

Lecture 6: Encapsulation, Visibility Modifiers and Methods

CSC 1214: Object-Oriented Programming

Outline

- Encapsulation
- Visibility Modifiers
- Method Declarations

Outline

- Encapsulation
- Visibility Modifiers
- Method Declarations

Encapsulation

- **Encapsulation** is a mechanism that is used to restrict access to an object's data and methods. Also known as *information hiding*
- Until now we haven't been concerned about how object's data can be exposed to the external world. Consider our Car example below:

```
class Car {  
    String numberPlate; // e.g. "UBC 080A"  
    double speed = 0.0; // in kilometers per hour  
    double maxSpeed; // in kilometers per hour  
    int year;  
    double oldCarSpeedLimit = 180.0;  
}
```

```
class CarDriver {  
    public static void main(String args[]){  
        Car myCar = new Car();  
        myCar.speed = 380.0;  
    }  
}
```



No
encapsulation!

Encapsulation

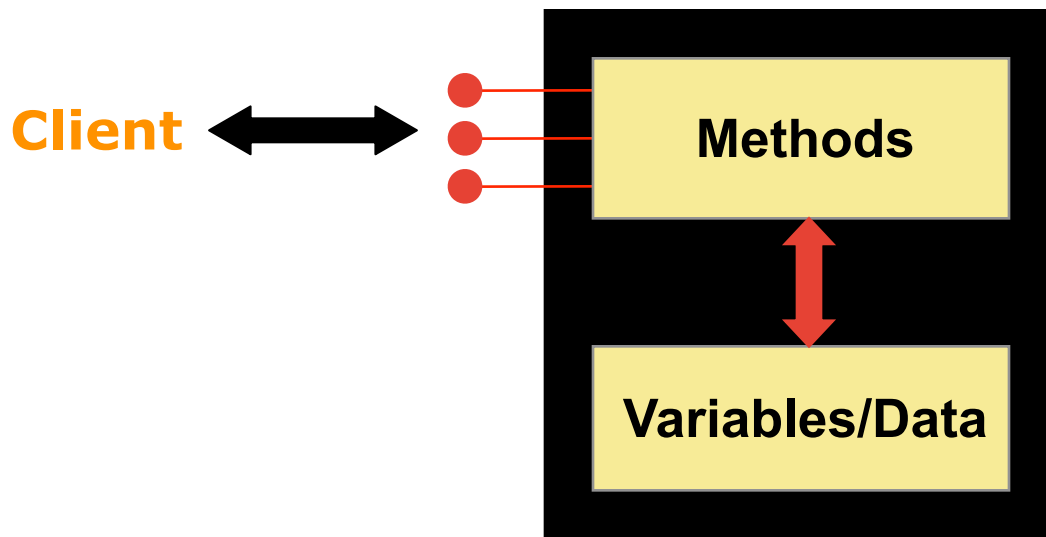
- In object-oriented programming an object has two views:
 - **internal** - the details of the variables and methods of the class that defines it
 - **external** - the services that an object provides and how the object interacts with the rest of the system/world
- From the external view, an object is an *encapsulated entity*, providing a set of specific services
- These services define the interface to the object

Encapsulation

- One object (called the *client*) may use another object for the services it provides
- The client of an object may request its services (call its methods), but it should not have to be aware of how those services are accomplished
- Any changes to the object's state (its variables) should be made by that object's methods
- We should make it difficult, if not impossible, for a client to access an object's variables directly

Encapsulation

- An encapsulated object can be thought of as a black box -- its inner workings are hidden from the client
- The client invokes the interface methods of the object, which manages the instance data



Visibility Modifiers

- In Java, we accomplish encapsulation through the appropriate use of visibility modifiers
- A modifier is a Java reserved word that specifies particular characteristics of a method or data
- Java has three visibility modifiers: **public**, **protected**, and **private**
- The **protected** modifier involves inheritance, which we will discuss later

Visibility Modifiers

- Members of a class that are declared with **public visibility** can be *referenced anywhere*.
- A class may also be designated public, which means that any other class can use the class definition. The name of a public class must match the filename, thus a file can have only one public class.
- Members of a class that are declared with **private visibility** can be *referenced only within that class*
- Members declared without a visibility modifier have **default visibility** and can be *referenced by any class in the same package*. We will discuss packages in Java later. 9

Using Visibility Modifiers to Enforce Encapsulation

Private fields can only be referenced from within this class

```
class Car {  
    private String numberPlate; // e.g. "UBC 080A"  
    private double speed = 0.0; // in kilometers per hour  
    private double maxSpeed; // in kilometers per hour  
    private int year;  
    private double oldCarSpeedLimit = 180.0;  
}
```

```
class CarDriver {  
    public static void main(String args[]) {  
        Car myCar = new Car();  
        myCar.speed = 380.0;  
    }  
}
```

Error!: The variable **speed** has private access in **Car**

Visibility Modifiers

- Public variables **violate encapsulation** because they allow the client to “reach in” and modify the values directly
- Therefore instance variables should not be declared with public visibility
- Though, it is acceptable to give a constant variable public visibility, which allows it to be used outside of the class
- Public constants do not violate encapsulation because, although the client can access it, its value cannot be changed

Visibility Modifiers

- Methods that provide the object's services are declared with public visibility so that they can be invoked by clients
- Public methods are also called **service methods**
- A method created simply to assist a service method is called a **support method**
- Since a support method is not intended to be called by a client, it should **not** be declared with public visibility

Visibility Modifiers

```
class Car {  
    private String numberPlate; // e.g. "UBC 080A"  
    private double speed = 0.0; // in kilometers per hour  
    private double maxSpeed; // in kilometers per hour  
    private int year;  
    private double oldCarSpeedLimit = 180.0;  
  
    public void setMaxSpeed (double newMaxSpeed){  
        if(this.isBefore90s() && newMaxSpeed>oldCarSpeedLimit){  
            System.out.println("Speed limit for cars older than 1990 is 180 KM/h");  
        } else {  
            this.maxSpeed = newMaxSpeed;  
        }  
    }  
  
    public double getSpeed () {  
        return this.speed;  
    }  
  
    private Boolean isBefore90s () {  
        return year < 1990;  
    }  
}
```

Service Methods

Support Method.
Private methods can only be accessed from within the class

Visibility Modifiers

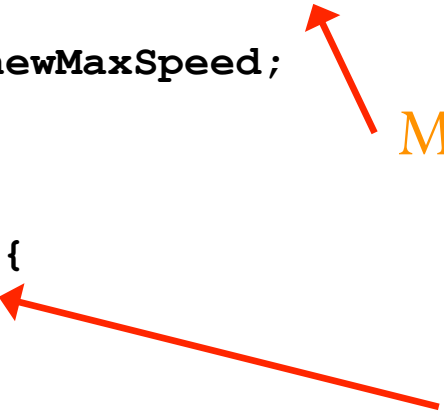
	<code>public</code>	<code>private</code>
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

Accessors and Mutators

- Because instance data is private, a class usually provides services to access and modify data values
- An accessor method returns the current value of a variable
- A mutator method changes the value of a variable
- The names of accessor and mutator methods take the form getX and setX, respectively, where X is the name of the variable
- They are sometimes called “getters” and “setters”

Accessors and Mutators

```
class Car {  
    private String numberPlate; // e.g. "UBC 080A"  
    private double speed = 0.0; // in kilometers per hour  
    private double maxSpeed; // in kilometers per hour  
    private int year;  
    private double oldCarSpeedLimit = 180.0;  
  
    public void setMaxSpeed (double newMaxSpeed){  
        if(this.isBefore90s() && newMaxSpeed>oldCarSpeedLimit){  
            System.out.println("Speed limit for cars older than 1990 is 180 KM/h");  
        } else {  
            this.maxSpeed = newMaxSpeed;  
        }  
    }  
  
    public double getSpeed () {  
        return this.speed;  
    }  
  
    private Boolean isBefore90s () {  
        return year < 1990;  
    }  
}
```



Mutator Method

Accessor Method

Accessors and Mutators

- The use of mutators gives the class designer the ability to restrict a client's options to modify an object's state
- A mutator is often designed so that the values of variables can be set only within particular limits
- For example, the setMaxSpeed of car should restrict the speed of cars older than 1990 to 180Km/hr.

Outline

- Encapsulation
- Visibility Modifiers
- Method Declarations

Method Declarations in Java

- A method declaration specifies the code that will be executed when the method is invoked (called)
- A method declaration begins with a method header

`int sum (int num1, int num2)`

The diagram shows the method header `int sum (int num1, int num2)` with three red arrows pointing to its components: one from the text 'return type' to the `int` before `sum`, one from the text 'method name' to `sum`, and one from the text 'parameter list' to the parentheses and contents `(int num1, int num2)`. A red curly brace is positioned under the parameter list.

return type

method name

parameter list

The parameter list specifies the type and name of each parameter

The name of a parameter in the method declaration is called a *formal parameter*

Method Body

- The method header is followed by the method body

```
int sum (int num1, int num2)
{
    int result = num1 + num2;
    return result;
}
```

**The return expression
must be consistent with
the return type**

**result
is local data**

**It is created each
time the method is
called, and is
destroyed when it
finishes executing**

The Return Statement

- The return type of a method indicates the type of value that the method sends back to the calling location
- A method that does not return a value has a **void** return type

```
public void setMaxSpeed (double newMaxSpeed) { ... }
```

- A return statement specifies the value that will be returned


```
return expression;
```

- Its **expression** must conform to the return type

Method Invocation

- When a method is called, the actual parameters in the invocation are copied into (bound to) the formal parameters in the method header

```
int total = obj.sum (10, 15);
```



```
int sum (int num1, int num2)
{
    int result = num1 + num2;
    return result;
}
```

The diagram illustrates the binding of actual parameters to formal parameters. Two red arrows originate from the values '10' and '15' in the invocation `obj.sum (10, 15);`. The first arrow points to the formal parameter `int num1` in the method signature `int sum (int num1, int num2)`. The second arrow points to the formal parameter `int num2` in the same signature.

Method Control Flow

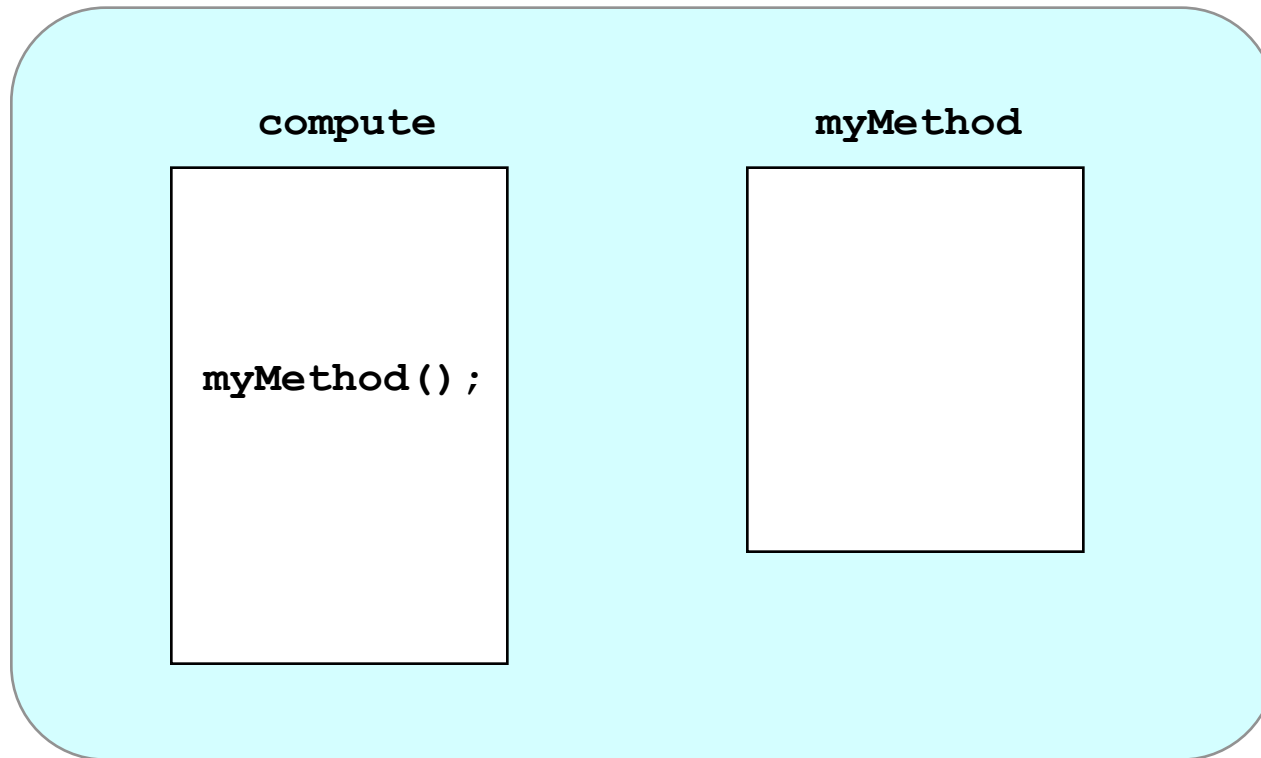
- When a method is invoked, the flow of control jumps to the method and executes its code
- When complete, the flow returns to the place where the method was called and continues
- The invocation may or may not return a value, depending on how the method is defined

Method Control Flow

Method Control Flow

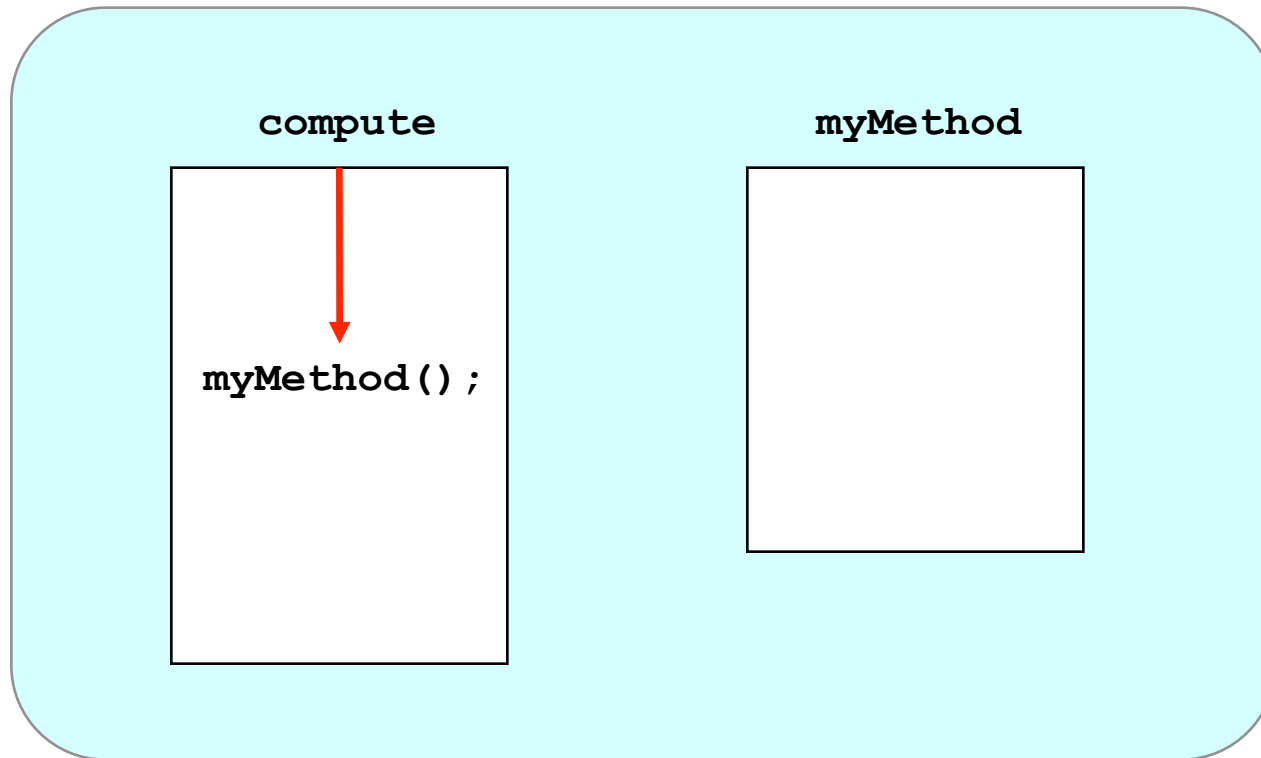
- If the called method is in the same class, only the method name is needed

Method Control Flow



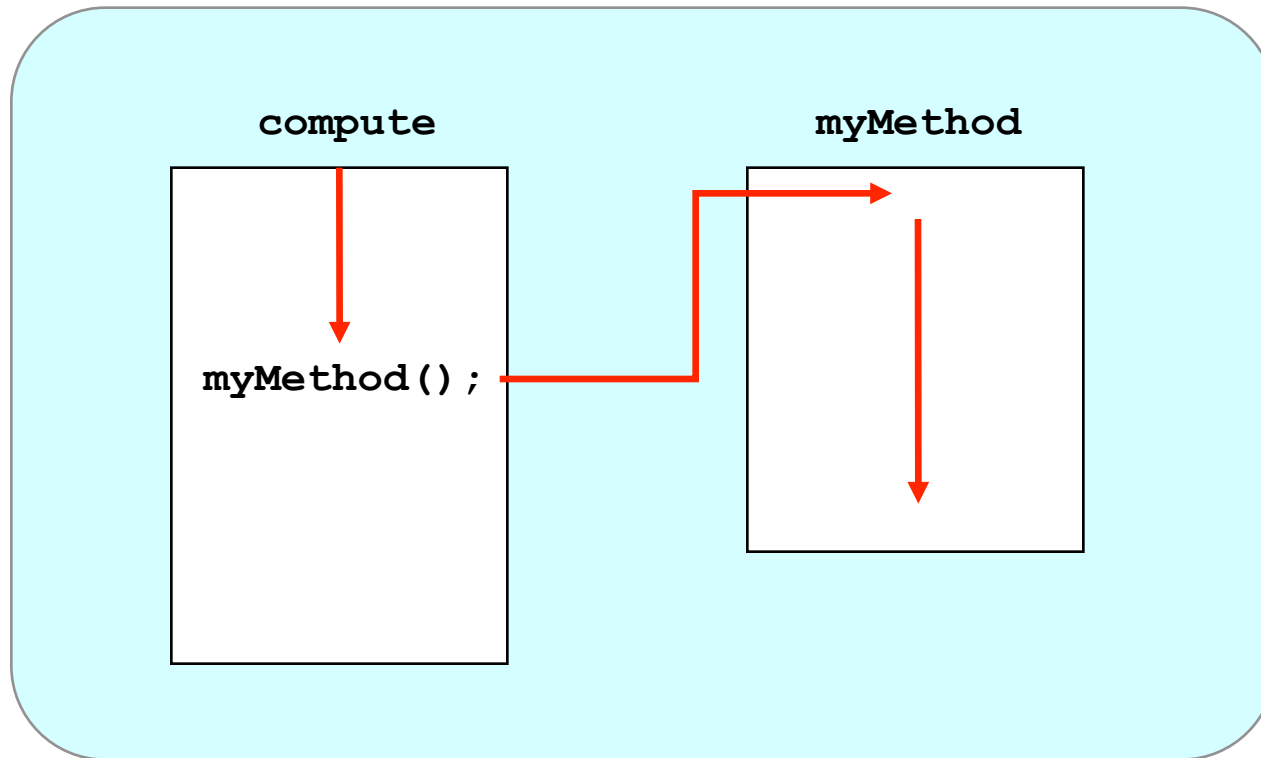
- If the called method is in the same class, only the method name is needed

Method Control Flow



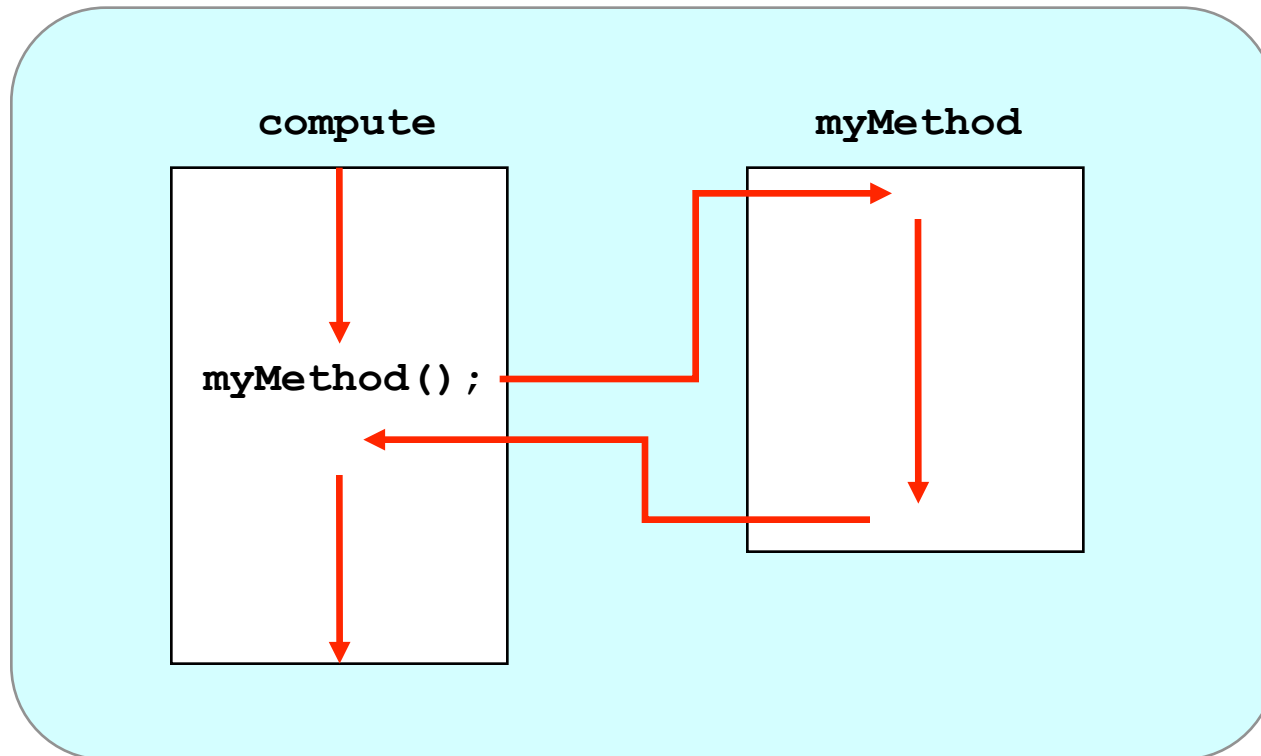
- If the called method is in the same class, only the method name is needed

Method Control Flow



- If the called method is in the same class, only the method name is needed

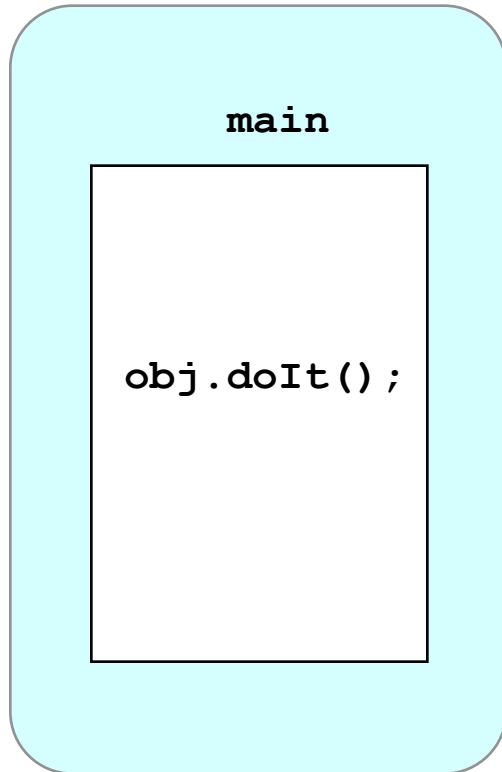
Method Control Flow



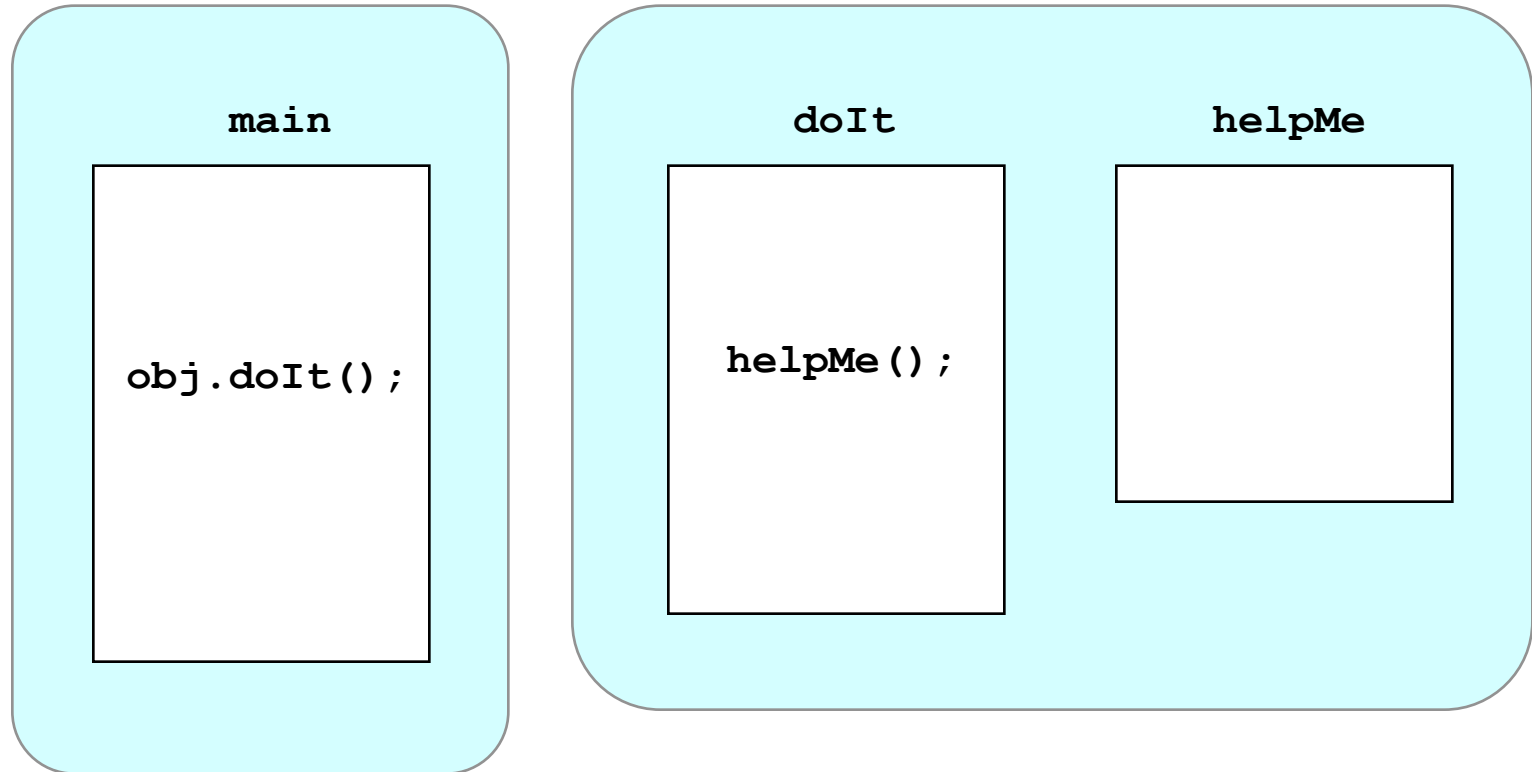
- If the called method is in the same class, only the method name is needed

Method Control Flow

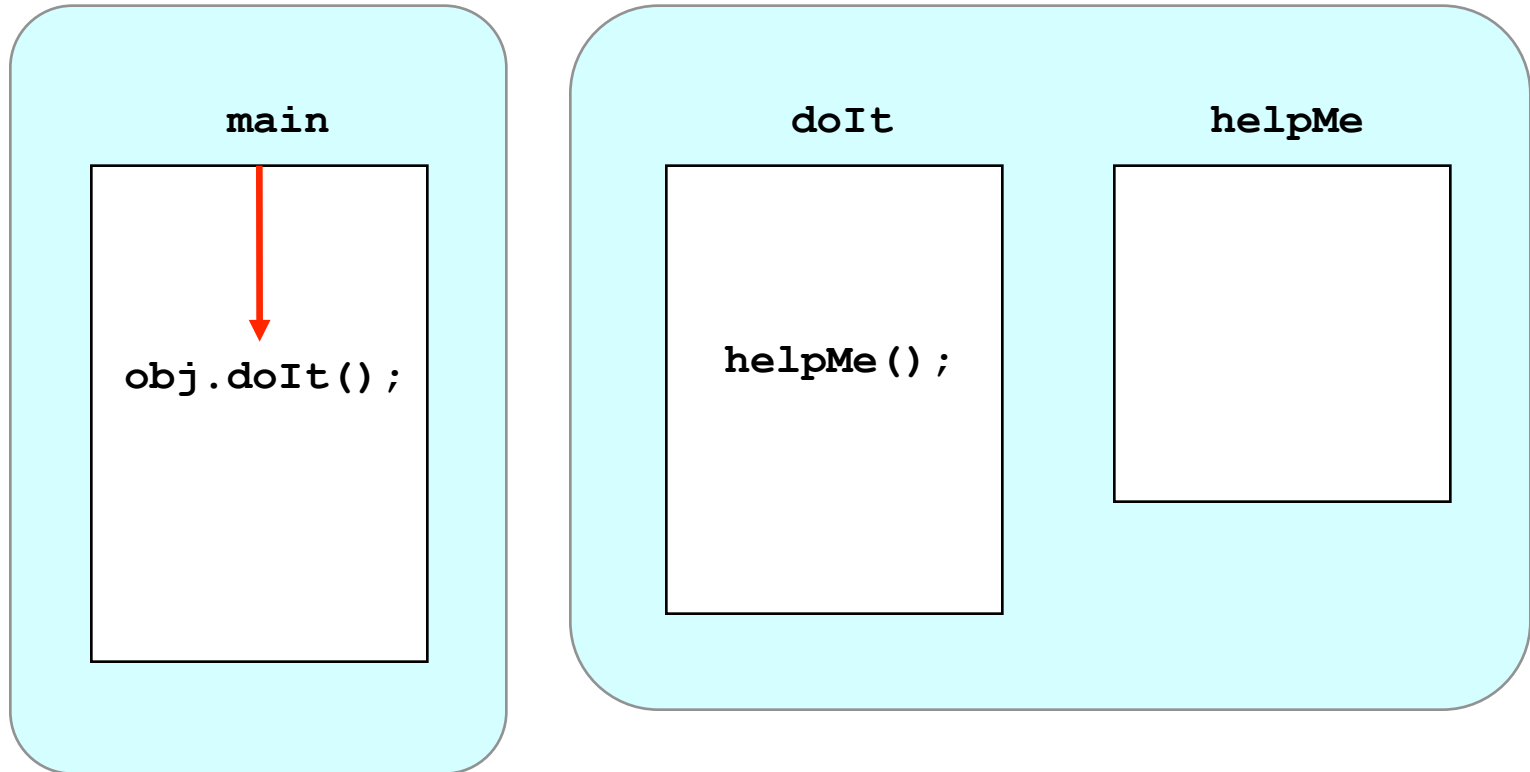
Method Control Flow



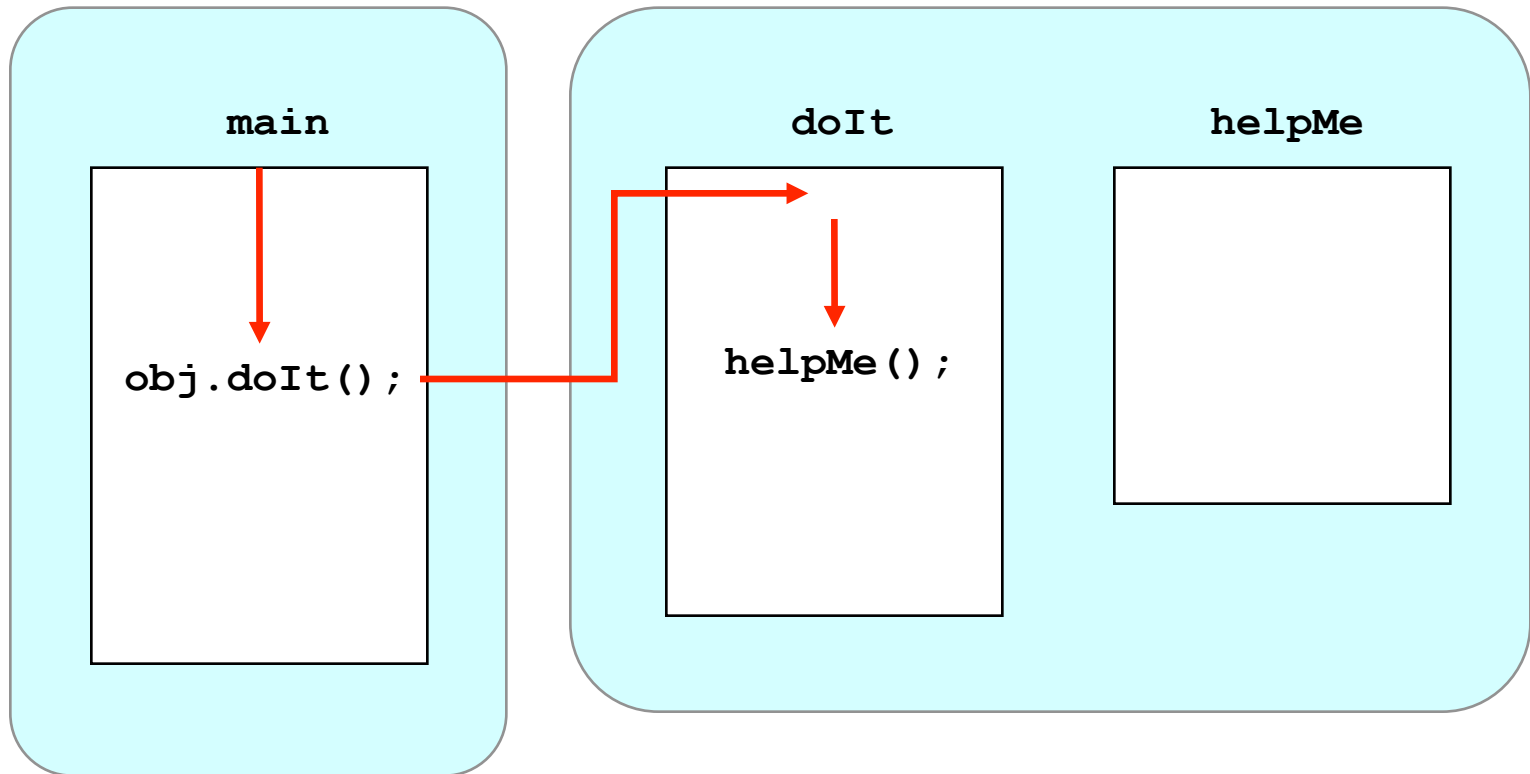
Method Control Flow



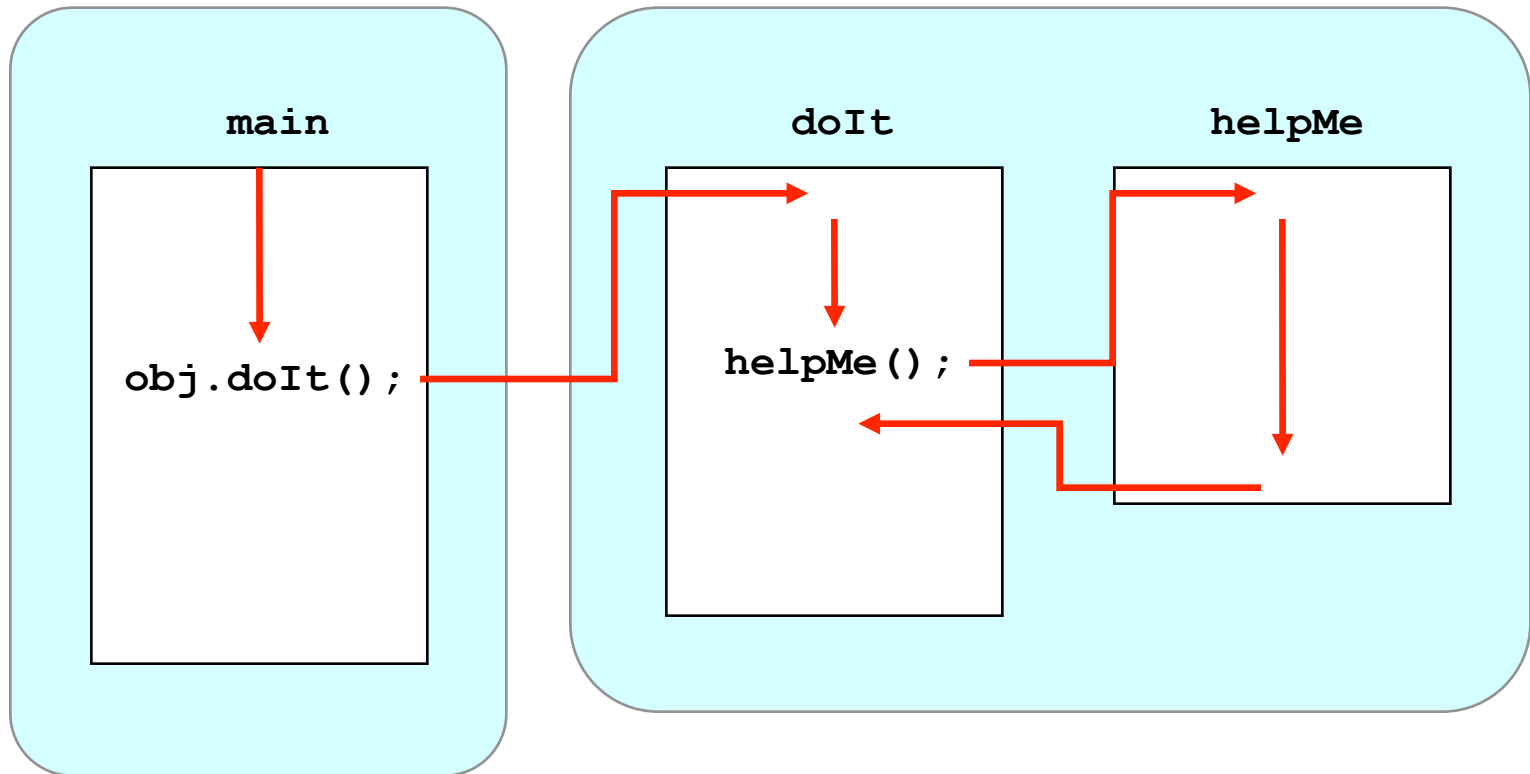
Method Control Flow



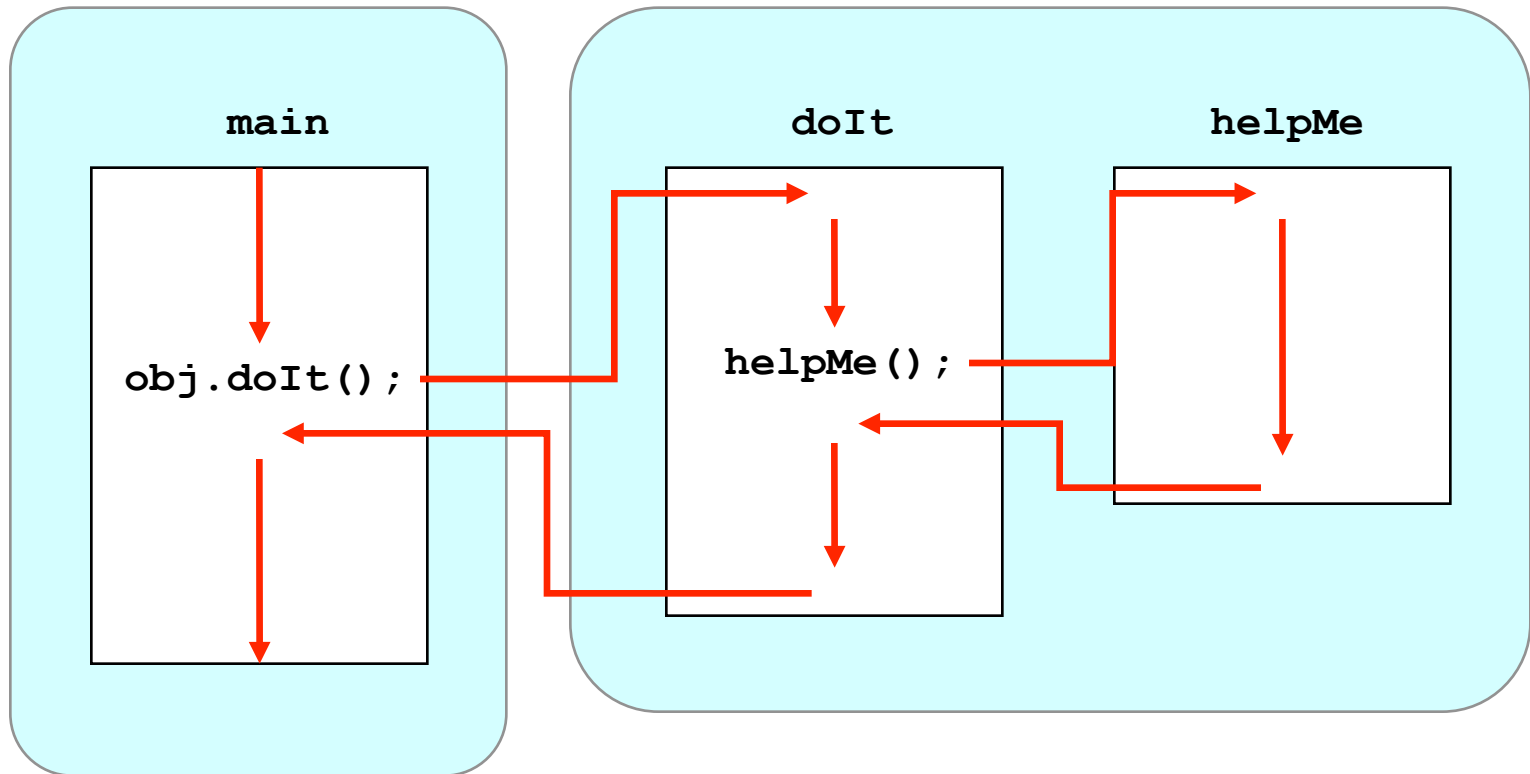
Method Control Flow



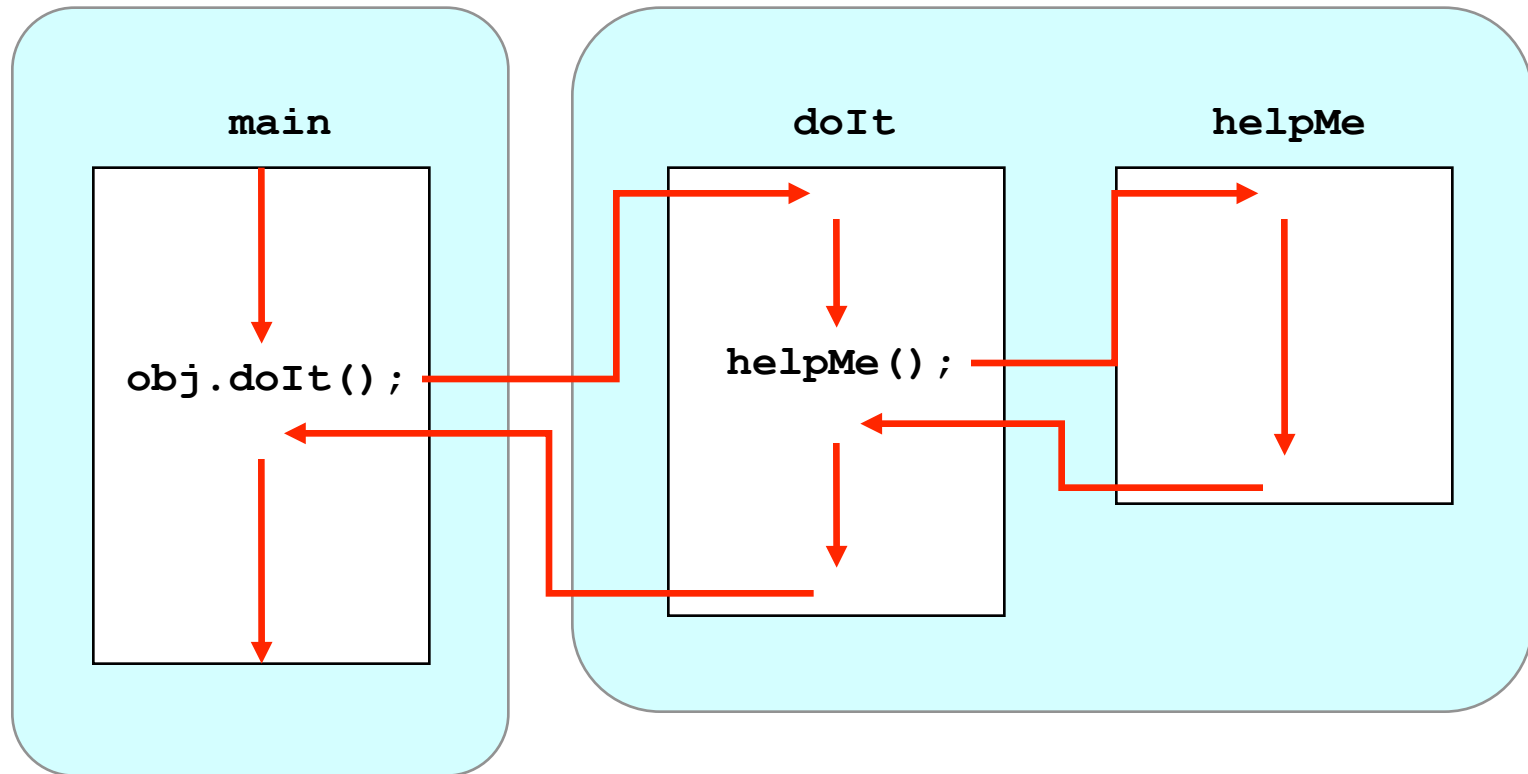
Method Control Flow



Method Control Flow



Method Control Flow



- The called method is often part of another class or object. In that case the object or class name, dot operator, and the method are required.


Variadic Methods

- A variadic method is a method that takes a variable number of arguments.
- Until now all our methods take a fixed number of arguments. For instance, the **sum** method only takes two numbers as arguments.
- A more useful version of the **sum** method should be able to accept any number of arguments.e.g., **sum(10,25)** , **sum(10,25,50)** , **sum(1,2,3,4,5,6,7)** , etc.

Variadic Methods in Java

The type of the last parameter is followed by an ellipsis (three dots, ...)

This feature is available as of Java 5.0



```
int sum (int... args) {  
    int total = 0;  
    for(int index = 0; index < args.length; index++){  
        total += args[index];  
    }  
    return total;  
}
```

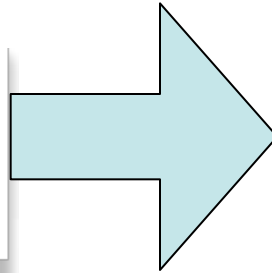
- Example:

```
int total1 = obj.sum (10, 15, 20);  
int total2 = obj.sum (10, 15, 20, 25);
```

Variadic Methods in Java

- Behind the scenes: arguments passed to a variadic method are converted into an array of the same-typed values

```
sum(10, 15, 20);
```

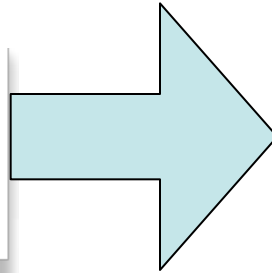


```
sum(new int[] {10,15, 20});
```


Variadic Methods in Java

- Behind the scenes: arguments passed to a variadic method are converted into an array of the same-typed values

```
sum(10, 15, 20);
```



```
sum(new int[] {10,15, 20});
```

```
int sum (int... args){  
    int total = 0;  
    for(int index = 0; index < args.length; index++){  
        total += args[index];  
    }  
    return total;  
}
```

Hence args is an array