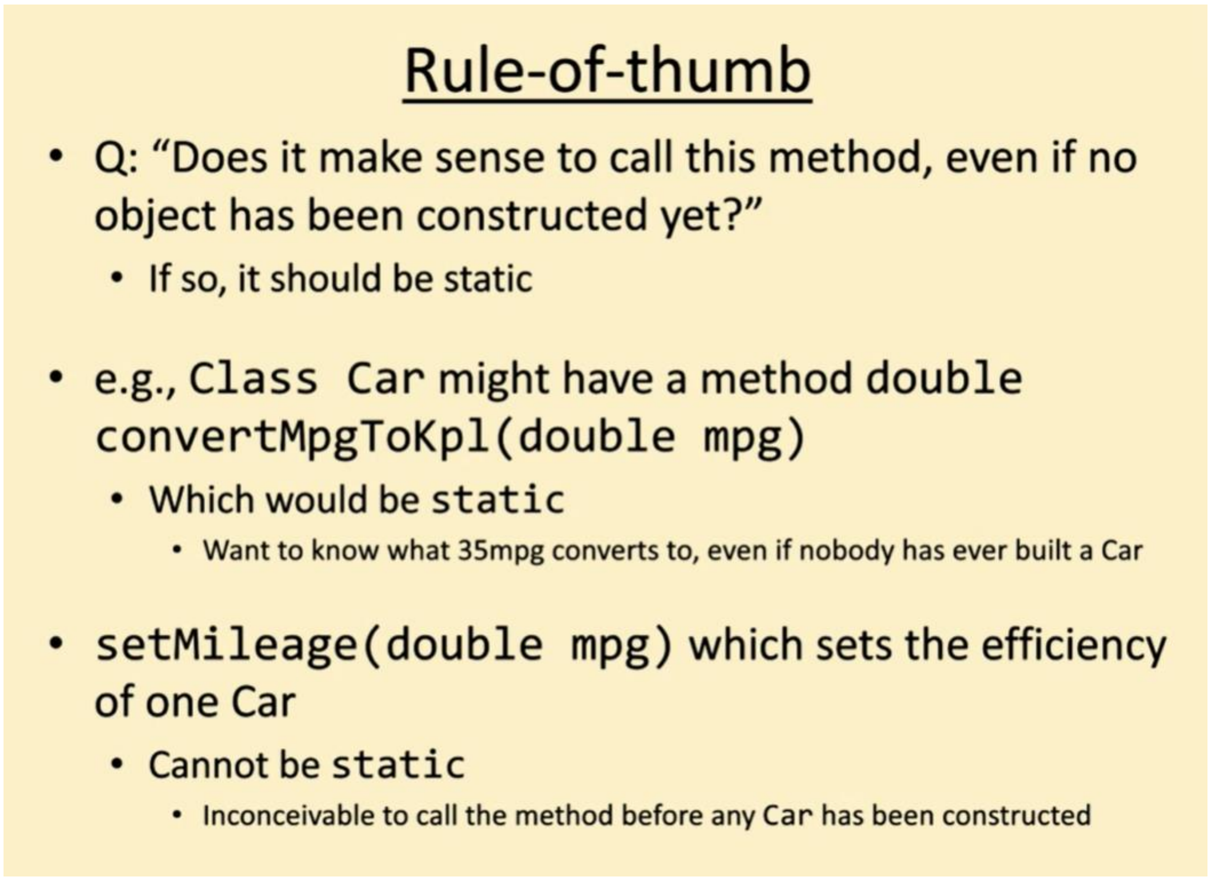
## Solution 2: Create an Object of the Class

```java
public class StaticCheck
{
    public void doSomething()
    {
        System.out.println("do Something");
    }

    public static void main(String[] args)
    {
        new StaticCheck().doSomething();
    }
}
```



## Solution 3: Declare the Method as `static`

```java
public class StaticCheck
{
    public static void doSomething()
    {
        System.out.println("do Something");
    }

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



## Rule-of-thumb

- Q: "Does it make sense to call this method, even if no object has been constructed yet?"
  - If so, it should be static
- e.g., `Class Car` might have a method `double convertMpgToKpl(double mpg)`
  - Which would be `static`
    - Want to know what 35mpg converts to, even if nobody has ever built a Car
- `setMileage(double mpg)` which sets the efficiency of one Car
  - Cannot be `static`
    - Inconceivable to call the method before any Car has been constructed

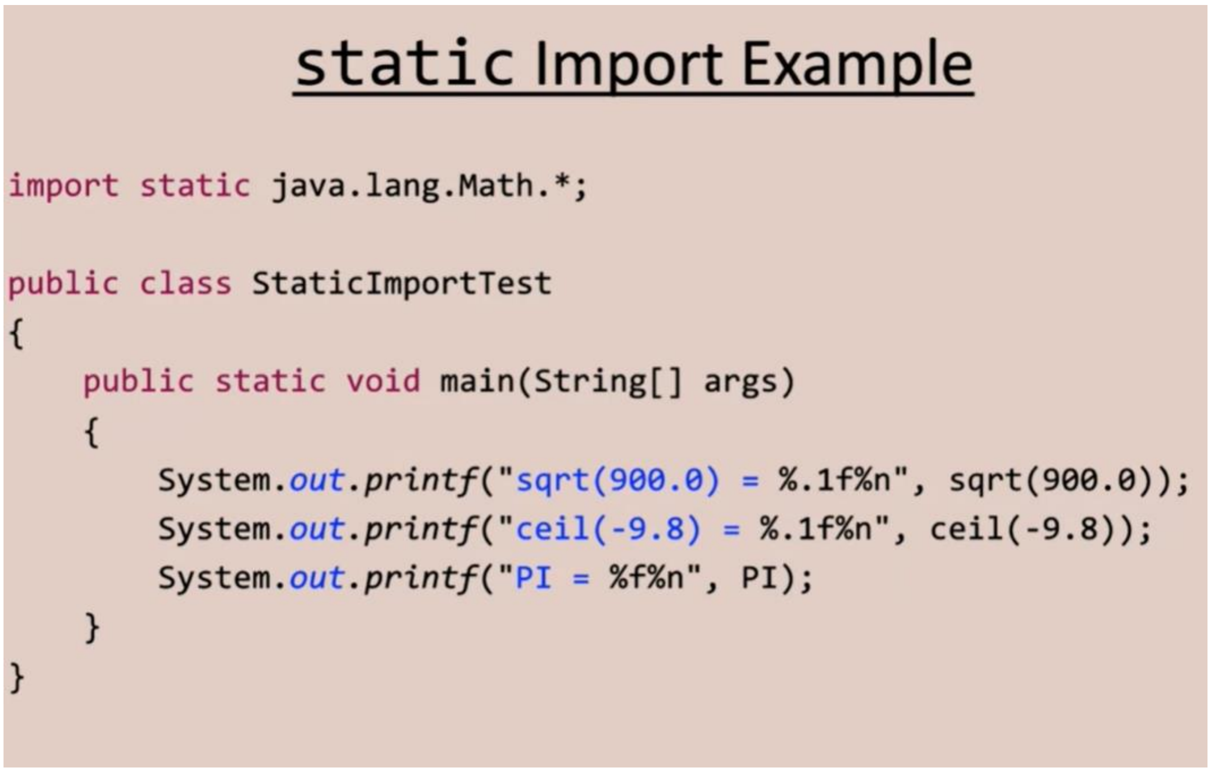
## `static` Import

- A `static` import declaration enables you to import the `static` members of a class or an interface
- You can access them via their *unqualified names* in your class
  - i.e. the class name and a dot (.) are not required when using an imported `static` member

## Problem:

Create a class that calculates the `Square Root`, `Ceiling` of a given value and the value of PI to a specified number of decimal places.

Make use of the static java.lang.Math library and call the in-built functions via their unqualified names.



## static Import Example

```java
import static java.lang.Math.*;

public class StaticImportTest
{
    public static void main(String[] args)
    {
        System.out.printf("sqrt(900.0) = %.1f%n", sqrt(900.0));
        System.out.printf("ceil(-9.8) = %.1f%n", ceil(-9.8));
        System.out.printf("PI = %f%n", PI);
    }
}
```