Introdução à programação orientada a objetos

java--programming-mooc-fi.translate.goog/part-4/1-introduction-to-object-oriented-programming

- Você está familiarizado com os conceitos de classe, objeto, construtor, métodos de objeto e variáveis de objeto.
- Você entende que uma classe define os métodos de um objeto e que os valores das variáveis de instância (objeto) são específicos do objeto.
- Você sabe como criar classes e objetos e como usar objetos em seus programas.

Começaremos agora nossa jornada no mundo da programação orientada a objetos. Começaremos focando na descrição de conceitos e dados usando objetos. A partir daí, aprenderemos como adicionar funcionalidades, ou seja, métodos ao nosso programa.

A programação orientada a objetos preocupa-se em isolar conceitos de um domínio de problema em entidades separadas e, em seguida, usar essas entidades para resolver problemas. Conceitos relacionados a um problema só podem ser considerados depois de identificados. Em outras palavras, podemos formar abstrações de problemas que tornem esses problemas mais fáceis de abordar.

Uma vez identificados os conceitos relacionados a um determinado problema, também podemos começar a construir construções que os representem em programas. Essas construções e as instâncias individuais formadas a partir delas, ou seja, objetos, são usadas na resolução do problema. A afirmação “os programas são construídos a partir de objetos pequenos, claros e cooperativos” pode ainda não fazer muito sentido. No entanto, parecerá mais sensato à medida que avançamos no curso, talvez até evidente.

Classes e objetos

Já usamos algumas das classes e objetos fornecidos pelo Java. Uma **classe** define os atributos dos objetos, ou seja, as informações relacionadas a eles (variáveis de instância), e seus comandos, ou seja, seus métodos. Os valores das variáveis de instância (ou seja, objeto) definem o estado interno de um objeto individual, enquanto os métodos definem a funcionalidade que ele oferece.

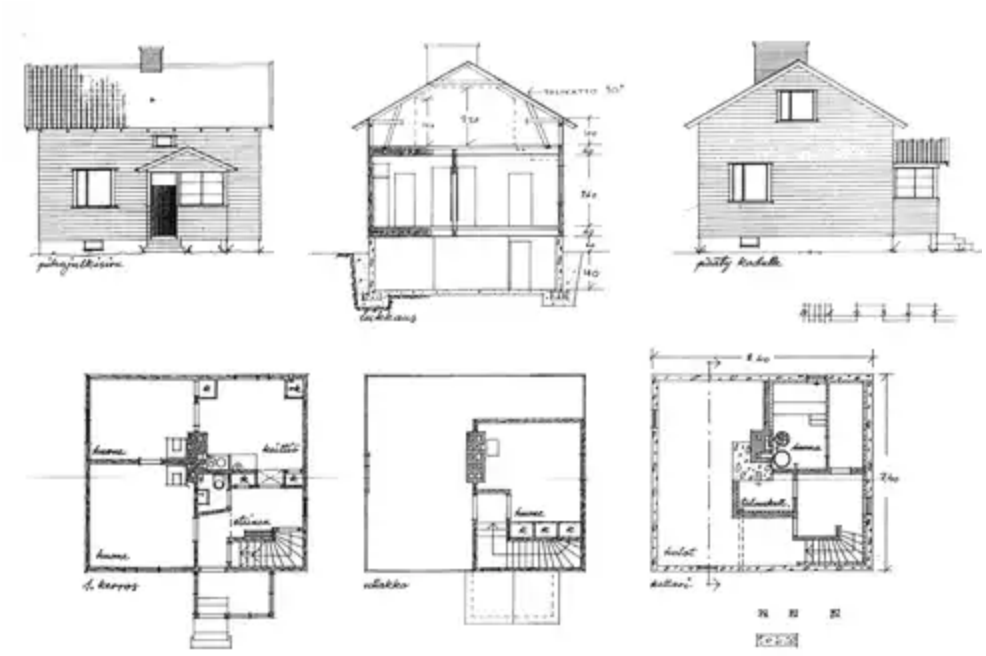
Um **Método** é um trecho de código-fonte escrito dentro de uma classe que foi nomeada e pode ser chamada. Um método sempre faz parte de alguma classe e costuma ser usado para modificar o estado interno de um objeto instanciado de uma classe.

As an example, `ArrayList` is a class offered by Java, and we've made use of objects instantiated from it in our programs. Below, an `ArrayList` object named `integers` is created and some integers are added to it.

```
// we create an object from the ArrayList class named integers
ArrayList<Integer> integers = new ArrayList<>();

// let's add the values 15, 34, 65, 111 to the integers object
integers.add(15);
integers.add(34);
integers.add(65);
integers.add(111);

// we print the size of the integers object
System.out.println(integers.size());
```

An object is always instantiated by calling a method that created an object, i.e., a **constructor** by using the `new` keyword.

A class lays out a blueprint for any objects that are instantiated from it. Let's draw from an analogy from outside the world of computers. Detached houses are most likely familiar to most, and we can safely assume the existence of drawings somewhere that determine what exactly a detached house is to be like. A class is a blueprint. In other words, it specifies what kinds of objects can be instantiated from it:



Individual objects, detached houses in this case, are all created based on the same blueprints - they're instances of the same class. The states of individual objects, i.e., their attributes (such as the wall color, the building material of the roof, the color of its foundations, the doors' materials and color, ...) may all vary, however. The following is an "object of a detached-house class":

Creating Classes

A class specifies what the objects instantiated from it are like.

- The **object's variables (instance variables)** specify the internal state of the object
- The **object's methods** specify what the object does

We'll now familiarize ourselves with creating our own classes and defining the variable that belong to them.

A class is defined to represent some meaningful entity, where a "meaningful entity" often refers to a real-world object or concept. If a computer program had to process personal information, it would perhaps be meaningful to define a seperate class `Person` consisting of methods and attributes related to an individual.

Let's begin. We'll assume that we have a project template that has an empty main program:

```
public class Main {

    public static void main(String[] args) {

    }
}
```

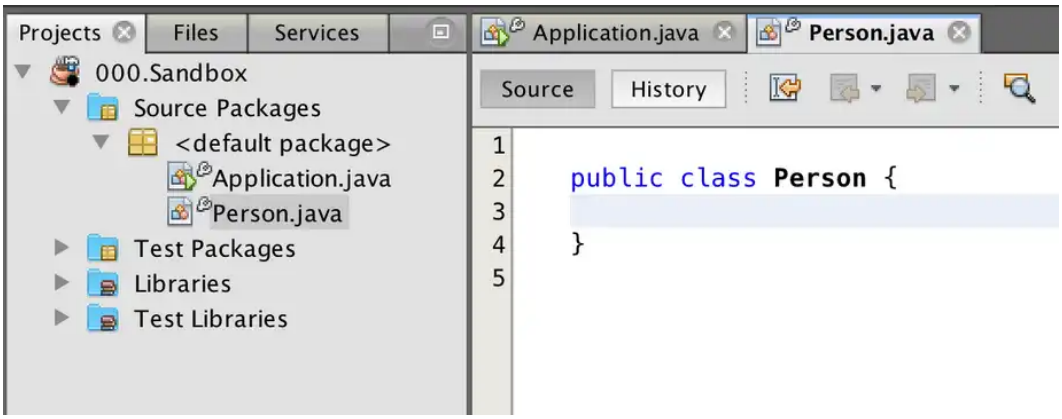
In NetBeans, a new class can be created by going to the *projects* section located on the left, right-clicking *new*, and then *java class*. The class is provided a name in the dialog that opens.

As with variables and methods, the name of a class should be as descriptive as possible. It's usual for a class to live on and take on a different form as a program develops. As such, the class may have to be renamed at some later point.

Let's create a class named `Person`. For this class, we create a separate file named `Person.java`. Our program now consists of two separate files since the main program is also in its own file. The `Person.java` file initially contains the class definition **public class Person** and the curly brackets that confine the contents of the class.

```
public class Person {

}
```

After creating a new file in NetBeans, the current state is as follows. In the image below, the class `Person` has been added to the SandboxExercise.



You can also draw a class diagram to depict a class. We'll become familiar with its notations as we go along. An empty person-named class looks like this:

A class defines the attributes and behaviors of objects that are created from it. Let's decide that each person object has a name and an age. It's natural to represent the name as a string, and the age as an integer. We'll go ahead and add these to our blueprint:

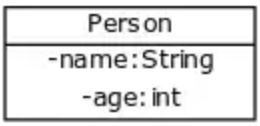
```
public class Person {
    private String name;
    private int age;
}
```

We specify above that each object created from the `Person` class has a `name` and an `age`. Variables defined inside a class are called **instance variables**, or object fields or object attributes. Other names also seem to exist.

Instance variables are written on the lines following the class definition `public class Person {`. Each variable is preceded by the keyword private. The keyword **private** means that the variables are "hidden" inside the object. This is known as **encapsulation**.

In the class diagram, the variables associated with the class are defined as "variableName: variableType". The minus sign before the variable name indicates that the variable is encapsulated (it has the keyword private).

We have now defined a blueprint — a class — for the person object. Each new person object has the variables `name` and `age`, which are able to hold object-specific values. The "state" of a person consists of the values assigned to their name and age.



Defining a Constructor

We want to set an initial state for an object that's created. Custom objects are created the same way as objects from pre-made Java classes, such as `ArrayList`, using the `new` keyword. It'd be convenient to pass values to the variables of that object as it's being created. For example, when creating a new person object, it's useful to be able to provide it with a name:

```
public static void main(String[] args) {
    Person ada = new Person("Ada");
    // ...
}
```

This is achieved by defining the method that creates the object, i.e., its constructor. The constructor is defined after the instance variables. In the following example, a constructor is defined for the Person class, which can be used to create a new Person object. The constructor sets the age of the object being created to 0, and the string passed to the constructor as a parameter as its name:

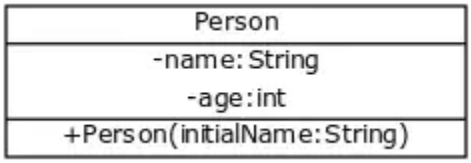
```
public class Person {
    private String name;
    private int age;

    public Person(String initialName) {
        this.age = 0;
        this.name = initialName;
    }
}
```

The constructor's name is always the same as the class name. The class in the example above is named Person, so the constructor will also have to be named Person. The constructor is also provided, as a parameter, the name of the person object to be created. The parameter is enclosed in parentheses and follows the constructor's name. The parentheses that contain optional parameters are followed by curly brackets. In between these brackets is the source code that the program executes when the constructor is called (e.g., `new Person ("Ada")`).

Objects are always created using a constructor.

A few things to note: the constructor contains the expression `this.age = 0`. This expression sets the instance variable `age` of the newly created object (i.e., "this" object's age) to 0. The second expression `this.name = initialName` likewise assigns the string passed as a parameter to the instance variable `name` of the object created.

If the programmer does not define a constructor for a class, Java automatically creates a default one for it. A default constructor is a constructor that doesn't do anything apart from creating the object. The object's variables remain uninitialized (generally, the value of any object references will be `null`, meaning that they do not point to anything, and the values of primitives will be `0`)



For example, an object can be created from the class below by making the call `new Person()`

```
public class Person {
    private String name;
    private int age;
}
```

If a constructor has been defined for a class, no default constructor exists. For the class below, calling `new Person` would cause an error, as Java cannot find a constructor in the class that has no parameters.

```
public class Person {
    private String name;
    private int age;

    public Person(String initialName) {
        this.age = 0;
        this.name = initialName;
    }
}
```

Defining Methods For an Object

We know how to create an object and initialize its variables. However, an object also needs methods to be able to do anything. As we've learned, a **method** is a named section of source code inside a class which can be invoked.

```
public class Person {
    private String name;
    private int age;

    public Person(String initialName) {
        this.age = 0;
        this.name = initialName;
    }

    public void printPerson() {
        System.out.println(this.name + ", age " + this.age + " years");
    }
}
```

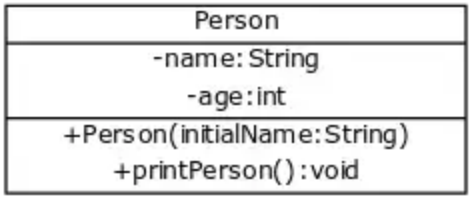
A method is written inside of the class beneath the constructor. The method name is preceded by `public void`, since the method is intended to be visible to the outside world (`public`), and it does not return a value (`void`).

We've used the modifier `static` in some of the methods that we've written. The `static` modifier indicates that the method in question does not belong to an object and thus cannot be used to access any variables that belong to objects.

Going forward, our methods will not include the `static` keyword if they're used to process information about objects created from a given class. If a method receives as parameters all the variables whose values it uses, it can have a `static` modifier.

In addition to the class name, instance variables and constructor, the class diagram now also includes the method `printPerson` . Since the method comes with the `public` modifier, the method name is prefixed with a plus sign. No parameters are defined for the method, so nothing is put inside the method's parentheses. The method is also marked with information indicating that it does not return a value, here `void` .

The method `printPerson` contains one line of code that makes use of the instance variables `name` and `age` — the class diagram says nothing about its internal implementations. Instance variables are referred to with the prefix `this` . All of the object's variables are visible and available from within the method.



Let's create three persons in the main program and request them to print themselves:

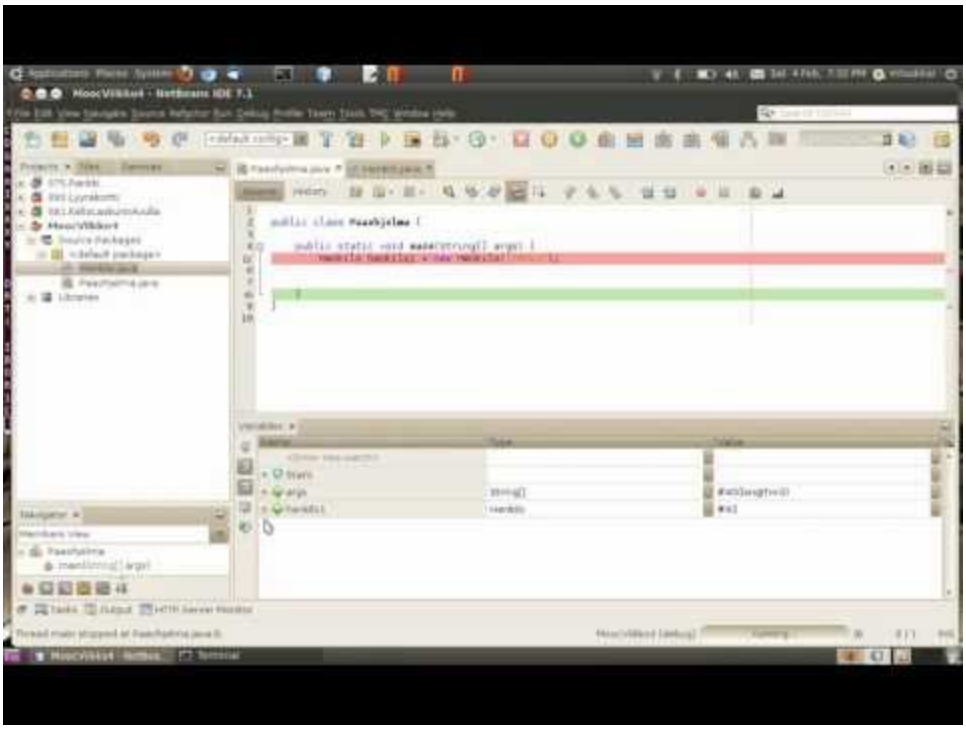
```
public class Main {

    public static void main(String[] args) {
        Person ada = new Person("Ada");
        Person antti = new Person("Antti");
        Person martin = new Person("Martin");

        ada.printPerson();
        antti.printPerson();
        martin.printPerson();
    }
}
```

Prints:

Sample output

Ada, age 0 years Antti, age 0 years Martin, age 0 years

This as a screencast:



Watch Video At: https://youtu.be/fWwXQ5n2gYo

Changing an Instance Variable's Value in a Method

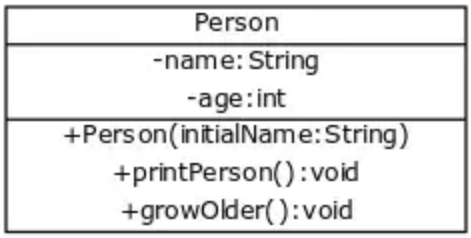
Let's add a method to the previously created person class that increments the age of the person by a year.

```
public class Person {
    private String name;
    private int age;

    public Person(String initialName) {
        this.age = 0;
        this.name = initialName;
    }

    public void printPerson() {
        System.out.println(this.name + ", age " + this.age + " years");
    }

    // growOlder() method has been added
    public void growOlder() {
        this.age = this.age + 1;
    }
}
```

The method is written inside the `Person` class just as the `printPerson` method was. The method increments the value of the instance variable `age` by one.

The class diagram also gets an update.

Let's call the method and see what happens:




```
public class Main {

    public static void main(String[] args) {
        Person ada = new Person("Ada");
        Person antti = new Person("Antti");

        ada.printPerson();
        antti.printPerson();
        System.out.println("");

        ada.growOlder();
        ada.growOlder();

        ada.printPerson();
        antti.printPerson();
    }
}
```

The program's print output is as follows:

Sample output

Ada, age 0 years Antti, age 0 years

Ada, age 2 years Antti, age 0 years

That is to say that when the two objects are "born" they're both zero-years old (`this.age = 0;` is executed in the constructor). The `ada` object's `growOlder` method is called twice. As the print output demonstrates, the age of Ada is 2 years after growing older. Calling the method on an object corresponding to Ada has no impact on the age of the other person object since each object instantiated from a class has its own instance variables.

The method can also contain conditional statements and loops. The growOlder method below limits aging to 30 years.

```
public class Person {
    private String name;
    private int age;

    public Person(String initialName) {
        this.age = 0;
        this.name = initialName;
    }

    public void printPerson() {
        System.out.println(this.name + ", age " + this.age + " years");
    }

    // no one exceeds the age of 30
    public void growOlder() {
        if (this.age < 30) {
            this.age = this.age + 1;
        }
    }
}
```

Returning a Value From a Method

A method can return a value. The methods we've created in our objects haven't so far returned anything. This has been marked by typing the keyword *void* in the method definition.

```
public class Door {
    public void knock() {
        // ...
    }
}
```

The keyword **void** means that the method does not return a value.

If we want the method to return a value, we need to replace the `void` keyword with the type of the variable to be returned. In the following example, the Teacher class has a method `grade` that always returns an integer-type (`int`) variable (in this case, the value 10). The value is always returned with the **return** command:

```
public class Teacher {
    public int grade() {
        return 10;
    }
}
```

The method above returns an `int` type variable of value 10 when called. For the return value to be used, it needs to be assigned to a variable. This happens the same way as regular value assignment, i.e., by using the equals sign:

```
public static void main(String[] args) {
    Teacher teacher = new Teacher();

    int grading = teacher.grade();

    System.out.println("The grade received is " + grading);
}
```

Sample output

The grade received is 10

The method's return value is assigned to a variable of type `int` value just as any other int value would be. The return value could also be used to form part of an expression.

```
public static void main(String[] args) {
    Teacher first = new Teacher();
    Teacher second = new Teacher();
    Teacher third = new Teacher();

    double average = (first.grade() + second.grade() + third.grade()) / 3.0;

    System.out.println("Grading average " + average);
}
```

Sample output

Grading average 10.0

All the variables we've encountered so far can also be returned by a method. To sum:

A method that returns nothing has the `void` modifier as the type of variable to be returned.

```
public void methodThatReturnsNothing() {
    // the method body
}
```

A method that returns an integer variable has the `int` modifier as the type of variable to be returned.

```
public int methodThatReturnsAnInteger() {
    // the method body, requires a return statement
}
```

A method that returns a string has the `String` modifier as the type of the variable to be returned

```
public String methodThatReturnsAString() {
    // the method body, requires a return statement
}
```

A method that returns a double-precision number has the `double` modifier as the type of the variable to be returned.

```
public double methodThatReturnsADouble() {
    // the method body, requires a return statement
}
```

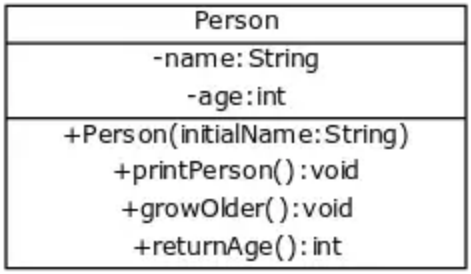
Let's continue with the Person class and add a `returnAge` method that returns the person's age.

```
public class Person {
    private String name;
    private int age;

    public Person(String initialName) {
        this.age = 0;
        this.name = initialName;
    }

    public void printPerson() {
        System.out.println(this.name + ", age " + this.age + " years");
    }

    public void growOlder() {
        if (this.age < 30) {
            this.age = this.age + 1;
        }
    }
    // the added method
    public int returnAge() {
        return this.age;
    }
```

The class in its entirety:

Let's illustrate how the method works:

```
public class Main {

    public static void main(String[] args) {
        Person pekka = new Person("Pekka");
        Person antti = new Person("Antti");

        pekka.growOlder();
        pekka.growOlder();

        antti.growOlder();

        System.out.println("Pekka's age: " +
pekka.returnAge());
        System.out.println("Antti's age: " +
antti.returnAge());
        int combined = pekka.returnAge() +
antti.returnAge();

        System.out.println("Pekka's and Antti's combined
age " + combined + " years");
    }
}
```



Sample output

Pekka's age 2 Antti's age 1

Pekka's and Antti's combined age 3 years

As we came to notice, methods can contain source code in the same way as other parts of our program. Methods can have conditionals or loops, and other methods can also be called from them.

Let's now write a method for the person that determines if the person is of legal age. The method returns a boolean - either `true` or `false`:

```
public class Person {
    // ...

    public boolean isOfLegalAge() {
        if (this.age < 18) {
            return false;
        }

        return true;
    }

    /*
     The method could have been written more succinctly in the following way:

    public boolean isOfLegalAge() {
        return this.age >= 18;
    }
    */
}
```

And let's test it out:

```
public static void main(String[] args) {
    Person pekka = new Person("Pekka");
    Person antti = new Person("Antti");

    int i = 0;
    while (i < 30) {
        pekka.growOlder();
        i = i + 1;
    }

    antti.growOlder();

    System.out.println("");

    if (antti.isOfLegalAge()) {
        System.out.print("of legal age: ");
        antti.printPerson();
    } else {
        System.out.print("underage: ");
        antti.printPerson();
    }

    if (pekka.isOfLegalAge()) {
        System.out.print("of legal age: ");
        pekka.printPerson();
    } else {
        System.out.print("underage: ");
        pekka.printPerson();
    }
}
```

Sample output

underage: Antti, age 1 years of legal age: Pekka, age 30 years

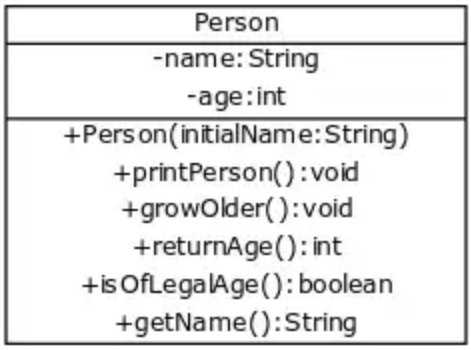
Let's fine-tune the solution a bit more. In its current form, a person can only be "printed" in a way that includes both the name and the age. Situations exist, however, where we may only want to know the name of an object. Let's write a separate method for this use case:

```
public class Person {
    // ...

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

The `getName` method returns the instance variable `name` to the caller. The name of this method is somewhat strange. It is the convention in Java to name a method that returns an instance variable exactly this way, i.e., `getVariableName`. Such methods are often referred to as "getters".

The class as a whole:

Let's mould the main program to use the new "getter" method:



```
public static void main(String[] args) {
    Person pekka = new Person("Pekka");
    Person antti = new Person("Antti");

    int i = 0;
    while (i < 30) {
        pekka.growOlder();
        i = i + 1;
    }

    antti.growOlder();

    System.out.println("");

    if (antti.isOfLegalAge()) {
        System.out.println(antti.getName() + " is of legal age");
    } else {
        System.out.println(antti.getName() + " is underage");
    }

    if (pekka.isOfLegalAge()) {
        System.out.println(pekka.getName() + " is of legal age");
    } else {
        System.out.println(pekka.getName() + " is underage ");
    }
}
```

The print output is starting to turn out quit neat:

Sample output

Antti is underage Pekka is of legal age

A string representation of an object and the toString-method

We are guilty of programming in a somewhat poor style by creating a method for printing the object, i.e., the `printPerson` method. A preferred way is to define a method for the object that returns a "string representation" of the object. The method returning the string representation is always `toString` in Java. Let's define this method for the person in the following example:

```
public class Person {
    // ...

    public String toString() {
        return this.name + ", age " + this.age + " years";
    }
}
```

The `toString` functions as `printPerson` does. However, it doesn't itself print anything but instead returns a string representation, which the calling method can execute for printing as needed.

The method is used in a somewhat surprising way:

```
public static void main(String[] args) {
    Person pekka = new Person("Pekka");
    Person antti = new Person("Antti");

    int i = 0;
    while (i < 30) {
        pekka.growOlder();
        i = i + 1;
    }

    antti.growOlder();

    System.out.println(antti); // same as System.out.println(antti.toString());
    System.out.println(pekka); // same as System.out.println(pekka.toString());
}
```

In principle, the `System.out.println` method requests the object's string representation and prints it. The call to the `toString` method returning the string representation does not have to be written explicitly, as Java adds it automatically. When a programmer writes:

```
System.out.println(antti);
```

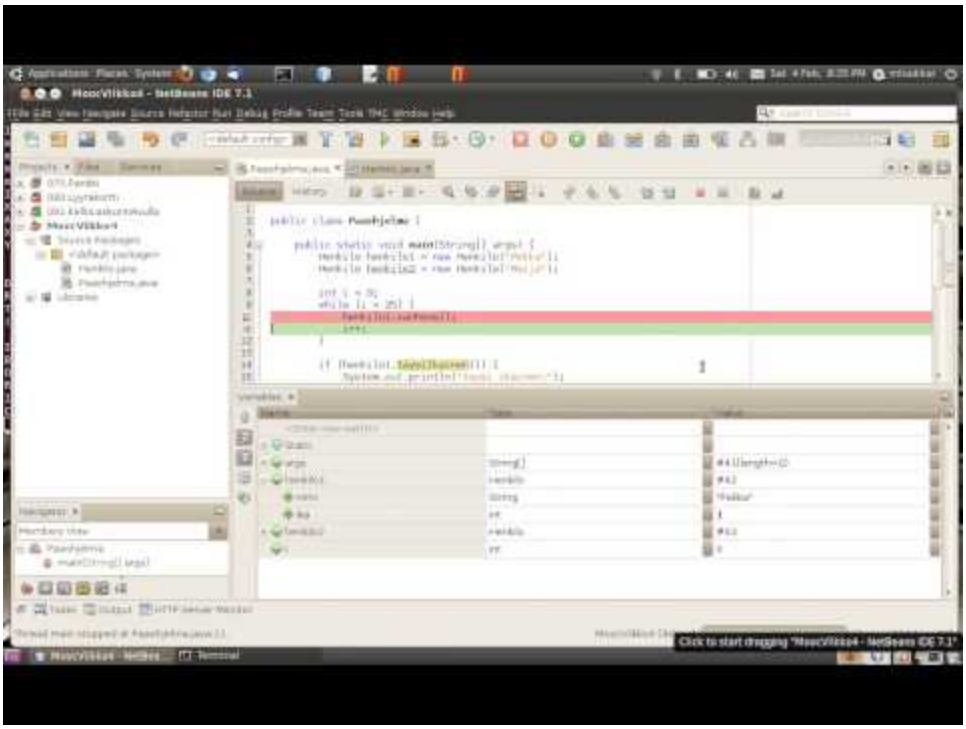
Java extends the call at run time to the following form:

```
System.out.println(antti.toString());
```

As such, the call `System.out.println(antti)` calls the `toString` method of the `antti` object and prints the string returned by it.

We can remove the now obsolete `printPerson` method from the Person class.

The second part of the screencast:



Watch Video At: https://youtu.be/d-56AxspStE

Method parameters

Let's continue with the `Person` class once more. We've decided that we want to calculate people's body mass indexes. To do this, we write methods for the person to set both the height and the weight, and also a method to calculate the body mass index. The new and changed parts of the Person object are as follows:

```
public class Person {
    private String name;
    private int age;
    private int weight;
    private int height;

    public Person(String initialName) {
        this.age = 0;
        this.weight = 0;
        this.height = 0;
        this.name = initialName;
    }

    public void setHeight(int newHeight) {
        this.height = newHeight;
    }

    public void setWeight(int newWeight) {
        this.weight = newWeight;
    }

    public double bodyMassIndex() {
        double heigthPerHundred = this.height / 100.0;
        return this.weight / (heigthPerHundred * heigthPerHundred);
    }

    // ...
}
```

The instance variables `height` and `weight` were added to the person. Values for these can be set using the `setHeight` and `setWeight` methods. Java's standard naming convention is used once again, that is, if the method's only purpose is to set a value to an instance variable, then it's named as `setVariableName`. Value-setting methods are often called "setters". The new methods are put to use in the following case:

```
public static void main(String[] args) {
    Person matti = new Person("Matti");
    Person juhana = new Person("Juhana");

    matti.setHeight(180);
    matti.setWeight(86);

    juhana.setHeight(175);
    juhana.setWeight(64);

    System.out.println(matti.getName() + ", body mass index is " + matti.bodyMassIndex());
    System.out.println(juhana.getName() + ", body mass index is " + juhana.bodyMassIndex());
}
```

Prints:

Sample output

Matti, body mass index is 26.54320987654321 Juhana, body mass index is 20.897959183673468

A parameter and instance variable having the same name!

In the preceding example, the `setHeight` method sets the value of the parameter `newHeight` to the instance variable `height`:

```
public void setHeight(int newHeight) {
    this.height = newHeight;
}
```

The parameter's name could also be the same as the instance variable's, so the following would also work:

```
public void setHeight(int height) {
    this.height = height;
}
```

In this case, `height` in the method refers specifically to a parameter named *height* and `this.height` to an instance variable of the same name. For example, the following example would not work as the code does not refer to the instance variable *height* at all. What the code does in effect is set the `height` variable received as a parameter to the value it already contains:

```
public void setHeight(int height) {
    // DON'T DO THIS!!!
    height = height;
}

public void setHeight(int height) {
    // DO THIS INSTEAD!!!
    this.height = height;
}
```

Calling an internal method

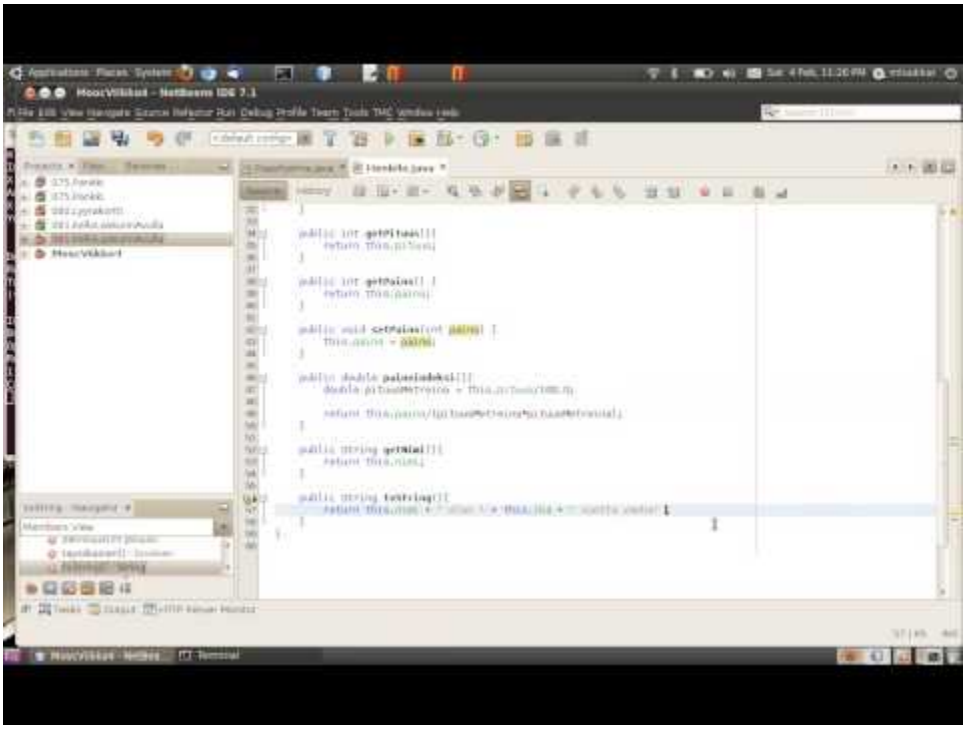
O objeto também pode chamar seus métodos. Por exemplo, se quisermos que a representação de string retornada por toString também informe o índice de massa corporal de uma pessoa, o próprio `bodyMassIndex` método do objeto deverá ser chamado no `toString` método:

```
public String toString() {
    return this.name + ", age " + this.age + " years, my body mass index is " +
this.bodyMassIndex();
}
```

Portanto, quando um objeto chama um método interno, o nome do método e este prefixo são suficientes. Uma maneira alternativa é chamar o método do próprio objeto no formato `bodyMassIndex()`, onde nenhuma ênfase é colocada no fato de que o método bodyMassIndex do próprio objeto está sendo chamado:

```
public String toString() {
    return this.name + ", age " + this.age + " years, my body mass index is " +
bodyMassIndex();
}
```

A terceira parte do screencast:



Watch Video At: https://youtu.be/YKwzIGuCLn8

Você provavelmente notou que alguns dos números apresentam erros de arredondamento. No exercício anterior, por exemplo, o saldo de 30,7 de Pekka pode ser impresso como `30.700000000000003`. Isso ocorre porque os números de ponto flutuante, como `double`, na verdade, são armazenados em formato binário. Ou seja, em zeros e uns usando apenas um número limitado de números. Como o número de números de ponto flutuante é infinito - (caso você esteja se perguntando "quão infinito?", pense em quantos valores de ponto flutuante ou decimais cabem entre os números 5 e 6, por exemplo). Todos os números de ponto flutuante simplesmente não podem ser representados por um número finito de zeros e uns. Assim, o computador deve colocar um limite na precisão dos números armazenados.

Normalmente, os saldos das contas, por exemplo, são salvos como números inteiros, de modo que, digamos, o valor 1 represente um centavo.