Scala 3 for Beginners

§

🔷 Made with | Obsidian

🗄 Type | blog 🎲 Category | computer-science </> Technologies | Scala, VS Code, PowerShell

📖 Website | Post Link

Scala is a strong, statically typed, high-level, general-purpose programming language that supports both object-oriented programming and functional programming. It was developed by Martin Odersky & a group of computer scientists at EPFL in Switzerland.

In this Blog Article, we'll discuss some historical background regarding Scala, the main advantages & disadvantages of the language, and provide an introduction to functional programming and its main traits. We'll then go over installation & configuration using VS Code & the powerful Metals extension, and creating, compiling & running projects using sbt.

We'll create our first Scala project and use it to discuss Scala's main features, general syntax, main constructs, and main types. We'll close this segment with some next steps for those interested in learning more about Scala & functional programming.

We'll be using Scala 3 scripts which can be found in the Blog Article Repo.

§

Table of Contents

§

What is Scala?

Scala, which stands for Scalable Language, is a strong, statically typed, high-level, general-purpose programming language that supports both object-oriented programming and functional programming. It was mainly developed with data-intensive applications in mind and is built on top of Java, which means that Scala compiles to Java bytecode and is executed by the **Java Virtual Machine** (*JVM*).

Scala was built with three main purposes in mind:

- To be scalable
- To be powerful
- To be safe

Each of these characteristics is carefully taken care of by a wide variety of carefully-crafted features, such as a robust type hierarchy structure, a wide collection of sophisticated higher-order methods, a reinvention of classes & objects combined with type-driven development, and much more.

1. A brief historical overview

Scala's journey begins in the early 2000s at the **École Polytechnique Fédérale de Lausanne** (*EPFL*), Switzerland, under the guidance of Martin Odersky. Drawing from his experiences with Java and functional programming languages, Odersky aimed to design a language that married the best aspects of both worlds. The first version of Scala was released publicly in 2004, and since then, it has evolved significantly.

For example, a turning point came in 2021 with the launch of Scala 3, introducing various new features and changes that made the language even more robust and versatile. While Scala has been growing slowly compared to other languages, it has been gaining more popularity since the Scala 3 release.

2. The foundations

Scala is built on top of Java; it uses the entire Java typeset and the JVM for compilation & execution. This means that many of Scala's concepts will be familiar if we are already familiar with Java or a similar language. Scala's interoperability with Java is one of its strongest suits. We can use Java libraries and frameworks in Scala and vice versa. The fact that Scala compiles to Java bytecode and runs on the Java Virtual Machine (JVM) provides the advantages of portability, efficiency, and access to the mature ecosystem of Java.

However, while Scala has strong ties with Java, it brings its own innovations and constructs. Scala introduces a rich type system, case classes, traits, and an advanced collection library. It also adds many functional programming features, such as first-class functions, immutability by default, and pattern matching.

A unique aspect of Scala is its fusion of object-oriented and functional programming. Every value is an object, and every operation is a method call in Scala, adhering to object-oriented design principles. At the same time, Scala adopts many principles of functional programming, like higher-order functions and strong support for recursion.

In essence, Scala takes the familiar foundation of Java but builds upon it with its own set of innovative features and design philosophy, creating a truly powerful language for modern programming needs.

§

Why Scala?

Choosing the right programming language is like choosing the right tool for a job. Scala often stands out as a top contender in software development. Why? Well, it boils down to five key attributes:

- Scalability
- Parallelism
- Security
- Adoption
- Functional programming

1. Scala(bility)

Let's start with the obvious, given the name: Scala is designed with scalability at its core. It has an innate ability to grow seamlessly alongside the demands of our projects. Whether we're working on a small script to automate a simple task, building out the core system for a startup, or even if we're deep in developing a sophisticated, distributed application, Scala just works.

And it's not just about the size of the projects we're working on. Scala allows us to write code in a style that suits us. Whether we want to write in a more traditional, object-oriented style or if we want to adopt functional programming, Scala enables us to scale our programming paradigm as well. This kind of flexibility is what makes Scala a truly scalable language, not just in terms of project size but also in terms of programming style and team skill levels.

2. Parallelism

In this era where hardware evolution means more cores, not just faster clock speeds, being able to write software that takes advantage of parallel computing capabilities is becoming increasingly important. This is where Scala comes into its own. Its functional nature and a robust collection of libraries make it significantly easier to write parallel and concurrent programs.

Scala's support for immutability and higher-order functions allows us to write code that is easier to reason about in a parallel execution context. Furthermore, libraries such as Akka enable us to write resilient message-driven applications, providing powerful abstractions for parallel and concurrent code.

3. Security

Scala's strong static type system and design decisions around immutability make our code safer. The Scala compiler has a remarkable eye for catching errors at compile-time rather than at run-time, dramatically reducing the likelihood of encountering bugs in production. This emphasis on security can save us a considerable amount of debugging time and make our software more robust.

Beyond this, Scala's expressiveness and type inference allows us to write concise code without sacrificing readability or safety. This means we spend less time on boilerplate code and more time thinking about the logic and security of our applications.

4. Adoption

Scala isn't just a language that's good on paper. It's got credibility among top-tier companies all around the world. Multinational tech companies like Twitter, LinkedIn, and Netflix have all integrated Scala into their tech stacks, and for good reasons.

5. Functional programming

Finally, Scala shines when it comes to functional programming. With its sophisticated blend of functional and OOP paradigms, Scala provides an excellent platform to delve into functional programming. It allows us to take a gentle step into the functional domain, letting us embrace it to the extent we're comfortable with, without forcing us into a complete paradigm shift overnight.

§

What to expect

Scala leverages multiple core functionalities, such as a rich type system, implicits, advanced pattern matching, powerful abstraction capabilities, and much more, to execute safely and highly-performantly. However, this makes Scala difficult to learn, even when discussing introductory topics.

Because of this, we must set realistic expectations: Scala takes time to get used to since the way of thinking for writing proper Scala code is different from other languages; it elegantly combines functional & OOP to create powerful abstractions; this means that we might have to think of how to approach problems differently.

1. The Scala way of thinking

We can, of course, write Scala code as if it were Python (*with some exceptions, of course*), but we would not be leveraging the full power that this language has in store for us. To properly write a Scala application, we must think the Scala way.

This encompasses one main rule of thumb: Scala seamlessly combines two <u>programming paradigms</u> while enforcing the key components of each one, unlike other languages that can be thought of as more "*relaxed*":

- **Functional**: Immutability, higher-order functions, pattern matching for comprehensions, collections, and functional transformations.
- **Object-Oriented**: Classes and objects, traits, inheritance and polymorphism, and encapsulation.

So, what does this mean? Well, in practical terms, the Scala way of thinking can be summarized in six core concepts:

- **Embracing immutability:** It's easy to reassign variables as we write our code, but this significantly affects compile time, execution performance, and safety. Thinking in terms of immutability forces us to be creative when approaching problems: one excellent example is using recursion in favor of loops.
- **Using pattern matching and deconstruction:** Pattern matching can be considered the more powerful brother of `if-else` statements: this technique can be used for checking types and deconstructing complex data structures.
- **Adjusting to a type-driven development:** If there's one fundamental aspect that can take time to get used to when coming from other dynamically-typed languages, it's type-driven development. It lets us think of our code less ambiguously; we don't have to guess the expected input & output types in a function because we closely delimit the rules of what can and cannot be done. We can use multiple techniques such as type annotations, type inheritance, bounds, and higher-kinded types for this.
- **Favoring composition over inheritance:** Although Scala supports inheritance, it encourages using composition and traits to promote code reuse and modularity.
- **Using powerful abstractions:** Scala enables the creation of powerful abstractions, such as type classes, implicit conversions, and context bounds. While Python has similar features, such as decorators and context managers, Scala's features are more versatile and expressive.
- **Taking advantage of concurrency and parallelism:** This one does not apply to all Scala applications but is one characteristic deeply embedded in the Scala ecosystem. It provides excellent support for concurrent and parallel programming through libraries like <u>Akka</u>, <u>Spark</u>, <u>Cats Effect</u>, and <u>Monix</u>. This differs from Python, which has the **Global Interpreter Lock** (*GIL*) that can limit parallelism.
- **Unit testing:** Unit testing is relevant in most modern programming languages but is particularly prevalent in Java & Scala: it ensures that individual units of code (*such as functions, methods, or classes*) work correctly and helps maintain code quality, catch regressions, and improve the overall robustness of the software. Scala has an entire ecosystem dedicated to unit testing, something we might not have done extensively in other languages; we can design & build entire test suites that, thanks to Metals, are extremely easy to execute and debug.

2. Versioning and IDEs

Also, we'll be using VS Code along with the Metals extension for this segment. However, there are more options for writing Scala code:

- <u>JetBrains IntelliJ IDEA</u>
- <u>ScalaIDE for Eclipse</u>
- <u>GNU Emacs</u>
- <u>Vim-Scala</u>
- <u>NetBeans</u>
- <u>Atom</u>

Although we can also use Scala 2 for some of the examples covered in this segment, it's best to use Scala 3 since the latter contains many powerful improvements over Scala 2.

---------§---------

Preparing our environment

Although environment setup in Scala is not as straightforward as in other languages, we have some great tools to aid us in the process: we will make sure to have the following components ready before starting to write code:

- The **Scala 3** language.
- A Scala 3 build tool, in this case, **sbt**.
- An IDE supporting Scala 3 syntax (*in this case, **VS Code***).
- An IDE productivity tool, in this case, **Metals**.

1. Installing Scala 3 & sbt using Coursier

For Scala 3 installation, we'll use a special tool called `Coursier`. This tool is a package & artifact manager that will:

- Install a JVM if none is present.
- Update the `JAVA_HOME` and `PATH` environment variables.
- Add `~\AppData\Local\Coursier\data\bin` to `PATH`.
- Install the following packages:
  - `ammonite`
  - `cs`
  - `coursier`
  - `scala`
  - `scalac`
  - `scala-cli`
  - `sbt`
  - `sbtn`
  - `scalafmt`
- Download & install the **Simple Build Tool** (*sbt*).

To begin, we'll do the following:

1. Download the latest release of `Coursier` matching our operating system.
2. Execute the installer.
3. Add `Coursier` to `PATH` in case we missed it.
4. Setup our Scala environment running the following in PowerShell:

CODE

```
cs setup --jvm adopt:11
```

To verify our installation, we can use the following commands:

CODE

```
java -version
scala --version
sbt about
```

Which should print the following:

```
java version "20.0.1" 2023-04-18
Java(TM) SE Runtime Environment (build 20.0.1+9-29)
Java HotSpot(TM) 64-Bit Server VM (build 20.0.1+9-29, mixed mode, sharing)

Scala code runner version 3.2.2 -- Copyright 2002-2023, LAMP/EPFL

This is sbt 1.8.2
```

Now that our installation is complete, we can install our IDE.

2. Installing VS Code

If we don't yet have VS Code installed, we can get it from the official downloads page. We need to select the Windows 8, 10, 11 executable and wait for it to download. When the installation is complete, we can verify by opening the Visual Studio Code application directly from the Windows start menu. A detailed configuration guide for VS Code is out of the scope of this article but can be consulted on the VS Code official documentation site.

3. Installing Metals for VS Code

Once we have Scala 3, sbt, and VS Code installed, we will proceed to install the Metals VS Code Extension:

1. Open VS code and head to the Extensions menu in the left panel. We can also open the Extensions menu by using the shortcut `Ctrl` + `Shift` + `X` or by opening the command palette by typing `F1` and searching for *Extensions: Install Extensions*.
2. We will search for `Scala (Metals)`, maintained by *Scalameta*, and install and enable it. We can also get the extension using this link.

Now that everything's in place, we're ready to start configuring our working environment.

---

§

The Scala REPL

As with many programming languages, Scala provides a **REPL** (*Read-Evaluate-Print-Loop*), a tool for evaluating expressions in Scala.

A Scala REPL can be opened from anywhere by using the following command:

CODE

```
scala
```

OUTPUT

```
Welcome to Scala 3.2.2 (11, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.
```

We can then input any expression we'd like to evaluate:

CODE

```
println("Hello Scala")
```

OUTPUT

```
Hello Scala
```

Although most of the code we'll be writing will be inside Scala files, we can also use the REPL to quickly evaluate any expression on-the-fly.

---

§

Creating a new project

A typical Scala 3 generalized workflow consists of the following steps:

1. Create a project folder.
2. Create a project using sbt and a selected template (*can be external or our own*).
3. Import our project's main `build.sbt` file.
4. Write our Scala 3 code in a file appended by the `.scala` extension.
5. Compile & run our Scala 3 source file using sbt.
6. Test our source code if a test suite is included in our project.
7. Debug as per required.

We'll follow each step in detail.

1. Creating a project

The first step is to create a project folder, where our project will reside:

```
cd Documents

New-Item -ItemType Directory -Path scala-for-beginners

cd scala-for-beginners
```

We can then create our project using a predefined template. Templates in `sbt` are predefined project structures or scaffolding that can be used as a starting point for new projects. Templates provide a convenient way to set up common project configurations, directory structures, and build settings.

There are several templates available for Scala `sbt`, including:

- **Basic Template:** This template provides a simple project structure with minimal configuration.
- **Giter8 Template**: Giter8 is a project templating tool that generates projects from templates hosted on GitHub. It allows us to choose from a wide range of community-maintained templates.
- **Play Framework Template**: Play Framework is a web development framework for Scala. This template creates a Play project with the necessary configurations and directory structure.

For this segment, we'll be using the basic template. Once we're in our project folder, we can execute the following command in PowerShell:

```
sbt new scala/scala3.g8
```

This will create a new project based on the Scala 3 seed template. It's important to use `scala3`. Otherwise, a project using Scala 2 will be created.

We will then need to provide a name for our project when prompted:

```
name [Scala 3 Project Template]: scala-for-beginners
```

This will create one main folder with the name of our project, as well as multiple subfolders.

2. The directory structure

If we take a look inside our main project folder, we'll see the following directory structure:

```
scala-for-beginners/
|
├── project/
|   ├── build.properties
|   └── plugins.sbt
|
├── src/
|   ├── main/
|   |   ├── resources/
|   |   └── scala/
|   |       └── Main.scala
|   |
|   └── test/
|       ├── resources/
|       └── scala/
|           └── MainSpec.scala
|
├── build.sbt/
├── .gitignore/
└── README.md
```

Let us explore each component in detail:

- `scala-for-beginners/` : Is the root directory of our sbt project, named after the project name.
- `project/` : Contains sbt build configuration files.
    - `build.properties` : Specifies the sbt version to be used for the project.
    - `plugins.sbt` : Contains sbt plugin dependencies for the project.

- `src/` : The source code directory, containing both main and test code.
  - `main/` : Contains the main application code and resources.
    - `scala/` : Contains the main Scala source code files.
      - `Main.scala` : A sample Scala source file (*not required to be named* `Main` *; it's just a convention to call the file containing the main methods* `Main.scala` *).*
  - `test/` : Contains the test code and resources.
    - `scala/` : Contains the Scala test source code files.
      - `MainSpec.scala/` : A sample Scala test source file.
- `build.sbt` : The main sbt build configuration file, where we define project settings, the Scala version we're using, library dependencies, and other build-related configurations. This file is extremely important and should always be located under the root directory.
- `.gitignore/` : A sample `.gitignore` file if we're working with code versioning tools such as GitHub or GitLab.
- `README.md` : A sample `readme` file that should be included as an entry point to our project.

There are three directories/files we'll usually modify during project development:

- `src/` : This directory is meant to be modified and contains our source code.
- `build.sbt` : This file is intended to be modified if required.
- `.gitignore` , `README.md` : These, of course, are intended to be modified since the already provided are just templates.

Apart from this base structure, we'll include one additional element. For this, we will head to `scala-for-beginners/src/` and create a new folder called `worksheets` . We will then `cd` into our new directory, and create a new file called `MyExamples.worksheet.sc` ; we'll see what this is in a moment.

3. Basic build configuration

When we created our project, a `build.sbt` file was also generated; this is the main build file for our application, and contains a structure like such:

CODE

```
val scala3Version = "3.2.2"

lazy val root = project
  .in(file("."))
  .settings(
    name := "scala-for-beginners",
    version := "0.1.0-SNAPSHOT",

    scalaVersion := scala3Version,

    libraryDependencies += "org.scalameta" %% "munit" % "0.7.29" % Test
  )
```

This file is intended to be modified; here, we can add dependencies, packages, and the Scala version we're using for this project.

Let us break this down in more detail:

- `scala3Version` : The scala version we're using for our project.
- `libraryDependencies` : The key denoting the packages we plan to install in our project. A dependency can be added by simply appending a new `key := dependency` entry:

```
libraryDependencies += "org.typelevel" %% "cats-core" % "2.7.0"
```

Where:

- `org.typelevel` : Is the organization or group ID of the library, typically representing the package structure. In our case, it's the group ID for the Cats library.
- `%%` : Is an `sbt` -specific shorthand that automatically appends the Scala binary version (*e.g., 3.20 for Scala 3.20*) to the library's name. This helps us fetch the correct library version compatible with our project's Scala version.
- `cats-core` : Is the library name or artifact ID. In our case, it's the core module of the Cats library.
- `%` : Separates the library's organization, name, and version in the dependency definition. It's used to build the complete dependency string.
- `2.7.0` : Is the library version we want to include in our project. It represents the specific release of the Cats library we'll be using.

4. Working with Scala worksheets

A couple of minutes ago, we created a new file called `MyExamples.worksheet.sc` . This file is a worksheet Scala file and will let us evaluate Scala code snippets incrementally as if we were using the Scala REPL.

Scala Worksheets use the `.sc` extension, and unlike a typical `.scala` file, they're not compiled but run by the interpreter in real-time.

Worksheet files can be located in the following directory:

```
projectname/src/worksheet/MyExamples.worksheet.sc
```

The name structure must be the following:

- Start with the parent directory name, followed by a dot.
- Include worksheet `keyword` .
- End with `.sc` extension.

Whenever we write code in the `worksheet.sc` file and save it, the Scala interpreter will automatically run the code and produce the output inside the file.

Let us write a simple line in our newly-created worksheet and check its output:

```
println("Hello, World")
```

```
println("Hello, World")
// Hello, World
```

If there's any problem with our code, the compiler will throw an error.

5. Working with Scala source files

Scala files have the `.scala` extension and are compiled when we run the project if a main class exists.

We can have multiple `.scala` files with multiple classes, objects, and methods inside, but there will be one main class required for each project. This class should be treated as the entry point to our application.

5.1 Programs

We'll see classes & objects in more detail later on, but for now, we must know that a main class can be defined using three different ways:

- Using a `Main` object with Java-like syntax.
- Using a `Main` object which extends App, and saves us the Java-like syntax.
- Using the `@main` annotation.

5.1.1 USING A MAIN OBJECT WITH JAVA-LIKE SYNTAX

This method includes a `Main` object along with a `main` method sticking to Java syntax:

CODE

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

If we compare this with a Java main Class & method, we can see it's very similar:

CODE

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

This way of declaring `Main` objects is too verbose, defeating one of Scala's purposes compared to Java: Creating a more concise language.

5.1.2 USING A MAIN OBJECT EXTENDING APP

This is a more concise version of the above since we can effectively save the Java-like syntactic verbosity, but we still declare a `Main` object:

```scala
object Main extends App {
  println("Hello, world!")
}
```

As we'll see later on, the `extends` keyword extends the behavior of `A` with `B`, in this case, `App` with `Main`; by extending the `App` trait, we automatically get a `main` method for our application, so we don't need to define it ourselves. We can write our application's code directly within the `Main` object body, and the `App` trait executes it.

One thing to note is that when using `object Main extends App`, our application's code will be executed when the `Main` object is run. This approach was commonly used in Scala 2, but it's worth noting that the `App` trait relies on the `DelayedInit` feature, which is deprecated in Scala 3. For new Scala 3 projects, it is recommended to use the `@main` annotation to define a main method or implement a `main` method within an object, as these approaches are more concise and do not rely on deprecated features.

5.1.3 USING THE @MAIN ANNOTATION

This is the more concise method since we don't even need to define a Main object or method explicitly:

CODE

```scala
@main def hello(): Unit = {
  println("Hello Scala")
}
```

When we annotate a method with `@main`, the Scala 3 compiler automatically generates a wrapper object with a `main` method that calls our annotated method. This wrapper object serves as the entry point for our application and does not require any explicit `main` method declaration.

If we're wondering which one to use, it's really up to us to decide. However, the preferred method is the last one since it's more concise and extremely easy to read.

6. Opening our project in VS Code

We will need to open our project in the root directory. This directory can be found by locating the `build.sbt` file, which will be the main build file for our project. We can then simply open VS Code and select import build:
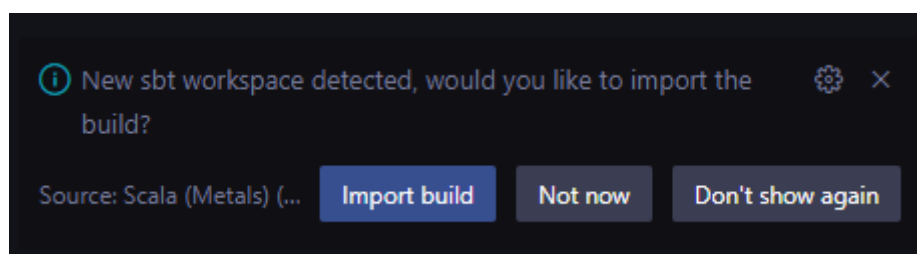

FIGURE 1: IMPORT BUILD PROMPT BY METALS

When we import the main build, some additional folders will be created in our root directory:

```
scala-for-beginners/
|
├── .bloop/
├── .bsp/
├── .metals/
└── plugins.sbt
```

We'll not be working directly with these folders. However, it's important to know they belong to the Metals extension and the `sbt` compiler.

7. Compiling and running a Scala project

If we have our `.scala` file, we can compile it and run it using sbt:

1. Go to our project's main directory.
2. Initialize an `sbt` prompt.
3. Execute the `run` command.

This command will run only if a main class is associated with our project. Else, it will throw an error.

We can try this command on our `Main.scala` predefined template, which should look like such:

CODE

```
@main def hello: Unit =
  println("Hello world!")
  println(msg)

def msg = "I was compiled by Scala 3. :)"
```

CODE

```
run
```

OUTPUT

```
[info] running hello
Hello world!
I was compiled by Scala 3. :)
[success] Total time: 0 s, completed May 12, 2023, 5:34:12 PM
```

This should run our main source file, and create a new directory under our project's root folder:

```
scala-for-beginners/
|
└── target/
```

The `target/` folder contains all the information produced by the compiler on our first compilation. This includes the Java Bytecode files, which are under `scala-for-beginners\target\scala-3.2.2\classes`.

Files with the `.class` extension are Java Bytecode files that can directly run on the JVM.

8. Working with the SBT shell

The build tool we installed earlier (`sbt`) provides a shell we can access to perform several operations on our Scala project, such as compiling, executing, testing, debugging, and more. We can access the `sbt` shell whenever we're under an existing `sbt` project (*that is, under the root folder of our project*).

An `sbt` instance can be executed by using the following command:

CODE

```
sbt
```

OUTPUT

```
[info] welcome to sbt 1.8.2 (Oracle Corporation Java 11)
[info] loading project definition from C:\Users\Pablo\OneDrive\Documents\Blog\computer-
science\scala-3-for-beginners\scala-for-beginners\project
[info] loading settings for project root from build.sbt ...
[info] set current project to scala-for-beginners (in build
file:/C:/Users/Pablo/OneDrive/Documents/Blog/computer-science/scala-3-for-beginners/scala-for-
beginners/)
[info] sbt server started at local:sbt-server-6ea3b4cc30af6310f6a2
[info] started sbt server
sbt:scala-for-beginners>
```

Any command we execute inside of this shell will be in reference to our newly-created project.

There are two main commands we'll be using extensively throughout this segment:

- `run` : Compiles and runs a main class, passing along arguments provided on the command line.
- `test` : Executes all tests.

If we have questions about any command in sbt, we can execute the following command:

CODE

```
help <command_name>
```

For example:

CODE

```
help project
```

OUTPUT

```
project

        Displays the name of the current project.

project name

        Changes to the project with the provided name.
        This command fails if there is no project with the given name.

project {uri}

        Changes to the root project in the build defined by `uri`.
        `uri` must have already been declared as part of the build, such as with
Project.dependsOn.

project {uri}name

        Changes to the project `name` in the build defined by `uri`.
        `uri` must have already been declared as part of the build, such as with
Project.dependsOn.

project /

        Changes to the initial project.

project ..

        Changes to the parent project of the current project.
        If there is no parent project, the current project is unchanged.

        Use n+1 dots to change to the nth parent.
        For example, 'project ....' is equivalent to three consecutive 'project ..' commands.
```

We can also consult the full `sbt` documentation here.

---

§

Basic syntax

Scala syntax is similar to Java's. However, it's considerably less verbose, more compact & concise, & more expressive. Especially Scala 3 possesses several syntax improvements over its predecessor, Scala 2.

1. Commenting

There are two ways we can use to comment in Scala:

- Single-line commenting
- Multi-line commenting

1.1 Single-line commenting

Single-line commenting is done via double forward slashes `//` :

CODE

```
// Single commenting
```

1.2 Multi-line commenting

Multi-line commenting is achieved by enclosing our comment in single forward slashes `/` and asterisks `*` :

CODE

```
/*
Multi-line
commenting
*/
```

2. Immutable variables

Variables in Scala can be mutable or immutable. Upon starting to write Scala code, we might be tempted to use mutable variables since this might seem "easier" at first glance but is not recommended in functional programming, and we'll see why in more detail. For now, we'll focus exclusively on immutable variables and only briefly mention mutable variables.

There are two main ways we can declare immutable variables in Scala 3, depending on the evaluation strategy we're after:

- Using `def` : Lazy evaluation (*this is not exactly a variable. Instead, it's a function declaration, but it can be used to declare variables also*).
- Using `val` : Eager evaluation.

2.1 Lazy evaluation

Lazy evaluation is an evaluation strategy that delays the evaluation of an expression until its value is needed. This means we can declare a variable or function, and the compiler will not evaluate until explicitly called.

Lazy evaluation in Scala for variables can be achieved by using the `def` keyword:

CODE

```
def mylazyvar = 21
```

We might recall that the keyword `def` is sometimes used to declare functions in other languages, and that also applies to Scala; we're essentially defining a method with no parameters and a returning value.

2.2 Eager evaluation

Eager evaluation is the opposite of lazy evaluation; it evaluates the expressions upon declaration. We can achieve this in Scala by using the `val` keyword.

We can declare a variable using the following syntax:

```
// Eager evaluation using val
val myeagerval = 31
```

3. Mutable variables

We mentioned that declaring mutable variables in Scala and functional languages is not a good idea. This is because of multiple reasons:

- **Predictability and Understanding**: When we use immutable data, it's easier to reason about our code because we don't have to worry about data changing unexpectedly. Once a `val` is set, we know it will stay the same.
- **Concurrency and Parallelism**: Immutable data is inherently thread-safe, meaning it can be used across multiple threads without synchronization. This is a big advantage when dealing with concurrent or parallel processing because it avoids issues with data inconsistency and race conditions.
- **Purity:** Functions in functional programming are usually pure, meaning they don't have side effects, and their output depends solely on their input. This is easier to achieve with immutable data.

In short, working with immutable variables results in safer and more predictable behavior upon our application's execution.

However, there are instances when we might want to declare mutable variables, for example, when dealing with iterators that include an accumulator variable or when performance is a concern and creating new immutable objects would be costly regarding memory or processing time.

We use the `var` keyword to declare a mutable variable:

CODE

```
// Eager evaluation using var
var myeagervar = 32

// Reassign variable
myeagervar = 33

// Print final evaluation
println(myeagervar)
```

OUTPUT

```
// 33
```

As with `val`, declarations with `var` are eager, meaning they will be evaluated when declared.

4. Printing & strings

Printing in Scala can be achieved using two main methods:

- `println()`

- `print()`

The first one is short for "*print line*" and adds a new line character at the end of each print statement, while the second one does not:

CODE

```scala
// Declaring a simple variable
val myString = "Howdy hey"

// Using println
println(myString)

// Using print
print(myString)
```

Since we're working inside a workbook, we'll be unable to see the difference between the two, but when compiling, the first adds the new line, while the second does not. This means that if we insert two `println()` statements in consecutive order, they will be printed in new lines, while two `print()` statements will print the content in the same line:

OUTPUT

```
Howdy hey
Howdy hey


Howdy heyHowdy hey
```

Printing variables to `stdout` is often accompanied by a combination of strings and variables. Scala provides a handy mechanism called string interpolation, which allows us to mix both types, similar to what f-strings in Python would do:

CODE

```scala
// Declaring a simple variable
val myImportantVal = 1000

// Print using string interpolation
println(s"This is one very important value: ${myImportantVal}")
```

OUTPUT

```
// This is one very important value: 1000
```

5. Optional braces & semicolons

Scala 3 has brought a really useful improvement to the table: The capacity to declare anything without the use of two components:

- Curly braces

- Semicolons

This is a relevant improvement since it makes the code clearer and easier to read; we rely on the indentation to provide structure to our declarations. However, this is optional; thus we can or cannot omit both components, and the result will be the same:

CODE

```
// Declaring a function including braces & semicolons
def myFun1 = {
    7 * 7;
}

// Declaring a function omitting braces & semicolons
def myFun2 =
    7 * 7
```

We'll see function declarations in a second, but the relevant thing here is to illustrate that both options are plausible; it simply depends on personal preference.

6. Indentation

Like its predecessor Scala 2, Scala 3 does not rely on indentation as part of its syntax. However, since curly brackets are not optional in Scala 3, indentation provides a visual guideline on variable & function declarations (*blocks in general*).

Because of this, we can declare a function without indenting:

CODE

```
def myUnindentedFun(a: Int): Unit =
println(a)

def myIndentedFun(a: Int): Unit =
    println(a)

myIndentedFun(7)
myUnindentedFun(7)
```

As expected, both will produce the same result:

OUTPUT

```
// 7
// 7
```

This also applies to other definitions when when we're using curly brackets `{}` to denote blocks:

CODE

```
val myUnindentedVar: Int =
{
val a = 1
a
}

val myIndentedVar: Int =
    {
        val a = 1
        a
    }
```

Even though the first approach is awful, it works as expected:

OUTPUT

```
myIndentedVar: Int = 1
myUnindentedVar: Int = 1
```

7. Functions

Functions are a core part of Scala's architecture; they let us abstract methods in multiple ways. Functions possess a wide variety of properties and can be often declared using combinations of syntactic sugar components.

A function in Scala is defined using the keyword `def`

7.1 Functions without arguments

As we've already seen, functions in Scala can be declared without any argument:

CODE

```
// Declaring a function without arguments
def simpleFun = 7 * 7
```

This function simply evaluates the multiplication operation and returns its value.

7.2 Functions with arguments

The more common use-case of functions in Scala is by including arguments that can be evaluated:

CODE

```
// Declaring a function with arguments
def moreElaborateFun(x: Int, y: Int) = x * y
```

However, in this case, we do need to include types for our arguments. Else, the compiler will return an error (*we'll see type definition in a second*).

Functions can be declared using one-line expressions or can also be continued in the next lines:

```
def multilineFunction(x: Int, y: Int) =
    val myvar_1 = 14
    val myvar_2 = 7
    x + y + (myvar_1 * myvar_2)
```

We can then call our function:

```
multilineFunction(5, 5)
```

And it will return the expected result:

```
res0: Int = 108
```

7.3 Anonymous functions

It might be of no surprise that Scala supports and encourages anonymous functions, also called literal functions, if they're required or could simplify our code.

Anonymous functions can be declared using the following syntax:

```
val myNewInt = (a: Int, b: Int) => a * b
```

Here, we're declaring a new variable, `myNewInt`, that will hold the resulting evaluation of our anonymous function, `(a: Int, b: Int) => a * b`.

The difference with a named function is that in the latter, we declare a function explicitly and separately from our variable assignment, then evaluate and assign.

In the case of an anonymous function, we do both evaluations in one single step. So when we call our variable with the two parameters, `a` and `b`, we evaluate the anonymous function and assign it:

```
println(myNewInt(5, 7))
```

```
// 35
```

Of course, here we're doing a fairly simple product operation, but anonymous functions can be extremely handy when we want to apply more complex operations to variables.

8. Blocks

Blocks in Scala are a scope-delimitation mechanism to group multiple expressions or statements together. They are declared using curly brackets `{}` and serve several purposes, including scoping and variable visibility; when we're defining variables, the Scala compiler provides a dedicated namespace.

If we recall, variables in Scala are immutable, meaning we need to get creative with naming conventions if we're working with extensive programs. What blocks allow us to do, is to create a restricted namespace, where we can use variable names that were used outside the block without any problem:

CODE

```
{
    val inside_1 = 14
    val inside_2 = 21
}

// Try to reference inside variable outside block
inside_2
```

OUTPUT

```
Not found: inside_2
```

This error is thrown since the variable is not in the global scope where it's being referenced.

Blocks are not common as standalone elements but are usually employed in variable or function definitions:

CODE

```
// Declare a variable using blocks
val my_extense_var = {
    val a = 5 * 4
    val b = 7 * 2
    b
}
```

Here, we're declaring an immutable variable using a block that contains a set of expressions, including two additional variables, `a`, `b`, and the eventual return of the last variable, `b`.

This might seem weird, but it's part of Scala's magic; we can define variables inside variables. What's cooler is that we don't even need to declare a `return` statement. In fact, Scala encourages not to use return statements since, by default, the last evaluated expression inside any block or function is returned.

9. Type declaration

Up until now, we have not included much data type declaration. But wait, didn't we say that Scala was a strong statically typed language? Yes, indeed, but the compiler is smart enough to infer the data types on compile time in some cases (*there are exceptions, for example, with lists or other more elaborate structures, where we have to explicitly declare the types of the variables inside our list*).

However, we're talking about a strong statically typed language here, so it's best practice to try to always explicitly declare the data types. This is because, unlike other languages like Python, Scala is strict in terms of which operations can be performed with which data types, and it might be easier if we have a clear sense of what data types we have before performing any operation on our variables.

Type declaration is performed using the colon symbol `:` , followed by the type we're using:

CODE

```scala
// Declaring a variable with its type
val myval_1: Int = 14

// Declaring a function with its type
def myfun_1: Int = 7 * 7

// Declaring a function with its type, including its arguments
def myfun_2(x: Int, y: Int): Int = x * y
```

---

§

Basic data types

Scala encourages type-driven development simply because it possesses a rich type system that lets us perform multiple manipulations and even create our own very complex type hierarchies, similar to what we would do with classes & objects in an OOP approach (*Scala combines these two approaches seamlessly*).

1. An introduction to type-driven development

Having discussed data type declarations, it's helpful to consider the data types available in Scala. This is because Scala programming can and should utilize types and type hierarchies extensively, a practice often referred to as **type-driven development** (*TDD*). Scala's robust type system, while not strictly adhering to the Hindley–Milner (HM) system found in some purely functional languages, allows for powerful type inference.

The concept of type-drive development is sometimes hard to grasp since we might be accustomed to other styles, such as OOP. Let us try to explain this using two examples:

1. **Object-Oriented Programming (OOP)**: We're building a toy car using LEGO blocks. The car's wheels are designed to roll, the doors can open and close, and so on. The blocks (*objects*) have specific shapes and sizes, and each has its own role in the structure. In essence, each object carries both data (*the specifics of the block*) and behavior (*what the block does in context*).
2. **Type-Driven Development (TDD)**: We're putting together a jigsaw puzzle. Each piece (*type*) has a specific shape, pre-defined where and how it fits into the overall picture. Before adding a piece, we know exactly what shape we're looking for due to the constraints provided by the pieces we've already placed.

If this was still confusing, we might want to start cooking a pizza to better illustrate:

1. **Object-Oriented Programming (OOP)**: We're cooking freestyle. We have ingredients (*objects*) and know what to do with them (methods). We decide on the fly what to do next, depending on how the dish is coming along. We might add onions, then decide to season with some peperoncino or maybe add some garlic.
2. **Type-Driven Development (TDD)** We're now following a recipe. We have a list of ingredients (*types*) and specific steps (*functions*) outlined. We follow the recipe step by step. The recipe tells us when to chop the onions when to add them, and when to season with peperoncino.

In short, both cooking styles can result in a delicious pizza, but they use different approaches; with OOP, we have flexibility but might make mistakes or changes along the way. With TDD, we have a solid plan that ensures the dish turns out as expected, provided we follow the recipe correctly.

However, this doesn't mean that TDD is rigid (*in the context of limited or not flexible*). On the contrary: The approach is flexible but under our own terms, which reduces the potential errors that might arise with the preparation of a pizza; it ensures that the pizza will turn out to be a pizza and not a suspicious blob of dubious precedence.

Of course, since we usually define the general rules in the first place, TDD requires technical knowledge of what can and cannot be done with certain types provided by the language, in this case, Scala. This knowledge is called type-system knowledge and is especially relevant in languages with rich type systems like Scala.

Now that we have more clarity about TDD, we can describe Scala's type system.

2. Scala's type system overview

Scala follows a hierarchical type system that contains its own defined types, as well as some inherited types from Java:
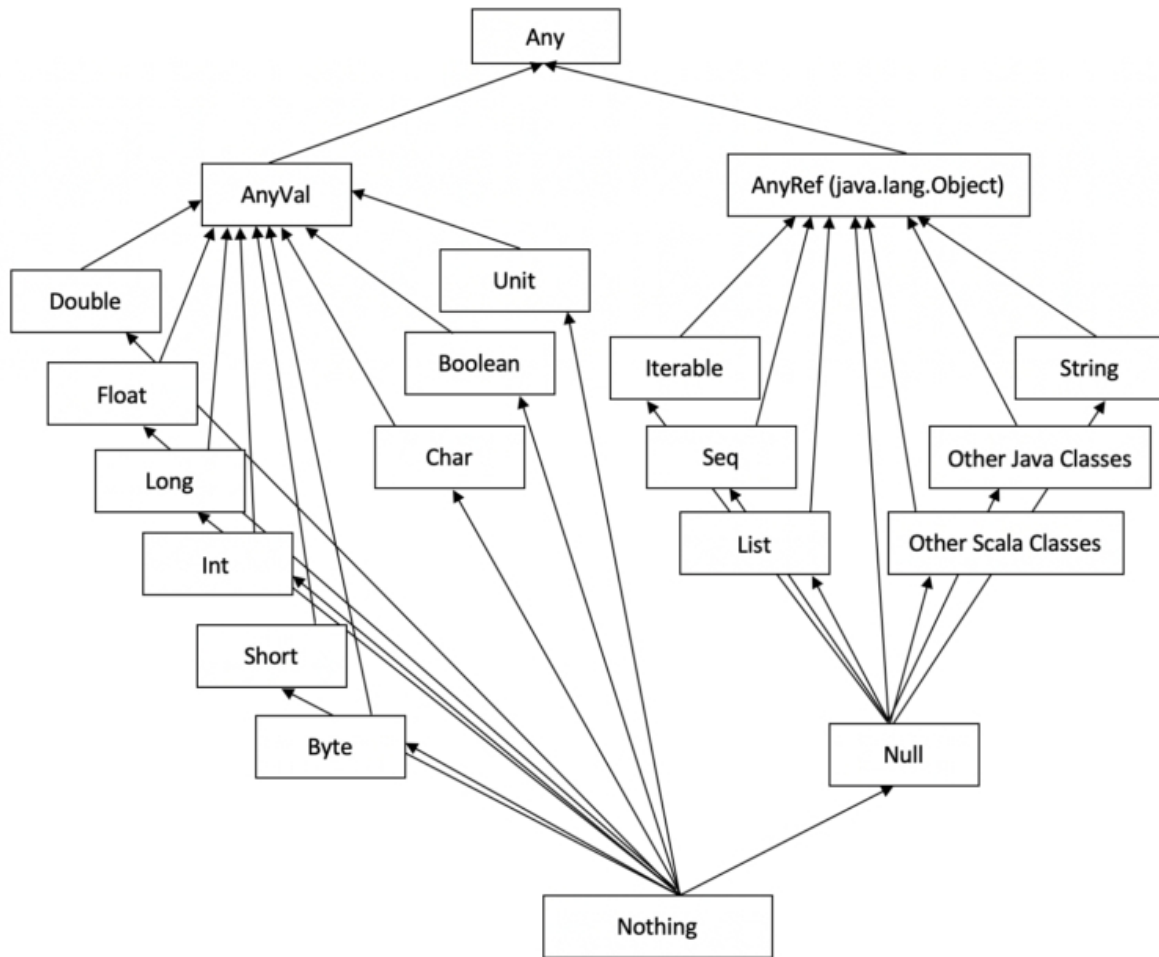
Let us break this hierarchy down into its individual components:

- `Any` : The supertype of all types, defining methods like `==` , `!=` , and `toString` .
  - `AnyVal` represents value types. These include `Double` , `Float` , `Long` , `Int` , `Short` , `Byte` , `Char` , `Unit` , and `Boolean` . They correspond to primitive types in Java and cannot be `null` .
    - `AnyRef` : Represents reference types. All non-value types are defined as reference types. This is equivalent to `java.lang.Object` .
- `Null` : A subtype of all `AnyRef` types, and its only instance is `null` .
- `Nothing` : A subtype of all other types, with no instances. It's used to denote abnormal termination and as a type for empty collections.

So, in short:

- `AnyVal` : Represents value types. They are lightweight, non-nullable, and usually have better performance characteristics.
- `AnyRef` : Represents reference types, meaning all types that are subclasses of `java.lang.Object` in Java. They are typically more complex, can be `null` , and include all user-defined and most built-in types.

Let us start from the more basic types and build from there.

3. Numeric types (AnyVal)

Scala provides six main numeric types, from `Byte` to `Double`. From bottom to top:

- `Byte`
- `Short`
- `Int`
- `Long`
- `Float`
- `Double`

3.1 Byte

A `Byte` in Scala represents an 8-bit signed integer. Its value range is **from -128 to 127**. We can declare a `Byte` variable using the following syntax:

CODE

```scala
val myByte: Byte = 127
```

Since we're dealing with a `Byte` type, we can perform arithmetic operations only with another `Byte`-type variable: Arithmetic operations in Scala return an `Int` value as a result. This is why these are not possible:

CODE

```scala
// Declare a Byte
val myByte: Byte = 127

// Impossible operations with Byte
val myByte_2: Byte = myByte - 7
val myByte_3: Byte = (myByte - 7.toByte)
```

OUTPUT

```
Found:    Int
Required: Bytemdoc
val myByte: Byte
```

While these are possible:

CODE

```scala
// Possible operations with Byte
val myByteToInt = myByte - 7
val myByte_2: Byte = (myByte - 7).toByte
```

OUTPUT

```
myByteToInt: Int = 120
myByte_2: Byte = 120
```

3.2 Short

A `Short` is a 16-bit signed integer data type. It has a value range from -32768 to 32767. We can declare a `Short` variable using the following syntax:

CODE

```
val myShort: Short = 32767
```

The same `Byte` rules apply to this data type.

3.3 Int

The `Int` type is probably the more well-known data type when referring to integer values. It represents a 32-bit signed integer data type. It has a value range from -2,147,483,648 to 2,147,483,647. The `Int` type is the default type when we're declaring an integer variable without its explicit type:

CODE

```
// Declare an Int with explicit type
val myIntExp: Int = 1

// Declare an Int with inferred type
val myIntInf = 1
```

Here, both declarations will result in the same data type:

OUTPUT

```
myIntExp: Int = 1
myIntInf: Int = 1
```

It's true that `Int` is the default integer value type in Scala, but it's not necessarily the most adequate when dealing with large numbers. If we want to define a number larger than what `Int` lets us do, we must use the `Long` type. This is true for all previous cases: We need to have clarity on what we're expecting from our variable since integer arithmetic can overflow and underflow without warning if the result of an operation exceeds the maximum or minimum value.

3.4 Long

The `Long` type is a 64-bit signed integer data type. It has a value range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. The `Long` type is used when the value to be stored exceeds the range provided by `Int`.

A Long value can be declared the same as with previous examples, but with the difference that we must append the `L` character (*for Long*) at the end of the number:

CODE

```
val myLong: Long = 9223372036854775807L
```

We append this `L` suffix because, by default, Scala treats integer literals as `Int`. The type annotation `: Long` tells Scala that the variable is of type `Long`, but it doesn't change how Scala interprets the literal number.

Consequently, this would result in an error:

CODE

```
val myLong: Long = 9223372036854775807
```

OUTPUT

```
integer number too large
```

3.5 Float

The `Float` type is a 32-bit floating point number. This means it can represent decimal numbers but with a limit to the amount of precision. It's sometimes referred to as a single-precision floating-point number in the IEEE 754 standard and has a precision of about seven decimal digits.

We can declare a `Float` variable as follows:

CODE

```
val myFloat: Float = 3.14159265358979323846264338327950288419
```

We can see that even though we include the complete decimal precision for π, we only get up to 7 decimal numbers in return:

OUTPUT

```
myFloat: Float = 3.1415927
```

We can also declare decimal numbers using scientific notation:

CODE

```
val myFloat_2: Float = 2.102030E1
```

Which will result in the following:

OUTPUT

```
myFloat_2: Float = 21.0203
```

If we'd like decimal numbers with increased precision, we can instead use a `Double` type.

3.6 Double

The `Double` type represents a 64-bit double-precision floating-point number. This type can be used to store decimal numbers with a larger range and precision than `Float`.

We can declare a Double type variable using the following syntax:

CODE

```
val myDouble: Double = 3.14159265358979323846264338327950288197
```

Which will result in the following:

OUTPUT

```
myDouble: Double = 3.141592653589793
```

As we can see, we get a precision of about 15 decimal numbers.

With this, we wrap up the numeric types under `AnyVal`. We can now proceed to non-numeric types.

4. Non-numeric types (AnyVal)

Non-numeric types under `AnyVal` in Scala include the following subclassification:

- Textual types
- `Boolean` type
- `Unit` type

In the case of `AnyVal`, the only textual type is Char, while the boolean type is the well-known `True` / `False` type.

4.1 Char

A `Char` is a 16-bit unsigned Unicode character type. It can represent any character in the Unicode character set, including ASCII and other characters from various languages worldwide. A 'Char' ranges from `U+0000` to `U+FFFF`.

Because a `Char` represents integer Unicode code point values, we can treat them as such, meaning we can perform arithmetic & relational comparison operations:

CODE

```
// Declare some chars
val myChar_1: Char = 'a'
val myChar_2: Char = 'b'
val myChar_3: Char = '1'

// Perform arithmetic operations with chars
myChar_1 > myChar_2
myChar_1 < myChar_2
myChar_1 >= myChar_2
myChar_1 <= myChar_2
myChar_1 != myChar_2
myChar_1 + myChar_2
myChar_1 - myChar_2
myChar_1 * 1
myChar_2 * 1
myChar_3 * 1
```

OUTPUT

```
// res5: Boolean = false
// res6: Boolean = true
// res7: Boolean = false
// res8: Boolean = true
// res9: Boolean = true
// res10: Int = 195
// res11: Int = -1
// res12: Int = 97
// res13: Int = 98
// res14: Int = 49
```

A `Char` is represented as a 16-bit unsigned integer in memory. This means it can take on a value between 0 and 65535. These numbers correspond to the Unicode values of the characters the `Char` can represent.

If we would like to convert a char to its unsigned integer type, we can do so:

CODE

```
// Convert Char to Int
myChar_1.toInt
```

OUTPUT

```
res15: Int = 97
```

4.2 Boolean type

The Boolean type in Scala, as in any other language, can take one of two possible values:

- `true`
- `false`

Keep in mind that in Scala, boolean values are declared in lowercase (*in other languages, this might change*):

CODE

```scala
val myBoolTrue: Boolean = true
val myBoolFalse: Boolean = false
```

4.2.1 LOGICAL OPERATORS IN BOOLEAN VARIABLES

We can use logical operators on boolean types:

CODE

```scala
// Boolean types
val myBoolTrue: Boolean = true
val myBoolFalse: Boolean = false

// AND Operator
myBoolTrue && myBoolTrue
myBoolTrue && myBoolFalse
myBoolFalse && myBoolTrue
myBoolFalse && myBoolFalse

// OR Operator
myBoolTrue || myBoolTrue
myBoolTrue || myBoolFalse
myBoolFalse || myBoolTrue
myBoolFalse || myBoolFalse
```

Which will result in the following:

```scala
// res16: Boolean = true
// res17: Boolean = false
// res18: Boolean = false
// res19: Boolean = false

// res20: Boolean = true
// res21: Boolean = true
// res22: Boolean = true
// res23: Boolean = false
```

4.3 Unit type

The `Unit` type is used when there is no meaningful value to return. It's similar to the `void` keyword in other languages like Java or C++.

For example, suppose we have a function that evaluates a certain expression and prints it out to `stdout` without returning the actual value as a result of the evaluation. In that case, we can use the `Unit` return type as follows:

```
def myFun(x: Int, y: Int): Unit =
    val a = x + y
    println(a)

myFun(1, 2)
```

```
// 3
```

5. Nothing type

`Nothing` is a subtype of every other type (*including* `Null`). It represents "*no value at all*". There are no values of type `Nothing`. It's used in Scala to signal abnormal termination. This is frequently used in methods with an abnormal exit, like throwing an exception.

We can define a function that throws an exception:

```
def myErrorFun(x: Int): Nothing = x match {
    case x:Int => throw new IllegalArgumentException(s"{s} is not of the correct type.")
}

myErrorFun(1)
```

```
java.lang.IllegalArgumentException: {s} is not of the correct type.
```

The only constraint is that this function will need to throw an exception since the type `Nothing` cannot be defined as a return type of other types (*e.g.,* `Int` , `Double` , *etc.*). In other words, we have to ensure that our function's return type is an exception or something returning `Nothing` . There are other ways to manage the return of two potential types; for example, if the exception is not met and we return an `Int` instead. We'll look at that later.

§

Control structures

Control structures are used to control the flow of execution of our program. Scala offers multiple ways to do this, but the two main ones are:

- Using the `if-else` construct.
- Using pattern matching.

The first option is more simple & straightforward but limited, while the second one is slightly more elaborate but much more powerful regarding reach & capabilities.

Let us look at both.

1. Conditionals

Conditionals in Scala are mostly used to evaluate simple single statements. They consist of a simple `if-else` structure and can be nested if we need more conditions, although this is discouraged in favor of pattern matching.

Conditional statements can be written using two different ways:

CODE

```
// Using no parenthesis and then
if a+b then c
else d

// Using parenthesis
if (a+b) c
else d
```

Let us declare a simple function that verifies if an integer value is greater than another number:

CODE

```
def getBiggest(x: Int, y: Int): Int =
    if (x >= y) x
    else y

getBigger(1, 2)
getBigger(4, 1)
```

OUTPUT

```
res25: Int = 2
res26: Int = 4
```

Multiple conditions can also be evaluated using two approaches:

- Logical operators
- Nested `if-else` constructs

We can include more than one condition on a single `if` statement by using logical operators:

CODE

```scala
def checkInBetween(x: Int, y: Int, z: Int): Boolean =
    if ((y > x) & (y < z)) true
    else false

checkInBetween(2, 3, 4)
checkInBetween(3, 2, 4)
```

```
res27: Boolean = true
res28: Boolean = false
```

We can also nest multiple if-else statements for more elaborate tests:

```scala
def checkSmallest(x: Int, y: Int, z: Int): Int =
    if ((x <= y) & (x <= z)) x
    else if ((y <= x) & (y <= z)) y
    else z

checkSmallest(1, 2, 3)
checkSmallest(5, 2, 10)
checkSmallest(1, 1, 2)
```

```
res29: Int = 1
res30: Int = 2
res31: Int = 1
```

Here, we only have one nested statement, so it's fairly easy to read, but whenever we add more conditions, our code will get messier; the `if-else` construct was not designed for multiple tests. This is where pattern matching comes into play.

2. Pattern matching

Pattern matching might be one of the most powerful features in Scala; it lets us destructure and examine complex data types clearly and concisely, providing a mechanism to execute code conditionally based on the structure or values of the data. It also enables seamless matching of sequence types, with capabilities to inspect sequence length and element position.

These are just two more sophisticated use cases, but we can also match using simpler statements. We'll start with a base definition using a code block:

```scala
val match_using_anonymous = (x: Int) => x match {
    case 1 => s"${x} is equals to 1"
    case 2 => s"${x} is equals to 2"
    case _ => s"${x} is neither 1 nor 2"
}

match_using_anonymous(1)
match_using_anonymous(10)
```

Output

```
res32: String = 1 is equals to 1
res33: String = 10 is neither 1 nor 2
```

Where:

- _ is a wildcard that matches any other value.

But this is a useless example, right? Who would want to check if a value is 1 or 2 using a pattern-matching implementation? Too much hustle, which can be simplified using a conditional construct.

That is correct. However, pattern matching is clearly not restricted to comparing integer numbers as we just did. In fact, this construct is much more powerful.

Since pattern matching can be used to deconstruct types, we can use it to check if a given class is a subclass of another class by trying to feed an animal, only if it's a dog:

Code

```scala
// Define abstract Animal class with unimplemented methods
abstract class Animal:
    val name: String
    val age: Int
    val species: String
    def salutes: Unit
end Animal

// Define Human class extending Animal
class Human(val name: String, val age: Int) extends Animal:
    val species: String = "human"
    def salutes =
        println(s"Hi, my name is ${this.name}, I'm ${this.age} years of age, and I'm a proud
${this.species}.")
end Human

// Define Doggo class extending Animal
class Doggo(val name: String, val age: Int) extends Animal:
    val species: String = "doggo"
    def salutes =
        println(s"Woof woof!!, My name is ${this.name}, I'm ${this.age} years of age, and I'm a
happy big ${this.species}.")
end Doggo

// Define a weird Animal subclass
class WeirdGuy(val name: String, val age: Int) extends Animal:
    val species: String = "weird guy"
    def salutes =
        println(s"Howdy doody, partner! I'm ${this.name}, ${this.age} olde, and I'm somewhat of a
${this.species}. Why don't ya grab yerself a cold one and sit a spell, ya hear?")

// Instanciate
val Chad = new Human("Chad", 23)
Chad.salutes

val Napoleon = new Doggo("Napoleon", 5)
Napoleon.salutes

val Skeeter = new WeirdGuy("Skeeter", 20)
Skeeter.salutes

// Declare feeding function (is not a method, since it does not belong to Animal or any of its
subclasses)
def feedAnimal(subject: Animal): String = subject match
    case subject: Doggo => s"Here you go ${subject.name} big boy, take some bacon!"
    case subject: Human => s"Hey my friend ${subject.name}, that bacon's not for you. Hands off
please!"
    case _ => "I don't know who or what you are, kindly leave my house or I'll call the police,
my good sir."

// Feed the dog animals only
feedAnimal(Napoleon)
feedAnimal(Chad)
feedAnimal(Skeeter)
```

```
// Hi, my name is Chad, I'm 23 years of age, and I'm a proud human.
// Woof woof!!, My name is Napoleon, I'm 5 years of age, and I'm a happy big doggo.
// Howdy doody, partner! I'm Skeeter, 20 olde, and I'm somewhat of a weird guy. Why don't ya grab
yerself a cold one and sit a spell, ya hear?

res37: String = Here you go Napoleon big boy, take some bacon!
res38: String = Hey my friend Chad, that bacon's not for you. Hands off please!
res39: String = I don't know who or what you are, kindly leave my house or I'll call the police,
my good sir.
```

Let us explain what's happening in more detail:

- We first declare an `Animal` abstract class (we'll discuss them later in more detail) with some unimplemented methods.
- We then declare three subclasses: `Human`, `Doggo`, and `WeirdGuy`
- Next, we implement their methods and create new instances for each subclass.
- Finally, we declare a feeding function that accepts an `Animal` instance and prints something to `stdout`, depending on the type of the subclass we're including. Since abstract classes cannot be instantiated, our only options are:
    - To try to feed a `Doggo`.
    - To try to feed a `Human`.
    - To try to feed a `WeirdGuy`.

Since we as humans can usually eat our own food, at least when we're grown-ups, and it would not be the best idea to let a weird guy inside our house, we can assume that a doggo will be the only candidate to be fed.

We check if the `Animal` is one of the two and decide whether to feed it or not. Lastly, we leave any other options at the end, including a not-so-friendly message.

This is just a simple pattern-matching example, but a wide variety of operations can be done using the `match-case` construct.

3. Loops

Loops in functional languages are sometimes not implemented in the traditional sense, at least not exactly. One of the characteristics of the functional style is that it discourages the use of `for` loops. This is because:

- Loops can and usually will involve a mutation of some variable(s). This, in turn, can result in impure implementations with side effects associated.
- Scala provides more powerful and concise ways to loop over things without explicitly writing a `for` loop, or even worse, a set of nested loops that quickly becomes unreadable, confusing, and bug-prone.
- Functional programming is more declarative than imperative, focusing on the "*what*" rather than the "*how*". Using higher-order functions or recursion instead of `for` loops is more in line with this style, as it focuses on the result of the operation rather than the specific steps needed to achieve it.

In short, using loops in Scala is like building a house with the wrong, expensive tools; we're better off using Python since, in that case, it's "*cheaper*" to learn, if that makes sense.

We'll briefly mention loops to illustrate why they're discouraged and what functional approaches we can use instead.

3.1 while loops

A `while` loop loops through a block of code as long as a specified condition is true. This specified condition usually involves some kind of variable mutation since, otherwise, the condition is never met, and the loop goes forever and ever.

We can define a simple `while` loop that checks for even numbers when dividing an initial number by 2:

CODE

```
// Declare a simple integer mutable variable
var y: Int = 100

// Declare a while loop
while (y % 2 == 0)
    println(s"${y} is still even")
    y /= 2
```

OUTPUT

```
// 100 is still even
// 50 is still even
```

Now for the functional approach, we implement a tail-recursive function.

CODE

```
// Declare a functional-style equivalent
def checkEven(y: Int): Unit =
    if (y / 2 % 2 != 0) println(s"${y} is still even")
    else checkEven(y / 2)

checkEven(100)
```

OUTPUT

```
// 50 is still even
```

In this last approach, we don't care for the even intermediate numbers; we just care for the last even number when divided by 2.

At first, the recursive approach might seem more confusing, but we're saving ourselves a potential future headache: We don't mutate any variable; thus we don't need to track its development over time.

3.2 for loops

In contrast to `while` loops, `for` loops loop over their body a defined number of times. These are helpful when we have a known range of values to cover. A typical `for` loop can be declared using the following syntax:

CODE

```
// Declare a for loop
for (x <- 1 to 10)
    println(x)
```

```
// 1
// 2
// 3
// 4
// 5
// 6
// 7
// 8
// 9
// 10
```

Now for the functional-style equivalent:

```
// Declare a functional-style equivalent
def printInteger(x: Int): Unit =
    if (x == 10) println(x)
    else printInteger(x + 1)

printInteger(1)
```

```
// 10
```

3.3 for comprehensions

In contrast to the previous two approaches, `for` comprehensions are syntactic constructs to build a new collection by applying some operations to each element of an existing collection.

We might already know comprehensions from other languages, such as Python:

```
# Comprehensions in Python

# List comprehension
mylist = [1, 2, 3, 4, 5, 6, 7, 8, 9]
mylist_squared = [x**2 for x in mylist]

print(mylist_squared)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In Scala, things are similar:

CODE

```
// Declare a list
val mynumbers: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

// Apply squared and create new list
val mynumbers_squared: List[Int] = for (n <- mynumbers) yield n * n
```

OUTPUT

```
mynumbers_squared: List[Int] = List(1, 4, 9, 16, 25, 36, 49, 64, 81)
```

In this example, `n <- numbers` is a generator that generates a new `n` for each element in `numbers`. `yield n * n` then creates a new element for the new list.

As with list comprehensions in Python, we can also add conditions to `for` comprehensions in Scala:

CODE

```
// for comprehensions with conditions
val numbers: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

// Apply modulus and filter by odd & even numbers
val numbers_odd: List[Int] = for (n <- numbers if n % 2 != 0) yield n
val numbers_even: List[Int] = for (n <- numbers if n % 2 == 0) yield n
```

OUTPUT

```
numbers_odd: List[Int] = List(1, 3, 5, 7, 9)
numbers_even: List[Int] = List(2, 4, 6, 8)
```

4. Exception handling

Exception handling in Scala is similar to that in Java and other languages, with `try`, `catch`, and `finally` blocks used to handle exceptions.

We can declare a simple exception-handling implementation by using the following syntax:

CODE

```scala
def parseStringToInt(x: String): Unit =
    try
        val parsed: Int = x.toInt
        println(parsed)
    catch
        case e: NumberFormatException => println(s"Couldn't parse '${x}' to an integer.")

parseStringToInt("7")
parseStringToInt("a")
```

OUTPUT

```
// 7
// Couldn't parse 'a' to an integer
```

There are other methods we can use to catch exceptions:

- Using the `scala.util.Try` method.
- Using `Option`
- Using `Either`

However, we'll not review those in this segment.

---

§

Collections

Collections in Scala are simply containers of things. They can be sequenced, linear sets of items like `List`, `Tuple`, `Option`, `Map`, `Set`, etc.

1. List

A `List` is an ordered collection of elements. It's a linear sequence that can contain duplicate elements. Elements in a list can be accessed using an index.

CODE

```scala
val myIntList: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
val myStringList: List[String] = List("a", "a", "b", "b", "c", "c")
val myFloatList: List[Float] = List(1f/2f, 1f/4f, 1f/8f)
val myMixedList: List[Matchable] = List(1, "2", 3.0)
```

OUTPUT

```
myIntList: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
myStringList: List[String] = List(a, a, b, b, c, c)
myFloatList: List[Float] = List(0.5, 0.25, 0.125)
myMixedList: List[Matchable] = List(1, 2, 3.0)
```

As we might have noticed, we had to declare the float list using at least one of the operands in each division: a `Float` or `Double`. This is because Scala handles integer division by discarding the remainder and keeping only the integer part of the result. In other words, if we'd like to perform a float division, we must provide at least one float (*numerator or denominator*).

Another interesting detail is that we use the `Matchable` type to be able to declare a list of mixed types. In general, it's not a good practice to create lists with mixed types in Scala (or in most other statically-typed languages) because it reduces the type safety that Scala's static typing provides. However, it's possible to do, and the code executes without errors.

2. Array

An `Array` is a mutable, indexed collection of elements. Arrays are fixed-size, and their length is unchangeable. Unlike Lists, we can change the value of elements in an array after it's been created.

One thing to note about arrays is that by default, they get printed using the `toString` representation, which is not very meaningful (*it shows the type and hashcode of the array, like `[I@77e8d3b1`*).

To print the contents of the array in a more readable way, we can use two different methods:

- `mkString`
- `foreach`

CODE

```
// Declare array
val myArray: Array[Int] = Array(1, 2, 3, 4, 5)

// Print string representation
print(myArray)

// Change array value using index (zero-based indexing)
myArray(0) = 0

// Print array elements using foreach
myArray.foreach(println)
```

OUTPUT

```
// [I@3112137

// 0
// 2
// 3
// 4
// 5
```

3. Set

A `Set` is a collection of distinct elements. Elements in a set are not ordered. Scala provides both mutable and immutable `Set`.

```scala
val mySet: Set[Int] = Set(1, 2, 3, 4, 5)
println(mySet)
```

```
// HashSet(5, 1, 2, 3, 4)
```

But wait a second...why is the `Set` printed in a different order than how it was originally declared? What happens is that, as mentioned, the `Set` collection in Scala does not maintain the order of elements. This is because `Set` is implemented as a `HashSet` by default, a data structure designed to optimize lookup times (*checking if an element is contained in the set*) rather than preserving insertion order.

In a `HashSet`, elements are organized based on their hash codes for quick lookup, which may not reflect the order in which they were added.

4. Map

A `Map` is a collection of key-value pairs. The keys in a map are unique. If we've worked with dictionaries in Python, this structure is what most resembles them:

```scala
// Declare Map of Int -> String
val myMapInt: Map[Int, String] = Map(1 -> "one", 2 -> "two", 3 -> "three")

// Index Map by Int key
myMapInt(1)

// Declare Map of String -> Int
val myMapString: Map[String, Int] = Map("one" -> 1, "two" -> 2, "three" -> 3)

// Index Map by String key
myMapString("two")
```

```
res57: String = one
res58: Int = 2
```

Keys can be of multiple types (`Int` , `String`); the only constraint is that they must be the same type and be unique values.

5. Tuple

A `Tuple` is an ordered group of elements. Unlike Lists and Arrays, Tuples are designed to hold elements of different types. They are immutable data structures useful when we want to return multiple values from a function.

We can declare tuples using three different ways:

```scala
// Declare a tuple using a single type
val myTupleSingle: Tuple = (1, 2, 3, 4, 5)

// Declare a tuple using multiple but explicitly declared types
val myTupleMulti: (String, Int, String, String, Int) = ("1", 2, "3", "4", 5)

// Declare a tuple using multiple inferred types
val myTupleMulti_2 = (1, "2", 3, "4", "5", '6', 7)
```

- In the first line, we declare a `Tuple` containing one data type (`Int`), but without explicitly declaring the types.
- In the second line, we declare a `Tuple` containing multiple data types but explicitly declaring the types of each value.
- In the last line, we declare a `Tuple` containing multiple data types, but without explicitly declaring the types.

6. Option

An `Option` is a more advanced collection representing a value that may or may not exist. An `Option` can be either `Some` value or `None` .

Let us imagine we have a `Map` structure mapping names with ages and would like to retrieve a value given a key. In this case, a key might not exist in the `Map` declaration, so we can use an `Option` to ensure we don't get errors when trying to index a non-existing key:

CODE

```scala
val personAge = Map("John" -> 30, "Amy" -> 25)

def getAge(name: String): Option[Int] = {
    personAge.get(name)
}

val johnAge = getAge("John")
val amyAge = getAge("Amy")
val unknownAge = getAge("Unknown")
```

OUTPUT

```
johnAge: Option[Int] = Some(30)
amyAge: Option[Int] = Some(25)
unknownAge: Option[Int] = None
```

What's happening here is:

- We first declare a `Map` with unique keys and values.
- We then declare a function called `getAge` , with the evaluation result as `Option` (*it might or might not exist, it might or might not be an* `Int`).

- Next, we index using some existing and non-existing keys.
- In the first two cases, we get `Some` as the output type, meaning it exists.
- In the last case, we get `None`, which means that no key was found; thus the corresponding `Age` integer value does not exist.

This is extremely useful because now, we can use pattern matching to make more sense of the data we're getting:

CODE

```scala
def checkAge(age: Option[Int]): Unit = age match
    case Some(age) => println(s"The age is: ${age}")
    case None => println(s"Sorry! Person does not exist.")

checkAge(johnAge)
checkAge(unknownAge)
```

OUTPUT

```
// The age is: 30
// Sorry! Person does not exist.
```

7. Vector

A `Vector` is a general-purpose, immutable data structure. It provides random access and updates in constant time, providing very fast append and prepend mechanisms.

Vectors support elements of the same type:

CODE

```scala
val myVector: Vector[Int] = Vector(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

§

Higher-order functions

Higher-order functions are more of an advanced topic, but they're worth mentioning since they're a key aspect of Scala & functional programming in general.

Higher-order functions are functions that:

- Accept functions as parameters
- Can return functions as results

This is somewhat confusing, so let us explain it with a simple example:

Suppose we want to implement a method that sums all the items in a given list while performing some intermediate operation beforehand. This is called a variation of the summation operation and is very common in mathematics.

For the product variation, the mathematical expression would look something like such:

$$\sum_{i=1}^{n} i^2$$

Normally, we would use one single function definition to approach this. However, one key aspect in functional programming is modularizing our functions, meaning one function ideally performs one single operation, and no more.

Thus, we can create two separate definitions that will complement each other:

CODE

```
// Defining a list of values to operate on
val myTargetList: List[Int] = List(1, 2, 3, 4)

// Define the summation function
def sumList(l: List[Int], f: Int => Int): Int =
    def sumNumbers(h: Int, t: List[Int], f: Int => Int): Int =
        if (t.isEmpty) f(h)
        else f(h) + sumNumbers(t.head, t.tail, f)
    sumNumbers(l.head, l.tail, f)

def intSquared(x: Int): Int =
    x*x

// Call function
sumList(myTargetList, intSquared)
```

Let us break this step by step:

- We first declare a simple target `List` containing integer values.
- We then define a `sumList` function, which accepts the following:
    - A list of integers `l`.
    - A function `f`, which accepts an `Int` and returns an `Int`.
- We then define another nested function. This one is declared because we need to keep track of the summation somehow; hence we include a head (`h`) and a tail (`t`) parameters, along with the function `f`.
- Next, we check if the tail `t` is empty (*empty list*). If it is, we return the result of the function `f` applied to the head `h` (*in this case, the last element of the list*)
- Else, if the list is not empty, we apply the function `f` to `h` and add that to a recursive call of `sumNumbers`, changing the head and tail values.
- Next, we make sure to call the `sumNumbers` function as initialization.
- Finally, we declare an `intSquared` function, which accepts an `Int` and returns the squared value as `Int`. We call our function and get what we expect.

OUTPUT

```
res61: Int = 30
```

This is confusing at first, but it's extremely powerful when getting used to it: We can compose functions in terms of other functions while explicitly declaring the expected types in every step. This is not only excellent for abstraction but is also safe and pure.

The nice thing about higher-order functions is that Scala already implements some extremely useful ones:

- `map` : We use it to iterate over a collection and then apply a function to each collection element.
- `flatMap` :
- `filter` :
- `reduce` :

1. map

Remember we declared a function that went over all elements of a list recursively and applied some operation, such as squaring each element? Well, `map` does exactly that, only we can implement it in a single line of code:

CODE

```scala
// Declare a simple list of integer values
val myList: List[Int] = List(1, 2, 3, 4, 5)

// Declare a squared method
def squaredInts(x: Int): Int =
    x*x

// Use Map on list
myList.map(squaredInts)

// Perform the same operation using an anonymous function
myList.map((x: Int) => x * x)
```

In both cases, the result will be the same, the only difference is that, in the latter, we'll write our entire declaration in a single line of code by leveraging anonymous functions:

OUTPUT

```
res62: List[Int] = List(1, 4, 9, 16, 25)
res63: List[Int] = List(1, 4, 9, 16, 25)
```

Suppose we want to multiply our elements by some scalar instead of squaring them. In that case, we can even simplify this even more by using a placeholder instead of the full anonymous function specification:

CODE

```scala
// Perform the same operation using a placeholder
myList.map(_ * 2)
```

Here, we are doubling each element:

OUTPUT

```
res64: List[Int] = List(2, 4, 6, 8, 10)
```

These are, of course, extremely simple operations. Still, we can imagine that we could construct very complex functions and apply them to a list of values other than integers (*even nested lists or other collections*).

For example, we can define a list of lists where we want to calculate the average of each list and return it as an `Int` value:

CODE

```
// Define a list of lists
val listOfLists: List[List[Float]] = List(
                              List(1, 2, 3, 4, 5),
                              List(6, 7, 8, 9, 10),
                              List(11, 12, 13, 14, 15)
                              )

// Declare single list average function
def averageOfLists(x: List[Float]): Float =
    val listLen = x.length
    def calculateAverage(h: Float, t: List[Float], cum: Float): Float =
        if (t.isEmpty) (cum + h)/listLen
        else calculateAverage(t.head, t.tail, h + cum)
    calculateAverage(x.head, x.tail, 0)

// Apply average function to list of lists using map
listOfLists.map(averageOfLists)
```

And the output will be a list of `Float` values containing the average values of each of the original nested lists:

OUTPUT

```
res65: List[Float] = List(3.0, 8.0, 13.0)
```

Map functions can also be chained, meaning we can apply a sequence of transformations using one expression:

CODE

```
listOfLists.flatten.map(x => x + 1).map(x => x * 2)
```

So, what are we doing here?

- We first flatten our list of lists, meaning we transform a list of lists into a list with all the nested list elements included.
- We then apply a simple operation, shifting each element by 1.
- We finally apply a third function that multiplies each shifted element by 2.

2. flatMap

We already saw what `map` & `flatmap` do separately. The `flatMap` method combines these two implementations into one convenient definition. So, in essence, `flatMap`:

- Applies a given function to each nested element inside the outer collection.

- Flattens the operated collection.

Let us define an example where we again have a list of lists and would like to come up with a single list of integers, where each integer inside the nested lists is multiplied y some constant:

CODE

```scala
// Define a collection of words
val myWordCollection: List[String] = List("Hello", "there", "my", "dear")

// Map & flatten
myWordCollection.flatMap(x => x.toList)
```

OUTPUT

```
res67: List[Char] = List(H, e, l, l, o, t, h, e, r, e, m, y, d, e, a, r)
```

If we recall our average example, we can create a more concise version using `flatMap` :

CODE

```scala
// Define a list of lists
val listOfLists_2: List[List[Float]] = List(
                                    List(1, 2, 3, 4, 5),
                                    List(6, 7, 8, 9, 10),
                                    List(11, 12, 13, 14, 15)
                                    )

// Calculate averages for each nested list
listOfLists_2.flatMap(x => List(x.sum / x.length))
```

OUTPUT

```
res68: List[Float] = List(3.0, 8.0, 13.0)
```

3. filter

The `filter` method does exactly that: It filters based on a given predicate (condition) that we specify to the method as a parameter. The only condition is that the predicate must be an expression that evaluates to a `Boolean` value.

For example, if we want to filter only even numbers, we can do so using the following syntax:

CODE

```
// Define a new list
val myRawList: List[Int] = List.range(1, 100)

// Filter even numbers
val myFilteredList: List[Int] = myRawList.filter(x => x % 2 == 0)
```

```
myFilteredList: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34,
36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82,
84, 86, 88, 90, 92, 94, 96, 98)
```

We can use any predicate and even include it as a more complex function, as we've done with the other methods. The bottom line is we only need an expression that checks if each of the values inside the collection is `true` or `false`; then, the true values are filtered out and returned as a new collection.

4. reduce

The `reduce` method belongs to a broader collection of methods:

- `reduce`
  - left
  - right
- `fold`
  - left
  - right
- `scan`
  - left
  - right

These methods are used to "*compress*" a collection by applying an operation called `op`. This "compression" simply means that we're squeezing our collection left or right, depending on the implementation, while applying an operation, which can be an addition, product, division, etc.

The left or right direction tells how we're squeezing our collection. For example, if we want to create a word from a set of string characters, we can do so by applying a `reduceLeft` method:

CODE

```
// Define a new list
val myAlphabetList: List[String] = List("a", "b", "c", "d")

// Reduce left
myAlphabetList.reduceLeft(_ + _)
```

OUTPUT

```
res69: String = abcd
```

If we again recall our average example, we can use `reduceLeft` to substitute a `sum` method:

CODE

```
// Declare a list of lists
val listOfLists_3: List[List[Float]] = List(
                            List(1, 2, 3, 4, 5),
                            List(6, 7, 8, 9, 10),
                            List(11, 12, 13, 14, 15)
                            )

// Calculate averages by reducing
listOfLists_3.map(x => x.reduceLeft(_ + _) / x.length)
```

Which will give us the expected result:

OUTPUT

```
res71: List[Float] = List(3.0, 8.0, 13.0)
```

What if we want to reverse our list of strings? Well, if we first reduce left and invert the order sequentially, we'll come up with an inverse concatenated version of our original list, which we can simply split into a new list:

CODE

```
// Define a new list
val myAlphabetList: List[String] = List("a", "b", "c", "d")

// Invert a list
myAlphabetList.reduceLeft((x,y) => y+x).toList
```

OUTPUT

```
res71: List[Char] = List(d, c, b, a)
```

This one is extremely interesting since we're taking two values for each pass and pasting `y` & `x` in the inverse order from their original sequence.

This is very useful since we're effectively saving ourselves the classic but painful loop implementation that takes `i` & `i + 1` and operates on those values; one single line of code can perform what we otherwise would have done in multiple lines.

This is the beauty of higher-order functions & declarative-style programming, and as we can imagine, they're extremely powerful and heavily encouraged in functional languages; the creators invested time & brains in these implementations, which we can simply use without asking questions; they just work.

§

Recursion

We already had a taste of what recursion is in previous examples. In fact, many of the implemented methods we used in higher-order functions are recursive in nature. Recursion is one of the most used features in functional programming because it leverages immutability to produce the same results that we would get from normal `for` -loop constructs.

As with many concepts in Scala, this one takes some time to get used to, but if used wisely, it can simplify our code by potentially thousands of lines while keeping our immutability condition intact.

Recursion is straightforward: Define a function, and call it from within while changing each call throughout. Recursive implementations have three main components:

1. **Base Case(s):** The simplest instance of the problem that can be solved directly. This is the condition under which the recursion stops. A recursive function must have at least one base case, but it could have multiple. The base case helps prevent the function from calling itself indefinitely.
2. **Recursive Case(s):** The part of the function that solves the problem by calling the function itself on a smaller or simpler instance of the problem.
3. **Progress:** Each recursive call should move closer to a base case to ensure the recursion eventually stops. This typically involves manipulating the input of the function so that it gets closer to the base case with each recursive call.

Let us explain this with an example: we have two integer numbers, `a` and `b` , and would like to sum everything in between. We can define our components previously:

- **Base case:** `a` is greater than `b` (*recursive call terminates*)
- **Recursive case:** `a` is less than `b` .
- **Progress:** `a` keeps incrementing while the base case is unmet.

CODE

```
def mySumFun(a: Int, b: Int): Int =
    def accumulatorFun(a: Int, b: Int, acc: Int): Int =
        if (a > b) acc
        else accumulatorFun(a + 1, b, acc + a)
    accumulatorFun(a, b, 0)

mySumFun(1, 10)
```

OUTPUT

```
res62: Int = 55
```

This type of recursive implementation (*at least the* `accumulatorFun` *implementation*) is called a tail-recursive implementation because it includes the recursive step as the last step and nothing more. The compiler can optimize these types of implementations and result in lower execution speeds and less stack memory used since we're not storing any intermediate values (*i.e., the last call,* `accumulatorFun(a + 1, b, acc + a)` *, is just that, and nothing more, so it gets evaluated without the need to store and "recall" previous values*)

Recursion is a vast topic and can get very confusing when dealing with more complex cases. However, in this segment, we'll stick with the generalities.

---
§

Next steps

Scala is feature-packed and has thousands of built-in, more advanced functionalities. We did not discuss the OOP aspect of Scala since it would require at least another full segment. However, plenty of resources exist for those interested in exploring Scala in more detail.

Suggested next topics to cover:

- Object-Oriented Programming
    - Classes
        - Non-abstract classes
        - Abstract classes
        - Constructors and initialization
    - Objects
    - Traits
    - Inheritance & polymorphism
        - The Liskov Substitution Principle
- Enums and ADTs
    - Enumerations
        - Basic enums
        - Parameterized enums
    - Algebraic Data Types (ADTs)
        - Sealed traits and case classes
        - Match types
- Packaging
    - Understanding packages and package hierarchy
    - Declaring packages
    - Importing & using packages
- Unit testing
    - Designing appropriate tests
    - Executing tests
    - Using the Scala built-in test suite

Furthermore, below are some resources for those interested in learning more about this awesome language:

Disclaimer: *None of the materials below contain referral links or other sponsored content. Everything included is of my own selection.*

First stops:

- **Scala Docs, Scala:** The recommended first stop, including everything required to start using Scala. This documentation page includes the following:
    - Getting Started
    - Tour of Scala
    - Scala Toolkit
    - Online Courses
    - Books
    - Tutorials
    - Cheatsheets
    - Style guide

- FAQs
- Changelogs between Scala versions
- **Scala Community, Scala:** A great community home page including several links, such as:
  - Scala Users
  - Discord Channels (*Scala, Scalameta, Type level, Play Framework*)

Free resources:

- **10 Advanced Programming Techniques**: Covers more advanced techniques using Scala as the main language.
  - Tail recursion
  - Higher-order functions
  - Currying
  - Monads
  - Pattern matching & extractors
  - Lazy evaluation
  - Implicits & type classes
  - Continuation-passing style (*CPS*)
  - Futures and Promises
  - Higher-kinded types
- **DevInsideYou**: The single-handedly best YouTube channel covering excellent tips, tricks, and full-blown tutorials of simple to complex topics.
- **Rock the JVM:** Another excellent YouTube channel covering various topics while providing comprehensive, step-by-step tutorials.
- **Baeldung/Scala:** An excellent free online resource providing detailed information about the Java ecosystem, including a vast section dedicated to Scala.

Paid resources:

- **Scala & Functional Programming Essentials, Rock the JVM on Udemy:** A very nice introductory-level course focusing on the essentials of Scala, provided by Rock the JVM founder Daniel Ciocîrlan.
- **Functional Programming in Scala Specialization, Coursera + EPFL:** An excellent intermediate-level, four-course specialization designed by EPFL and taught by Scala founder Martin Odersky, focusing on:
  - Functional programming
  - Functional design
  - Parallel programming
  - Big Data analysis with Scala & Spark

General bibliographical resources:

- **Programming in Scala, Fifth Edition, Artima:** The best book to learn Scala, particularly for Java developers, having Martin Odersky as co-author of the book.
- **Functional Programming in Scala, Manning:** Almost every programming language has at least one "*infamous*" book that strongly focuses on computational and even mathematical theory. This red book is the case for Scala; it provides a comprehensive theoretical review of advanced functional programming concepts, mixing in heavy mathematical backgrounds such as set theory, lambda calculus, type theory, and more. It even goes as far as discussing approaches for writing entire applications using purely functional code.
- **Structure and Interpretation of Computer Programs, Harold Abelson & Gerald Jay Sussman, MIT Press:** This one offers a deep dive into the principles of functional programming while accompanying plenty of examples & exercises using a combination of math, diagrams, & code.

Mathematical bibliographical resources:

- **Naive Set Theory, Paul R. Halmos, Springer:** Now that we're discussing set theory, it's relevant to at least have general knowledge of what this branch of mathematics studies, simply because functional programming inherits a lot of concepts from this discipline.

- **An Introduction to Functional Programming Through Lambda Calculus, Greg Michaelson**: as we discussed, functional programming is based on lambda calculus. Having a general understanding of what this branch of mathematics is all about could better facilitate understanding of functional programming.
- **Types and Programming Languages, Benjamin C. Pierce, MIT Press:** Since Scala combines and encourages the use of functional programming along with type-driven development, it's useful to at least have a general notion of how type systems are designed.

Contributing to the Scala development:

- **Contributions, Scala:** For those already versed in the language and wishing to take the next step, this is a great page containing multiple resources for contributing to the Scala language development.
- **The Scala Programming Language, GitHub:** The official GitHub Scala page, including 50+ repositories at the time of the writing of this article.
- **lampepfl/dotty:** When Scala 3 was still under development, and Scala 2 was the active version, all the new features (*now Scala 3*) were included in an implementation called dotty. The official EPFL page can also be consulted here.

---

§

Conclusions

Well, there we have it. We journeyed through Scala from its beginnings to why it's a worthwhile language for anyone interested in high performant & elegantly-written applications. We got our workspace ready and started tinkering with Scala REPL and projects.

Next, we discussed Scala syntax, tangled with the most relevant data types, and navigated through some of the most relevant control structures available, including the powerful pattern-matching construct. We discussed a set of the most pertinent collections, walked through higher-order functions, and dipped our toes into the pool of recursion.

As we wrap this up, we're now positioned on the brink of the vast landscape of Scala, one of the most powerful languages available today. We've covered a lot of ground, but there's always more to learn. And especially if we're coming from a different paradigm, Scala provides an excellent way to do things differently, smarter, and most of all, safer.

---

§

References

- Scala Documentation, Scala
- Mutable & Immutable Collections, Scala
- Tail Recursion in Scala, Baeldung
- For Comprehensions, Scala
- Scala Conditional Expressions, Baeldung
- Pattern Matching, Scala
- Types and the Type System, Scala

---

§

Copyright