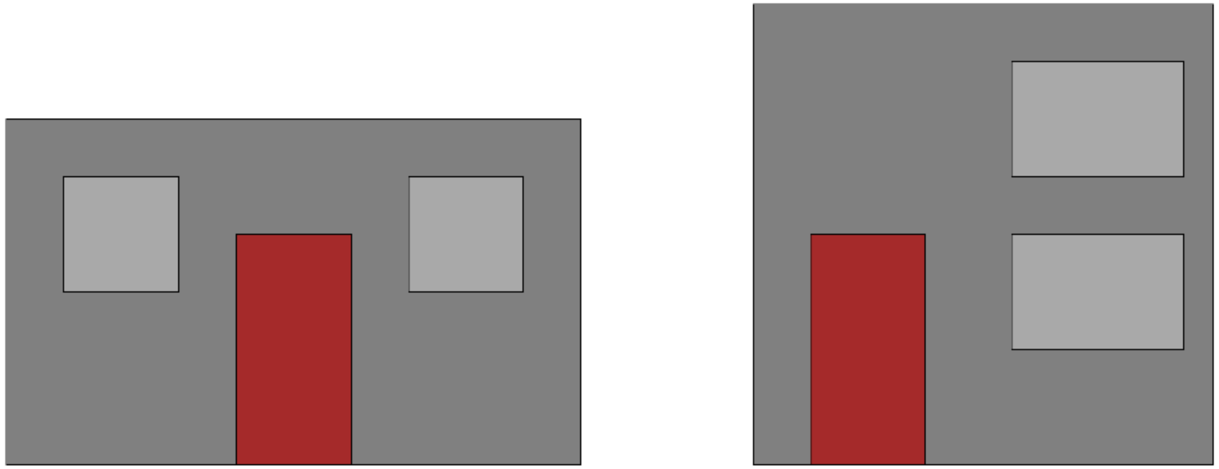
EPFL

Methods and Parameters

Effective Programming in Scala

Variability and Similarity

Consider that your neighbours also want to paint their houses. All the houses have one door and two windows, however they don't all have the same size:



Alice's house and Bob's house.

Can you tell which house requires more paint?

Variability and Similarity (2)

Here is the program that computes the surface to paint on Alice's house:

```
val facade = 5 * 3
val door   = 2 * 1
val window = 1 * 1
facade - door - window * 2
```

And on Bob's house:

```
val facade = 4 * 4
val door   = 2 * 1
val window = 1.5 * 1
facade - door - window * 2
```

Variability and Similarity (2)

Here is the program that computes the surface to paint on Alice's house:

```
val facade = 5 * 3
val door   = 2 * 1
val window = 1 * 1
facade - door - window * 2
```

And on Bob's house:

```
val facade = 4 * 4
val door   = 2 * 1
val window = 1.5 * 1
facade - door - window * 2
```

Note how **similar** these two programs are.

Principle of Abstraction

> *Where* ***similar*** *functions are carried out by* distinct *pieces of code, it is generally beneficial to combine them into* one *by abstracting out the* ***varying*** *parts.*

Benjamin C. Pierce

What are the benefits?

Principle of Abstraction

> *Where* ***similar*** *functions are carried out by* distinct *pieces of code, it is generally beneficial to combine them into* one *by abstracting out the* ***varying*** *parts.*

Benjamin C. Pierce

What are the benefits?

- Reduces maintenance: the implementation is centralized in one place,
- Structures code into abstraction levels.

Principle of Abstraction

> *Where* ***similar*** *functions are carried out by* distinct *pieces of code, it is generally beneficial to combine them into* one *by abstracting out the* ***varying*** *parts.*

Benjamin C. Pierce

What are the benefits?

- Reduces maintenance: the implementation is centralized in one place,
- Structures code into abstraction levels.

What are the **varying** parts in our case?

Principle of Abstraction

> *Where* ***similar*** *functions are carried out by* distinct *pieces of code, it is generally beneficial to combine them into* one *by abstracting out the* ***varying*** parts.

Benjamin C. Pierce

What are the benefits?

- Reduces maintenance: the implementation is centralized in one place,
- Structures code into abstraction levels.

What are the **varying** parts in our case?

- The areas of windows and facades.

Methods

We can **abstract out** the facade and windows areas by defining a **method**:

```
val door = 2 * 1
def house(facade: Double, window: Double) =
  facade - door - window * 2
```

And then, we can compute the area of Alice's house:

```
house(5 * 3, 1 * 1)
```

And of Bob's house:

```
house(4 * 4, 1.5 * 1)
```

Methods

```
def house(facade: Double, window: Double) =
  facade - door - window * 2
```

- **Methods** are introduced by the `def` keyword
- The method `house` takes two **parameters**: `facade` and `window`
- We have to specify the type of method parameters

Digression: Type Widening

```
def house(facade: Double, window: Double) = ...
```

We have decided to use the Double data type to carry the areas of the facade and the windows, because Bob's house dimensions could not be represented in Int values (the area of his windows is 1.5 m^2).

```
house(5 * 3, 1 * 1)
```

When we write a literal number such as 42 where the compiler expects a value of type Double, it automatically widens the type of the literal number from Int to Double.

Note that it doesn't work the other way around: a floating-point literal such as 3.14 will be rejected by the compiler if an Int is expected.

Blocks

The body of a method can span several lines:

```
def house(facade: Double, window: Double): Double =
  val door = 2 * 1
  val subtractedArea = door + window * 2
  facade - subtractedArea
```

This is useful to introduce intermediate definitions before returning the result.

All the lines with the same *level of indentation* make a block. The block ends with the resulting expression.

End of Definitions

Optionally, you can mark the end of a definition with an end clause:

```
def house(facade: Double, window: Double): Double =
  val door = 2 * 1
  val subtractedArea = door + window * 2
  facade - subtractedArea
end house
```

The end clauses are optional, their purpose is only to improve readability.

Blocks (Scala 2 Compatibility)

In Scala 2, blocks had to be delimited by braces (indentation was not significant):

```scala
def house(facade: Double, window: Double): Double = {
  val door = 2 * 1
  val subtractedArea = door + window * 2
  facade - subtractedArea
}
```

Scala 3 still supports this syntax.

Explicit Result Type

```
def house(facade: Double, window: Double): Double =
  val door = 2 * 1
  val subtractedArea = door + window * 2
  facade - subtractedArea
```

Although this is not necessary, we recommend writing explicit result types for methods. This allows developers reading the code to immediately see the return type of a method, without having to go through its entire implementation.

Lexical Scope

It is worth noting that names introduced within a block are not visible from the outside of that block:

```
val tenSquared: Int =
  val ten = 10
  ten * ten

ten
```

Lexical Scope

It is worth noting that names introduced within a block are not visible from the outside of that block:

```
val tenSquared: Int =
  val ten = 10
  ten * ten

ten
```

error: not found: value ten

Exercise

Find out the similarities and varying parts in these two programs and introduce a more general program that can be used to implement them.

(Note: single-line comments in Scala use the same syntax as in C, C++, Java, etc.)

> *How many minutes does it take for Alice and Bob to run a marathon, given their average speed (in km/h)?*

```scala
val distance = 42.195 // (km)
val speed    = 12 // (km/h)
val duration = distance / speed
duration * 60 // (minutes)
```

```scala
val distance = 42.195 // (km)
val speed    = 14 // (km/h)
val duration = distance / speed
duration * 60 // (minutes)
```

Exercise (2)

```
def marathonDuration(speed: Double): Double =
  val distance = 42.195 // (km)
  val duration = distance / speed
  duration * 60 // (minutes)
```

Exercise (2)

```
def marathonDuration(speed: Double): Double =
  val distance = 42.195 // (km)
  val duration = distance / speed
  duration * 60 // (minutes)

marathonDuration(12)

marathonDuration(14)
```

Named Parameters

Sometimes it helps readability to explicitly mention the name of the parameter for which we pass an argument (especially when the arguments are literal values):

```
marathonDuration(speed = 12)
```

Summary

Parameters allow us to:

- implement a program with unknown inputs
- **apply** the same program to different sets of inputs

Blocks let us write intermediate definitions before returning a result. Definitions that are inside a block are not visible outside of the block.