The runtime stack

Where do variables live? The local variables of a method live inside the method's activation record (also called stack frame).

(But all objects are on the heap. The stack frame only stores the variable names and references to the heap.)

When a method starts executing, its stack frame is created. When the method returns, its stack frame is destroyed.

The runtime system (JVM) keeps stack frames on a stack.

The top of the stack is the stack from of the currently executing method. A stack is suitable for storing stack frames, since the start and return time of methods form a nesting structure (like balanced parentheses).

(This stack is built into the JVM! It is not a Scala object—we cannot access the stack of activation records ourselves.)

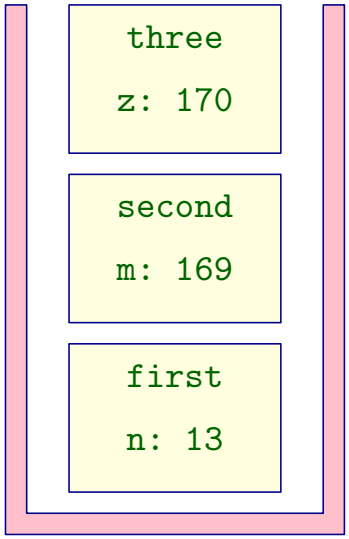
Example

```
def first(n: Int) {
  second(n)
  second(n * n)
}

def second(m: Int) {
  three(m)
  three(m+1)
  three(m+2)
}

def three(z: Int) {
  Thread.dumpStack()
}
first(13)
```

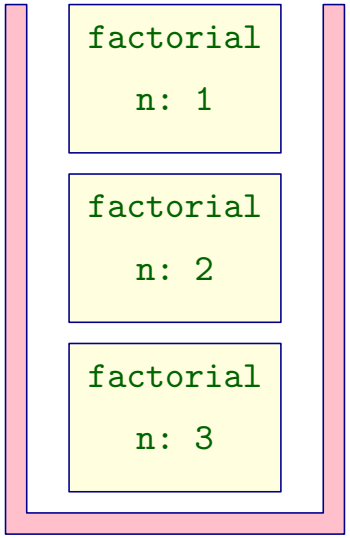
example1.scala, example2.scala



Recursion

The runtime stack makes recursion possible.

```
def factorial(n : Int) : Long = {
  if (n <= 1)     // base case
    1
  else
    n * factorial(n - 1)
}
```



Exceptions

When a runtime error occurs, the program terminates with an exception message:

```
scala> val a = 3
a: Int = 3
scala> a / 0
java.lang.ArithmeticException: / by zero
scala> val s = "abc"
s: java.lang.String = abc
scala> s.toInt
java.lang.NumberFormatException:
  For input string: "abc"
scala> val F = scala.io.Source.fromFile("test.txt")
java.io.FileNotFoundException: test.txt
  (No such file or directory)
```

Handling exceptions

If an exception occurs inside a `try` clause, execution continues with a matching exception handler in the `catch` clause:

```
val str = readLine("Enter a number> ")
try {
  val x = str.toInt
  printf("You said: %d\n", x)
} catch {
  case e: NumberFormatException =>
    printf("'%s' is not a number\n", str);
}
```

Exceptions keep the interface of the method `toInt` clean. (Compare the C function `strtol`.)

catch1.scala

Catching across function calls

```
def test(s: String): Int = {
  (s.toDouble * 100).toInt
}
def show(s: String) {
  try {
    println(test(s))
  } catch {
    case e: NumberFormatException =>
      println("Incorrect input")
  }
}
```

```
scala> show("123.456")
12345
scala> show("123a456")
Incorrect input
```

catch2.scala

Handling exceptions

If an exception occurs, the normal flow of control is interrupted. Execution continues in the innermost `catch` block with a matching exception handler.

```
def f(n: Int) =  g(n)

def g(n: Int) {
  val m = 100 / n
  printf("The result is %d\n", m)
}

try {
  f(n)
} catch {
  case e: ArithmeticException
  => println("I can't handle this value!")
}
```

except1.scala

Throwing exceptions

When we detect an error in the input data, we can throw an exception ourselves:

```
if (n < 0)
  throw new IllegalArgumentException
```

except2.scala

Exceptions are often used to detect errors in the input data.

We can catch the exception at a suitable place in the program and print an error message, or handle the problem in some other way.

The exception may happen deep inside several function calls:

```
Welcome to KAIST SuperCalculator!
> 3 + 5 * (12.0 + (4 + 6.0 * @))
Syntax error
```

Throwing exceptions

Exceptions are Scala objects derived from `Exception`:

```
class SyntaxError extends Exception
```

When we detect a situation that we cannot handle locally, we can throw an exception:

```
if (!tok.token.isSymbol(")"))
  throw new SyntaxError
```

Exceptions are normal objects and can have additional fields and methods.

calculator3.scala