Methods and method arguments

- Method call without arguments:
- `dog1.eat();`

```
public void eat() {

}
```

- Method call with one argument:
- `dog1.setName("Pluto");`

```
public void setName(String name) {

}
```

- Method call with many arguments:
- `dog1.init("Pluto", 3, Color.GREEN);`

```
public void init(String name, int age, Color color) {

}
```

Methods and return types

- Method call that returns a String:
- `String name = dog1.getName();`

```
public String getName() {
    return name;
}
```

- The returned value does not need to be handled:
- `dog1.getName();`
- This code will compile but will not do anything useful

- Methods that don't return anything declared to return void:

```
public void setName(String name) {
    this.name = name;
}
```

Variables

Instance variables

- Instance variables are declared in the class, not in a method
- They exist as long as the object exists
- They are accessible at least from all of the methods in the object

Local variables

- Variables declared inside a method belongs to that method
- They only exist during the execution of the method

Method arguments

- Arguments to a method are given local names inside the method
- Arguments are copied by value
- Primitive types:
  - A copy of the value is sent to the method
  - The original value is not affected by changes to the copy in the method
- Objects (object refereces):
  - A copy of the reference is sent to the method
  - Both copies of the reference refer to the same object
  - Changes to to object through any of the copies will affect the original object

this

- An object can reference itself with the object reference *this*
- This is necessary if an instance variable and a local variable have the same name
- this.name refers to the instance variable
- name refers to the local variable

```java
public class Dog {
    String name;

    public void setName(String name) {
        this.name = name;
    }
}
```

Overloaded methods

- Methods in a class can have the same name as long as the method arguments differ

```java
public class Test {
    public void test(int number) {
    }

    public void test(String number) {
    }

    public void test(int n, long l) {
    }
}
```

Encapsulation

- Encapsulation is one of the fundamental OOP concepts
- Encapsulation means data hiding
- The variables of a class are hidden from the outside, and can only be accessed from methods inside the class
- Methods that are meant for internal use should also be private
- Only expose methods with public access if they are meant to be accessed from the outside!
- The JavaBean standard builds upon encapsulation with private variables and getter and setter methods to access the variables

The JavaBean Standard

- The JavaBean standard is for creating reusable software componants in Java
- JavaBean:
    - encapsulates variables and methods
    - are serializable
    - have a zero-argument constructor
    - allow access to variables using getter and setter methods
- Naming convention for getters and setters:
    - getter for a variable *name* of type String: public String getName()
    - setter for a variable *name* of type String: public void setName(String name)

How to achieve encapsulation

- Declare all variables of a class as private
- Provide public getter methods to the variables that should be visible from the outside
- Provide public setter methods to the variables that should be able to be modified from the outside
- Provide constructors to set variables when objects are instantiated
- Declare all methods that are meant for internal use as private, and only the methods that are menat for external use as public

Access modifiers

- private – only accessible from within the class
- default (no access modifier) – only accessible from within the class and the package
- protected – only accessible from within the class, package and subclass
- public – accessible from everyware

Access Modifier	**Within class**	**Within package**	**Outside package and in subclass**	**Outside package**
private	Y	N	N	N
default	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

Benefits of Encapsulation

- The variables of a class can be made read-only or write-only
- A class can have total control over what is stored in its fields
- Systems become less complex to maintain if there are fewer allowed dependencies between classes
- It’s easier to change the internal functions of a class if it is not exposed externally