Functions and Data

Functions and Data

In this section, we'll learn how functions create and encapsulate data structures.

Example

Rational Numbers

We want to design a package for doing rational arithmetic.

A rational number $\frac{x}{y}$ is represented by two integers:

- its *numerator* x, and
- its *denominator* y.

Rational Addition

Suppose we want to implement the addition of two rational numbers.

```
def addRationalNumerator(n1: Int, d1: Int, n2: Int, d2: Int): Int
def addRationalDenominator(n1: Int, d1: Int, n2: Int, d2: Int): Int
```

but it would be difficult to manage all these numerators and denominators.

A better choice is to combine the numerator and denominator of a rational number in a data structure.

Classes

In Scala, we do this by defining a *class*:

```
class Rational(x: Int, y: Int) {
  def numer = x
  def denom = y
}
```

This definition introduces two entities:

- A new *type*, named `Rational`.
- A *constructor* `Rational` to create elements of this type.

Scala keeps the names of types and values in *different namespaces*.
So there's no conflict between the two defintions of `Rational`.

Objects

We call the elements of a class type *objects*.

We create an object by prefixing an application of the constructor of the class with the operator `new`.

Example

```
new Rational(1, 2)
```

Members of an Object

Objects of the class `Rational` have two *members*, `numer` and `denom`.

We select the members of an object with the infix operator '.' (like in Java).

Example

```
val x = new Rational(1, 2)   > x: Rational = Rational@2abe0e27
x.numer                      > 1
x.denom                      > 2
```

Rational Arithmetic

We can now define the arithmetic functions that implement the standard rules.

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$$

$$\frac{n_1}{d_1} / \frac{n_2}{d_2} = \frac{n_1 d_2}{d_1 n_2}$$

$$\frac{n_1}{d_1} = \frac{n_2}{d_2} \quad \text{iff} \quad n_1 d_2 = d_1 n_2$$

Implementing Rational Arithmetic

```
def addRational(r: Rational, s: Rational): Rational =
  new Rational(
    r.numer * s.denom + s.numer * r.denom,
    r.denom * s.denom)

def makeString(r: Rational) =
  r.numer + "/" + r.denom
```

```
makeString(addRational(new Rational(1, 2), new Rational(2, 3)))   > 7/6
```

Methods

One can go further and also package functions operating on a data abstraction in the data abstraction itself.

Such functions are called *methods*.

Example

Rational numbers now would have, in addition to the functions `numer` and `denom`, the functions `add`, `sub`, `mul`, `div`, `equal`, `toString`.

Methods for Rationals

Here's a possible implementation:

```
class Rational(x: Int, y: Int) {
  def numer = x
  def denom = y
  def add(r: Rational) =
    new Rational(numer * r.denom + r.numer * denom,
                 denom * r.denom)
  def mul(r: Rational) = ...
  ...
  override def toString = numer + "/" + denom
}
```

Remark: the modifier `override` declares that `toString` redefines a method that already exists (in the class `java.lang.Object`).

Calling Methods

Here is how one might use the new `Rational` abstraction:

```
val x = new Rational(1, 3)
val y = new Rational(5, 7)
val z = new Rational(3, 2)
x.add(y).mul(z)
```

Exercise

1. In your worksheet, add a method `neg` to class Rational that is used like this:

```
x.neg           // evaluates to -x
```

2. Add a method `sub` to subtract two rational numbers.
3. With the values of `x`, `y`, `z` as given in the previous slide, what is the result of

```
x - y - z
```

?

More Fun with Rationals

Data Abstraction

The previous example has shown that rational numbers aren't always represented in their simplest form. (Why?)

One would expect the rational numbers to be *simplified*:

- reduce them to their smallest numerator and denominator by dividing both with a divisor.

We could implement this in each rational operation, but it would be easy to forget this division in an operation.

A better alternative consists of simplifying the representation in the class when the objects are constructed:

Rationals with Data Abstraction

```
class Rational(x: Int, y: Int) {
  private def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)
  private val g = gcd(x, y)
  def numer = x / g
  def denom = y / g
  ...
}
```

`gcd` and `g` are *private* members; we can only access them from inside the `Rational` class.

In this example, we calculate `gcd` immediately, so that its value can be re-used in the calculations of `numer` and `denom`.

Rationals with Data Abstraction (2)

It is also possible to call `gcd` in the code of `numer` and `denom`:

```
class Rational(x: Int, y: Int) {
  private def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)
  def numer = x / gcd(x, y)
  def denom = y / gcd(x, y)
}
```

This can be advantageous if it is expected that the functions `numer` and `denom` are called infrequently.

Rationals with Data Abstraction (3)

It is equally possible to turn `numer` and `denom` into `val`s, so that they are computed only once:

```
class Rational(x: Int, y: Int) {
  private def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)
  val numer = x / gcd(x, y)
  val denom = y / gcd(x, y)
}
```

This can be advantageous if the functions `numer` and `denom` are called often.

The Client's View

Clients observe exactly the same behavior in each case.

This ability to choose different implementations of the data without affecting clients is called *data abstraction*.

It is a cornerstone of software engineering.

Self Reference

On the inside of a class, the name `this` represents the object on which the current method is executed.

Example

Add the functions `less` and `max` to the class `Rational`.

```
class Rational(x: Int, y: Int) {
  ...
  def less(that: Rational) =
    numer * that.denom < that.numer * denom

  def max(that: Rational) =
    if (this.less(that)) that else this
}
```

Self Reference (2)

Note that a simple name `x`, which refers to another member of the class, is an abbreviation of `this.x`. Thus, an equivalent way to formulate `less` is as follows.

```
def less(that: Rational) =
  this.numer * that.denom < that.numer * this.denom
```

Preconditions

Let's say our `Rational` class requires that the denominator is positive.

We can enforce this by calling the `require` function.

```
class Rational(x: Int, y: Int) {
  require(y > 0, "denominator must be positive")
  ...
}
```

`require` is a predefined function.

It takes a condition and an optional message string.

If the condition passed to `require` is `false`, an `IllegalArgumentException` is thrown with the given message string.

Assertions

Besides `require`, there is also `assert`.

Assert also takes a condition and an optional message string as parameters. E.g.

```
val x = sqrt(y)
assert(x >= 0)
```

Like `require`, a failing `assert` will also throw an exception, but it's a different one: `AssertionError` for `assert`, `IllegalArgumentException` for `require`.

This reflects a difference in intent

- `require` is used to enforce a precondition on the caller of a function.
- `assert` is used as to check the code of the function itself.

Constructors

In Scala, a class implicitly introduces a constructor. This one is called the *primary constructor* of the class.

The primary constructor

- takes the parameters of the class
- and executes all statements in the class body (such as the `require` a couple of slides back).

Auxiliary Constructors

Scala also allows the declaration of *auxiliary constructors*.

These are methods named `this`

ExampleAdding an auxiliary constructor to the class `Rational`.

```
class Rational(x: Int, y: Int) {
  def this(x: Int) = this(x, 1)
  ...
}
```

```
new Rational(2)   > 2/1
```

Exercise

Modify the `Rational` class so that rational numbers are kept unsimplified internally, but the simplification is applied when numbers are converted to strings.

Do clients observe the same behavior when interacting with the rational class?

○	yes
○	no
●	yes for small sizes of denominators and nominators and small numbers of operations.

Evaluation and Operators

Classes and Substitutions

We previously defined the meaning of a function application using a computation model based on substitution. Now we extend this model to classes and objects.

Question: How is an instantiation of the class `new C(`$e_1, ..., e_m$`)` evaluted?

Answer: The expression arguments $e_1, ..., e_m$ are evaluated like the arguments of a normal function. That's it.

The resulting expresion, say, `new C(`$v_1, ..., v_m$`)`, is already a value.

Classes and Substitutions

Now suppose that we have a class definition,

$$\texttt{class C}(\texttt{x}_1, ..., \texttt{x}_m)\{ \ ... \ \texttt{def f}(\texttt{y}_1, ..., \texttt{y}_n) = \texttt{b} \ ... \ \}$$

where

- The formal parameters of the class are $\texttt{x}_1, ..., \texttt{x}_m$.
- The class defines a method $\texttt{f}$ with formal parameters $\texttt{y}_1, ..., \texttt{y}_n$.

(The list of function parameters can be absent. For simplicity, we have omitted the parameter types.)

Question: How is the following expression evaluated?

$$\texttt{new C}(\texttt{v}_1, ..., \texttt{v}_m).\texttt{f}(\texttt{w}_1, ..., \texttt{w}_n)$$

Classes and Substitutions (2)

Answer: The expression $\texttt{new C}(v_1, ..., v_m).\texttt{f}(w_1, ..., w_n)$ is rewritten to:

$$[w_1/y_1, ..., w_n/y_n][v_1/x_1, ..., v_m/x_m][\texttt{new C}(v_1, ..., v_m)/\texttt{this}]\,b$$

There are three substitutions at work here:

- the substitution of the formal parameters $y_1, ..., y_n$ of the function $\texttt{f}$ by the arguments $w_1, ..., w_n$,
- the substitution of the formal parameters $x_1, ..., x_m$ of the class $\texttt{C}$ by the class arguments $v_1, ..., v_m$,
- the substitution of the self reference *this* by the value of the object $\texttt{new C}(v_1, ..., v_n)$.

class C ($x_1, ..., x_m$) {
  def f ($y_1, ..., y_m$) = b ... this ...
}

Object Rewriting Examples

```
new Rational(1, 2).numer
```

Object Rewriting Examples

```
new Rational(1, 2).numer
```

$\longrightarrow$ $[1/x, 2/y]$ [] [new Rational(1, 2)/this] x

Object Rewriting Examples

```
new Rational(1, 2).numer
```

$\rightarrow$ [1/x, 2/y] [] [new Rational(1, 2)/this] x

$=$ 1

Object Rewriting Examples

```
new Rational(1, 2).numer
```

$\longrightarrow$ [1/x, 2/y] [] [new Rational(1, 2)/this] x

$=$ 1

```
new Rational(1, 2).less(new Rational(2, 3))
```

Object Rewriting Examples

```
new Rational(1, 2).numer
```

$\rightarrow$ [1/x, 2/y] [] [new Rational(1, 2)/this] x

$=$ 1

```
new Rational(1, 2).less(new Rational(2, 3))
```

$\rightarrow$ [1/x, 2/y] [newRational(2, 3)/that] [new Rational(1, 2)/this]

```
    this.numer * that.denom < that.numer * this.denom
```

Object Rewriting Examples

```
new Rational(1, 2).numer
```

$\rightarrow$ $[1/x, 2/y]$ [] $[$`new Rational(1, 2)`$/$`this`$]$ `x`

$=$ `1`

```
new Rational(1, 2).less(new Rational(2, 3))
```

$\rightarrow$ $[1/x, 2/y]$ $[$`newRational(2, 3)`$/$`that`$]$ $[$`new Rational(1, 2)`$/$`this`$]$

```
    this.numer * that.denom < that.numer * this.denom
```

$=$

```
new Rational(1, 2).numer * new Rational(2, 3).denom <
    new Rational(2, 3).numer * new Rational(1, 2).denom
```

Object Rewriting Examples

```
new Rational(1, 2).numer
```

$\longrightarrow$ $[1/\texttt{x}, 2/\texttt{y}]$ [] $[\texttt{new Rational}(1, 2)/\texttt{this}]$ x

$=$ 1

```
new Rational(1, 2).less(new Rational(2, 3))
```

$\longrightarrow$ $[1/\texttt{x}, 2/\texttt{y}]$ $[\texttt{newRational}(2, 3)/\texttt{that}]$ $[\texttt{new Rational}(1, 2)/\texttt{this}]$

```
    this.numer * that.denom < that.numer * this.denom
```

$=$

```
new Rational(1, 2).numer * new Rational(2, 3).denom <
    new Rational(2, 3).numer * new Rational(1, 2).denom
```

$\twoheadrightarrow$ `1 * 3 < 2 * 2`

$\twoheadrightarrow$ `true`

Operators

In principle, the rational numbers defined by `Rational` are as natural as integers.

But for the user of these abstractions, there is a noticeable difference:

- We write `x + y`, if `x` and `y` are integers, but
- We write `r.add(s)` if `r` and `s` are rational numbers.

In Scala, we can eliminate this difference. We procede in two steps.

Step 1: Infix Notation

Any method with a parameter can be used like an infix operator.

It is therefore possible to write

```
r add s                                    r.add(s)
r less s          /* in place of */        r.less(s)
r max s                                    r.max(s)
```

Step 2: Relaxed Identifiers

Operators can be used as identifiers.

Thus, an identifier can be:

- *Alphanumeric*: starting with a letter, followed by a sequence of letters or numbers
- *Symbolic*: starting with an operator symbol, followed by other operator symbols.
- The underscore character '_' counts as a letter.
- Alphanumeric identifiers can also end in an underscore, followed by some operator symbols.

Examples of identifiers:

```
x1    *    +?%&    vector_++    counter_=
```

Operators for Rationals

A more natural definition of class `Rational`:

```
class Rational(x: Int, y: Int) {
  private def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)
  private val g = gcd(x, y)
  def numer = x / g
  def denom = y / g
  def + (r: Rational) =
    new Rational(
      numer * r.denom + r.numer * denom,
      denom * r.denom)
  def - (r: Rational) = ...
  def * (r: Rational) = ...
  ...
}
```

Operators for Rationals

… and rational numbers can be used like `Int` or `Double`:

```
val x = new Rational(1, 2)
val y = new Rational(1, 3)
(x * x) + (y * y)
```

Precedence Rules

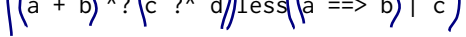
The *precedence* of an operator is determined by its first character.

The following table lists the characters in increasing order of priority precedence:

```
(all letters)
|
^
&
< >
= !
:
+ -
* / %
(all other special characters)
```

Exercise

Provide a fully parenthized version of



Every binary operation needs to be put into parentheses, but the structure of the expression should not change.

Class Hierarchies

Abstract Classes

Consider the task of writing a class for sets of integers with the following operations.

```
abstract class IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
}
```

`IntSet` is an *abstract class*.

Abstract classes can contain members which are missing an implementation (in our case, `incl` and `contains`).

Consequently, no instances of an abstract class can be created with the operator `new`.

Class Extensions

Let's consider implementing sets as binary trees.

There are two types of possible trees: a tree for the empty set, and a tree consisting of an integer and two sub-trees.

Here are their implementations:

```
class Empty extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmpty(x, new Empty, new Empty)
}
```

Class Extensions (2)

```
class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {

  def contains(x: Int): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true

  def incl(x: Int): IntSet =
    if (x < elem) new NonEmpty(elem, left incl x, right)
    else if (x > elem) new NonEmpty(elem, left, right incl x)
    else this
}
```

Terminology

`Empty` and `NonEmpty` both *extend* the class `IntSet`.

This implies that the types `Empty` and `NonEmpty` *conform* to the type `IntSet`

- an object of type `Empty` or `NonEmpty` can be used wherever an object of type `IntSet` is required.

Base Classes and Subclasses

`IntSet` is called the *superclass* of `Empty` and `NonEmpty`.

`Empty` and `NonEmpty` are *subclasses* of `IntSet`.

In Scala, any user-defined class extends another class.

If no superclass is given, the standard class `Object` in the Java package `java.lang` is assumed.

The direct or indirect superclasses of a class `C` are called *base classes* of `C`.

So, the base classes of `NonEmpty` are `IntSet` and `Object`.

Implementation and Overriding

The definitions of `contains` and `incl` in the classes `Empty` and `NonEmpty` *implement* the abstract functions in the base trait `IntSet`.

It is also possible to *redefine* an existing, non-abstract definition in a subclass by using `override`.

Example

```
abstract class Base {
  def foo = 1
  def bar: Int
}
```

```
class Sub extends Base {
  override def foo = 2
  def bar = 3
}
```

Object Definitions

In the `IntSet` example, one could argue that there is really only a single empty `IntSet`.

So it seems overkill to have the user create many instances of it.

We can express this case better with an *object definition*:

```
object Empty extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmpty(x, Empty, Empty)
}
```

This defines a *singleton object* named `Empty`.

No other `Empty` instances can be (or need to be) created.

Singleton objects are values, so `Empty` evaluates to itself.

Programs

So far we have executed all Scala code from the REPL or the worksheet.

But it is also possible to create standalone applications in Scala.

Each such application contains an object with a `main` method.

For instance, here is the "Hello World!" program in Scala.

```
object Hello {
  def main(args: Array[String]) = println("hello world!")
}
```

Once this program is compiled, you can start it from the command line with

```
> scala Hello
```

Exercise

Write a method `union` for forming the union of two sets. You should implement the following abstract class.

```
abstract class IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
  def union(other: IntSet): IntSet
}
```

Dynamic Binding

Object-oriented languages (including Scala) implement *dynamic method dispatch*.

This means that the code invoked by a method call depends on the runtime type of the object that contains the method.

Example

```
Empty contains 1
```

Dynamic Binding

Object-oriented languages (including Scala) implement *dynamic method dispatch*.

This means that the code invoked by a method call depends on the runtime type of the object that contains the method.

Example

```
Empty contains 1
```

$\rightarrow$ `[1/x] [Empty/this] false`

Dynamic Binding

Object-oriented languages (including Scala) implement *dynamic method dispatch*.

This means that the code invoked by a method call depends on the runtime type of the object that contains the method.

Example

```
Empty contains 1
```

$\rightarrow$ `[1/x] [Empty/this] false`

$=$ `false`

Dynamic Binding (2)

Another evaluation using `NonEmpty`:

```
(new NonEmpty(7, Empty, Empty)) contains 7
```

Dynamic Binding (2)

Another evaluation using `NonEmpty`:

```
(new NonEmpty(7, Empty, Empty)) contains 7
```

$\rightarrow$ $[7/\texttt{elem}]$ $[7/\texttt{x}]$ $[\texttt{new NonEmpty(7, Empty, Empty)}/\texttt{this}]$

```
    if (x < elem) this.left contains x
      else if (x > elem) this.right contains x else true
```

Dynamic Binding (2)

Another evaluation using `NonEmpty`:

```
(new NonEmpty(7, Empty, Empty)) contains 7
```

$\rightarrow$ $[7/\text{elem}]\ [7/\text{x}]\ [\text{new NonEmpty}(7, \text{Empty}, \text{Empty})/\text{this}]$

```
    if (x < elem) this.left contains x
      else if (x > elem) this.right contains x else true
```

$=$

```
if (7 < 7) new NonEmpty(7, Empty, Empty).left contains 7
    else if (7 > 7) new NonEmpty(7, Empty, Empty).right
        contains 7 else true
```

Dynamic Binding (2)

Another evaluation using `NonEmpty`:

```
(new NonEmpty(7, Empty, Empty)) contains 7
```

$\rightarrow$ $[7/\texttt{elem}]$ $[7/\texttt{x}]$ $[\texttt{new NonEmpty(7, Empty, Empty)}/\texttt{this}]$

```
    if (x < elem) this.left contains x
      else if (x > elem) this.right contains x else true
```

$=$

```
if (7 < 7) new NonEmpty(7, Empty, Empty).left contains 7
    else if (7 > 7) new NonEmpty(7, Empty, Empty).right
        contains 7 else true
```

$\rightarrow$ `true`

Something to Ponder

Dynamic dispatch of methods is analogous to calls to higher-order functions.

Question:

Can we implement one concept in terms of the other?

- Objects in terms of higher-order functions?
- Higher-order functions in terms of objects?

How Classes are Organized

Packages

Classes and objects are organized in packages.

To place a class or object inside a package, use a package clause at the top of your source file.

```
package progfun.examples

object Hello { ... }
```

This would place `Hello` in the package `progfun.examples`.

You can then refer to `Hello` by its *fully qualified name* `progfun.examples.Hello`. For instance, to run the `Hello` program:

```
> scala progfun.examples.Hello
```

Imports

Say we have a class `Rational` in package `week3`.

You can use the class using its fully qualified name:

```
val r = new week3.Rational(1, 2)
```

Alternatively, you can use an import:

```
import week3.Rational
val r = new Rational(1, 2)
```

Forms of Imports

Imports come in several forms:

```
import week3.Rational           // imports just Rational
import week3.{Rational, Hello}  // imports both Rational and Hello
import week3._                  // imports everything in package week3
```

The first two forms are called *named imports*.

The last form is called a *wildcard import*.

You can import from either a package or an object.

Automatic Imports

Some entities are automatically imported in any Scala program.

These are:

- All members of package `scala`
- All members of package `java.lang`
- All members of the singleton object `scala.Predef`.

Here are the fully qualified names of some types and functions which you have seen so far:

`Int`	`scala.Int`
`Boolean`	`scala.Boolean`
`Object`	`java.lang.Object`
`require`	`scala.Predef.require`
`assert`	`scala.Predef.assert`

Scaladoc

You can explore the standard Scala library using the scaladoc web pages.

You can start at

www.scala-lang.org/api/current

Traits

In Java, as well as in Scala, a class can only have one superclass.

But what if a class has several natural supertypes to which it conforms or from which it wants to inherit code?

Here, you could use `traits`.

A trait is declared like an abstract class, just with `trait` instead of `abstract class`.

```
trait Planar {
  def height: Int
  def width: Int
  def surface = height * width
}
```

Single inheritance

Traits (2)

Classes, objects and traits can inherit from at most one class but arbitrary many traits.

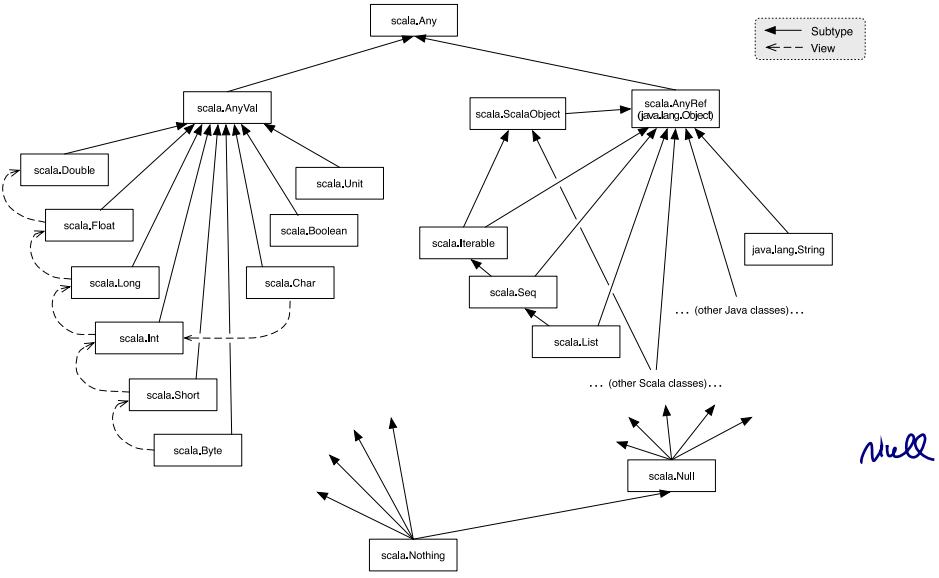
Example:

```
class Square extends Shape with Planar with Movable ...
```

Traits resemble interfaces in Java, but are more powerful because they can contains fields and concrete methods.

On the other hand, traits cannot have (value) parameters, only classes can.

Scala's Class Hierarchy



Top Types

At the top of the type hierarchy we find:

`Any`	the base type of all types Methods: ‘==‘, ‘!=‘, ‘equals‘, ‘hashCode, ‘toString‘
`AnyRef`	The base type of all reference types; Alias of ‘java.lang.Object‘
`AnyVal`	The base type of all primitive types.

The Nothing Type

`Nothing` is at the bottom of Scala's type hierarchy. It is a subtype of every other type.

There is no value of type `Nothing`.

Why is that useful?

- To signal abnormal termination
- As an element type of empty collections (see next session)

Set [Nothing]

Exceptions

Scala's exception handling is similar to Java's.

The expression

```
throw Exc
```

aborts evaluation with the exception `Exc`.

The type of this expression is `Nothing`.

The Null Type

Every reference class type also has `null` as a value.

The type of `null` is `Null`.

`Null` is a subtype of every class that inherits from `Object`; it is incompatible with subtypes of `AnyVal`.

```
val x = null          // x: Null
val y: String = null // y: String
val z: Int = null     // error: type mismatch
```

Exercise

What is the type of

```
if (true) 1 else false
```

- 0 Int
- 0 Boolean
- 0 AnyVal
- 0 Object
- 0 Any