Programming paradigms

Scala and functional programming

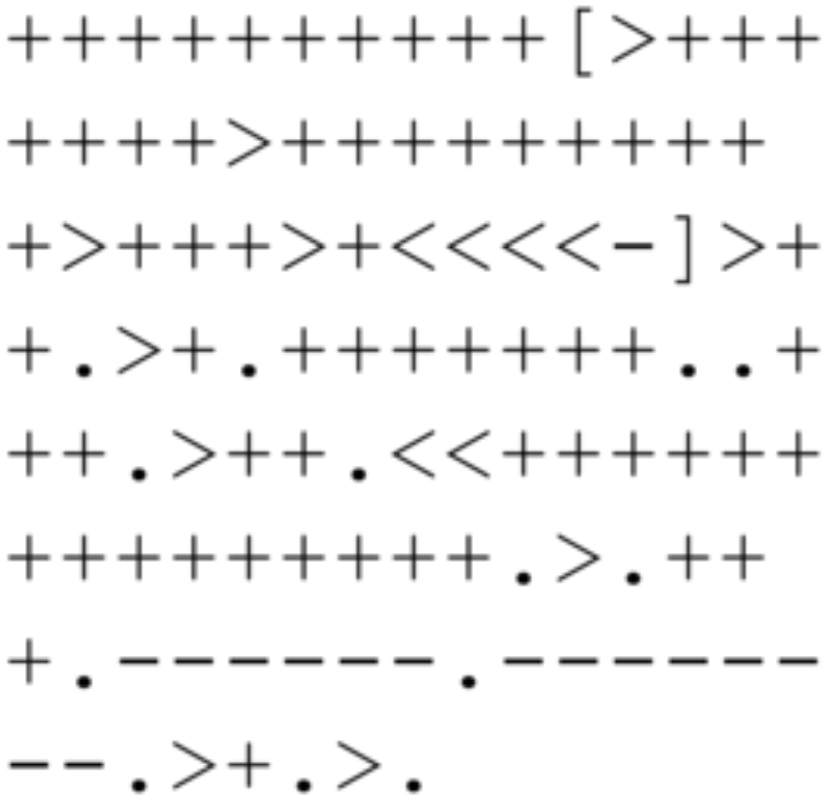
Semantic

Syntax

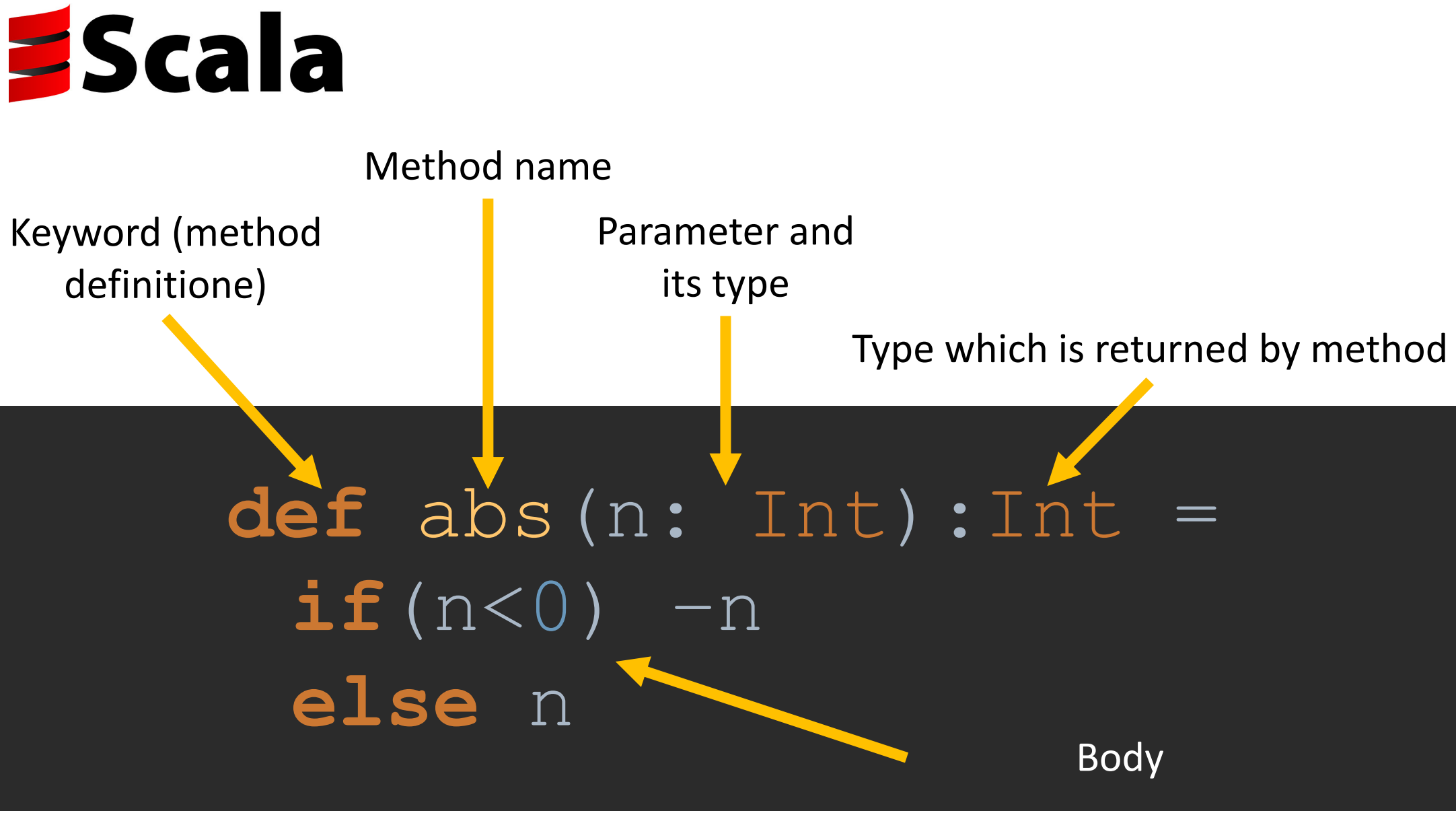
the meaning of…

the form or structure of…

…expression, statement and program unit



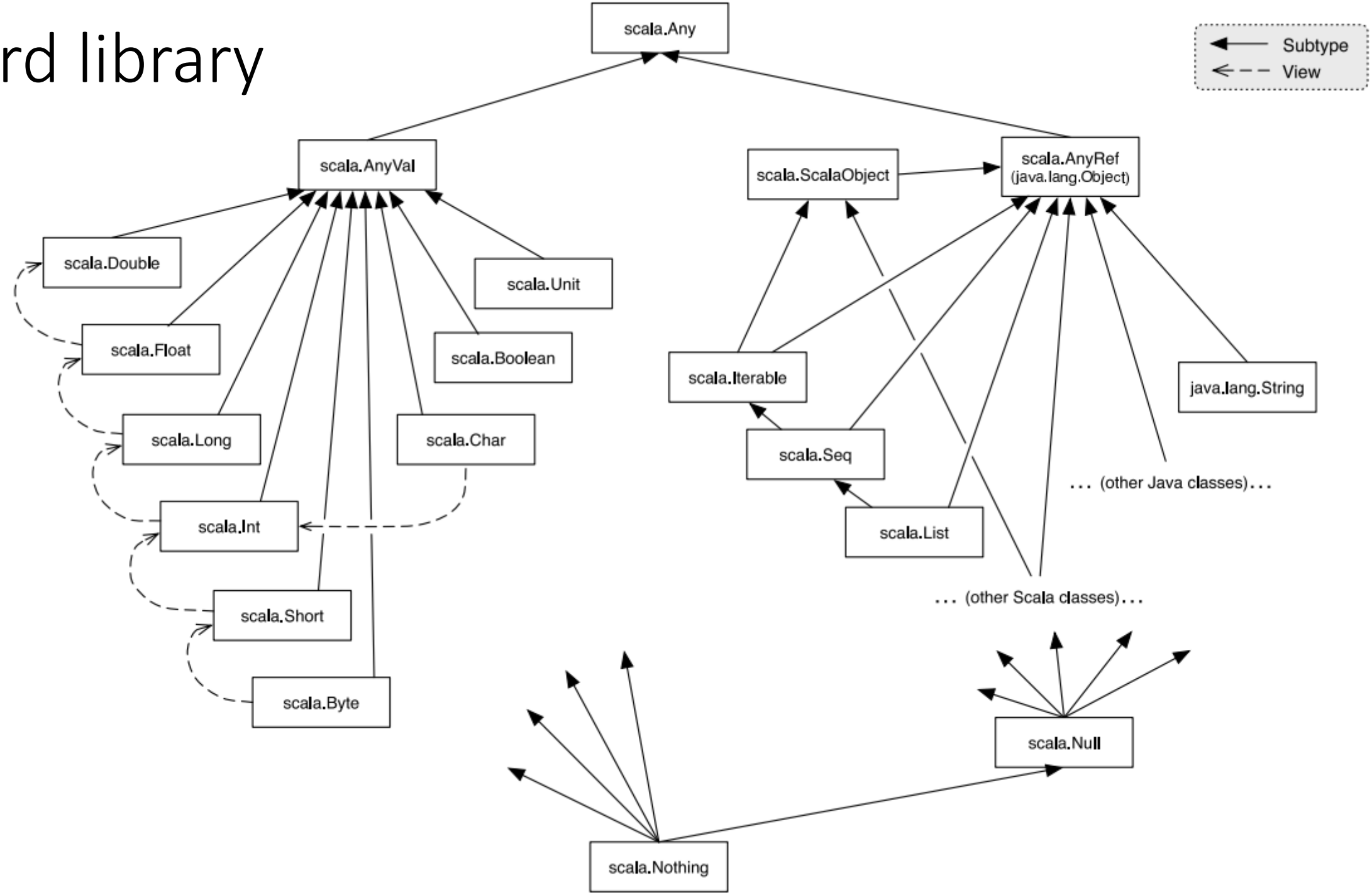
Unfamiliar syntax != complexity


Scala
Method name
Keyword (method definitione)
Parameter and its type
Type which is returned by method
def abs(n: Int):Int =
if(n<0) -n
else n
Body

Data Types

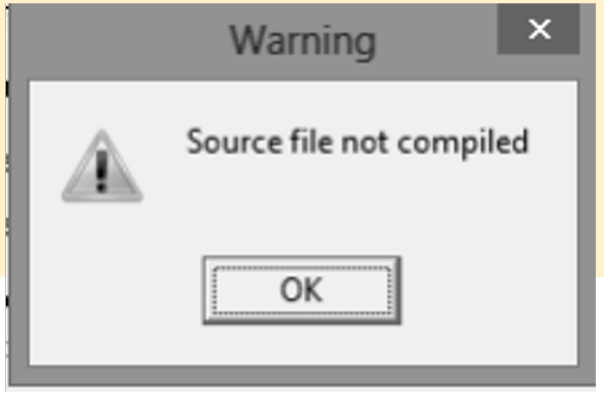
Byte - 8 bit signed value. Range from -128 to 127
Short - 16 bit signed value. Range -32768 to 32767
Int - 32 bit signed value. Range -2147483648 to 2147483647
Long - 64 bit signed value. -9223372036854775808 to 9223372036854775807
Float - 32 bit IEEE 754 single-precision float
Double - 64 bit IEEE 754 double-precision float
Char - 16 bit unsigned Unicode character. Range from U+0000 to U+FFFF
String - A sequence of Chars
Boolean - Either the literal true or the literal false
Unit - Corresponds to no value
Null - null or empty reference
Nothing - The subtype of every other type; includes no values
Any - The supertype of any type; any object is of type Any
AnyRef - The supertype of any reference type

Standard library



Expression vs Statement based language

```
if (x > 0) {
    true;
} else {
    false;
}
```



Expression vs Statement based language

```
boolean value;
if (x > 0) {
     value = true;
} else {
     value = false;
}
```

Java is statement based language!

Expression vs Statement based language

```
boolean value;
if (x > 0) {
     value = true;
}
```

Expression vs Statement based language

```
if (x > 0) {
    true
} else {
    false
}
```

Scala control flow is expression based language!

Expression vs Statement based language

```
val value = if (x > 0) true else false
```

Expression vs Statement based language

Statements have side effect

Expressions evaluate to values

Referential transparency

An expression is referentially transparent if it can be replaced with its value without changing the program’s behavior.

Referential Transparency

```
val x: Int = ???
val y = x*x
```

```
val x = 20
```

```
val x = { println("PP"); 20 }
```

NOTE: ??? - let you write a not-yet implemented method

Referential transparency

“Pure function” is a common term in functional world that is tightly related to RT

Lets assume that all functions that are referentially transparent are called “pure functions”.*

https://www.reddit.com/r/haskell/comments/21y560/purity_and_referential_transparency_are_different/

```
def abs(n: Int):Int =
  if(n<0) -n
  else n
```

```
def abs(n: Int)=
  if(n<0) -n
  else n
```

Anonymous function

Anonymous function:

```
((x:Int) => x+x) (2)
```

A functional expression can be associated with an identifier:

```
val double = (x:Int) => x+x
double(2)
```

```
def double(x:Int) = x+x
```

Currying:
uncurried and curried representation

Currying is the process of transforming a function that takes multiple arguments into a function that takes just a single argument and returns another function if any arguments are still needed.

```
def plus(x:Int, y:Int) = x+y
plus(2,1)

def plus2(x:Int)(y:Int) = x+y
plus2(2)(1)
```

Why we may need currying

```
public static int add(int x, int y){
    return x+y;
}

public static int add7(int x){
    return add(x, 7);
}
```

Why we may need currying

```
public static int multiply(int x, int y){
    return x*y;
}

public static int multiplyBy8(int x){
    return multiply(x, 8);
}
```

Why we may need currying

```
public static int add7ThenMultiplyBy8(int x){
    return multiplyBy8(add7(x));
}
```

Why we need currying

```
def add(x: Int, y: Int) = x + y
def add7(x: Int) = add(x, 7)


def addCurring(x: Int)(y: Int): Int = x + y


val add7to = addCurring(7) _
```

Why we need currying

```
def add(x: Int, y: Int) = x + y
def add7(x: Int) = add(x, 7)

def addCurring(x: Int)(y: Int): Int = x + y

val add7To = addCurring(7) _

val add7ToX = (add _).curried(7)
```

Why we need currying

```
def add(x: Int)(y: Int): Int = x + y

def multiply(x: Int)(y: Int): Int = x * y

def multiplyBy8TheNumberPlus7 =  multiply(8) _
    compose(add(7))

def add7ThenMultiplyBy8 =  add(7) _
    andThen(multiply(8))
```

Recursion

Mutual recursion:
two or more functions call each other.

```
def evenR(n: Int): Boolean =
  if (n==0) true else oddR(n-1)
def oddR(n: Int) :Boolean =
  if (n==0) false else evenR(n-1)
```

Recursion

Do we have any problem here?

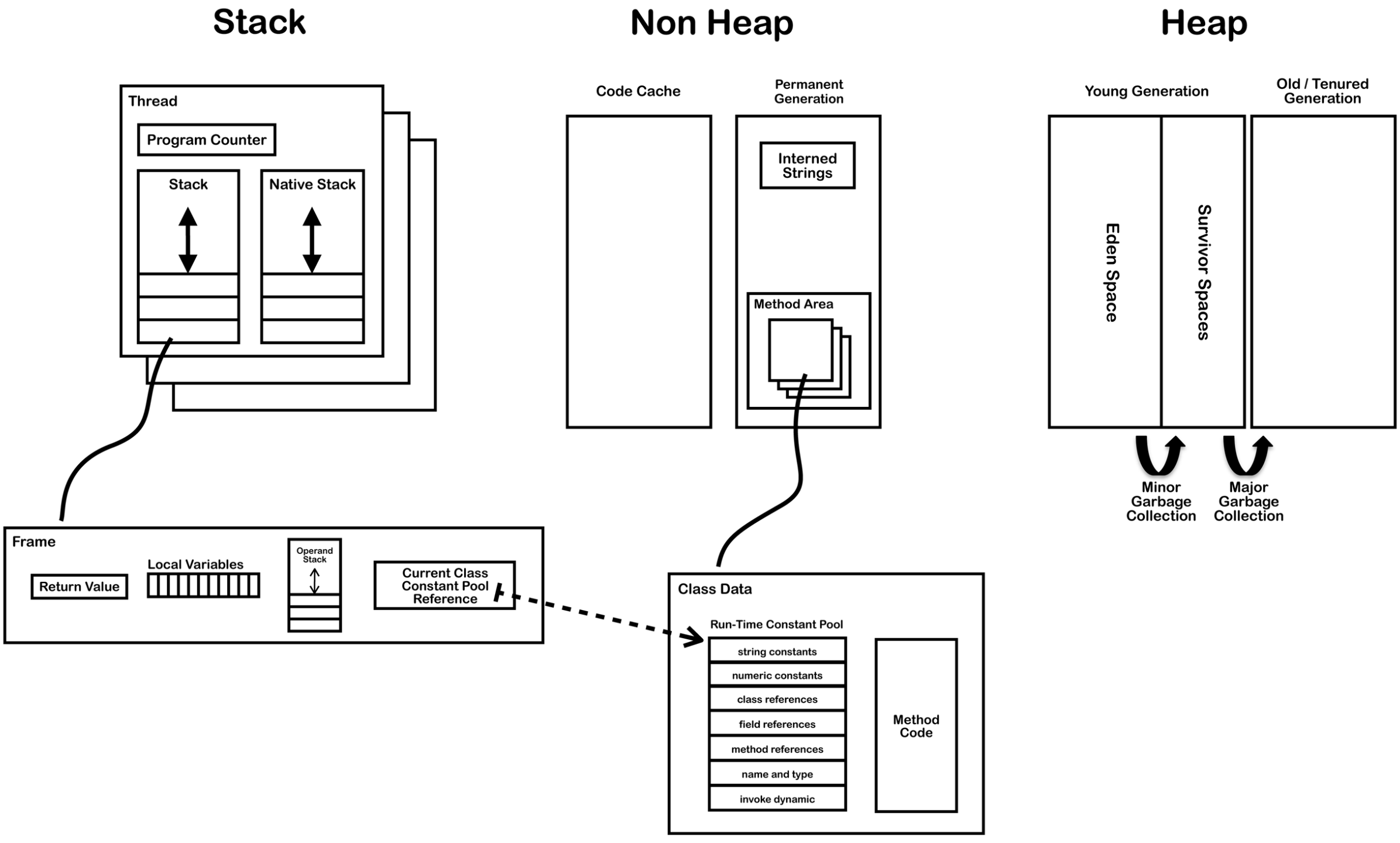
```
def fibo( n : Int) : Int     {
  if(n == 0 || n == 1
  else fib1( n-1 ) + fib1( n-2 )
}
```

Stack

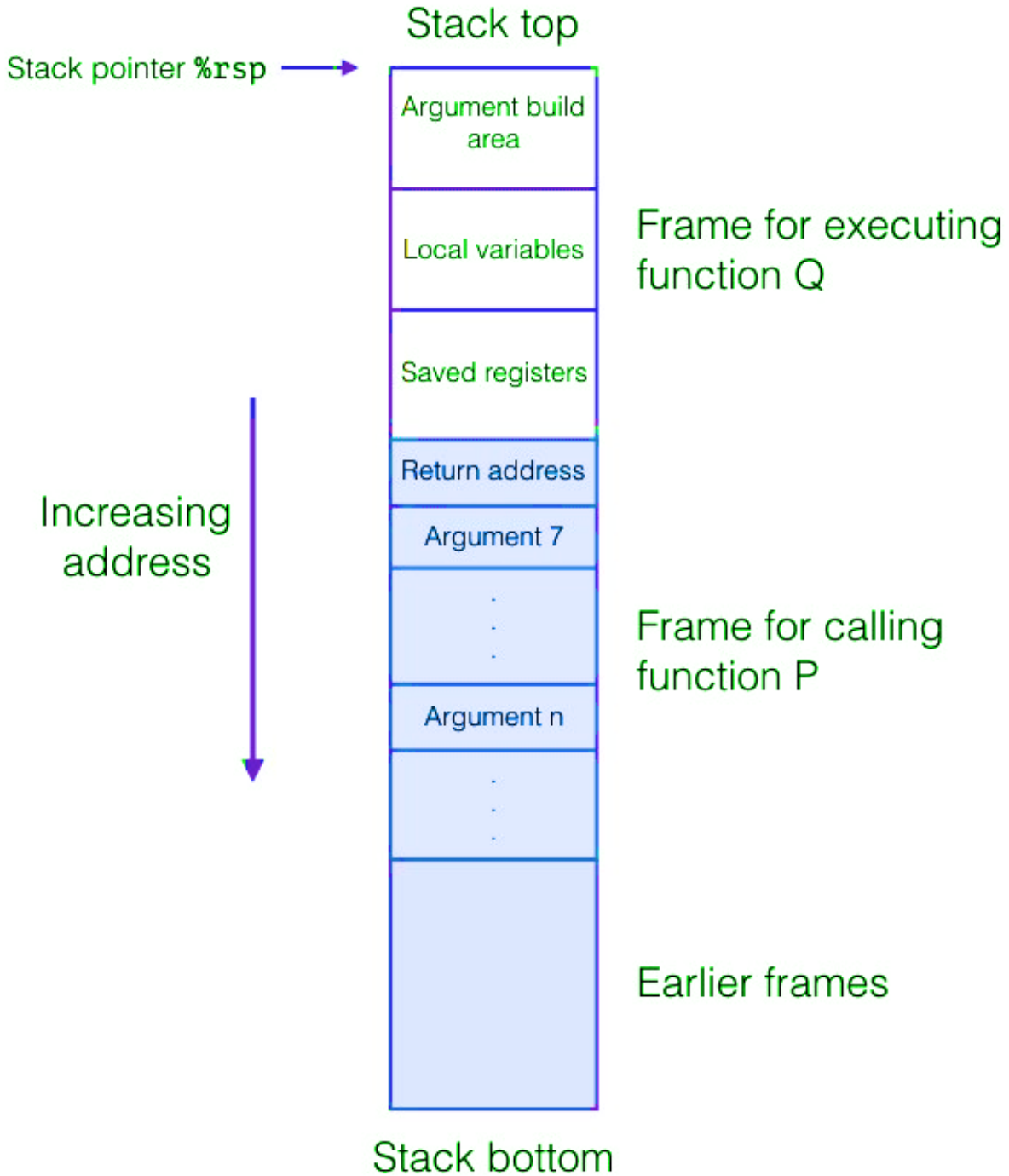
Heap

Stack and heap

Stack is used for static memory allocation
and
Heap for dynamic memory allocation,
both stored in the computer's RAM.


Stack
Thread
Program Counter
Stack
Native Stack
Frame
Return Value
Local Variables
Operand Stack
Current Class Constant Pool Reference
Non Heap
Code Cache
Permanent Generation
Interned Strings
Method Area
Class Data
Run-Time Constant Pool
string constants
numeric constants
class references
field references
method references
name and type
invoke dynamic
Method Code
Heap
Young Generation
Old / Tenured Generation
Eden Space
Survivor Spaces
Minor Garbage Collection
Major Garbage Collection

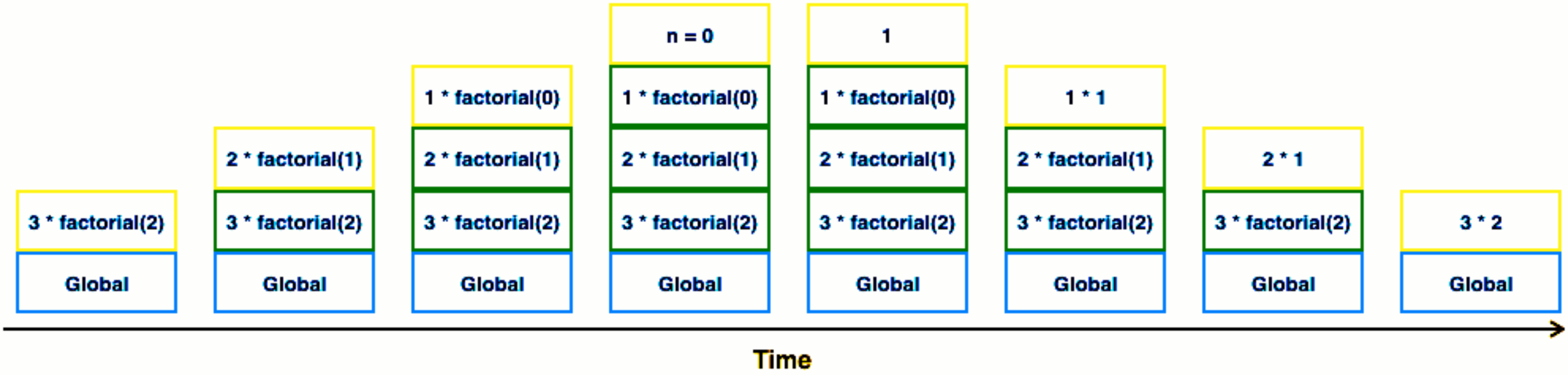
JVM Stack

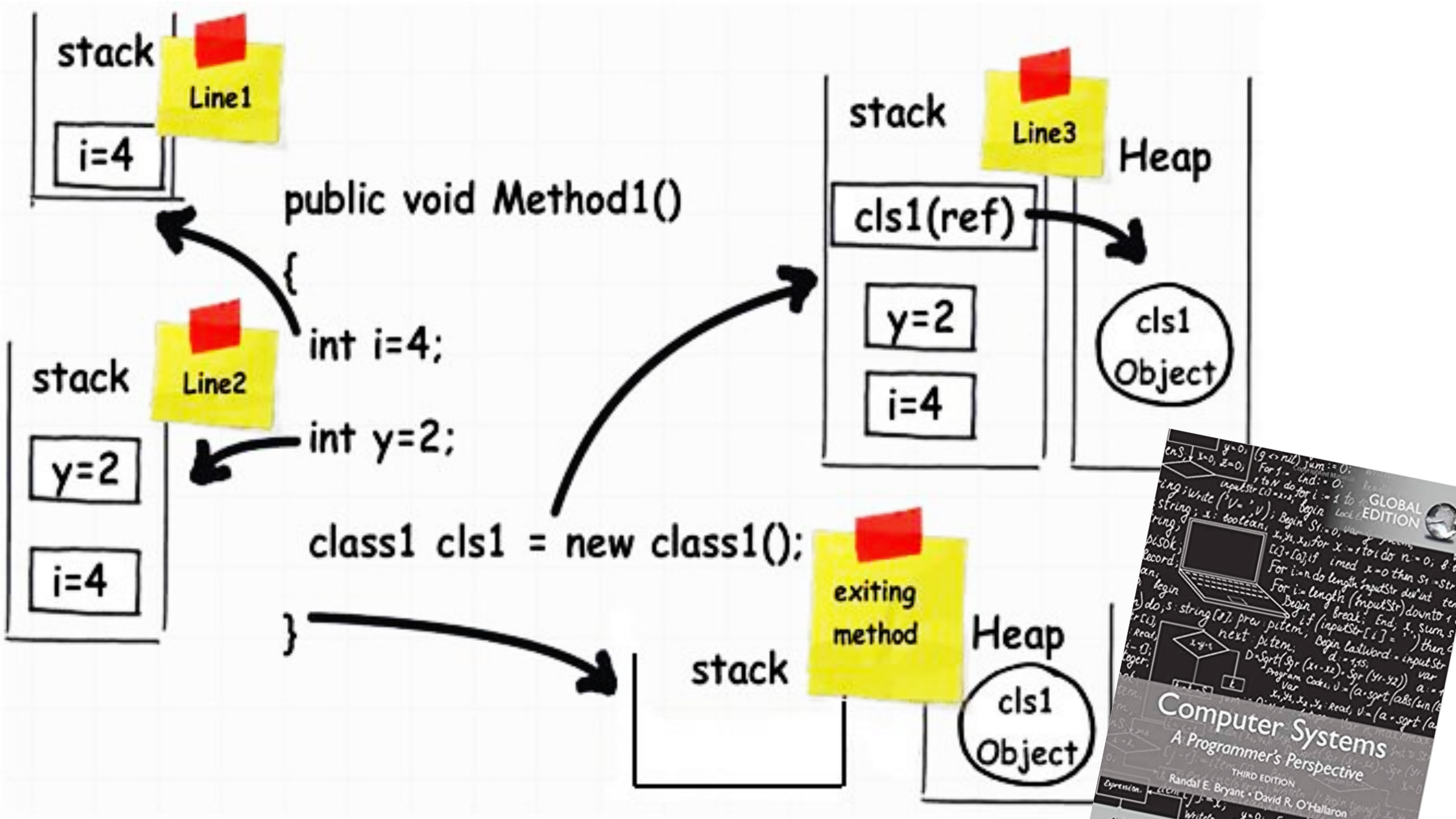

Stack top
Stack pointer %rsp
Argument build area
Local variables
Saved registers
Return address
Argument 7
Argument n
Increasing address
Frame for executing function Q
Frame for calling function P
Earlier frames
Stack bottom

JVM Stack

- Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method.
- Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method.
- As soon as method ends, the block becomes unused and become available for next method.
  Stack memory size is very less compared to Heap memory.

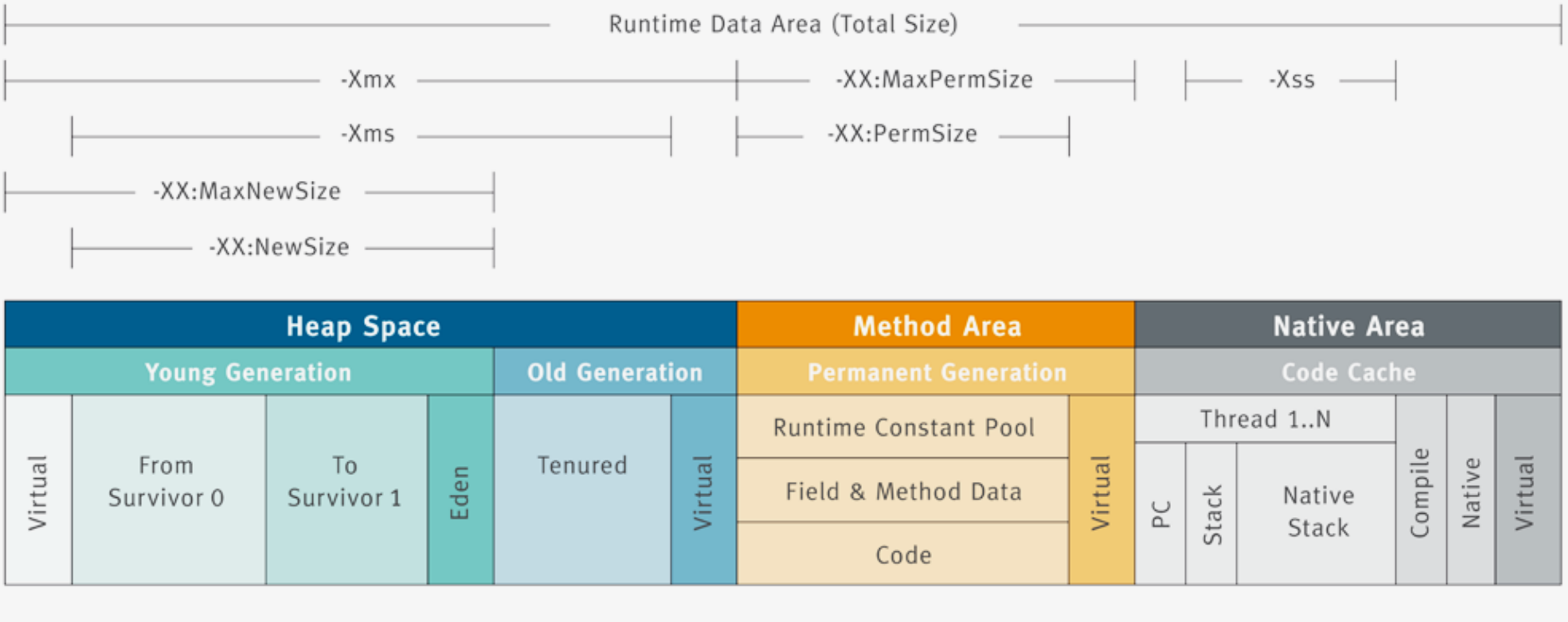
Recursion and the stack




stack
Line1
i=4
public void Method1()
{
int i=4;
stack
Line2
y=2
i=4
int y=2;
class1 cls1 = new class1();
}
stack
exiting method
stack
cls1(ref)
Line3
y=2
i=4
Heap
cls1 Object
Heap
cls1 Object
GLOBAL EDITION
Computer Systems
A Programmer's Perspective
THIRD EDITION
Randal E. Bryant • David R. O'Hallaron

JVM Heap

- Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it’s always created in the Heap space.
- Garbage Collection runs on the heap memory to free the memory used by objects that doesn’t have any reference. Any object created in the heap space has global access and can be referenced from anywhere of the application


Runtime Data Area (Total Size)
-Xmx
-XX:MaxPermSize
-Xss
-Xms
-XX:PermSize
-XX:MaxNewSize
-XX:NewSize
Heap Space
Method Area
Native Area
Young Generation
Old Generation
Permanent Generation
Code Cache
Virtual
From Survivor 0
To Survivor 1
Eden
Tenured
Virtual
Runtime Constant Pool
Field & Method Data
Code
Virtual
Thread 1..N
PC
Stack
Native Stack
Compile
Native
Virtual

Tail call / Tail recursion

Tail call is nothing else as initiate a function at the end of the current function.
Tail recursion is a call the same function at the end of this function.

```
def succTail(n: Int) = {
  def succIter(n:Int, accum:Int): Int =
    if (n==0) accum else succIter(n-1, accum+1)
  succIter(n,1)
}
```

Now, with tail recursion, code optimization is applied:
another stack frame is NOT ADDED to the call stack.

Tail recursion – limitations (Scala)

In Scala, tail recursion is optimized if the function calls itself.

In other cases, this optimization is not carried out because on a Java Virtual Machine (JVM) this is very "difficult".

So mutually recursive functions don't have this optimization ☹


DETAIL

Progrmmaing language

Static/Dynamic Typing is about **when** type information is acquired (Either at compile time or at runtime)

Strong/Weak Typing is about **how strictly** types are distinguished (e.g. whether the language tries to do an implicit conversion from strings to numbers).

You Should
Know This!

Example (Static/Dynamic)

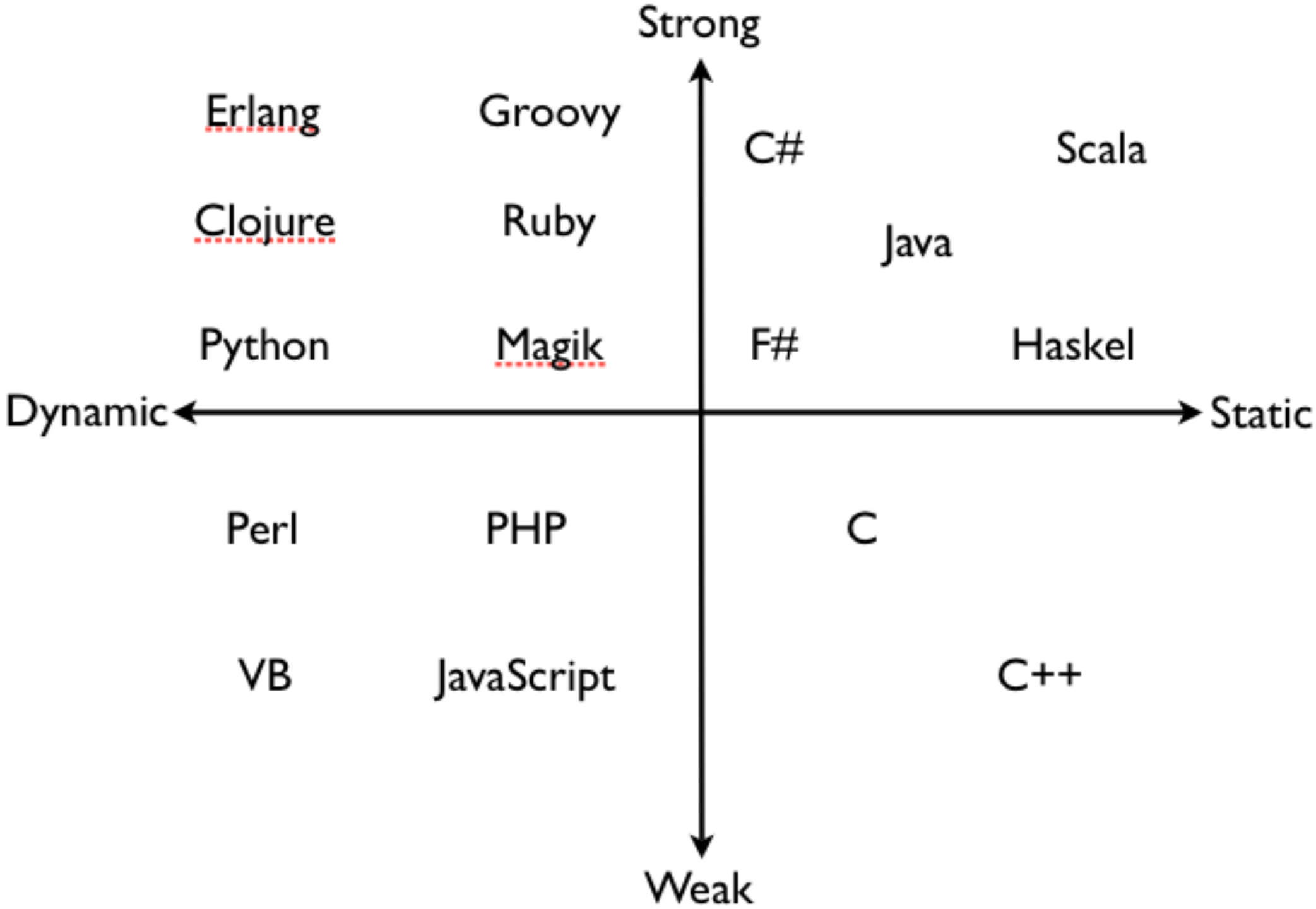
```
String str = "Hello"; //statically typed as string
str = 5; //would throw an error
//since java is statically typed
```

```
str = "Hello" # it is a string
str = 5 # now it is an integer
```

Example (Strong/Weak)


```
$str = 5 + "hello"; // equals 5 because "hello"
//is implicitly casted to 0
// PHP is weakly typed.
```

```
str = 5 + "hello" # would throw an error \
#since it does not want to cast one type
#to the other implicitly.
```


Strong
Erlang
Groovy
C#
Scala
Clojure
Ruby
Java
Python
Magik
F#
Haskel
Dynamic
Static
Perl
PHP
C
VB
JavaScript
C++
Weak

Pattern matching

Machers

```
val x = (false, 10)
val (z,y) = x
???
```

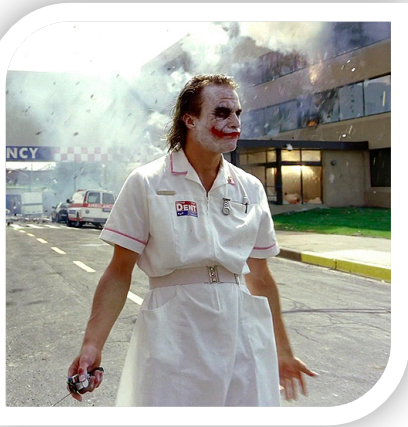
```
val x = (false, 10)

val (false,y) = x
val (true,z) = x
???
```

Definicje zmiennych z dopasowaniem

```
val xs = List("Ala", "ma", "kota")
val List(x1, x2, x3) = xs
val h::t = xs
```

Wildcard (Joker)



Wildcard like a variable, fits with each value but does not form a binding.

```
val (z,_) = (false, 10)
```

match

```
def imply1(pb:(Boolean,Boolean)) =
  pb match {
    case (false,false) => true
    case (false,true) => true
    case (true,false) => false
    case (true,true) => true
  }

def imply2(pb:(Boolean,Boolean)) =
  pb match {
    case (true,false) => false
    case _ => true
  }
```

Guarded matches

```
def avg1(p:(Double,Double)) =
  p match {
    case (x,x) => x // < --
    case (x,y) => (x+y)/2.0
  }

def avg2(p:(Double,Double)) =
  p match {
    case (x,y) if x == y => x
    case (x,y) => (x+y)/2.0
  }
```

Guarded matches

```
def avg3(p:(Double,Double)) = {
  val (x,y) = p
  if (x==y) x else (x+y)/2.0
}
```

michal.szczepanik@pwr.edu.pl