TND002 Object-oriented programming

Lecture 2: Static methods and reference variables

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

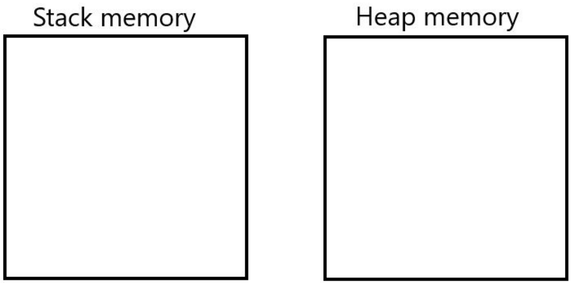
January 11, 2023

Outline of the lecture

- Aspects of Java's memory management.
- Static methods and procedures.
- Method overloading.
- Primitive variables and reference variables.

Java's memory management

Java subdivides the computer's RAM into two memory types.



Code that gets created when you compile your program goes into the stack memory.

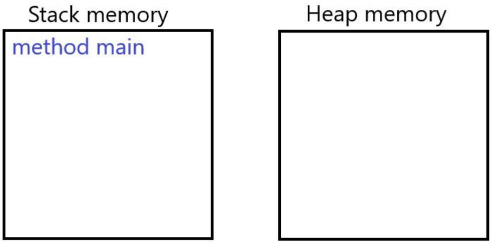
Code that gets created while the code is running goes into the heap memory.

Compiling our program

I will usually leave out the line starting with *package*

```
public class FirstProgram {
public static void main(String[] args) {
System.out.println("Hello world");} }
```

The keyword *static* lets *main* go into the stack.



Compiling our program

The JVM goes to *main(String[] args)* when you click on "run"

$\Rightarrow$ only one method *main(..)* can be run at any time.

No method calls *main(..)*. Its return type is thus always *void*.

The memory requirements of code, which goes into *main(..)* and thus the stack memory, must be known at compile time.

In principle, this could work here because the only line of code

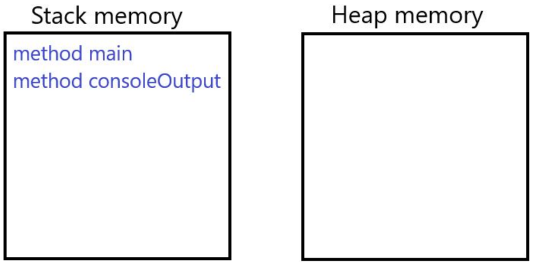
System.out.println("Hello world");

has a string with 11 characters of type *char* (11 $\times$ 8 bits).

Compiling our program

A static method can call another one in the same class.

```
public class FirstProgram {
public static void main(String[] args) {
consoleOutput("Hello world"); }
public static void consoleOutput(String arg) {
System.out.println(arg);} }
```



Static method call

```
public class FirstProgram {

public static void main(String[] args) {
consoleOutput("Hello world"); }

public static void consoleOutput(String arg) {
System.out.println(arg);} }
```

The new static method *consoleOutput(String arg)* can be placed in a class before or after *main(..)*.

It must be placed inside the black class braces.

Problem: We can only write strings to the console even though *System.out.println(arg)* can deal with other variable types.

Overloading methods

We can overload the method: we declare two methods with the same name but different argument lists.

```
public class FirstProgram {
public static void main(String[] args) {
consoleOutput("Hello world"); }
public static void consoleOutput(String arg) {
System.out.println(arg);}
public static void consoleOutput(int arg) {
System.out.println(arg);} }
```

The correct static method is selected at compile time.

Methods with the return type *void* are called procedures. We will not distinguish between methods and procedures.

Return types

Methods can also return values (for example a String):

```
public class FirstProgram {
public static void main(String[] args) {
String test = consoleOutput("Hello world");
System.out.println(test);}
public static String consoleOutput(String arg) {
System.out.println(arg); return arg; } }
```

consoleOutput(arg) returns the string we send in.

Java allows us to put a method call into an argument list:

```
System.out.println(consoleOutput("Hello world"));
```

Motivating the need for heap memory

Let us vary the length of a string.

```
public class FirstProgram {

public static void main(String[] args) {
String arg = "Hello"; System.out.println(arg);
arg = "Hello world"; System.out.println(arg); }
```

We declare the string as *String arg;*

We declare and initialize the string as *String arg = "Hello"*;

We assign a new value to it by *arg = "Hello world"*;

The memory need of *arg* changes from 5 to 11 chars.

Expensive to do in the stack memory: the code must be stopped to rearrange the memory content.

Primitive and reference variables

Primitive variables like *int, char, float* have a fixed size that is known at compile time. They can go into the stack memory.

Strings can change their length while the code is running: they cannot be primitive variables.

Every byte of the computer's RAM has an address.

Addresses are defined as sequences of 4 (32 bit) or 8 (64 bit) hexadecimal numbers: *0-9, a-f* .

String arg; declares *arg* as a reference variable that contains a memory address. It points to a part of the heap memory.

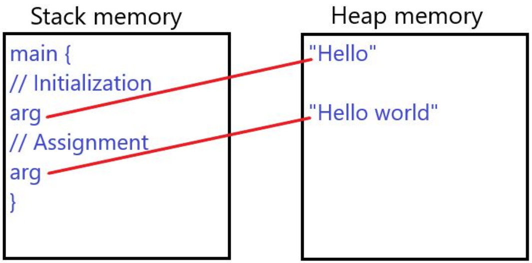
The size of *arg* is 32 bit or 64 bit: it has a fixed size. The reference variable *arg* can go into the stack memory.

Accessing heap memory

The content of the String goes into the heap.

```
public class FirstProgram {

public static void main(String[] args) {
String arg = "Hello"; // Initialization
arg = "Hello world"; // Assignment
System.out.println(arg);}
```



Why is the heap good?

```
String s1 = "Hello1"; // "Hello1" is placed at the beginning of the heap.
String s2 = "Hello2"; // "Hello2" is placed in the heap directly after "Hello1".
s1 = "Hello world"; // Does not fit into the original slot.
```

The JVM moves the content of s1 into a new memory segment with a starting address above those used by s2.

The JVM does not need to stop the running code of *main* in the stack memory.

In time, the heap memory fills up. Every now and then the code is stopped and the content in the heap memory is rearranged.

Java takes care of this for us (automatic garbage collection).

Filling a variable with content

```
int i = 10;
```

The declaration *int i;* reserves 4 bytes of memory for the integer. The initialization *i=10;* puts the content into *i*.

```
String s = "Hello world";
```

The declaration *String s;* reserves memory for a 32/64 bit address in stack.

"Hello world"; creates a memory segment 11 bytes long in the heap and fills it with the text. It returns its heap address.

The symbol = copies the returned heap address into *s*.

Declaration, instantiation, initialization, deletion

String s = "Hello world"; is the shorthand notation for

```
String s = new String("Hello");
```

String s is the declaration and *s* is a reference variable that can hold the address of a string variable.

The *new* operator reserves space for a string in the heap. This is called instantiation.

String("Hello"); initializes the memory segment in the heap.

new returns an address and = copies the address into *s*.

```
s = null; // setting the address in s to null separates reference
variable in stack from content in heap.
```

Summary

We discussed

- aspects of Java's memory management.
- static methods and procedures.
- method overloading.
- primitive variables and reference variables.