TDT4165 Scala 101

Waclaw Kusnierczyk | waclaw.kusnierczyk@gmail.com

Agenda

1. What is Scala?
2. Hello :)
3. Execute!
4. OOPs
5. Pattern matching (or not)
6. Generics
7. Traits

What is Scala?

Scala is a **scalable**, general-purpose programming language that

- is **compiled** (to Java bytecode), but has a strong **scripting** feel;
- is **statically typed**, but types can often be **inferred** by the compiler;
- has **object-oriented** programming features;
- has **functional** programming features;
- supports **operator overloading** and many other features absent from Java;
- and **much more**.

Agenda

1. What is Scala?
2. Hello :)
3. Execute!
4. OOPs
5. Pattern matching (or not)
6. Generics
7. Traits

Hello :)

Consider a simple `hello world` program in Java:

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

To make code executable, we need

- a class with a `public static void (String[])` method.

To run the program, we need to

- **compile** the source: `javac <source file name>`;
- **execute** the binary: `java <class file name>`.

Hello :)

We can almost figuratively translate this to Scala:

```scala
object HelloWorld {
    def main(args: Array[String]): Unit = {
        println("Hello, World")
    }
}
```

Hello :)

We can almost figuratively translate this to Scala:

```scala
object HelloWorld {
    def main(args: Array[String]): Unit = {
        println("Hello, World")
    }
}
```

Note:

- `object` is a singleton instance **without** a class having ever been defined;
- `Array[String]` and `Unit` declare types **after** the declared objects;
- stuff is **public** unless explicitly **private** or **protected**.

Hello :)

We can almost figuratively translate this to Scala:

```scala
object HelloWorld {
    def main(args: Array[String]): Unit = {
        println("Hello, World")
    }
}
```

Note:

- `object` is a singleton instance **without** a class having ever been defined;
- `Array[String]` and `Unit` declare types **after** the declared objects;
- stuff is **public** unless explicitly **private** or **protected**.

Question: Where does `println` come from?

Hello :)

Why be verbose? In Scala, we can avoid some of the syntactic noise:

```scala
object HelloWorld {
    def main(args: Array[String]) =
        println("Hello, World")
}
```

Hello :)

Why be verbose? In Scala, we can avoid some of the syntactic noise:

```scala
object HelloWorld {
    def main(args: Array[String]) =
        println("Hello, World")
}
```

Note:

- `main` calls `println` that does not return any value;
- hence, `main` does not return any value;
- the return type of `main` is thus inferred to be `Unit` (`void` in Java speak), so no need to be explicit;
- calling `main` on any `Array[String]` argument is equivalent to calling `println` on `"Hello, World!"`.

Hello :)

We don't even need to explicitly implement `main`:

```
object HelloWorld extends App {
    println("Hello, World!")
}
```

Hello :)

We don't even need to explicitly implement `main`:

```scala
object HelloWorld extends App {
    println("Hello, World!")
}
```

Note:

- we `extend` the *trait* `App`: that's **inheritance** in action;
- `App` wraps our code in a ready-to-go `main`.

More on traits later.

Question: How do we run Scala code?

Agenda

1. What is Scala?
2. Hello :)
3. Execute!
4. OOPs
5. Pattern matching (or not)
6. Generics
7. Traits

Execute!

There are several ways to run Scala code:

- **compile** the source, **execute** the binary (as in Java):

```
$ scalac hello.scala  # produces a binary
$ scala HelloWorld    # executes the binary
```

- **run** code directly from the source:

```
$ scala hello.scala   # executes the source
```

Note:

- the object `HelloWorld` does **not** have to be defined in a file named `HelloWorld.scala`.

Execute!

There are several ways to run Scala code:

- run the code as a **script**:

```
$ ./hello.sh          # if hello.sh is executable
$ bash hello.sh       # otherwise
```

where `hello.sh` may be of two forms:

```
#!/bin/bash
exec scala "$0" "$@"
#!
object HelloWorld extends App {
    println("Hello, World!")
}
HelloWorld.main(null)
```

```
#!/bin/bash
exec scala "$0" "$@"
#!
println("Hello, World!)
```

Execute!

There are several ways to run Scala code:

- start **interactive** Scala shell (just type `scala`), then type in the code:

```
scala> object HelloWorld {
 | def main(args: Array[String]) = println("Hello, World!") }
defined module HelloWorld
scala> HelloWorld.main(null)
Hello, World!
scala> println("Hello, World!")
Hello, World!
```

Execute!

There are several ways to run Scala code:

- use **SBT**, the Scala build tool:

```
$ sbt                  # choose action interactively in sbt shell
$ sbt run              # have sbt run code and exit
```

Note:

- if multiple executable units are found (objects extending `App` or implementing `main`), you will be given a list to choose from:

```
Multiple main classes detected, select one to run:
[1] HelloWorld
[2] tdt4165.HelloWorld
Enter number:
```

Agenda

1. What is Scala?
2. Hello :)
3. Execute!
4. OOPs
5. Pattern matching (or not)
6. Generics
7. Traits

Object-oriented Scala

Use classes to specify the behaviour of a group of similar objects:

```scala
class Point(x: Int, y: Int) {
  def show() = println(s"Point($x, $y)") }
```

Note:

- there is **no explicit constructor** method;
- parameters of the class (`x`, `y`) are accessible from within the class.

Object-oriented Scala

Use classes to specify the behaviour of a group of similar objects:

```
class Point(x: Int, y: Int) {
  def show() = println(s"Point($x, $y)") }
```

Note:

- there is **no explicit constructor** method;
- parameters of the class (`x`, `y`) are accessible from within the class.

Question: What does `s"Point($x, $y)"` stand for?

Object-oriented Scala

Instantiate classes with `new`:

```
val point = new Point(0, 0)
point.show()
```

Note:

- `val` declares a new *value* (an **immutable variable**, so to speak);
- `new Point(0, 0)` returns an object of type `Point`;
- hence, the compiler can **infer** the type of `point`;
- hence, we do not need to; but we might:

  ```
  val point: Point = new Point(0, 0)
  point.show()
  ```

Object-oriented Scala

Instantiate classes with `new`:

```
val point = new Point(0, 0)
point.show()
```

Note:

- `val` declares a new *value* (an **immutable variable**, so to speak);
- `new Point(0, 0)` returns an object of type `Point`;
- hence, the compiler can **infer** the type of `point`;
- hence, we do not need to; but we might:

  ```
  val point: Point = new Point(0, 0)
  point.show()
  ```

Question: Why would we?

Object-oriented Scala

The values `x` and `y` are visible exclusively inside the class `Point`:

```
val point = new Point(0, 0)
println(s"point = Point(${point.x}, ${point.y})")  // x and y not found
```

Object-oriented Scala

The values `x` and `y` are visible exclusively inside the class `Point`:

```scala
val point = new Point(0, 0)
println(s"point = Point(${point.x}, ${point.y})")  // x and y not found
```

Questions:

- Why not `s"point = Point($point.x, $point.y)"`?
- Can we modify `x` and `y` (inside `Point`)?

Object-oriented Scala

We can 'promote' class parameters to become true instance fields:

```
class Point(val x: Int, val y: Int) {
  def show() = println(s"Point($x, $y)") }
```

Fields are (by default) public:

```
val point = new Point(0, 0)
println(point.x)       // ok
```

Object-oriented Scala

We can 'promote' class parameters to become true instance fields:

```scala
class Point(val x: Int, val y: Int) {
  def show() = println(s"Point($x, $y)") }
```

Fields are (by default) public:

```scala
val point = new Point(0, 0)
println(point.x)       // ok
```

Question: Are the fields modifiable or mutable?

Object-oriented Scala

We can 'promote' class parameters to become true instance fields:

```
class Point(val x: Int, val y: Int) {
  def show() = println(s"Point($x, $y)") }
```

Fields are (by default) public:

```
val point = new Point(0, 0)
println(point.x)       // ok
```

Question: Are the fields modifiable or mutable? No, they aren't:

```
point.x = 1            // error
```

Object-oriented Scala

Let's augment the class to enable updating the coordinates:

```
class Point(var x: Int, var y: Int) {
    def move(dx: Int, dy: Int) = {
        x = x + dx
        y = y + dy }
    ... }
val point = new Point(1, 1)
```

Object-oriented Scala

Let's augment the class to enable updating the coordinates:

```
class Point(var x: Int, var y: Int) {
    def move(dx: Int, dy: Int) = {
        x = x + dx
        y = y + dy }
    ... }
val point = new Point(1, 1)
```

Questions:

- If `point` is a `val` (immutable variable), can `move` change its coordinates?
- Can the fields `x` and `y` be modified from outside of the class?

Object-oriented Scala

Let's augment the class to enable updating the coordinates:

```scala
class Point(var x: Int, var y: Int) {
    def move(dx: Int, dy: Int) = {
        x = x + dx
        y = y + dy }
    ... }
val point = new Point(1, 1)
```

Questions:

- If `point` is a `val` (immutable variable), can `move` change its coordinates?
- Can the fields `x` and `y` be modified from outside of the class? Yes!

```scala
point.move(1, -1)       // Point(2, 0)
point.y = 2             // Point(2, 2)
```

Agenda

1. What is Scala?
2. Hello :)
3. Execute!
4. OOPs
5. Pattern matching (or not)
6. Generics
7. Traits

Pattern-matching

We'd like to be able to compare a points for equality with other objects. Easy:

```
class Point(val x: Int, val y: Int) {
    override def equals(any: Any): Boolean = {
        if (!any.isInstanceOf[Point])
            return false
        val that = any.asInstanceOf[Point]
        return that.x == this.x && that.y == this.y }
    ... }
```

Note:

- if you don't intend to modify fields, make them immutable (`vals`);
- the method `equals` is inherited from `Any` (Scala's `Object`);
- any **method overrides** must be explicitly acknowledged.

Pattern-matching

We'd like to be able to compare a points for equality with other objects. Easy:

```
class Point(val x: Int, val y: Int) {
    override def equals(any: Any): Boolean = {
        if (!any.isInstanceOf[Point])
            return false
        val that = any.asInstanceOf[Point]
        return that.x == this.x && that.y == this.y }
    ... }
```

Note:

- if you don't intend to modify fields, make them immutable (`vals`);
- the method `equals` is inherited from `Any` (Scala's `Object`);
- any **method overrides** must be explicitly acknowledged.

Question: This is kinda ugly. Can we do better?

Pattern-matching

If we turn `Point` into a **case class**, we can use **pattern-matching**:

```
case class Point(x: Int, y: Int) {
    override def equals(any: Any) = any match {
        case Point(xx, yy) => xx == x && yy == y
        case _ => false }
    ... }
```

Note:

- in a `case` class, the parameters are `val` by default (accessible, immutable);
- `match` performs **pattern matching** between an object and one or more `case` patterns;
- a `case` pattern compares objects **structurally**, **capturing** variables (fields, etc.) as appropriate.
- the pattern _ is a **catch-all** pattern that provides a default result.

Pattern-matching

We can now easily compare a point to other objects:

```
val point = Point(0, 0)
point.equals(Point(1, 1))   // nope
point.equals(Point(0, 0))   // yepp
point.equals("Point(0, 0)") // nope!
```

Note:

- a case class does not need `new` to construct instances.

Pattern-matching

We can now easily compare a point to other objects:

```
val point = Point(0, 0)
point.equals(Point(1, 1))   // nope
point.equals(Point(0, 0))   // yepp
point.equals("Point(0, 0)") // nope!
```

Note:

- a case class does not need `new` to construct instances.

Agenda

1. What is Scala?
2. Hello :)
3. Execute!
4. OOPs
5. Pattern matching (or not)
6. Generics
7. Traits

Generics

We could use different types of values for point coordinates:

```
case class Point[T](x: T, y: T) {
    override def equals(any: Any) = any match {
        case Point(xx: T, yy: T) => xx == x && yy == y
        case Point(xx, yy) => // do some magic
        case _ => false }
    ... }
```

Note:

- `Point[T]` is a **generic class**, where `T` is a type variable;
- both arguments are of type `T`;
- the pattern `case Point(xx: T, yy: T)` matches all and only `T`-points.

Generics

We could use different types of values for point coordinates:

```scala
case class Point[T](x: T, y: T) {
    override def equals(any: Any) = any match {
        case Point(xx: T, yy: T) => xx == x && yy == y
        case Point(xx, yy) => // do some magic
        case _ => false }
    ... }
```

Note:

- `Point[T]` is a **generic class**, where `T` is a type variable;
- both arguments are of type `T`;
- the pattern `case Point(xx: T, yy: T)` matches all and only `T`-points.

Question: What is the outcome of:

```scala
Point(0, 1/2).equals(Point(0, 0.5))
```

Agenda

1. What is Scala?
2. Hello :)
3. Execute!
4. OOPs
5. Pattern matching (or not)
6. Generics
7. Traits

Traits

Suppose we are allergic to `!` and prefer `a unequals b` to `!a.equals(b)`.

```
case class Point[T](x: T, y: T) {
    override def equals(any: Any) = any match {
        case Point(xx: T, yy: T) => xx == x && yy == y
        case _ => false }
    def unequals(any: Any) = !equal(any)
    ... }
```

We can now write:

```
Point(0, 0) unequals Point(1, 1)  // true
```

instead of:

```
!Point(0, 0).equals(Point(1, 1))  // true
```

(Yeah, **silly**. But note the elegant **infix notation**!)

Traits

We don't want to implement both `equals` and `unequals` for each new class.

Let's implement a reusable **trait**:

```
trait Unequal {
    def unequals(any: Any) = !equals(any) }
case class Point[T](x: T, y: T) extends Unequal {
    ... }
```

Note:

- `Unequals` is not a class, and cannot be instantiated;
- the **trait provides** a method that can be immediately used by any class that extends it;
- it refers to `equals`, a method from `Any` that every class either inherits or overrides.

Traits

We don't want to implement both `equals` and `unequals` for each new class.

Let's implement a reusable **trait**:

```
trait Unequal {
    def unequals(any: Any) = !equals(any) }
case class Point[T](x: T, y: T) extends Unequal {
    ... }
```

Note:

- `Unequals` is not a class, and cannot be instantiated;
- the **trait provides** a method that can be immediately used by any class that extends it;
- it refers to `equals`, a method from `Any` that every class either inherits or overrides.

Question: How novel is this?

Traits

We don't want to implement both `equals` and `unequals` for each new class.

Let's implement a reusable **trait**:

```
trait Unequal {
    def unequals(any: Any) = !equals(any) }
case class Point[T](x: T, y: T) extends Unequal {
    ... }
```

Note:

- `Unequals` is not a class, and cannot be instantiated;
- the **trait provides** a method that can be immediately used by any class that extends it;
- it refers to `equals`, a method from `Any` that every class either inherits or overrides.

Question: How novel is this? (Think interfaces with default method implementations in Java.)

TDT4165: Concurrency in Scala, Part I

October 28, 2015

Processes and Threads

A **process**

- is an instance of a computer program that is being executed;
- is largely **self-contained**: has its own, **isolated** share of memory and other computational resources;
- cannot directly access the memory of other processes;
- cannot simultaneously use the same resources as other processes.

Processes and Threads

A **process**

- is an instance of a computer program that is being executed;
- is largely **self-contained**: has its own, **isolated** share of memory and other computational resources;
- cannot directly access the memory of other processes;
- cannot simultaneously use the same resources as other processes.

A **thread**

- is a **unit of execution** within a process;
- has its own program stack and program counter;
- **shares memory and resources** with other threads within the process.

Processes and Threads

A **process**

- is an instance of a computer program that is being executed;
- is largely **self-contained**: has its own, **isolated** share of memory and other computational resources;
- cannot directly access the memory of other processes;
- cannot simultaneously use the same resources as other processes.

A **thread**

- is a **unit of execution** within a process;
- has its own program stack and program counter;
- **shares memory and resources** with other threads within the process.

Note: We will be concerned mostly with threads (i.e., **multithreaded** programs).

Processes and Threads

For debugging, it will be useful to **identify** threads.

```
object ThreadName extends App {
  val name = Thread.currentThread.getName
  println(s"thread: $name") }
```

Note:

- for convenience, we **extend the trait** `App` to inherit `main` (see `scala.App`);
- `Thread` is a **Java** class (see `java.lang.Thread`);
- `currentThread` is a `static` method of `Thread` that *returns a reference to the currently executing thread object* (an instance of `Thread`);
- `getName` is an instance method of `Thread` that *returns this thread's name*;

Processes and Threads

For debugging, it will be useful to **identify** threads.

```
object ThreadName extends App {
  val name = Thread.currentThread.getName
  println(s"thread: $name") }
```

Note:

- for convenience, we **extend the trait** `App` to inherit `main` (see `scala.App`);
- `Thread` is a **Java** class (see `java.lang.Thread`);
- `currentThread` is a `static` method of `Thread` that *returns a reference to the currently executing thread object* (an instance of `Thread`);
- `getName` is an instance method of `Thread` that *returns this thread's name*;

Question:

- Why `currentThread` and `getName`, and not `currentThread()` and `getName()`?

Processes... **Methods**

Method definition syntax:

- `currentThread` and `getName` are **nullary** methods: they take **no arguments** (and **return** a value);
- in Scala, nullary methods can be defined with or without (empty) parentheses;
- methods defined with empty parentheses can be called with or without them.

```
def f1 = "f1"    // def f1: String = ...
def f2() = "f2"  // def f2(): String = ...
```

Processes... **Methods**

Method definition syntax:

- `currentThread` and `getName` are **nullary** methods: they take **no arguments** (and **return** a value);
- in Scala, nullary methods can be defined with or without (empty) parentheses;
- methods defined with empty parentheses can be called with or without them.

```
def f1 = "f1"     // def f1: String = ...
def f2() = "f2"   // def f2(): String = ...
```

Note:

- both `f1` and `f2` are nullary methods returning a value of type `String`;
- their definitions are **roughly equivalent**, but
- there are differences in how they can be called.

Processes... Methods

How a nullary method should be called depends on its definition.

```
def f1 = "f1"     // def f1: String = ...
def f2() = "f2"   // def f2(): String = ...
```

```
println(f1)       // (1)
println(f1())     // (2)
println(f1(0))    // (3)
println(f2)       // (4)
println(f2())     // (5)
println(f2(0))    // (6)
```

Questions:

- Are any/all of the calls legal? If legal, what is the output?

Processes... **Methods**

How a nullary method should be called depends on its definition.

```
def f1 = "f1"     // def f1: String = ...
def f2() = "f2"   // def f2(): String = ...
```

```
println(f1)       // (1)
println(f1())     // (2)
println(f1(0))    // (3)
println(f2)       // (4)
println(f2())     // (5)
println(f2(0))    // (6)
```

Questions:

- Are any/all of the calls legal? If legal, what is the output?

Answer:

- 1: `f1` | 2: **incidentally illegal** | 3: `f`, **incidentally legal**
- 4: `f2` | 5: `f2` | 6: illegal.

Processes... **Methods**

It is **customary** to use `()` in both definitions and calls to methods that **do not return** values (i.e., have `Unit` return type).

```
def p1 = println("p1")    // def p1: Unit = ...
def p2() = println("p2")  // def p2(): Unit = ..., preferred
```

```
p1      // (1)
p1()    // (2)
p2      // (3)
p2()    // (4)
```

Question:

- Are any/all of the calls legal?

Processes... Methods

It is **customary** to use `()` in both definitions and calls to methods that **do not return** values (i.e., have `Unit` return type).

```
def p1 = println("p1")    // def p1: Unit = ...
def p2() = println("p2")  // def p2(): Unit = ..., preferred
```

```
p1      // (1)
p1()    // (2)
p2      // (3)
p2()    // (4)
```

Question:

- Are any/all of the calls legal?

Answer:

- 1: legal | 2: illegal
- 3: legal | 4: legal, **preferred**

Processes and Threads

A convenience **library** method to print messages with thread id prefixed:

```
package cis.utils
object ThreadUtils {
  def log(message: String): Unit =
    println(s"[${Thread.currentThread.getName}] $message") }
```

Note: Unlike in Java, in Scala

- a class (or object) does **not** have to reside in a file named exactly after the class;
- the `package` declaration does **not** have to reflect the directory structure above the source file;
- a source file may include **more than one** class (or object).

Processes... Project Structure

To ease building, organize code into a **standard structure**.

```
$ tree
|-- src
|   `-- main
|       |-- resources
|       `-- scala
|           |-- examples
|           |   `-- ThreadsDemo.scala  # code with main
|           `-- utils
|               `-- Threads.scala      # library code
`-- build.sbt
```

Include a rudimentary **build file** `build.sbt` at the project root:

```
scalaVersion := "2.11.7"
```

Processes and Threads

We can now conveniently print diagnostic messages **prefixed** with a thread id:

```
package cis.example
import cis.utils.ThreadUtils._
object MainThreadDemo extends App {
  log(s"greetings from $this!") }
```

Processes and Threads

We can now conveniently print diagnostic messages **prefixed** with a thread id:

```
package cis.example
import cis.utils.ThreadUtils._
object MainThreadDemo extends App {
  log(s"greetings from $this!") }
```

Note:

- the blank in `ThreadUtils._` matches anything, hence
- we are importing the **whole content** of the `ThreadUtils` object;
- equivalent to `import static cis.utils.ThreadUtils.*` in Java;
- we can use `log` as if it were a locally defined method.
- some prefer explicit imports, e.g., `import cis.utils.ThreadUtils.log`

Processes and Threads

We can now conveniently print diagnostic messages **prefixed** with a thread id:

```
package cis.example
import cis.utils.ThreadUtils._
object MainThreadDemo extends App {
  log(s"greetings from $this!") }
```

Note:

- the blank in `ThreadUtils._` matches anything, hence
- we are importing the **whole content** of the `ThreadUtils` object;
- equivalent to `import static cis.utils.ThreadUtils.*` in Java;
- we can use `log` as if it were a locally defined method.
- some prefer explicit imports, e.g., `import cis.utils.ThreadUtils.log`

Question:

- What does `$this` evaluate to within the message?

Processes and Threads

Let us start a new thread:

```
object NewThreadDemo extends App {
  val thread = new Thread {
    override def run() = log(s"greetings from $this!") } }
```

Note:

- `log` is imported from `ThreadUtils`, as before;
- from now on, **we will skip** `package` and `import` lines where they are obvious;
- as with nullary functions, the **constructor** `Thread()` is called **without parentheses**;
- we create an instance of an anonymous class that extends `Thread`, overriding its `run` nethod.

Processes and Threads

Let us start a new thread:

```
object NewThreadDemo extends App {
  val thread = new Thread {
    override def run() = log(s"greetings from $this!") } }
```

Note:

- `log` is imported from `ThreadUtils`, as before;
- from now on, **we will skip** `package` and `import` lines where they are obvious;
- as with nullary functions, the **constructor** `Thread()` is called **without parentheses**;
- we create an instance of an anonymous class that extends `Thread`, overriding its `run` nethod.

Question:

- What will be the result of running this program?

Processes and Threads

To avoid repetitive typing, let's add the thread creating code to our library:

```
object ThreadUtils {
  def thread(body: => Unit): Thread =
    new Thread {
      override def run() = body }
  ... }
```

Processes and Threads

To avoid repetitive typing, let's add the thread creating code to our library:

```
object ThreadUtils {
  def thread(body: => Unit): Thread =
    new Thread {
      override def run() = body }
  ... }
```

Note: We update `ThreadUtils` by adding

- the method `thread` that
- takes as an argument an **unevaluated statement** `body`, and
- returns a new instance (a thread) of an anonymous class that
- extends `Thread` implementing `run` that
- executes `body`.

Processes and Threads

To avoid repetitive typing, let's add the thread creating code to our library:

```
object ThreadUtils {
  def thread(body: => Unit): Thread =
    new Thread {
      override def run() = body }
  ... }
```

Note: We update `ThreadUtils` by adding

- the method `thread` that
- takes as an argument an **unevaluated statement** `body`, and
- returns a new instance (a thread) of an anonymous class that
- extends `Thread` implementing `run` that
- executes `body`.

Question:

- What exactly is `body`?

Processes... **Function Objects**

Scala allows us to create **anonymous function** (and **procedure**) objects.

```
val double = new Function1[Int, Int] {
  def apply(int: Int): Int = 2 * int }
```

Note: `double`

- is a **unary** function from `Int` to `Int` that
- is an instance of an **anonymous class extending** the trait `Function1`, which
- **implements** `apply` declared in `Function1`.

Processes... Function Objects

Scala allows us to create **anonymous function** (and **procedure**) objects.

```
val double = new Function1[Int, Int] {
  def apply(int: Int): Int = 2 * int }
```

Note: `double`

- is a **unary** function from `Int` to `Int` that
- is an instance of an **anonymous class extending** the trait `Function1`, which
- **implements** `apply` declared in `Function1`.

Questions:

- Don't we have to **explicitly** `override` parent methods?
- Why don't we need (though we can) `override` `def apply` here?

Processes... Function Objects

Being lazy programmers, we appreciate more **compact syntax**.

```
val double = (int: Int) => 2 * int
```

Note: As previously,

- `double` is an anonymous unary function with
- **explicit** input type `Int`, and
- **implicit** output type (`Int`).

Processes... Function Objects

Being lazy programmers, we appreciate more **compact syntax**.

```
val double = (int: Int) => 2 * int
```

Note: As previously,

- `double` is an anonymous unary function with
- **explicit** input type `Int`, and
- **implicit** output type (`Int`).

Questions:

- Is the type of `double` itself explicit or implicit? What is `double`'s type?

Processes... Function Objects

Being lazy programmers, we appreciate more **compact syntax**.

```
val double = (int: Int) => 2 * int
```

Note: As previously,

- `double` is an anonymous unary function with
- **explicit** input type `Int`, and
- **implicit** output type (`Int`).

Questions:

- Is the type of `double` itself explicit or implicit? What is `double`'s type?

Consider:

```
val double: Int => Int = (int: Int) => 2 * int
```

Processes... Function Objects

Functions in Scala are **first-class objects**.

```scala
def twice[T] =
  (func: (T => T)) =>
    (input: T) => func(func(input))
```

Note:

- `twice` is a `T`-generic method whose value is an anonymous function that
- **takes** as an argument a **function** `func` from `T` to `T` and
- **returns** a **function** that takes an `input` of type `T` and
- returns the result of `func` applied to the result of `func` applied to `input`.

Processes... Function Objects

Functions in Scala are **first-class objects**.

```
def twice[T] =
  (func: (T => T)) =>
    (input: T) => func(func(input))
```

Note:

- `twice` is a `T`-generic method whose value is an anonymous function that
- **takes** as an argument a **function** `func` from `T` to `T` and
- **returns** a **function** that takes an `input` of type `T` and
- returns the result of `func` applied to the result of `func` applied to `input`.

Questions:

- What is the value of `twice(twice[Int])(double)(2)`?
- What is the type of `twice[T]`?

Processes and Threads

Wrapping code to be evaluated by a thread into a nullary procedure or function (a **thunk**), we can:

- **delay** its evaluation until a call to the thunk is made;
- **pass** it in unevaluated form **as an argument** to a function.

```
object ThreadUtils {
  def thread(body: () => Unit): Thread =
    new Thread { override def run() = body() }
  ... }
```

Note:

- with `body: () => Unit ... run() = body()`, you **pass by value** a nullary procedure to be called with no arguments by the thread;
- with `body: => Unit ... run() = body`, you **pass by name** an unevaluated statement to be evaluated by the thread;
- `=> Unit` is a **syntactic extension** that is resolved to `() => Unit` at compile time. (Hint: try to overload `def thread` with both singatures.)

Processes and Threads

Consider this example:

```
object NewThreadDemo extends App {
  log("1")
  val t = thread { log("2"); Thread.sleep(1000); log("3") }
  log("4")
  t.start()
  log("5")
  Thread.sleep(1000)
  log("6")
  t.join()
  log("7") }
```

Note:

- `Thread.sleep(1000)` makes the **calling** thread **pause for at least** 1000 ms;
- `t.join()` makes the **calling** thread (main) **wait until** thread `t` **completes**.

Questions: What can we say about the output?

Processes and Threads

Consider this example:

```
object NewThreadDemo extends App {
  log("1")
  val t = thread { log("2"); Thread.sleep(1000); log("3") }
  log("4")
  t.start()
  log("5")
  Thread.sleep(1000)
  log("6")
  t.join()
  log("7") }
```

Note:

- `Thread.sleep(1000)` makes the **calling** thread **pause for at least** 1000 ms;
- `t.join()` makes the **calling** thread (main) **wait until** thread `t` **completes**.

Questions: What can we say about the output? **Test it!**

Processes and Threads

If a program is executed in **more than one thread**, then...

Processes and Threads

If a program is executed in **more than one thread**, then...

- the overall behaviour will usually be **nondeterministic**;
- the execution may lead to **unexpected results**;
- the execution may lead to **incorrect results**;
- the execution may lead to **no results** at all;
- **testing** is difficult and tricky.

Processes and Threads

If a program is executed in **more than one thread**, then...

- the overall behaviour will usually be **nondeterministic**;
- the execution may lead to **unexpected results**;
- the execution may lead to **incorrect results**;
- the execution may lead to **no results** at all;
- **testing** is difficult and tricky.

How do you change a lightbulb in...

- Object-oriented programming? You don't. Instead, you tell the lamp to do it.
- Functional programming? You can't.
- Logical programming? You imply that it is changed.
- **Concurrent programming**? You take the lamp to a secure area where nobody else can try to change the lightbulb while you're changing it, and do an atomic swap of the old lightbulb with the new lightbulb.

Processes and Threads

```
log("1")
val t = thread { log("2"); Thread.sleep(1000); log("3") }
log("4")
t.start()
log("5")
Thread.sleep(1000)
log("6")
t.join()
log("7")
```

In the `NewThreadDemo` example, we can say that the output

- **always** starts with `1 4`;
- **always** ends with `7`;
- **always** contains `2` before `3` and `5` before `6`;
- **almost certainly** contains `2` before `6` and `5` before `3`;
- **may** contain `2` before `5` or vice versa, and likewise with `3` and `6`.

Processes and Threads

Let us create another convenience method in `ThreadUtils`:

```
def start(body: => Unit, delay: Int = 0): Thread = {
  val t = thread { Thread.sleep(delay); body }
  t.start()
  t }
```

Note:

- `start` calls the previously defined `thread`;
- the `body` passed to `thread` contains a `Thread.sleep` call and the `body` passed to `start`;
- `delay` has a **default** value `0` and is **optional** in calls to `start`;
- a call to `start` will create a new thread that will execute `body` after a set `delay`, start the thread, and return it.

Processes and Threads

Mutable state and concurrent execution **interact unpredictably**.

```
var state = 0
val threads = (1 to n).map(i => start({ state = (state + i) * i }, delay))
threads.foreach(_.join())
```

Note:

- `1.to(n)` is a **sequence** of `Int`s from `1` to `n`, inclusive;
- `1 to n` is **infix** form of `1.to(n)`;
- `map` **applies** a unary function to each element of the sequence, and
- returns a **new sequence** of the results;
- `i => start(...)` takes an `Int` and returns a **started** `Thread` that
- sleeps for `delay` miliseconds, and then **destructively updates** `state` using `i`;
- `threads` is a **sequences of threads**;
- `foreach` **iterates** over `threads` to join each in turn;
- `_.join()` is **syntactic sugar** for `thread => thread.join()`.

Processes and Threads

```
def test(n: Int, delay: Int = 0) = {
  var state = 0
  val threads = (1 to n).map(i => start({ state = (state + i) * i }, delay))
  threads.foreach(_.join())
  state }
```

Question: What is the value of `test(2)`? `test(3)`?

- The **order** of execution of multiple threads is **unpredictable**.
- For `test(2)`, we would expect

Processes and Threads

```scala
def test(n: Int, delay: Int = 0) = {
  var state = 0
  val threads = (1 to n).map(i => start({ state = (state + i) * i }, delay))
  threads.foreach(_.join())
  state }
```

Question: What is the value of `test(2)`? `test(3)`?

- The **order** of execution of multiple threads is **unpredictable**.
- For `test(2)`, we would expect either ((0+1)*1+2)*2 = 6 or ((0+2)*2+1)*1 = 5.
- For `test(3)`, we would expect 22, 23, 24, 27, or 28.

Processes and Threads

```
def test(n: Int, delay: Int = 0) = {
  var state = 0
  val threads = (1 to n).map(i => start({ state = (state + i) * i }, delay))
  threads.foreach(_.join())
  state }
```

Question: What is the value of `test(2)`? `test(3)`?

- The **order** of execution of multiple threads is **unpredictable**.
- For `test(2)`, we would expect either ((0+1)*1+2)*2 = 6 or ((0+2)*2+1)*1 = 5.
- For `test(3)`, we would expect 22, 23, 24, 27, or 28.

But (with sufficiently large `delay`) **we actually get**:

- 1, 4, 5, 6 (for `test(2)`);
- 1, 4, 5, 6, 9, 10, 12, 21, 22, 23, 24, 27, 28 (for `test(3)`), etc.

Processes and Threads

```
def test(n: Int, delay: Int = 0) = {
  var state = 0
  val threads = (1 to n).map(i => start({ state = (state + i) * i }, delay))
  threads.foreach(_.join())
  state }
```

Question: What is the value of `test(2)`? `test(3)`?

- The **order** of execution of multiple threads is **unpredictable**.
- For `test(2)`, we would expect either ((0+1)*1+2)*2 = 6 or ((0+2)*2+1)*1 = 5.
- For `test(3)`, we would expect 22, 23, 24, 27, or 28.

But (with sufficiently large `delay`) **we actually get**:

- 1, 4, 5, 6 (for `test(2)`);
- 1, 4, 5, 6, 9, 10, 12, 21, 22, 23, 24, 27, 28 (for `test(3)`), etc.

Questions: Why are all those **unexpected** results **smaller** than those expected?

Processes and Threads

With concurrency and mutable state, predicting is not easy. Experiment!

```scala
def tests(ntests: Int = 100, nthreads: Int = 1, delay: Int = 0) =
  (1 to ntests).par
    .map(_ => test(nthreads, delay))
    .groupBy(identity).mapValues(_.size)
    .toSeq.seq.sortBy(_._1)
```

Note:

- `par` **parallelizes** the sequence (further steps can be run concurrently);
- `map` **executes** `test` for each value in `(1 to ntests)`;
- `groupBy` **aggregates** the return values into a `Map` with unique results as keys and sequences of those as values;
- `mapValues` **calculates** the length of each sequence value in the map;
- `toSeq` **converts** the map into a sequence of (result, size) tuples;
- `seq` **de-parallelizes** the sequence;
- `sortBy` **reorders** the tuples by the first element.

Processes and Threads

The update of `state` is not atomic:

```
state = (state + i) * i
```

is equivalent to

```
val temp1 = state
val temp2 = temp1 + i
val temp3 = temp2 * i
state = temp3
```

Note:

- `state` is the only **mutable**, **shared** state variable;
- `i`, `temp1`, `temp2`, and `temp3` are thread-local and immutable;
- threads may access `state` in **between read and write** operations.

Processes and Threads

In `test(2)`, one possible sequence of events is

- thread 1 | `val temp1 = state` (`state` = 0, thus `temp1` = 0)
- thread 2 | `val temp1 = state` (`state` = 0, thus `temp1` = 0)
- thread 2 | `val temp2 = temp1 + i` (`i` = 2, `temp1` = 0, thus `temp2` = 2)
- thread 1 | `val temp2 = temp1 + i` (`i` = 1, `temp1` = 0, thus `temp2` = 1)
- thread 1 | `val temp3 = temp2 * i` (`i` = 1, `temp2` = 1, thus `temp3` = 1)
- thread 2 | `val temp3 = temp2 * i` (`i` = 2, `temp2` = 2, thus `temp3` = 4)
- thread 2 | `state = temp3` (`temp3` = 4, thus `state` = 4)
- thread 1 | `state = temp3` (`temp3` = 1, thus `state` = 1)

Processes and Threads

In `test(2)`, one possible sequence of events is

- thread 1 | `val temp1 = state` (`state` = 0, thus `temp1` = 0)
- thread 2 | `val temp1 = state` (`state` = 0, thus `temp1` = 0)
- thread 2 | `val temp2 = temp1 + i` (`i` = 2, `temp1` = 0, thus `temp2` = 2)
- thread 1 | `val temp2 = temp1 + i` (`i` = 1, `temp1` = 0, thus `temp2` = 1)
- thread 1 | `val temp3 = temp2 * i` (`i` = 1, `temp2` = 1, thus `temp3` = 1)
- thread 2 | `val temp3 = temp2 * i` (`i` = 2, `temp2` = 2, thus `temp3` = 4)
- thread 2 | `state = temp3` (`temp3` = 4, thus `state` = 4)
- thread 1 | `state = temp3` (`temp3` = 1, thus `state` = 1)

Oops!

Synchronization

To prevent undesirable thread interactions, **protect shared**, **mutable** state.

```
def test(n: Int, delay: Int = 0) = {
  var state = 0
  val threads = (1 to n)
    .map(i => start(this.synchronized { state = (state + i) * i }, delay)
  ... }
```

Note:

- `this` refers to the **enclosing object** (one that `test` is a method of);
- the `synchronized` statement wraps a block to be executed **atomically**;
- `synchronized` causes a thread to **obtain a lock** on `this`;
- the thread **keeps the lock** until ready with executing the block;
- when `this` is locked, **no other thread** can lock it; hence
- **only one thread** at a time can be executing the block;
- the block can be executed only as a **whole** (atomically).

Synchronization

In `test(2)`, one possible sequence of events is

- thread 1 | **lock**
- thread 1 | `val temp1 = state` (`state` = 0, thus `temp1` = 0)
- thread 2 | **cannot lock**
- thread 1 | `val temp2 = temp1 + i` (`i` = 1, `temp1` = 0, thus `temp2` = 1)
- thread 1 | `val temp3 = temp2 * i` (`i` = 1, `temp2` = 1, thus `temp3` = 1)
- thread 2 | **cannot lock**
- thread 1 | `state = temp3` (`temp3` = 1, thus `state` = 1)
- thread 1 | **unlock**
- thread 2 | **lock**
- thread 2 | `val temp1 = state` (`state` = 1, thus `temp1` = 1)
- thread 2 | `val temp2 = temp1 + i` (`i` = 2, `temp1` = 1, thus `temp2` = 3)
- thread 2 | `val temp3 = temp2 * i` (`i` = 2, `temp2` = 3, thus `temp3` = 6)
- thread 2 | `state = temp3` (`temp3` = 6, thus `state` = 6)
- thread 2 | **unlock**

Quiz!

Threads synchronizing on **the same object** may **execute different code**.

Question: What is the most likely value to be printed by this program?

Quiz!

Threads synchronizing on **the same object** may **execute different code**.

Question: What is the most likely value to be printed by this program?

```
object Quiz extends App {
  var state = 10
  start(this.synchronized { state *= 10 }, 1000)
  start(this.synchronized { state += 10 }, 1000)
  println(s"$state") }
```

Synchronization

Synchronization solves one problem, but introduces others.

Consider the classic problem of **dining philosophers**.

```
case class Fork()
case class Philo(left: Fork, right: Fork) {
  def eat() = left.synchronized { right.synchronized { log(s"$this") } } }
class Dinner(val n: Int) {
  val forks = (0 until n).map(_ => Fork())
  val philos = (0 until n).map(i => Philo(forks(i), forks((i+1)%n))) }
```

Synchronization

Synchronization solves one problem, but introduces others.

Consider the classic problem of **dining philosophers**.

```scala
case class Fork()
case class Philo(left: Fork, right: Fork) {
  def eat() = left.synchronized { right.synchronized { log(s"$this") } } }
class Dinner(val n: Int) {
  val forks = (0 until n).map(_ => Fork())
  val philos = (0 until n).map(i => Philo(forks(i), forks((i+1)%n))) }
```

Question: What will happen if we throw a party?

```scala
new Dinner(10).philos.foreach(_.eat())
```

Synchronization

Synchronization solves one problem, but introduces others.

Consider the classic problem of **dining philosophers**.

```
case class Fork()
case class Philo(left: Fork, right: Fork) {
  def eat() = left.synchronized { right.synchronized { log(s"$this") } } }
class Dinner(val n: Int) {
  val forks = (0 until n).map(_ => Fork())
  val philos = (0 until n).map(i => Philo(forks(i), forks((i+1)%n))) }
```

Question: What will happen if we throw a party?

```
new Dinner(10).philos.foreach(_.eat())
```

It's a **kølapp party**, no deadlock.

Synchronization

How about a real party?

```
new Dinner(10).philos.par.foreach(_.eat())
```

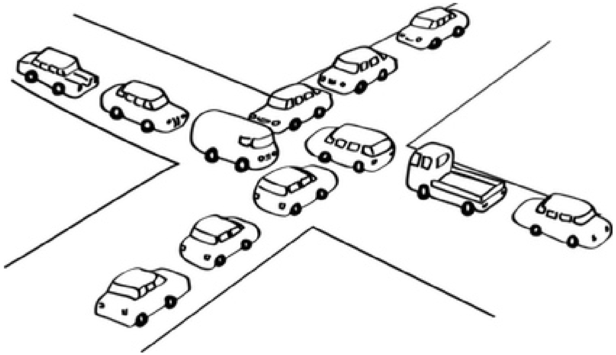
Note:

- the philosophers start (or attempt to start) eating simultaneously;
- each philosopher grabs the left fork;
- no philosopher can grab the right fork;
- the **execution halts** indefinitely.

Synchronization

Deadlock is a dangerous consequence of too much synchronization.

- The dining philosophers has some trivial solutions.
- In general, chances for deadlock can be reduced by synchronizing over the **smallest possible blocks** of code and on the **most local objects** possible.
- Other ways to assure atomicity may help avoid deadlocks entirely.



Compare and Swap

Consider the task of generating successive unique ids.

```
object UIDGenerator {
  var uid = 0;
  def getNext = {
    uid += 1
    uid } }
```

Note:

- This implementation is **flawed**.
- The bug is solved by wrapping the update `uid += 1` in a `synchronized` block.
- It may be more convenient to use a **compare-and-swap** (CAS) operation.

Compare and Swap

```scala
import java.util.concurrent.atomic.AtomicInteger
import scala.annotation.tailrec
object UIDGenerator {
  val uid = new AtomicInteger(0)
  @tailrec def getNext: Int = {
    val current = uid.get
    val updated = current + 1
    if uid.compareAndSet(current, updated) return updated
    getNext }
```

Note:

- an `AtomicInteger` is a **wrapper** around an `Int` that
- allows one to **replace** the content if one knows the current value
- `getNext` is a **recursive** (self-calling) method that gets the current value, calculates an update, and tries to swap them;
- `getNext` **fails** if another thread has modified `uid` since `current` was obtained;
- `@tailrec` states that `getNext` is **tail recursive**.

TDT4165: Concurrency in Scala, Part II

November 4, 2015

Lazy Values

A **lazy value** is a value that

- remains **uninitialized** until it is needed;
- is initialized the **first time** it is used,
- is initialized only **once**.

```
lazy val value = { /* some horribly involved computation */ }
```

Note:

- lazy values are created with the keyword `lazy`;
- there are **no lazy variables**.

Lazy Values

A **lazy value** is a value that

- remains **uninitialized** until it is needed;
- is initialized the **first time** it is used,
- is initialized only **once**.

```
lazy val value = { /* some horribly involved computation */ }
```

Note:

- lazy values are created with the keyword `lazy`;
- there are **no lazy variables**.

With lazy values, potentially complex computations

- are **avoided** if their result is never used;
- may have **surprising results** in sequential execution context;
- may have **unpredictable results** in concurrent execution context.

Lazy Values

In a sequential program, lazy values may lead to surprising, but predictable behaviour.

```
var square = 121
lazy val root = math.sqrt(square).toInt
var square = 0
log(s"square: $square, root: $root")
```

Question: What can we say about the output?

Lazy Values

In a sequential program, lazy values may lead to surprising, but predictable behaviour.

```
var square = 121
lazy val root = math.sqrt(square).toInt
var square = 0
log(s"square: $square, root: $root")
```

Question: What can we say about the output?

Note:

- `root` is first used in the `println` statement;
- `root` is uninitialized until then;
- when `root` is initialized, `square` is already reassigned the value 0;
- hence, `root` = 0.

Lazy Values

In a concurrent program, lazy values may lead to unpredictable behaviour.

```
var root = 0
lazy val square = root * root
val thread = (1 to 3)
    .map(i => start({ root = i; log(s"square: $square") }, 100))
```

Question: What can we say about the output?

Lazy Values

In a concurrent program, lazy values may lead to unpredictable behaviour.

```
var root = 0
lazy val square = root * root
val thread = (1 to 3)
    .map(i => start({ root = i; log(s"square: $square") }, 100))
```

Question: What can we say about the output?

Note:

- three different threads are started (in addition to main);
- the **first thread** that reaches its `println` statement **forces** `square` to be initialized;
- the remaining threads see `square` **already initialized** and print the same value as the first thread.

Lazy Values

Streams (aka *lazy lists*) are a prime use example for lazy values.

```
def ints(first: Int): Stream[Int] =
  Stream.cons(first, ints(first + 1))
def primes(ints: Stream[Int]): Stream[Int] =
  Stream.cons(ints.head, primes(ints.tail filter (_ % ints.head != 0)))
def primes(n: Int): List[Int] =
  primes(ints(2)) take n
```

Note:

- `Stream.cons` lazily **prepends** the first argument to the second (a stream);
- `ints(n)` **lazily builds** an **infinite stream** of integers starting at `n`;
- `primes(ints)` lazily builds an infinite stream of prime numbers found in `ints`;
- `primes(n)` lazily builds a **finite stream** of the first `n` primes.

Lazy Values

Lazy values **delay** computations, but do not run them concurrently.

```
val n = 1000
val ps = primes(n)
...
ps.foreach(/* process each prime */)
```

Note:

- there is **no delay** when `ps` is assigned, and the thread proceeds;
- there is **delay** when `ps.foreach` is called;
- the whole computation of `primes(n)` is **enforced** before the first iteration of `foreach`;
- `val ps = primes(n)` is **functionally equivalent** to `lazy val ps = primes(n).toList`.

Lazy Values

Lazy values **delay** computations, but do not run them concurrently.

```
val n = 1000
val ps = primes(n)
...
ps.foreach(/* process each prime */)
```

Question: How can we update the code so that:

- the computation of `primes(n)` actually **starts** when `ps` is assigned,
- the original thread **proceeds immediately** with further code,
- the thread is able to **check** whether computation of `ps` is complete, and then
- **if complete**, use `ps` without having to uselessly wait for it,
- otherwise, proceed with other tasks **instead of blocking**.

Lazy Values

We can use shared state and wait for the child thread to complete:

```
val n = 1000
var ps: List[Int] = List.empty
val t = start { ps = primes(n) toList }
...
t.join()
ps.foreach(/* process each prime */)
```

Note: This approach is

- explicitly **waiting on thread completion**, not on the result;
- unable to **block conditionally**, only if the asynchronous computation is ready;
- error-prone: shared state + concurrency = evil

Futures

```scala
val n = 1000
val future = Future { primes(n) }
...
while (!future.isCompleted) { /* proceed with other tasks while waiting */ }
future.value match {
  case Some(Success(ps)) => ps.foreach(/* process the prime */)
  case _ => }
```

Note:

- `Future` is an **object**;
- `Future { primes(n) }` is short for `Future.apply(primes(n))`;
- `Future.apply` returns an instance of the **class** `Future`;
- `future` is a wrapper that will eventually be filled with a value;
- `future.isCompleted` is a **non-blocking** check of the future's state;
- `future.value` is the result of the computation.

Futures

```
trait Future[+T] extends Awaitable[T] {
  def value: Option[Try[T]]
  ... }
sealed abstract class Option[+T] ...
case object None extends Option[Nothing] ...
final case class Some[+T](x: T) extends Option[T] ...
sealed abstract class Try[+T] ...
final case class Success[+T](value: T) extends Try[T] ...
final case class Failure[+T](exception: Throwable) extends Try[T] ...
```

Note:

- `future.value` is an instance of `Option[Try[Stream[Int]]]`;
- until `future` is **completed**, `future.value` is `None`;
- if `primes(n)` **succeeded**, `future.value` is `Some(Success(ps))`;
- if `primes(n)` **failed** with exception `e`, `future.value` is `Some(Failure(e))`;
- `[+T]` denotes **type covariance**: if `U` is subclass of `T`, then `X[U]` is subclass of `X[T]` (think `<? extends T>` in Java).

Futures

Code depending on a future can be executed in a **callback**:

```
val future = Future { primes(n) }
future foreach {
  case ps => ps.foreach(p => log(s"$p") }
future foreach {
  case ps => ps.zipWithIndex.foreach(ip => log(s"#${ip._2}: ${ip._1}") }
...
```

Note:

- there are two separate **callbacks** installed on `future`;
- the two `future foreach` statements **return immediately**, irrespectively of `future.value`;
- the current thread proceeds without blocking;
- once `future` is complete, the callbacks can run **concurrently** to each other and the current thread.

Futures

Callbacks can also handle failures—futures with `Failure[T]` as the value:

```
val future = Future { primes(n) }
future foreach { case ps => /* process primes */ }
(future failed) foreach { case e => /* process exception */ }
...
```

Note:

- `future foreach` **will only execute if** `future.value` is `Some(Success(ps))`;
- `(future failed) foreach` **will only execute if** `future.value` is `Some(Failure(e))`;
- `(future failed) foreach` is syntactic variant of `future.failed foreach` (and `future.failed.foreach`);
- `future onSuccess` and `future onFailure` are equivalent to the above;
- `future onComplete` installs a callback the will process `future` on both success and failure.

Promises

Futures are

- **read-only** placeholders for values that **may not yet exist**.

Promises are

- **write-only**, **single-assignment** placeholders that
- **complete** futures using the `success` method, or
- **fail** futures using the `failure` method.

```
val promise: Promise[T] = Promise[T]
val future: Future[T] = promise.future
```

Note:

- a promise has an **associated future** object which
- can be used to **test** for and **read** the value that
- can be **set** through the promise object.

Promises

```
val promise = Promise[T]
val future = promise future
future foreach { case result => /* handle result */ }
future.failed foreach { case error => /* handle failure */ }
start { promise success new T(/* provide value */) }
start { promise failure new Exception(/* provide failure message */) }
```

Note:

- `Promise[T]` **constructs** a new promise object (for some type `T`);
- `promise future` is the **corresponding future** object;
- `promise success` **attempts** to assign the promise a **success value**;
- `promise failure` **attempts** to assign the promise a **failure exception**;
- **only one** of the above concurrent threads may succeed;
- when one did, the other **must fail**.

Actors

The **actor model** of concurrent computing allows

- mutable state to be **entirely hidden** within individual actors;
- communication between actors to be **limited to passing messages**;
- **no memory** to be shared between actors;
- each actor to process its message queue **sequentially**; thus
- the order of processing of messages sent to different actors is **non-deterministic**,
- the order of processing of messages sent to the same actor is **deterministic**.

The Scala `Actors` library has been deprecated, the current default is the `akka` library—see akka.io.

Actors

Consider a counter that can be used to keep track of passengers on a flight:

```
import akka.actor.Actor
class Counter(var count: Int = 0) extends Actor {
  override def receive: Receive = {
    case '+' => count += 1
    case '_' => count = 0
    case '.' => context.stop(self)
    case _ =>  } }
```

Note:

- a `Counter` has an internal **mutable** state, intially set to 0;
- the method `receive` **matches a message** against a list of patterns;
- if it matches `'+'`, the counter increases its state by 1;
- if it matches `'_'`, the counter resets its state to 0;
- if it matches `'.'`, the counter **stops receiving** further messages;
- otherwise, the message is simply **ignored**.

Actors

Actors are constructed using `Props`, `ActorRef` configuration objects:

```scala
val props: Props = Props(classOf[Counter], 0)
```

Note:

- `classOf[T]` is the **runtime representation** of a class (think `T.class` in Java);
- `Props()` is short for `Props.apply()`;
- `Props(classOf[Counter], 0)` **configures** counters with initial state 0;
- providing 0 in `Props` is not necessary as `Counter` has a **default** for `count`.

Actors

Actors are constructed using `Props`, `ActorRef` configuration objects:

```
val props: Props = Props(classOf[Counter], 0)
```

Note:

- `classOf[T]` is the **runtime representation** of a class (think `T.class` in Java);
- `Props()` is short for `Props.apply()`;
- `Props(classOf[Counter], 0)` **configures** counters with initial state 0;
- providing 0 in `Props` is not necessary as `Counter` has a **default** for `count`.

It is customary to provide a **companion object** for the actor class:

```
object Counter {
  def props = Props(new Counter)
  def props(init: Int = 0): Props = Props(classOf[Counter], init) }
```

- `props` is a `Props`-builder method.

Actors

Actors are constructed and run within an `ActorSystem`:

```
val system: ActorSystem = ActorSystem("Counters")
val counter: ActorRef = system.actorOf(Counter.props, name="counter")
```

Note:

- `ActorSystem()` is short for `ActorSystem.apply()`;
- `system` is an `ActorSystem` with the name `"Counters"`;
- `Counter.props` creates a `Props` object;
- `counter` is an `ActorRef`, a **reference to an actor** within `system`;
- `counter` has the name `"counter"` within `system`.

Actors

Actors receive and process messages:

```
counter ! '+'  // 1
counter ! '+'  // 2
counter ! '_'  // 3
counter ! '?'  // 4
counter ! '.'  // 5
counter ! '+'  // 6
```

Actors

Actors receive and process messages:

```
counter ! '+'  // 1
counter ! '+'  // 2
counter ! '_'  // 3
counter ! '?'  // 4
counter ! '.'  // 5
counter ! '+'  // 6
```

Note:

- `1` and `2` bump `counter` to 1, then 2;
- `3` resets `counter` to 0;
- `4` sends an unrecognized message;
- `5` stops `counter` from receiving further messages;
- `6` is never delivered.

Actors

Actors can handle messages differently depending on their state:

```
class Counter(var count: Int = 0) extends Actor {
  def receive = unlocked
  def unlocked: Receive = {
     case '+' => count += 1
     ...
     case '*' => context.become(locked) }
  def locked: Receive = {
     case '*' => context.become(unlocked)
     case _ => } }
```

Note:

- initially, `receive` is `unlocked` and the counter acts as previously;
- if the message is `'*'`, `receive` is changed to `locked`;
- `locked` ignores all messages except for `'*'`, on which it restores `receive` to `unlocked`.

Actors

Actors can **send responses** to their senders:

```
class Counter(var count: Int = 0) extends Actor {
  def receive = {
    case '=' => sender ! count
    ... } }
...
val response = counter ? '='
response match { case Some(Success(count)) => /* handle count */ }
```

Note:

- `counter ? '='` sends an **ask** request to `counter`;
- in `receive`, if the message is =, the actor sends `count` to the **sender** of the message;
- `sender` is a **method inherited** from `Actor`;
- `response` is a **future**;
- if the operation suceeded, `response` contains `Some(Success(count))`.