TND002 Object-oriented programming

Lecture 4: Working with a class and documenting its content

Mark Eric Dieckmann,
MIT division,
ITN, Linköping University.

January 22, 2023

Outline of the lecture

- Shallow and deep copies of reference variables.
- The keyword *this*.
- Using a reference variable inside its class.
- Documenting the content of a class.
- The class String and static arrays of reference variables.

Starting point

We start with the class **Human** (no package line).

```
public class Human {
private String name;
public Human(String arg){ setName(arg);}
public void setName(String arg){ name = arg;}
public String getName(){ return name;}
public String toString(){ return "The name is: " + getName();}}
```

We can call methods from other methods / constructors.

arg has a local scope in the constructor and in *setName(arg)* (we can use the same name twice).

The only relevant attribute to copy is *name*.

Copy of an instance

```
public class MainClass {
public static void main(String[] args) {
Human h1 = new Human("Mark");
Human h2; h2=h1;
System.out.println(h1); System.out.println(h2); }}
```

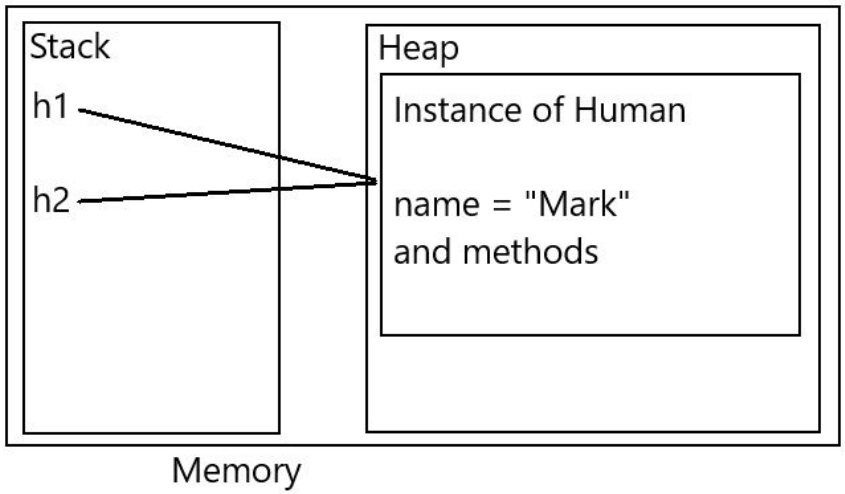
Result: "The name is: Mark" followed by "The name is: Mark"

```
public class MainClass {
public static void main(String[] args) {
Human h1 = new Human("Mark");
Human h2; h2=h1; h1.setName("Karl");
System.out.println(h1); System.out.println(h2); }}
```

Result: "The name is: Karl" followed by "The name is: Karl"

Shallow copy

```
Human h1 = new Human("Mark");
Human h2; h2 = h1;
```



We created one instance of **Human** with *name*="Mark".

h2=h1; copies the address value of *h1* to *h2*.

Both refer to the same object.

This is a shallow copy.

The command *h1.setName("Karl");* changes the name in the object connected to *h1*, which is also connected to *h2*.

Hence, it changes the result of *h1.toString()* and *h2.toString()*.

Method for a shallow copy

We can also define a method for a shallow copy to illustrate two important features of Java classes.

```
public class Human {
private String name;
public Human(String arg){ setName(arg);}
public void setName(String arg){ name = arg;}
public Human shallowCopy(){ return this;}
public String getName(){ return name;}
public String toString(){ return "The name is: " + name;}}
```

Feature 1: The return type of a method can equal its class.

shallowCopy() returns an address (reference) to an object in the heap but not the object itself.

Method for a shallow copy

Feature 2: the keyword *this* in the method

```
public Human shallowCopy(){ return this;}
```

```
public class MainClass {
public static void main(String[] args) {
Human h1 = new Human("Mark");
Human h2=h1, h3 = h1.shallowCopy();
h1.setName("Karl");
System.out.println(h1); System.out.println(h2);
System.out.println(h3); }}
```

We obtain the three lines "The name is: Karl" on the console.

The address in *h1.this* equals the address of *h1* and is accessible inside the object attached to *h1*.

Deep copy

Now we want to duplicate the object; both instances should have the same values of the attributes (instance variables).

We introduce the method *deepCopy()*, which creates a new instance of **Human** with the same value for *name*.

```
public class Human {
private String name;
public Human(String arg){ setName(arg);}
public void setName(String arg){ name = arg;}
public Human deepCopy(){ return new Human(name);}
public Human shallowCopy(){ return this;}
public String getName(){ return name;}
public String toString(){ return "The name is: " + name;}}
```

Using instances in class methods of the same class

We want to implement a class method, which takes the names of two instances of **Human** and adds them up. Consider:

```
public class Human {
private String name;
public Human(String arg){ name = arg; }
public void setName(String arg){ name = arg;}
public String getName(){ return name;}}
```

```
public class MainClass {
public static void main(String[] args) {
Human h1 = new Human("Mark");
Human h2 = new Human("Karl");
System.out.println(Human.addNames(h1, h2));} }
```

Using instances in class methods of the same class

```
public class Human {
private String name;
public void setName(String arg){ name = arg;}
public String getName(){ return name;}
public static String addNames(Human arg1, Human arg2){
String s1 = arg1.getName();
String s2 = arg2.getName();
String result = s1 + " and " + s2;
return result; } }
```

We can write it more compact as

```
public static String addNames(Human arg1, Human arg2)
{ return arg1.getName() + " and " + arg2.getName();}
```

Using instances in instance methods of the same class

The class method is called as *Human.addNames(h1, h2);*

If we design an instance method, then the call is done with a reference to one of both humans *h1.addNames(h2);*

We can overload the method name (different argument lists).

```
public String addNames(Human arg)
{ return name + " and " + arg.getName(); }
```

name is the name of *h1* and *arg.getName()* that of *h2*;

Inside a class, we can access its private members and replace *arg.getName()* by *arg.name*.

Arranging information

A class can contain

instance / class variables and instance / class methods.

constructors.

access modifiers (private and public).

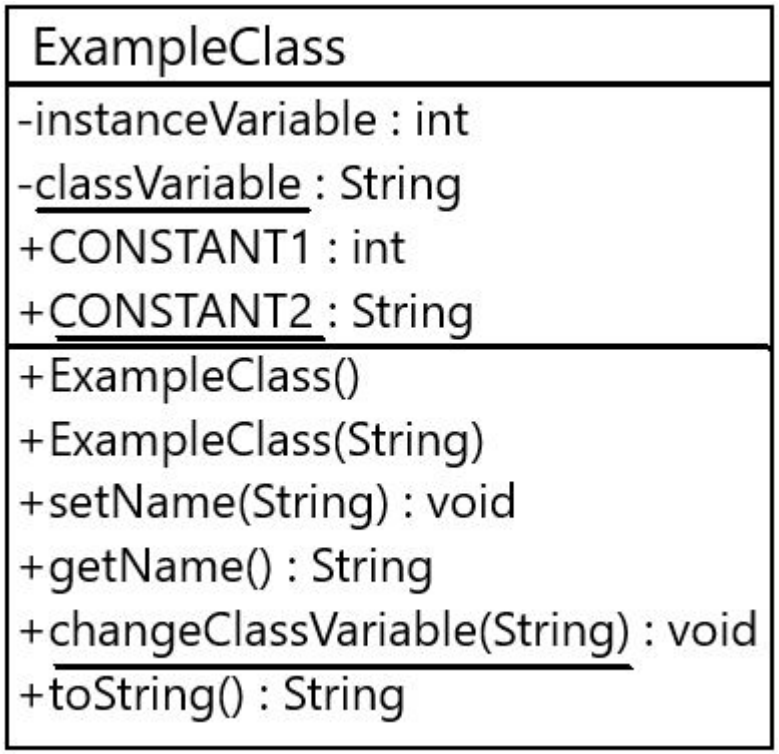
constant variables declared with the keyword final:

```
final int i = 10; final static double d = 3.0; final String s = "Hello world";
```

Class diagrams provide an easy and compact means to communicate this information to a programmer or user.

Class diagrams

This class diagram presents **ExampleClass**.



The class name is on top.

Variables with a global scope go into the middle box.

access modifier "-" is private and "+" is public.

The variable name is on the left and the type on the right.

underlining means "static".

uppercase means constant.

Methods are listed in the bottom part. The method name and the argument list is on the left. The return type on the right.

Constructors come first and have no return type.

The class String

The right hand side in *String s = new String("Hello world");* is a constructor call: **String** is a class.

Java provides the short form *String s = "Hello world";*, which is translated internally into the correct call.

Class names in Java start with an uppercase letter. Primitive variables (int, double) with a lowercase letter.

The most relevant instance variable of String is the content "Hello world". How about its methods?

"String." lists the class methods marked by a "S" for static.

"String s;" followed by "s." lists the available instance methods.

We show how to use important methods in Lesson 1b.

Static arrays (fixed size) of Strings

Arrays are reference variables in Java and instantiated as

```
String[] result = new String[3]; // We do not use () here.
```

String[] result; declares a reference variable for a string array.

Once you instantiate it (reserve memory on the heap) you need to set its size. *result* can take 3 strings.

We do not use the *()*: we are instantiating the array but we are not initializing its strings.

Initialization: *result[0] = new String();*, *result[0] = "Hello";* or at the first declaration as *String[] result = { "a", "b", "c"};*

result.length tells you how many elements the array has.

Summary

We have discussed

- Shallow and deep copies of reference variables.
- The keyword this.
- How to use a reference variable inside its class.
- Documenting the content of a class.
- The class String and static arrays of reference variables.