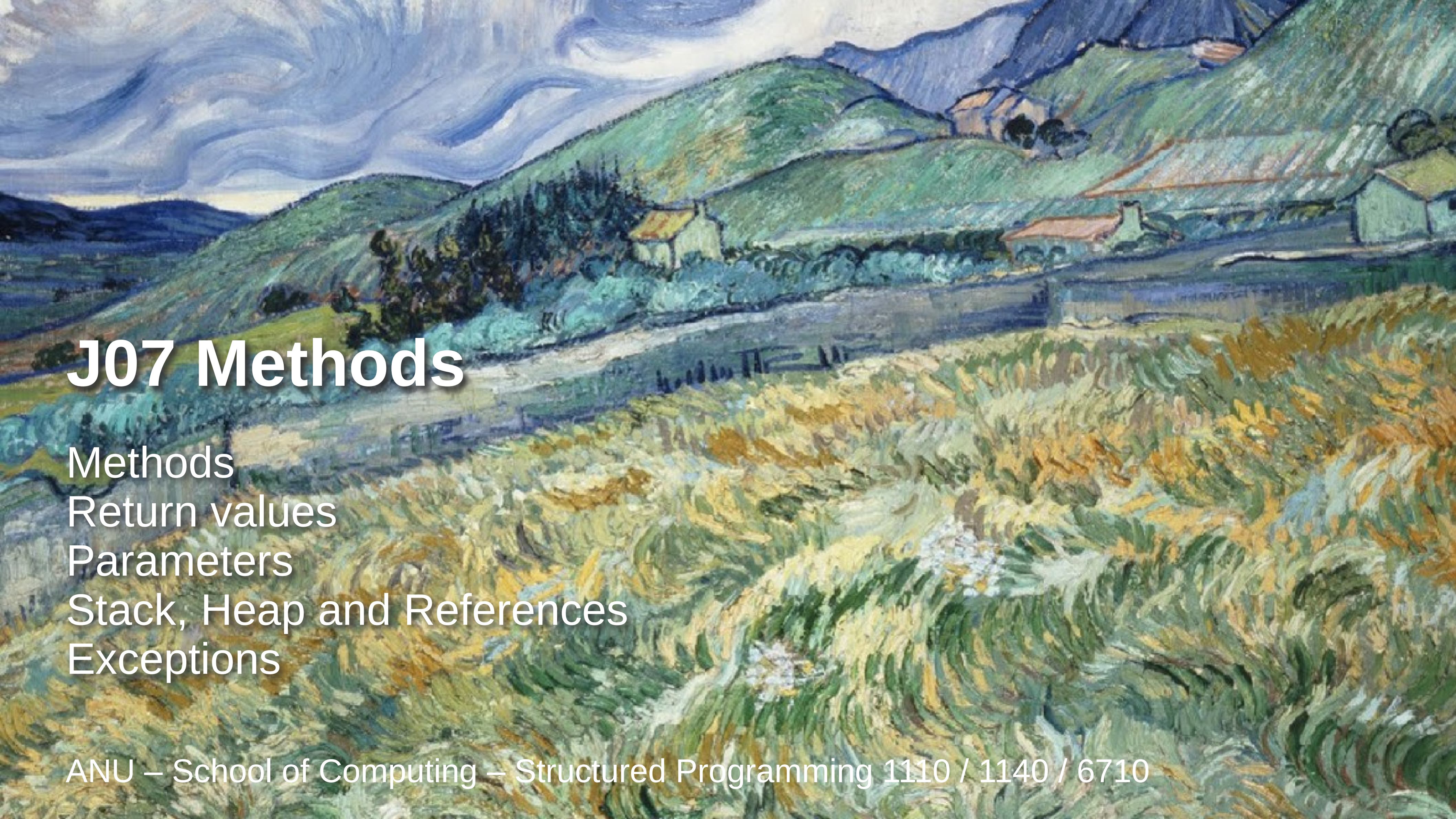


J07 Methods

Methods
Return values
Parameters
Stack, Heap and References
Exceptions

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

Methods

- A subroutine
  - Reusable code to perform a specific task
  - Modularity, encapsulation
- May take arguments (parameters)
- May return a value

Method Declaration

Method declarations will have the following, in order:

- Any **modifiers** (`public`, `private`, etc.)
- **return type**
- **method name**
- **parameters**, in parentheses
- Any **exceptions** the method may throw
- The method **body** (code)

```
class String {…
 public byte[] getBytes(String charsetName) throws UnsupportedEncodingException {
 … }
… }
```

Class and Instance methods

A method declared with the `static` modifier is a **class method** (otherwise it is an **instance method**).

- Class methods
  - May operate on **class fields** only
- Instance methods
  - May operate on class and **instance fields**

Returning a Value from a Method

The `return` statement exits the current method.

Methods `return` to caller when:

- all statements in method executed, or
- a `return` statement is reached, or
- the method throws an exception (later)

Methods declared `void` do not return a value.

All other methods must return a value of the declared type (or a subclass of the declared type, described later).

Parameters (method arguments)

Parameters are the mechanism for passing information from one method to another method (or constructor).

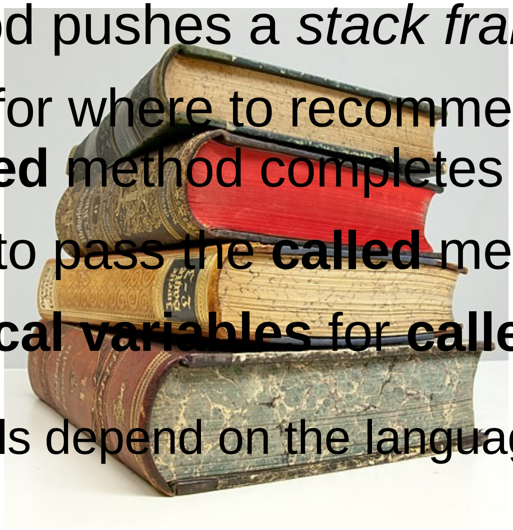
The syntax is easy, following method declaration add pass values / variables to appropriate type in the correct order:

```
    byte[] bytes = myString.getBytes(“UTF-8”);
```

The semantics of passing parameters is not so simple…

Warning! next few slides delve into some implementation details.

The Call Stack: Method after Method after...

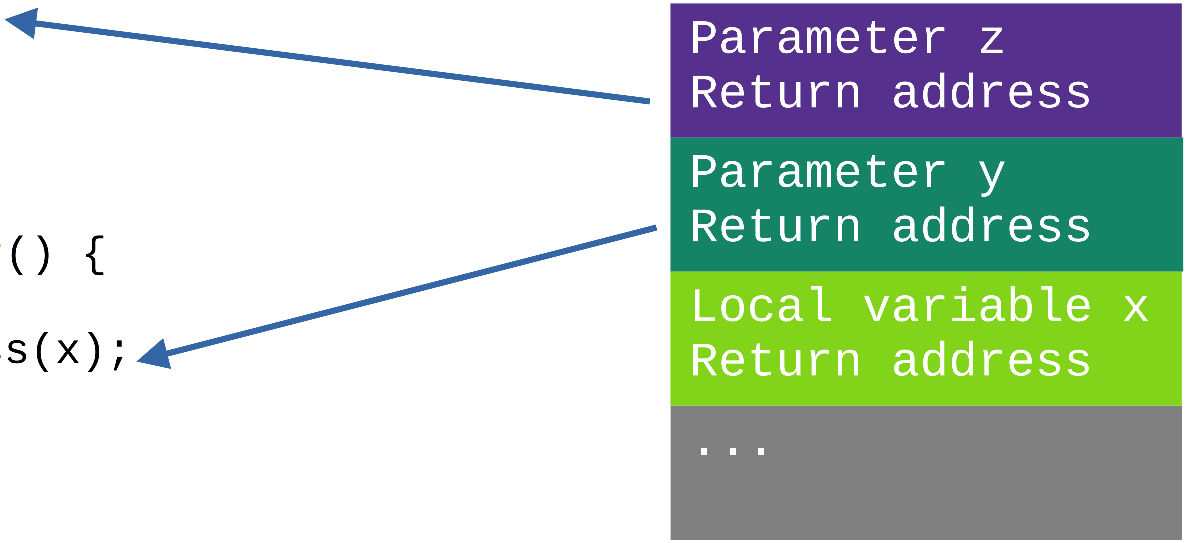


- Call Stack: a data structure that tracks method calls
  - Not directly interacted with in high-level languages like Java
  - Fast, efficient way to implement method calls
  - Each call to a method pushes a *stack frame* to the stack with*:
    - **Return address** for where to recommence execution in the **calling** method after **called** method completes
    - The **parameters** to pass the **called** method
    - Space to store **local variables** for **called** method

* Not specific to Java, the details depend on the language, compiler, instruction set, operating system etc...

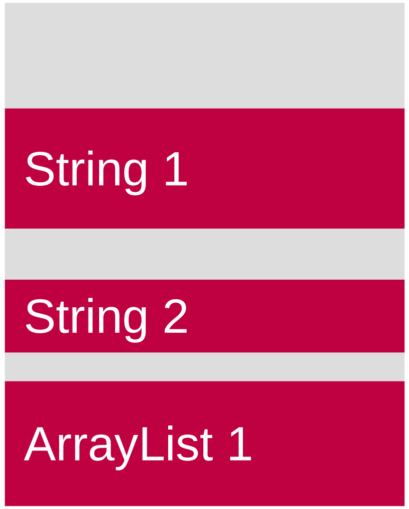
The Call Stack

```
static int twice(int z) {
    return 2 * z;
}

static int process(int y) {
    y = twice(y);
    y = y + 1;
    return y;
}

static int number() {
    int x = 100;
    return process(x);
}
```



The Heap

- In Java, objects (non-primative types) are *not directly* stored on the stack, rather they are stored (allocated) on *the heap*.
- **The heap**: a large region(s) of memory used to dynamically store objects, i.e. their instance fields.



Variables and References

Variables are for either:

- **Primitive Types**: value stored directly
- **Reference Types (all objects)**: “value” is a pointer / address to an object stored on the heap
  - We don’t directly interact with this pointer
  - Except, can be set to `null` (pointer to nowhere)
  - Method calls, fields automatically access the object pointed to
    - `NullPointerException` thrown if reference is `null`

Parameters

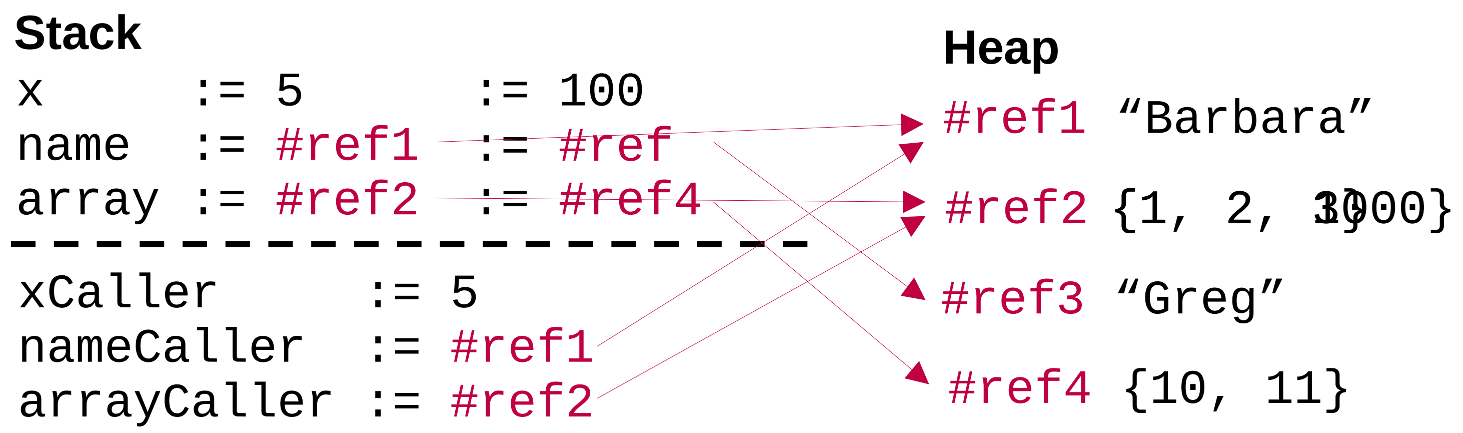
- **Primitive types** passed by value (copied into stack frame)
  - Changes to parameter are **not seen** by caller
- **Reference types** passed by value (copied into stack frame)
  - Changes to the *reference* are **not seen** by caller
  - Changes* to *object referred* to **are seen** by caller

* Some types (e.g., String) are designed to be *immutable* – no public methods modify any class or instance fields.

Parameter Passing

```java
public static void main(…) {
    int xCaller = 5;
    String nameCaller = "Barbara"
    int[] arrayCaller = new int[] {1, 2, 3};
    method(xCaller, nameCaller, arrayCaller);
}
```

```java
static void method(int x,
                   String name,
                   int[] array) {
    x = 100;
    name = "Greg";
    array[2] = 1000;
    array = new int[]{10, 11};
}
```



```
Stack                                       Heap
x      := 5        := 100                   #ref1 “Barbara”
name   := #ref1    := #ref
array  := #ref2    := #ref4                 #ref2 {1, 2, 3}1000}
---------------------
xCaller     := 5                            #ref3 “Greg”
nameCaller  := #ref1
arrayCaller := #ref2                        #ref4 {10, 11}
```

Exceptions Basics

- A method can either execute normally and return a value (passing execution back to caller), or throw an exception to signal something went wrong.
- When an exception is thrown, exception control flow kicks in: *unwinds* the call stack until either a method further down the stack “handles” the exception, or the process exits.
- We will revisit the types of exception and how to catch them later on. For now you will just likely want exceptions to crash your program so it is obvious something went wrong.

Image Attributions

- By Ch. Maderthoner - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=37987842
- By Vysotsky - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=81482154