آموزش اسکالا

یاسین امینی

فهرست مطالب

1 مقدمه

در حال حاضر که در حال نگارش مطلب هستم، هیچ گونه منبع فارسی‌ای برای آموزش این زبان وجود ندارد و من سعی دارم که آشنایی مختصر و مفیدی برکارکردها و ویژگی‌های این زبان برنامه‌نویسی داشته باشم.

موضوعاتی که در این کتاب یاد می‌گیرید:

- اسکالا چیست؟
- روش برنامه‌نویسی اسکالا؟
- تفاوت بین برنامه‌نویسی سنتی و تابعی؟
- توسعه برنامه با زبان برنامه‌نویسی اسکالا؟
- نگاشت/کاهش در اسکالا؟
- ....

2 تاریخچه و معرفی

اسکالا یک زبان سطح بالا و موازی و همه منظوره است که امروزه تقریباً در بیشتر حوزه ها از جمله کلان داده، داده کاوی، وب، اندروید و ... کاربرد دارد و برای خود جایی در دنیای توسعه نرم‌افزار باز کرده است. این زبان برای اولین بار در سال ۲۰۰۳ توسط مارتین اودرسکی[1] معرفی گردید. زبان اسکالا مخفف scaleable language به معنی زبان مقیاس پذیر است یعنی می‌تواند برای انواع مختلفی از پروژه‌ها، از پروژه‌های کوچک تا بزرگ مورد استفاده قرار بگیرد.

زبان برنامه‌نویسی اسکالا از نخستین زبان‌های برنامه نویسی بود که دو مفهوم شی گرایی و برنامه نویسی تابعی را با هم ادغام کرد. اسکالا بر روی ماشین مجازی جاوا یا jvm اجرا می‌شود، یعنی در ابتدا کدهای اسکالا به بایت کد تبدیل شده و سپس تفسیر می شوند. اسکالا قابلیت اجرای کدها و کتابخانه‌های جاوا را نیز دارا می‌باشد. یعنی این قابلیت وجود دارد که کدهای جاوا را بدون هیچ تغییری اجرا نماید.

تعریف متغیرها در اسکالا ایستا است یعنی باید حتماً نوع متغیر(در اسکالا کلاس متغیر ، چون همه چیز در اسکالا شی است) را مشخص باشد. به دلیل امکاناتی که در زمینه ی تعریف متغیرهای تغییر ناپذیر دارد می‌تواند در زمینه برنامه‌های همزمان و موازی بسیار خوب عمل کند. ایستا بودن تعریف متغیر در اسکالا سبب افزایش سرعت، خطایابی آسان و ... می‌شود.

اسکالا به برنامه‌نویسان و توسعه‌دهندگان این اجازه را می‌دهد که کدهای خود را به صورت کارآمد و بهینه بنویسند و سرعت توسعه نرم‌افزار را به شکل چشم‌گیری بالا می‌برد. این ویژگی

[1]Martin Odersky

اسکالا سبب شد که بسیاری از استفاده کنندگان اصلی جاوا، به اسکالا مهاجرت کنند. اسکالا امکانات ویژه‌ای نیز جهت همگام‌سازی و موازی‌سازی فراهم می‌کند.

2.1 نصب

برای نصب اسکالا باید در قدم اول از نصب بودن جاوا بروی سیستم خود اطمینان حاصل نمایید و سپس اقدام به نصب اسکالا کنید. برای نصب بررسی نصب بودن جاوا دستور java -version را وارد نمایید، اگر خروجی این دستور نسخه جاوا را نمایش داد بدین معنی است که جاوا در سیستم شما به درستی نصب شده است. اگر هنگام وارد کردن دستور با خطا مواجه شدید یعنی جاوا به درستی نصب نشده است.

بعد از این مرحله به آدرس زیر مراجعه نموده و نسخه متناسب با سیستم عامل خود را دانلود کرده و اقدام به نصب آن نمایید.

https://www.scala-lang.org/download/

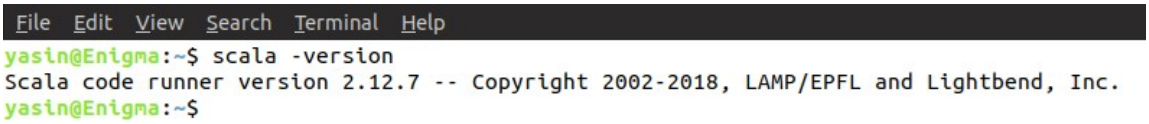
آدرس بالا فایل جاوا را در اختیار شما قرار می‌دهد، برای نصب کردن اینگونه فایل‌ها کافی است دستور زیر را وارد کنید.

```
$ java -jar scala.jar
```

در سیستم‌های مبتنی بر یونیکس برای نصب اسکالا به ترتیب زیر عمل نمایید:

```
$ sudo apt install scala
```

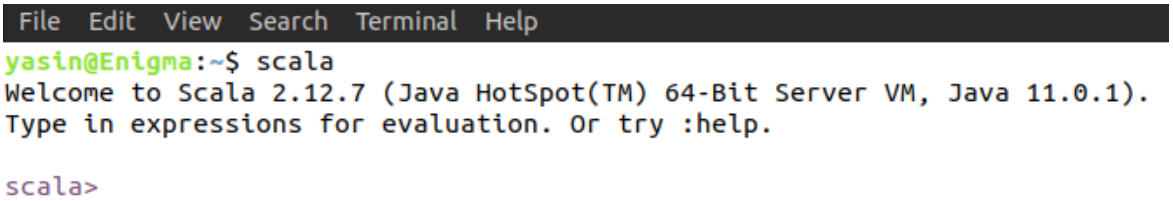
برای اطمینان از نصب بودن scala –version را وارد نموده و اگر اطلاعاتی از نسخه موجود به عنوان خروجی نمایش داده شود، یعنی به درستی نصب شده است (همانند تصویر ۱).



تصویر ۱ نصب موفقیت آمیز اسکالا بروی سیستم عامل اوبونتو

2.2 نوشتن اولین برنامه

طبق قاعده‌ای تعریف نشده شروع کد نویسی در هر زبان برنامه‌نویسی با برنامه «سلام دنیا» آغاز می‌شود. با وارد کردن scala در خط فرمان یک محیط تعاملی برای کار با زبان اسکالا باز می‌شود همانند تصویر ۲.



تصویر ۲ محیط تعاملی زبان برنامه‌نویسی اسکالا

حالا در این محیط اولین برنامه، یعنی «سلام دنیا» یا به قول خارجی‌ها «Hello, World» را اجرا می‌کنیم. برای چاپ عبارت در خروجی در زبان برنامه‌نویسی اسکالا نیز، مانند بیشتر زبان‌ها از print یا println استفاده می‌شود. پس اولین برنامه در دنیا اسکالا به ترتیب زیر خواهد بود.

```
scala> println("Hello, World!")
```

بعد از وارد کردن این دستور Enter را وارد و خروجی مورد نظر به شما نمایش داده می‌شود.

```
Output: Hello, World!
```

به این محیط که در آن هر دستور را وارد و به سرعت خروجی آن را مشاهده می‌کنید، محیط تعاملی[2] گفته می‌شود. این محیط‌ها با در اختیار گذاشتن قابلیت REPL[3] در اختیار توسعه‌دهندگان سرعت خطایابی را بالا برده و برای مباحث آموزشی بسیار مناسب هستند.

[2] Interactive

[3] Read-Evaluate-Print-Loop

REPL خلاصه‌سازی شده چهار کلمه خواندن، ارزیابی، چاپ و حلقه می‌باشد. یعنی در اجرای هر دستور چهار مرحله زیر طی می‌شود:

- ✓ **خواندن:** دستور وارد شده توسط کاربر، خوانده می‌شود.
- ✓ **ارزیابی:** زمانی که کلید Enter وارد شد، عبارت وارد شده توسط کاربر از نظر نحوی و معنایی مورد بررسی قرار می‌گیرد.
- ✓ **چاپ:** زمانی که دستور وارد شده توسط کاربر مورد ارزیابی قرار گرفت و نتیجه ارزیابی درست بود، نتیجه دستور به کاربر نمایش داده می‌شود.
- ✓ **حلقه:** بعد از طی مراحل بالا، محیط یک بار دیگر آماده دریافت ورودی جدید می‌شود.

حال می‌توانید از این محیط تعاملی به عنوان یک ماشین حساب برای اجرای دستورات مختلف یا برای آزمایش دستورات ساده که به سرعت نتیجه را دریافت کنید، استفاده نمائید (تصویر ۳).

```
scala> 1 + 2 + 3
res7: Int = 6

scala> 1 + 2 * 4
res8: Int = 9

scala> 2 * 9 / 3
res9: Int = 6

scala> 1 + 2 * 3 / 4 - 1
res10: Int = 1

scala> 212 * 23
res11: Int = 4876

scala> 432 / 43
res12: Int = 10

scala> 432 % 43
res13: Int = 2
```

تصویر ۳ انجام کارهای عملیاتی در محیط تعاملی زبان برنامه‌نویسی اسکالا

بعد از مدتی کار کردن با این محیط تعاملی متوجه خواهید شد که دارای محدودیت‌های زیادی است. یکی از اصلی‌ترین محدودیت‌های آن در دستورات چند خطی یا در حین کپی کردن یک برنامه کامل خود را نشان می‌دهد. برای رفع این محدودیت مجموعه‌ای از ویژگی‌ها در این خط فرمان تعبیه شده است که آن را برای کارهای تست و عیب‌یابی بسیار مناسب می‌سازد. در ادامه به برخی از این ویژگی‌ها اشاره می‌کنیم:

برای دسترسی به لیست تمامی این ویژگی‌ها می‌توانید از help: استفاده کنید، که خروجی آن به شکل تصویر ۴ خواهد بود.

```
scala> :help
All commands can be abbreviated, e.g., :he instead of :help.
:completions <string>    output completions for the given string
:edit <id>|<line>        edit history
:help [command]          print this summary or command-specific help
:history [num]           show the history (optional num is commands to show)
:h? <string>             search the history
:imports [name name ...] show import history, identifying sources of names
:implicits [-v]          show the implicits in scope
:javap <path|class>      disassemble a file or class name
:line <id>|<line>        place line(s) at the end of history
:load <path>             interpret lines in a file
:paste [-raw] [path]     enter paste mode or paste a file
:power                   enable power user mode
:quit                    exit the interpreter
:replay [options]        reset the repl and replay all previous commands
:require <path>          add a jar to the classpath
:reset [options]         reset the repl to its initial state, forgetting all session entries
:save <path>             save replayable session to a file
:sh <command line>       run a shell command (result is implicitly => List[String])
:settings <options>      update compiler options, if possible; see reset
:silent                  disable/enable automatic printing of results
:type [-v] <expr>        display the type of an expression without evaluating it
:kind [-v] <type>        display the kind of a type. see also :help kind
:warnings                show the suppressed warnings from the most recent line which had any
```

تصویر ۴ مجموعه‌ای از دستورات مفید برای کارکرد بهتر خط فرمان اسکالا

همانطور که در شکل ۴ مشاهده می‌کنید، دستورات زیادی جهت عمل‌کرد بهتر این محیط تعاملی تعبیه شده است.

گزینه‌های کمکی در محیط تعاملی

- :paste
- :history
- :h?
- :quit
- :warnings
- :load
- :edit
- …

به عنوان مثال به کمک دستور paste: می‌توانید تعداد خطوط بسیار زیادی را به صورت یک قطعه کد نوشته و سپس با کمک ctrl + d آن را اجرا نمایید. از history: برای دیدن دستورات قبلی وارد شده استفاده کنید. از warnings: برای دیدن هشدارهایی که در پشت صحنه کد روی داده است، استفاده می‌شود، یا از quit: برای خروج و...

یکی دیگر از مشکلاتی که در محیط تعامل وجود دارد این است که اگر ورودی کاربر دچار تغییر شود، برنامه نیز باید تغییر پیدا کند. به عنوان مثال در نظر بگیرید می‌خواهیم مساحت یک مستطیل را محاسبه کنیم که طول آن برابر ۱۰ و عرض آن برابر ۵ می‌باشد. برنامه محاسبه مساحت مستطیل در تصویر ۵ نشان داده شده است.

```
scala> val length = 10
length: Int = 10

scala> val width = 5
width: Int = 5

scala> var s = length * width
s: Int = 50

scala>
```

تصویر ۵ برنامه محاسبه مساحت مستطیل

حال اگر در اندازه‌گیری طول یا عرض مستطیل خطایی روی داده باشد، باید مجدداً این دستورات را مقداردهی کرده و آن را دوباره اجرا نماییم. برای رفع این مشکل هم مانند سایر زبان‌های برنامه‌نویسی عمل کرده و کدهای برنامه را در یک فایل نوشته و سپس فایل را اجرا نمایید.

برنامه‌های اسکالا در فایل با پسوند scala. ذخیره می‌شوند و برای کامپایل آن‌ها می‌توان از دستور scalac filename.scala استفاده کرد که خروجی آن بایت کد خواهد بود که می‌توانید با scala filename آن را اجرا نمایید. کدهای نوشته شده در زبان اسکالا را می‌توانید به شکل زبان‌های اسکریپتی نیز اجرا نمایید، یعنی نیازی به کامپایل کردن کدها وجود ندارد و کافی است که دستور scala filename را اجرا نمایید. البته شما می‌توانید بانصب پلاگین های موجود در سایت اصلی بروی محیط های توسعه مختلف این کارها را به IDE ها بسپارید. در سایت اسکالا یک نسخه از اکلیپس مخصوص توسعه اسکالا نیز وجود دارد که می‌توانید از آن استفاده نمائید. یکی دیگر از حالت‌های اجرا در حالت اسکریپتی می‌باشد یعنی کد اسکالا را در یک فایل با پسوند .scala ذخیره کرده و آن را با دستور scala filename اجرا می‌نماییم.

2.3 نکات اولیه در زبان برنامه‌نویسی اسکالا

- ✓ زبان اسکالا به حروف بزرگ و کوچک حساس است.
- ✓ نامگذاری کلاس و متغیرها با روش camel case پیروی می کند.
- ✓ نام کلاس‌ها با حروف بزرگ شروع می‌شود و اگر دو قسمتی باشد قسمت دوم هم با حرف بزرگ نوشته می شود. مانند HelloWorld
- ✓ نام متدها با حرف کوچک آغاز می‌گردد و اگر قسمت دوم داشته باشد با حرف بزرگ نوشته می شود. helloWorld
- ✓ نام شی یا کلاس اصلی برنامه و فایل با یکی باشد مثلاً هر دو باید HelloWorld باشند.
- ✓ اجرای برنامه از متد main شروع می‌شود و باید در هر برنامه اسکالا وجود داشته باشد.

- ✓ سه نوع کامنت در زبان اسکالا وجود دارد، کامنت یک خطی با // ،کامنت چند خطی با /* multi line comment */ و نوشتن مستندات برای توابع و کلاس‌ها نیز به ترتیب زیر می باشد.

```
/**
* documents for method or class
* @param
* @return
*/
```

- ✓ فاصله و خط خالی اضافی نادیده گرفته می شود.
- ✓ در اسکالا امکان افزودن دو یا چند کتابخانه در یک خط وجود دارد مثلاً :

```
import java.util.Scanner ;
import java.util.Arrays ;
```

این نحوه افزودن کتابخانه مخصوص زبان جاواست و در اسکالا می‌توان آن را بسیار مختصرتر انجام داد که موجب افزایش سرعت توسعه می گردد.

```
import java.util.{Scanner , Arrays}
```

- ✓ کلمات کلیدی زبان اسکالا در جدول ۱

جدول ۱ کلمات کلیدی زبان اسکالا

abstract	case	catch	class
def	do	else	extends
false	final	finally	for
forSome	if	implicit	import
lazy	match	new	Null
object	override	package	private
protected	return	sealed	super
this	throw	trait	try
true	type	val	var
while	with	yield	
-	:	=	=>
<-	<:	<%	>:
#	@		

- ✓ شناسه در زبان اسکالا همانند دیگر زبان‌ها با حرف یـا زیرخـط(underline) شـروع شده و در ادامه می‌تواند عدد یا حرف بیاید.

- ✓ بزرگترین و آشکارترین تفاوت نحوی بین جاوا و اسکالا در این است که در اسکالا گذاشتن «;» اختیاری است و در جاوا اجباری. در اسکالا توصیه می‌شود که از گذاشتن «;» خودداری نمایید.
- ✓ کلاس : هر کلاس را می‌توان یک قالب در نظر گرفت که رفتار و خصوصیات مرتبط با کلاس را توصیف می نماید.
- ✓ شی : شی یک نمونه از کلاس است، درواقع کلاس یک مفهوم انتزاعی است و با شی است که کلاس معنا می یابد. مثلاً اگر انسان را یک کلاس در نظر بگیریم، انسان به خودی خود یک مفهوم است و زمانی که می‌گوییم علی یک انسان است معنا می یابد درواقع علی یک شی از کلاس انسان می باشد.
- ✓ متغیر : متغیرها درواقع حکم خصوصیات یک کلاس را دارند. مانند نام و سن در کلاس انسان.
- ✓ متد : درواقع متدها بیان کننده ی رفتار های مرتبط با کلاس می‌باشند مثلاً پیاده روی کردن یک متد از کلاس انسان است.
- ✓ رفتار (trait) : رفتارها درواقع کلاس‌هایی هستند که متغیرهای درون آن‌ها محصورسازی شده‌اند و در تابع ها فقط امضای تابع مشخص می‌گردد و پیاده‌سازی نمی‌شود بلکه در هنگام ساختن شی از آن پیاده‌سازی می گردد.
- ✓ عبارات رزرو شده :
  - ✓ \n : خط جدید
  - ✓ \t : به اندازه یم تب فاصله ایجاد می نماید
  - ✓ \\ : \ برای نمایش
  - ✓ \” : ” برای نمایش
  - ✓ \’ : ’ برای نمایش

- ✓ \b : قبلی را حذف می‌کند

2.4 متغیرها و نوع داده

اسکالا در نوع داده موجود از زبان جاوا الهام گرفته است و تمام نوع داده‌های موجود در جاوا در اسکالا نیز وجود دارد. فقط نکته‌ای که وجود دارد این است که تمام نوع داده‌های موجود در زبان اسکالا کلاس می‌باشند و برای تعریف متغیر در واقع باید از آن‌ها شی ساخت و می‌توان بروی انواع داده‌های موجود تابع فراخوانی کرد. یعنی مثلاً وقتی که ما می نویسیم ۱ + ۲ دو شی ۱ و ۲ را ایجاد کرده‌ایم و تابع + را روی اولی که دومی را به عنوان پارامتر ورودی می‌گیرد فراخوانی کرده‌ایم. یعنی مثال بالا را می‌توانیم به صورت زیر نیز بنویسیم : ۲+.۱ . گاهی نیز برای افزایش سرعت اسکالا شی داده ایجاد شده در زمان کامپایل به نوع داده‌ای از جاوا تبدیل می گردد. در جدول ۲ انواع داده‌های قابل تعریف در زبان اسکالا را مشاهده می‌نمایید.

جدول ۲ انواع داده موجود در اسکالا

نوع داده	تعداد بیت	بازه
Byte	۸	۱ ـ $۲^{۸}$ تا $۲^{۸}$ـ
Short	۱۶	۱ ـ $۲^{۱۶}$ تا $۲^{۱۶}$ـ
Int	۳۲	۱ ـ $۲^{۳۲}$ تا $۲^{۳۲}$ـ
Long	۶۴	۱ ـ $۲^{۶۴}$ تا $۲^{۶۴}$ـ
Float	۳۲	طبق استاندارد IEEE 754
Double	۶۴	طبق استاندارد IEEE 754

Boolean	۸	بیت قابل آدرس دهی نیست!
Char	۱۶	پشتیبانی از یونیکد U+0000 تا U+FFFF
String	-	مجموعه‌ای از کاراکترها
Unit	-	عدم وجود مقدار
Null	-	مقدار null یا اشاره به متغیر خالی
Nothing	-	زیرنوع تمامی نوع‌ها، بر عدم وجود مقدار دلالت دارد
Any	-	تمامی نوع داده‌ها، زیر مجموعه این نوع هستند
AnyRef	-	والد تمام انواع داده Any

Unit نوع داده‌ای است که برای تعریف توابعی استفاده می‌شود که هیچ مقداری را برنمی‌گردانند و به معنی ( ) خالی است. تابع main در اسکالا تابعی است که خروجی و مقدار برگردانده شده آن از جنس Unit می‌باشد.

انتساب نوع به متغیر در اسکالا اختیاری است. زبان برنامه‌نویسی اسکالا از هر دو نوع تعریف نوع ایستا و پویا پشتیبانی می‌کند. در تعریف نوع ایستا هنگام تعریف یک متغیر نوع آن نیز تعیین می‌شود اما در تعریف نوع پویا، نوع متغیر تعیین نمی‌شود و توسط خود کامپایلر تعیین می‌شود.

2.5 رشته

در اسکالا نیز رشته‌ها غیرقابل تغییر هستند، یعنی متدهایی که برای کار با رشته‌ها وجود دارد بروی رشته اصلی تغییر ایجاد نمی‌کنند، مگر آن که دوباره در رشته قرار داده شوند. نحوه ایجاد رشته در اسکالا به دو ترتیب زیر می‌باشد.

```
var str = " one line String "
```

از این تعریف برای رشته‌های یک خطی استفاده می‌گردد. اگر رشته چند خط باشد باید چگونه آن را در یک متغیر بنویسیم؟

```
var str = """ multi line String """
```

برای رشته‌های چند خطی از سه تا " استفاده می‌گردد. در زبان اسکالا رشته‌ها دارای توابع بسیار زیادی هستند. تعدادی از توابع کاربردی رشته‌ها را در جدول شماره مشاهده می‌نمایید.

در نسخه‌های اخیر اسکالا روش‌هایی برای کار با رشته‌ها و فرمت‌بندی رشته ایجاد شده است. برای استفاده از این ویژگی‌ها باید یکی از سه مقدار s، f، raw را قبل از رشته وارد نمایید. اگر از s استفاده کنید می‌توانید متغیرهای غیررشته‌ای را داخل رشته با پیشوند $ به کار ببرید. در صورت استفاده از f مشابه با printf در زبان سی می‌توانید رشته را فرمت‌بندی نمایید (%d، %i، %f، %) و در صورتی که از raw استفاده کنید رشته خام موجود در بین "" را برمی‌گرداند (یعنی تمامی \n و ... را نادیده می‌گیرد). برای درک بهتر این مفاهیم به تصویر ۶ توجه نمایید.

```
object Demo {
   def main(args: Array[String]) {
      var name = "yasin"
      var height = 1.9
      println(s"name: $name")
      println(s"12 * 6 = ${12*6}")

      println(f"$name%s, $height%f2.2d")

      println(raw"Hello\n world\n")
   }
}
```

تصویر ۶ فرمت‌بندی رشته‌ها در زبان اسکالا

در این جدول ۳ مقدار str1 برابر با "String" و مقدار str2 برابـر بـا "one" در نظـر گرفتـه شود.

جدول ۳ انواع متدهای رشته در جاوا

متد	ورودی	خروجی	نمونه
length	خالی	عدد، طول رشته را برمی‌گرداند.	str1.length( ) => 6
concat	رشته	دو رشته را به هم متصل می‌کند. مشابه با جمع دو رشته با + است.	str1.concat(str2) => Stringone
charAt	شماره کاراکتر مورد نظر	کاراکتر مورد نظر را برمی‌گرداند.	str1.charAt(2) => r
compare To	رشته	اختلاف عددی از دو رشته را برمی‌گرداند.	str1.compare(str2) => -28
trim	خالی	تمامی کاراکترهای خالی ابتدا و انتهای رشته را حذف می‌کند.	var str = " one " str.trim( ) => one
toUpper Case	خالی	تمامی کاراکترها را به حروف بزرگ تبدیل می‌کند.	str1.toUpperCase( ) => STRING
toLower Case	خالی	تمامی کاراکترها را به حروف کوچک تبدیل می‌کند.	str1.toLowerCase( ) => string

toString	خالی	شیء مورد نظر را تبدیل به رشته می‌کند.	var a = 12 a.toString() => "12"
toCharArray	خالی	رشته را به آرایه‌ای از کاراکترها تبدیل می‌کند.	str1.toCharArray() => Array[Char] = Array(S, t, r, i, n, g)
substring	دو عدد به عنوان شروع و پایان	زیر رشته‌ای از رشته مورد نظر از نقطه ابتدا تا پایان تولید می‌کند.	str1.substring(1,4) => tri
split	عبارت باقاعده (regex)	رشته براساس عبارت باقاعده به آرایه‌ای از رشته‌های تقسیم می‌کند.	str1.split("") => Array[String] = Array(S, t, r, i, n, g)
replace	رشته قبلی، رشته جدید	رشته قبلی را با رشته جدید جایگزین می‌کند.	str1.replace("S", "s") => string
replaceAll	رشته قبلی، رشته جدید	تمامی رشته‌های قبلی را با رشته جدید جایگزین می‌کند.	str1.replaceAll("S", "s") => string
indexOf	رشته، کاراکتر و ...	اندیس کاراکتر یا نقطه شروع رشته وارد شده را برمی‌گرداند.	str1.IndexOf("S") => 0

در جدول شماره ما تنها تعدادی از متدهای موجود برای کار با رشته‌ها را به شما معرفی نمودیم. برای آشنایی بیشتر با رشته‌ها و متدهای آن به مستندات اسکالا مراجعه فرمایید.

رشته‌ها در زبان اسکالا، مانند جاوا غیرقابل تغییر هستند و برای ایجاد رشته‌های پویا باید از StringBuilder استفاده گردد.

2.6 تعریف متغیر

به هر چیزی که بتواند ارزش‌ها و مقادیر گوناگون را بپذیرد، متغیر گفته می‌شود. در حقیقت متغیر اسمی است که به یک خانه از حافظه داده می‌شود و بسته به نوع داده می‌تواند مقداری از حافظه را رزرو کرده و انواع مقدار در آن ذخیره گردد. به منظور تعریف متغیر در اسکالا از کلمه کلیدی val برای مقادیر ثابت و var برای تعریف متغیر مورد استفاده قرار می‌گیرد. اگر یک متغیر با val تعریف شود، مقدار آن تا پایان برنامه ثابت بوده و تغیر نمی‌کند، اما اگر با var تعریف شود، مقدار آن می‌تواند در صورت نیاز تغیر کند.

اسکلت تعریف متغیر در زبان اسکالا به صورت زیر می‌باشد:

val یا var نام متغیر : نوع داده = مقدار

تعیین نوع داده اختیاری است و می‌تواند خالی باشد که کامپایلر خود اسکالا تعیین نوع را انجام می‌دهد. پیشنهاد می‌گردد که تعریف نوع به صورت دستی انجام شود تا سرعت کامپایل برنامه بیشتر شود. کامپایلر اسکالا نوع متغیر را با توجه به مقدار اختصاص داده شده به متغیر تشخیص می‌دهد[4].

val valName : DataType = Value => define immutable variable

[4] به این ویژگی Variable Type Interface گفته می‌شود.

var varName : DataType = Value => define mutable variable

یکی دیگر از ویژگی‌های زبان اسکالا این است که یک تابع می‌تواند چند مقدار برگرداند یا به طور همزمان چندین متغیر مقداردهی شوند. برای مقداردهی چند متغیر به صورت همزمان از کلاس Pair استفاده می‌شود مانند مثال زیر. کلاس Pair کلاسی برای ایجاد تاپل است. تاپل (Tuple) یک ساختار داده مشابه آرایه است که می‌تواند انواع داده در آن ذخیره گردد (در قسمت ساختمان داده‌های اسکالا تاپل‌ها را مورد بررسی قرار می‌دهیم).

val (myVar1: DataType, myVar2: DataType) = Pair(var1, var2)

در تصویر ۷، تعریف انواع متغیرها و مقداردهی آن‌ها را مشاهده می‌نمایید.

```scala
object Variable {
  def main (args: Array[String]){
    val fName: String = "yasin"
    val lName: String = "amini"
    var age: Short = 23
    var weight: Float = 71.2f
    var salary: Long = 1234324234
    var isMarried: Boolean = false

    println(s"First name: $fName")
    println(s"Lost name: $lName")
    println(s"Age: $age")
    println(s"Weight: $weight")
    println(s"Salary: $salary")
    println(s"Married: $isMarried")
  }
}
```

تصویر ۷ تعریف انواع متغیر در اسکالا

حال برنامه بالا را به گونه‌ای تغییر دهید که مقداردهی هر دو متغیر نام و فامیلی در یک خط صورت بپذیرد. در خط ۶ برای تعریف نوع اعشاری Float یک f در مقابل عدد مورد نظر گذاشته می‌شود، در غیراینصورت برنامه دچار خطا می‌شود (در اسکالا به صورت پیش‌فرض اعداد اعشاری Double در نظر گرفته می‌شوند). در تابع println نیز برای فرمت‌بندی رشته در ابتدای رشته یک s قرار داده و داخل رشته با $ و نام متغیر آن را به رشته تبدیل می‌نماییم (سعی کنید این کار را با روش‌های دیگری برای این کار وجود دارد، انجام دهید). برنامه را یکبار بدون تعریف نوع داده بازنویسی نمایید آیا خروجی برنامه با خروجی برنامه حاضر یکسان خواهد بود؟

محدوده متغیرها می‌تواند بر عمل‌کرد متغیر تاثیرگذار باشد. در زبان اسکالا سه محدوده دسترسی برای متغیرها وجود دارد: متغیرهای مرتبط به یک شی، آرگومان‌های تابع، متغیرهای محلی.

- ✓ متغیرهای محلی

به متغیرهایی که در یک محدوده خاص مانند درون حلقه، درون تابع یا متد، درون عبارت‌های کنترلی تعریف می‌شوند، متغیرهای محلی می‌گویند. دسترسی به متغیرهای محلی محدود به همان محدوده‌ای که متغیر در آن تعریف شده است و درخواست‌های خارج از محدوده تعریف شده با خطا مواجهه می‌شوند.

- ✓ آرگومان‌های تابع

آرگومان‌های یک تابع، متغیرهایی هستند که هنگام فراخوانی یک تابع به عنوان ورودی به تابع ارسال می‌شوند. محدوده این متغیرها درون تابع بوده و در تمامی محدوده تابع قابل استفاده می‌باشند.

- ✓ متغیرهای مرتبط با شی خاص

متغیرهایی که هنگام ساختن یک شی خاص مقداردهی می‌شوند، فیلدهای یک شی نامیده می‌شوند. این فیلد یا خصوصیات در تمامی محدوده شی و محدوده دسترسی کلاس قابل استفاده می‌باشد (درباره سطح دسترسی کلاس‌ها در آینده بحث می‌کنیم).

2.7 عملگرها

به نمادی که به کامپایلر انجام یک عمل خاص را گزارش می‌دهد، عملگر گفته می‌شود. انواع مختلفی از عملگرها مانند عملگرهای ریاضی، عملگرهای منطقی، عملگرهای بیتی، عملگرهای رابطه‌ای و انتسابی وجود دارند.

به نمادی که عملگر بروی آن عملی را انجام می‌دهد، عملوند گفته می‌شود.

- **عملگرهای ریاضی**

عملگرهایی که برای انجام اعمال ریاضی مانند جمع و تفریق و ضرب و ... استفاده می‌شوند، عملگرهای ریاضی را در جدول ۴ مشاهده می‌نمایید.

جدول ۴ انواع عملگرهای ریاضی در اسکالا

عملگر	توضیحات	نمونه
+	جمع دو عملوند	۱۰ + ۲۰ = ۳۰
-	تفریق دو عملوند	۱۰ ـ ۲۰ = -۱۰
*	ضرب دو عملوند	۱۰ * ۲۰ = ۲۰۰
/	تقسیم دو عملوند	۱۰ / ۲۰ = ۰
٪	محاسبه باقی‌مانده تقسیم دو عدد	۱۰ ٪ ۲۰ = ۱۰

➢ عملگرهای رابطه‌ای

به عملگرهایی که رابطه بین دو عملونـد را بررسـی می‌کننـد و خـروجی آن‌هـا بـه صـورت بـولی (درست یا غلط) خواهد بود، عملگرهای رابطه‌ای گفتـه می‌شـود. انـواع مختلـف عملگرهـای رابطه‌ای که در اسکالا پشتیبانی می‌شوند را در جدول ۵ مشاهده می‌نمایید.

جدول ۵ عملگرهای رابطه‌ای اسکالا

عملگر	توضیحات	نمونه
==	دو عملوند با هم برابر هستند یا خیر	false <= (۱۰ == ۲۰) true <= (۱۰ == ۱۰)
=!	دو عملوند نابرابرند یا خیر	true <= (۱۰ != ۲۰) false <= (۱۰ != ۱۰)
>	عملوند اول از عملوند دوم بزرگ‌تر است یا خیر	false <= (۱۰ > ۲۰) false <= (۱۰ > ۱۰) true <= (۲۰ > ۱۰)
<	عملوند اول از عملوند دوم کوچک‌تر است یا خیر	true <= (۱۰ < ۲۰) false <= (۱۰ < ۱۰) false <= (۲۰ < ۱۰)
>=	عملوند اول بزرگ‌تر، مساوی عملوند دوم است یا خیر	false <= (۱۰ >= ۲۰) true <= (۱۰ >= ۱۰)

false <= (۲۰ >= ۱۰)		
true <= (۱۰ <= ۲۰) true <= (۱۰ <= ۱۰) false <= (۲۰ <= ۱۰)	عملوند اول کوچک‌تر، مساوی عملوند دوم است یا خیر	<=

➢ عملگرهای منطقی

به عملگرهایی که بروی عملوندها اعمال منطقی مانند اجتماع و اشتراک و تناقض را انجام می‌دهند، گفته می‌شود. عملگرهای منطقی پشتیبانی شده در زبان اسکالا را در جدول ۶ مشاهده می‌نمایید.

جدول ۶ عملگرهای منطقی در اسکالا

عملگر	توضیحات	نمونه
&&	"و" منطقی	true && true => true true && false => false false && false => false
||	"یا" منطقی	true || true => true true || false => true false || false => false
!	"تناقض" منطقی	!true => false !false => true

➢ عملگرهای بیتی

عملگرهایی که اعمال منطقی را بروی بیت‌های اعمال می‌کنند. در جدول ۷ عملگرهای بیتی موجود در اسکالا را مشاهده می‌نمایید. توجه داشته باشید که عملگر تناقض یا ~ بروی یک عملوند اعمال می‌گردد و مقدار را برعکس می‌نماید. Xor عملگری است که تنها در صورتی جواب آن یک خواهد بود که عملگر اول و دوم مخالف هم باشند.

جدول ۷ عملگرهای بیتی اسکالا

عملگر اول	عملگر دوم	& یا "و" بیتی	ا یا "و" بیتی	~ یا "تناقض" بیتی	xor
۰	۰	۰	۰	۱	۰
۱	۰	۰	۱	۰	۱
۰	۱	۰	۱	۱	۱
۱	۱	۱	۱	۰	۰

عملگرهای بیتی دیگری نیز همچون شیفت راست (<<) و شیفت چپ (>>) وجود دارد که عملوند را به تعداد عدد ورودی به راست یا چپ شیفت می‌دهند.

اگر مقدار عملوند برابر ۰۰۰۱ باشد با سه شیفت به چپ به ۱۰۰۰ و با ۱ شیفت به راست به ۰۰۰۰ تبدیل می‌شود. در تصویر ۸، انجام اعمال بیتی در اسکالا را مشاهده می‌نمایید.

```
scala> var a = 12
a: Int = 12

scala> var b = 40
b: Int = 40

scala> a & b
res2: Int = 8

scala> a | b
res3: Int = 44

scala> a ^ b
res4: Int = 36

scala> ~a
res5: Int = -13

scala> a << 2
res6: Int = 48

scala> a >> 2
res7: Int = 3
```

تصویر ۸ اعمال بیتی

اگر به تصویر ۸ توجه کنید متوجه می‌شوید که شیفت به راست مانند تقسیم بر ۲ و شیفت به چپ مشابه با ضرب در ۲ عمل می‌کند. برای درک بهتر این عملگرها، نتایج تصویر شماره را به صورت دستی تحلیل نمایید.

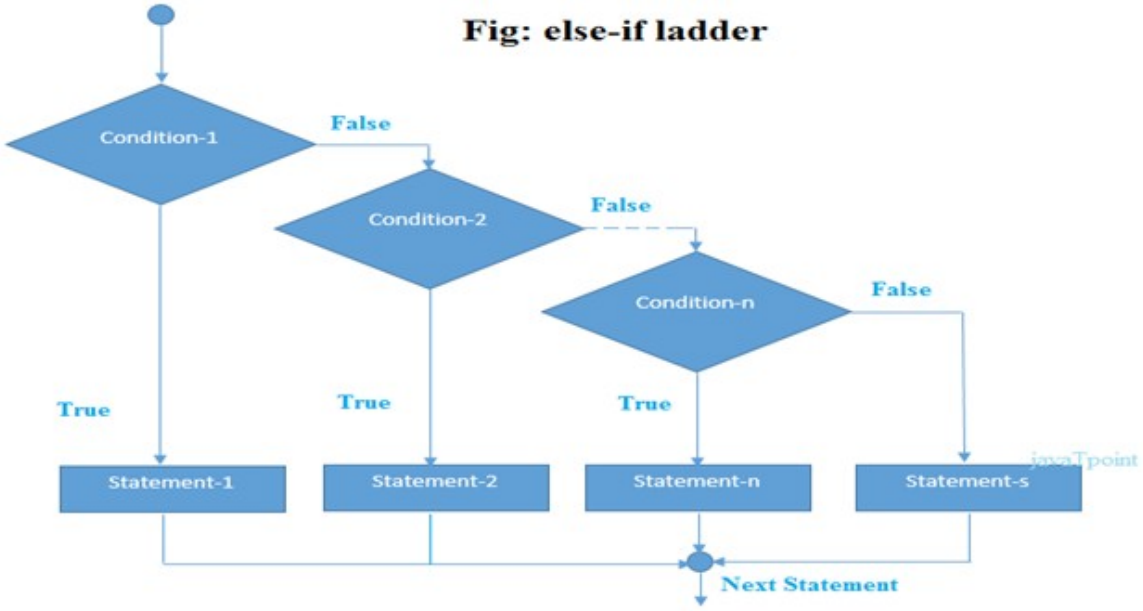
➢ عملگرهای انتساب

عملگرهای انتساب به عملگرهایی اتلاق می‌شود که یک مقدار را به یک متغیر نسبت می‌دهند مانند عملگر =، که مقدار سمت راست = را به متغیر سمت چپ نسبت می‌دهد، مانند a = ۲. عملگرهای دیگر را نیز می‌توانیم به اختصار بنویسیم به عنوان مثال به جای نوشتن a = a + ۱ می‌توانیم آن را به شکل a +=۱ خلاصه نماییم.

سوال: حاصل عبارت ۳ + ۴ * ۵ برابر با ۳۵ است یا ۲۳ ؟ در اینجاست که اولویت عملگرها اهمیت پیدا می‌کند. در اسکالا نیز مانند سایر زبان‌های برنامه‌نویسی اولویت عملگر به ترتیب () []، ! ~، * / ٪، + ـ، >> <<، > < >= <=، == !=، &، ^، |، &&، = می‌باشد.

3 دستورات شرطی و کنترلی

عبارت کنترلی همان تصمیماتی هستند که ما در زندگی روزمره باهاشون روبرو هستیم و آن‌ها را از ابعاد و جنبه‌های مختلف بررسی کرده و براساس معیارها یکی از موارد را انتخاب می‌نماییم. به عنوان مثال در فصول سرد سال، هنگام بیرون رفتن، اگر هوا ابری یا بارانی باشد با خود چتر می‌بریم، در غیراینصورت اگر برف باشد لباس گرم و چتر می‌بریم و اگر هیچ کدام از شرایط بالا نباشد (نه هوا سرد باشد و نه بارانی و ابری) از لباس متناسب با اون شرایط استفاده می‌کنیم. حالا اگر بخواهیم این را به زبان تصویر و فلوچارت درک کنید به تصویر ۸ توجه نمایید.



تصویر ۹ فلوچارت دستورات شرطی، منبع: javapoint

دستورات شرطی موجود در زبان برنامه‌نویسی اسکالا شباهت بسیاری زیادی به دستورات شرطی زبان‌های جاوا، سی و... و کلاً زبان‌های مبتنی بر سی دارد. در ادامه اسکلت‌بندی و ساختار دستورات شرطی از جنس if را مشاهده می‌نمایید.

```
If (شرط۱) {
        // بدنه شرط
}
else If (شرط۲) {
        // بدنه شرط
}else {
        // بدنه شرط
}
```

در اسکالا نیز مانند سایر زبان‌های برنامه‌نویسی دستورات شرطی یک خطی وجود دارد. نمونه‌ای از این نوع دستورات را در تصویر ۱۰ مشاهده می‌نمایید.

```scala
object Decision {
    def main(args: Array[String]) {
      val age = 23
      val result = if (age>13 && age<20) "teenager" else if (age<13) "baby :)" else "young"
      println(result)
    }
}
```

تصویر ۱۰ دستور شرطی یک خطی

یکی دیگر از ویژگی‌های جالب اسکالا این است که می‌توانیم نتایج دستورات شرطی را به یک تابع نسبت دهیم. این امکان به این ویژگی تابعی بودن اسکالا برمی‌گردد که در آن همه چیز مقدار است. یعنی می‌توانیم اینگونه بگوییم که دستورات شرطی نیز مقداری را به عنوان خروجی برمی‌گردانند (تصویر ۱۱).

```
object Decision {
    def main(args: Array[String]) {
      val number = -10
      val numberTwo = 100
      println("|" + number + "| = " + absFunc(number))
      println("|" + numberTwo + "| = " + absFunc(numberTwo))
    }

   def absFunc(number: Int) = if (number>0) number else -number
}
```

تصویر ۱۱ انتساب مقدار برگردانده شده از if به تابع

یک نوع دستور شرطی دیگر به نام match/case در زبان اسکالا وجود دارد. این دستور ساختاری مشابه با switch/case سایر زبان‌ها دارد، ولی قدرت و انعطاف بیشتری دارد به همین علت زبان‌های جدید نیز بیشتر از این مورد استفاده می‌کنند. در تصویر ۱۲ یک نمونه برنامه که به کمک match/case نوشته شده است، را مشاهده می‌کنید.

```scala
object Decision {
    def main(args: Array[String]) {
      val number = 8
      match number {
       case 1 => println("One")
       case 2 => println("Two")
       case 3 => println("Three")
       case 4 => println("Four")
       case 5 => println("Five")
       case 6 => println("Six")
       case 7 => println("Seven")
       case 8 => println("Eight")
       case 9 => println("Nine")
       case _ => println("> Ten")
      }
    }
}
```

تصویر ۱۲ استفاده از pattern-match

در برنامه بالا – نشان‌دهنده حالت default است. یعنی اگر هیچ یک از موارد بالا نباشند این حالت برگزیده می‌شود. توجه داشته باشید که مانند سایر زبان‌ها دیگر نیز، نیازی به گذاشتن break در آخر هر case نداریم و خود زبان آن‌ها را یکی مورد آزمایش قرار می‌دهد.

یکی از نکاتی که باید به آن دقت فرمایید این است که match با تمام کلاس شی‌ها کار می‌کند و محدود با یک شی خاص نیست. و همچنین می‌تواند مقداری را نیز برگرداند، کلاً در اسکالا کلیه عبارات شرطی می‌توانند داخل متغیر ذخیره شوند (تصویر ۱۳).

```
object MatchReturn {
    def myFunc(a:String)={
        var retValue= a match {
            case "one" => a*2
            case "two" => a*5
            case _ => -1
        }
        retValue;
    }
    def main(args: Array[String]) {
        println(myFunc("three"))
        println(myFunc("two"))
        println(myFunc("one"))
    }
}
```

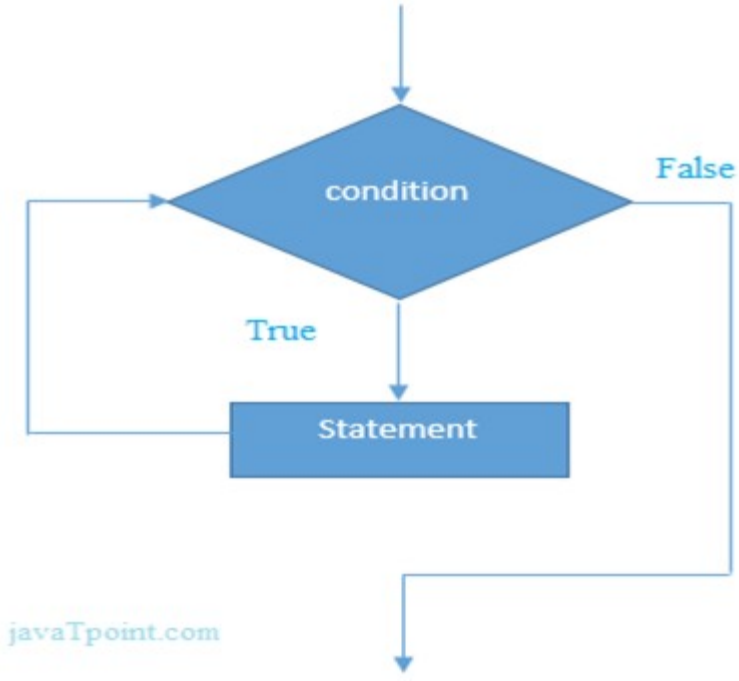
تصویر ۱۳ ذخیره حاصل دستورات شرطی در متغیر

output ==> -1 , twotwotwotwotwo , oneone

در اینجا یک نکته دیگر نیز قابل مشهود است که در اسکالا ضرب عدد در رشته معنادار است و به تعداد عدد تکرار می‌گردد.

4 حلقه‌ها

اگر بخواهیم کار یا عملی به صورت مکرر تکرار بشه از حلقه‌ها استفاده می‌کنیم. برای این کار در زبان‌های برنامه‌نویسی مختلف روش‌های متنوعی وجود دارد که زبان اسکالا از دو عبارت while و for و do-while پشتیبانی می‌کند. در شکل ۱۴ فلوچارت کارکرد حلقه را مشاهده می‌فرمایید. توجه داشته باشید که انواع حلقه‌ها هیچ تفاوتی با هم نداشته و هر کدام قابل تغییر به دیگری هستند. البته این امکان وجود دارد که کاری با یک نوع حلقه خاص سریع‌تر با تعداد دستورات کم‌تری انجام شود.



تصویر ۱۴ فلوچارت حلقه، منبع: javaPoint

در ادامه می‌خواهیم مثال شمارش و چاپ اعداد تا یک بازه خاص توسط هر سه نوع حلقه را مورد بررسی قرار دهیم (تصاویر ۱۵ تا ۱۸).

```
object LoopOne {
  def main(args: Array[String]){
    var number = 10
    while (number > 0){
      print(number + " ")
      number -=1
    }
  }
}
```

تصویر ۱۵ حلقه while در زبان اسکالا

Output:

10 9 8 7 6 5 3 2 1

سوال) اگر مثال تصویر ۱۵ را با استفاده از val دوباره نویسی کنیم، آیا نتیجه با نتیجه این برنامه یکسان خواهد بود؟

با توجه به مثال بالا اگر بخواهیم یک الگوی کلی برای تعریف حلقه while در زبان اسکالا تعریف نماییم، به ترتیب زیر خواهد بود:

while (شرط) {

بدنه حلقه

}

یعنی برای نوشتن یک حلقه بی‌نهایت کافیست که شرط حلقه همیشه درست باشد یعنی از عباراتی شبیه به "0 <- 1" یا "true" و ... استفاده نماییم. نمونه‌ای از حلقه بی‌نهایت را در تصویر ۱۶ مشاهده می‌نمایید.

```scala
object Infinity {
  def main(args: Array[String]){
    var number = 0
    while (true){
      println(number)
      number +=1
    }
  }
}
```

تصویر ۱۶ حلقه بی‌نهایت در اسکالا

اگر برنامه موجود در تصویر ۱۶ را اجرا نمایید، مشاهده خواهید کرد که تا اعداد بسیار بزرگ را چاپ می‌کند (برای توقف برنامه از ctrl + c استفاده کنید و برنامه تصحیح شده را در تصویر ۱۷ مشاهده فرمایید.

```
object DoWhile {
  def main(args: Array[String]){
    var number = 0
    do {
      println(number)
      number -=1
    } while(number>0)
  }
}
```

تصویر ۱۷ حلقه do-while در اسکالا

Output:

0

حال شاید برای شما سؤال پیش بیاید که چه تفاوتی بین دو حلقه while و do-while وجود دارد؟ حلقه‌های do-while در هر برنامه حداقل یک بار اجرا می‌شوند، اما در حلقه‌های while این امکان وجود دارد که حلقه اجرا نشود. برای درک بهتر این تفاوت بهتر است که

مثال موجود در تصویر ۱۷ را با حلقه while بازنویسی نمایید. در تصویر ۱۸ نحوه استفاده از حلقه for در زبان اسکالا را مشاهده می‌کنید.

```
object DoWhile {
  def main(args: Array[String]){
   for (i <- 1 to 10){
    print(i + " ")
   }
  }
}
```

تصویر ۱۸ حلقه for در زبان اسکالا

Output:

1 2 3 4 5 6 7 8 9 10

یکی از تفاوت‌های واضح بین اسکالا و جاوا در حلقه‌های for می‌باشد. در اسکالا فرم کلی حلقه‌ها به صورت زیر خواهد بود. یکی از نقاط قوت اسکالا قابلیت‌های بسیار زیاد آن در قسمت حلقه‌ها می‌باشد.

```
for ( I ← شروع to انتها ) {
        بدنه حلقه
}
```

کلمه کلید to اعداد را در یک بازه مشخص تولید می‌کند. به جای to می‌توانید از کلمه کلید until استفاده کنید. در to از کلمه شروع تا خود خاتمه را شامل می‌شود، اما در until از شروع تا یک عدد مانده به خاتمه را شامل می‌شود. یعنی اگر مثال تصویر ۱۸ توسط until بازنویسی شود، خروجی آن به ترتیب زیر خواهد بود:

Output:

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

5 ساختمان داده‌ها

اسکالا از نظر تعداد ساختمان داده زبان بسیار قدرتمندی است و شامل تعداد زیاد ساختمان داده مختلف با قابلیت‌های مختلف است. ساختمان داده‌ها می‌توانند قابل تغییر[5] (انجام عملیات بروی آن‌ها سبب تغییر مقدار اصلی ساختار می‌شود) یا غیرقابل تغییر[6] (انجام عملیات بروی آن‌ها مقدار اصلی ساختار را تغییر نمی‌دهد مانند رشته‌ها) باشند. ساختمان داده‌ها به دو دسته سست (Lazy) و محکم (Strict) دسته‌بندی می‌شوند. ساختمان داده‌ها Lazy به ساختمان داده‌هایی گفته می‌شود که پیچیدگی زمانی و مکانی آن‌ها، تا زمانی که آن‌ها را در عملیاتی خاص قرار ندهید صفر است. یعنی در حالت عادی هیچ سرباری را

---

[5] mutable

[6] immutable

برای پردازش‌گر و حافظه ایجاد نمی‌کنند. در ادامه ساختمان داده‌های موجود در زبان اسکالا را مورد بررسی قرار می‌گیرند.

5.1 آرایه

آرایه ساختمان داده‌ای است که برای ذخیره داده‌های هم نوع به کار می‌رود. طول آرایه معمولاً ثابت است. اگر طول آرایه متغیر باشد یا بتوان در آن‌ها عناصر غیر هم نوع نیز ذخیره نمود به آن لیست گفته می‌شود. آرایه‌ها به دو دسته آرایه تک بعدی و آرایه دوبعدی تقسیم می‌شوند. دلیل بوجود آمدن آرایه جلوگیری از تکرار زیاد متغیرهای تک مقداری بود. به عنوان مثال به جای ذخیره اسامی یک کلاس در متغیرهای جداگانه همه آن را در یک ساختار داده واحد ذخیره می‌نماییم.

برای تعریف آرایه می‌توانید به شیوه‌های زیر عمل نمایید:

```
var classNames: Array[String] = new Array[String](3)
```

در اینجا آرایه‌ای یک بعد به طول سه ایجاد شده است. یعنی می‌توانید سه اسم را در آن ذخیره نمایید.

```
Var className = new Array[String](3)
```

همانند تعریف بالا است با این تفاوت که نوع متغیر ذکر نشده است.

در زبان اسکالا مانند بیشتر زبان‌های برنامه‌نویسی اندیش آرایه‌ها از صفر شروع می‌شود. برای مقداردهی آرایه‌ها می‌توانید به روش‌های زیر عمل نمایید:

```
className(0) = "Ali Ghaderi"
className(1) = "Behnam Amiri"
className(2) = "Yasin Amini"
```

یا همه مقداردهی‌ها را به صورت یک‌جا انجام دهید:

```
 className = Array("Ali Ghaderi", "Behnam Amiri", "Yasin Amini")
```

برای دسترسی به عناصر یک آرایه نیز می‌توانید به صورت زیر عمل نمایید:

```
for (i <- 0 to (className.length - 1)){
        println(className(i))
}
```

یا به روشی که در زبان‌های دیگر به آن foreach می‌گویند عمل نمایید:

```
for (i <- className) {
        println(i)
}
```

در تصویر ۱۹ برنامه جامعی را از آرایه شامل چاپ آرایه، پیدا کردن مجموع عناصر آرایه و پیدا کردن بیشترین و کمترین مقدار در آرایه‌ها را مشاهده می‌نمایید.

```
object Array {
   def main(args: Array[String]) {
      var myList = Array(1,2,3,4,5,6,7,8,9,10)

      // چاپ عناصر آرایه
      for ( x <- myList ) {
         println( x )
      }

      // محاسبه مجموع عناصر آرایه
      var total = 0

      for ( i <- 0 to (myList.length - 1)) {
         total += myList(i)
      }
      println("Total is " + total)

      // پیدا کردن عنصر بیشینه و کمینه
      var max = myList(0)
      var min = myList(0)
      for ( i <- 1 to (myList.length - 1) ) {
         if (myList(i) > max) max = myList(i)
         if (myList(i) < min) min = myList(i)
      }

      println("Max is " + max)
      println("Min is " + min)
   }
}
```

تصویر ۱۹ کار با آرایه‌ها در اسکالا

Output:

1

.

.

10

Total is 55

Max is 10

Min is 1

یکی از تفاوت‌های آرایه‌ها در جاوا و زبان اسکالا در این است که در اسکالا امکان تعریف آرایه با عناصر با نوع مختلف وجود دارد. این ویژگی به دلیل جامع بودن نوع‌های اسکالا است که اگر یک در یک آرایه عناصر با نوع مختلف وجود داشته باشد به آن نوع Any نسبت می‌دهد.

Var person = Array("Yasin", 23, 75.2, false)

همانگونه که در مثال بالا مشاهده می‌کنید به یک آرایه عناصری از انواع داده‌ها را نسبت داده‌ایم.

در بسیاری از شرایط‌ها شما نیاز به ایجاد آرایه‌های دوبعدی دارید. آرایه دوبعد، همان آرایه تک بعدی است با این تفاوت که عناصر آن نیز هر کدام آرایه هستند. برای ایجاد آرایه‌های

دوبعدی در زبان اسکالا باید ابتدا کتابخانه Array._ را به پروژه خود اضافه نمایید. در مثال پایین نحوه ایجاد یک آرایه ۴ در ۴ را مشاهده می‌کنید.

```
import Array._
```

برای استفاده از آرایه‌های دوبعدی باید ابتدا این کتابخانه را پروژه اضافه کنید.

```
var matrix = ofDim[Int](3,3)
```

در تصویر ۲۰ نحوه ایجاد و نمایش آرایه‌های دوبعدی یا ماتریس را مشاهده می‌کنید.

```scala
import Array._

object Matrix {
   def main(args: Array[String]) {
      var myMatrix = ofDim[Int](3,3)

      // ساخت ماتریس
      for (i <- 0 to 2) {
         for ( j <- 0 to 2) {
            myMatrix(i)(j) = j;
         }
      }

      // چاپ عناصر موجود در ماتریس
      for (i <- 0 to 2) {
         for ( j <- 0 to 2) {
            print(" " + myMatrix(i)(j));
         }
         println();
      }
   }
}
```

تصویر ۲۰ کار با ماتریس‌ها در اسکالا

برای ایجاد آرایه‌های عدد می‌توانید از range استفاده نمایید. range مانند یک شمارنده عمل می‌کند و سه ورودی می‌تواند داشته باشد. ورودی اول برای شروع، ورودی دوم برای عدد انتهایی و ورودی سوم تعیین کننده گام شمارش است (اگر به تابع range دو ورودی ارسال گردد، این تابع مقدار گام را به طور پیش‌فرض برابر یک در نظر می‌گیرد).

```
var array = range(1, 10) => Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
var array = range(1,10,2) => Array[Int] = Array(1, 3, 5, 7, 9)
```

برای آشنایی بیشتر در مورد آرایه‌ها می‌توانید متدهای کلاس _.Array را مستندات اسکالا مطالعه فرمایید.

5.2 لیست

ساختمان داده‌ای مشابه با آرایه که از لینک لیست ساخته شده است. همه عناصر موجود در لیست دارای نوع یکسانی هستند (توجه داشته باشید که لیست‌ها همانند آرایه می‌توانند دارای عناصر از جنس رشته، عدد، عدد ممیزدار و ... باشند. اما در اسکالا به دلیل ساختار سلسله مراتبی نوع‌های داده، همه آن‌ها یک نوع کلی که همه را پوشش دهد پیدا می‌کنند).

لیست یک ساختمان داده غیرقابل تغییر است و متدهای آن سبب تغییر مقادیر لیست نمی‌شوند. برای تعریف لیست از الگوی زیر استفاده نمایید.

val/var نام لیست : List[نوع عناصر لیست] = List(عناصر لیست)

مانند

```
val fruits : List[String] = List("apples", "oranges", "pears")
```

یا

```
val fruits = "apples" :: ("oranges" :: ("pears" :: Nil))
```

در روش دوم، در آخر حتماً باید Nil گذاشته شود.

```
val numbers : List[Int] = List(1,2,3,4,5)
val nums = 1 :: (2 :: (3 :: (4 :: Nil)))
```

```
val empty : List[Nothing] = List()
val empty = Nil
```

ایجاد لیست دوبعدی

```
val dim: List[List[Int]] =
  List(
    List(1, 0, 0),
```

```
    List(0, 1, 0),
    List(0, 0, 1)
  )

val dim = (1 :: (0 :: (0 :: Nil))) ::
          (0 :: (1 :: (0 :: Nil))) ::
          (0 :: (0 :: (1 :: Nil))) :: Nil
```

برای دسترسی به عناصر لیست مانند آرایه‌ها از پرانتز استفاده می‌شود. اندیس لیست نیز از صفر شروع می‌شود (تصویر ۲۱).

```scala
object ListOperation {
   def main(args: Array[String]) {
      val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
      val nums = Nil

      println( "Head of fruit : " + fruit.head )
      println( "start of fruit : " + fruit(0) )
      println( "Tail of fruit : " + fruit.tail )
      println( "last of fruit : " + fruit(fruit.length-1) )
      println( "Check if fruit is empty : " + fruit.isEmpty )
      println( "Check if nums is empty : " + nums.isEmpty )
   }
}
```

تصویر ۲۱ کار با لیست در اسکالا

Output:

```
Head of fruit : apples

start of fruit : apples

Tail of fruit : List(oranges, pears)

last of fruit : pears

Check if fruit is empty : false

Check if nums is empty : true
```

از متد isEmpty برای بررسی خالی بودن یا نبودن لیست استفاده می‌شود. اگر لیست خالی باشد مقدار false و اگر لیست خالی نباشد مقدار true برمی‌گرداند. متد head عنصر اول لیست و متد tail بقیه عناصر لیست (غیر از عنصر اول را برمی‌گرداند). در لیست بالا مقدار fruit(0) = "cherry" به درستی اجرا می‌شود؟ همانگونه که در ابتدای معرفی لیست‌ها اشاره گردید، لیست‌ها غیرقابل تغییر هستند. یعنی اجرای دستور گفته شده سبب ایجاد خطا می‌شود.

برای اتصال لیست از ::: یا :::. و یا متد concat استفاده می‌شود. عملگر ::: لیست دوم را به انتهای لیست اول اضافه می‌کند. :::. لیست اول را به انتهای لیست دوم اضافه می‌کند. در متد concat نیز هر کدام از لیست‌ها اول نوشته شوند، لیست دوم به آن اضافه می‌گردد (تصویر ۲۲).

```scala
object ListOperation {
   def main(args: Array[String]) {
      val nums1: List[Int] = List(1,2,3,4,5)
      val nums2: List[Int] = List(6,7,8,9,10)

      println(nums1:::nums2)
      println(List.concat(nums1,nums2))
      println(nums1.:::(nums2))
      println(List.concat(nums2,nums1))
   }
}
```

تصویر ۲۲ کار با متدهای لیست

Output: List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

List(6, 7, 8, 9, 10, 1, 2, 3, 4, 5)

List(6, 7, 8, 9, 10, 1, 2, 3, 4, 5)

متد reverse سبب معکوس شدن ترتیب عناصر لیست می‌گردد.

val fruits = "apple":: ("orange" :: ("pears":: Nil))

println(fruits.reverse) => List(pears, oranges, apples)

متد fill به تعداد اعداد پاس شده به آن یک عنصر را تکرار می‌کند.

Val nums = List.fill(10)(2)

print(nums) => List(2, 2, 2, 2, 2, 2, 2, 2, 2, 2)

متد tabulate نیز همانند fill برای ایجاد لیست استفاده می‌شود. ورودی اول نشان‌دهنده اندازه لیست و ورودی دوم تابعی است که لیست بر مبنای آن ساخته می‌شود.

val mul = List.tabulate(4,5)(_ * _)

print(mul) => List(List(0, 0, 0, 0, 0), List(0, 1, 2, 3, 4), List(0, 2, 4, 6, 8), List(0, 3, 6, 9, 12))

برای درک بهتر متدهای گفته شده در یک مثال واحد در تصویر ۲۳ آورده شده است.

```
object ListOperation {
   def main(args: Array[String]) {
      val nums1: List[Int] = List(1,2,3,4,5)
      val nums2 = List.tabulate(10)(_*4)
      val nums3 = List.fill(5,4)("Hello")

      println(nums1.reverse)
      println(nums2)
      println(nums3)
   }
}
```

تصویر ۲۳ کار با متدهای لیست

Output:

List(5, 4, 3, 2, 1)

List(0, 4, 8, 12, 16, 20, 24, 28, 32, 36)

List(List(Hello, Hello, Hello, Hello), List(Hello, Hello, Hello, Hello), List(Hello, Hello, Hello, Hello), List(Hello, Hello, Hello, Hello), List(Hello, Hello, Hello, Hello))

لیست‌ها نیز همانند آرایه‌ها دارای تعداد بسیار زیادی متد است که بررسی تک تک آن‌ها خارج از حجم این کتابچه می‌باشد. برای آشنایی با متدهای بیشتر به مستندات و منابع معرفی شده در قسمت منابع و مراجع، مراجعه فرمایید.

5.3 Set

مجموعه‌ها ویژگی‌هایی مشابه با مجموعه‌های ریاضی را دارند. در مجموعه‌ها امکان تعریف عنصر تکراری وجود ندارد. مجموعه‌های می‌توانند غیرقابل تغییر یا قابل تغییر باشند. در هنگام تعریف مجموعه‌ها اسکالا به صورت پیش‌فرض آن‌ها را غیرقابل تغییر در نظر می‌گیرد. برای استفاده از مجموعه‌های قابل تغییر باید از کلاس scala.collection.mutable.Set استفاده شود. الگوی تعریف مجموعه‌ها به صورت زیر می‌باشد:

var/val نام مجموعه : Set[نوع مجموعه] = Set(عناصر مجموعه)

همانند تمامی متغیرهای موجود در اسکالا گذاشتن نوع مجموعه نیز اختیاری است.

```
Var nums : Set[Int] = Set(1,2,3,4,1,2,3,4)
print(nums) => Set(1,2,3,4)
```

عملیات‌های head، tail، isEmpty همانند لیست‌ها می‌باشد. برای اتصال دو مجموعه نیز از عملگرهای ++ و ++. استفاده می‌شود. عملگر ++، مجموعه دوم را به ابتدای مجموعه اول متصل می‌کند و ++.، مجموعه اول را به انتهای مجموعه اول وصل می‌نماید. اغلب متدهای بین ساختمان داده‌ها مشترک است. به عنوان مثال در مجموعه‌ها نیز همانند لیست‌ها متدهایی مانند min و max به ترتیب کوچک‌ترین و بزرگ‌ترین عنصر مجموعه و لیست را برمی‌گردانند. در مجموعه‌ها همانند مجموعه‌های ریاضی امکان انجام عملیات‌های اجتماع، اشتراک، تفاضل و ... وجود دارد. برای درک بهتر در مورد مجموعه‌ها با مثالی جامعی مطالب گفته شده در مورد مجموعه‌ها را جمع‌بندی می‌نماییم (تصویر ۲۴).

```
object SetOperation {
   def main(args: Array[String]) {
      val nums1: Set[Int] = Set(1,2,3,4,5)
      val nums2: Set[Int] = Set(1,2,3,4,5,6,7,8,9,10)
      val nums3: Set[Int] = Set(6,7,8,9,10)
      val nums4: Set[Int] = Set()


      println( "Head of nums1 : " + nums1.head )
      println( "Tail of nums1 : " + nums1.tail )
      println( "Check if nums1 is empty : " + nums1.isEmpty )
      println( "Check if nums4 is empty : " + nums4.isEmpty )

      println(nums1++nums3)
      println(nums1.++(nums3))

      println(nums2.max)
      println(nums2.min)

      // & = intersect
      println(nums1.&(nums2))
      // | = union
      println(nums1.|(nums2))

   }
}
```

تصویر ۲۴ کار با set در اسکالا

Output:

Head of nums1 : 5

```
Tail of nums1 : Set(1, 2, 3, 4)
Check if nums1 is empty : false
Check if nums4 is empty : true
Set(5, 10, 1, 6, 9, 2, 7, 3, 8, 4)
Set(5, 10, 1, 6, 9, 2, 7, 3, 8, 4)
10
1
Set(5, 1, 2, 3, 4)
Set(5, 10, 1, 6, 9, 2, 7, 3, 8, 4)
```

5.4 Map

Map یکی از ساختمان داده‌های اسکالا است که داده‌ها در آن به صورت جفت کلید/مقدار ذخیره می‌شوند (مشابه با dictionary در پایتون و hash table در جاوا). در این ساختمان داده کلیدها منحصر به فرد هستند. این ساختمان داده نیز می‌تواند قابل تغییر یا غیرقابل تغییر باشد. در هنگام تعریف Map کامپایلر اسکالا به صورت پیش‌فرض آن را غیرقابل تغییر در نظر می‌گیرد. برای استفاده از Map قابل تغییر باید از کتابخانه scala.collection.mutable.Map استفاده نمایید. الگوی تعریف Map به صورت زیر می‌باشد.

val/var نام متغیر : Map[نوع کلید, نوع مقدار] = Map(عناصر Map)

```
val persons = Map("yasin" -> 23, "ali" -> 45)
```

برای دسترسی به مقادیر می‌توانید در داخل پرانتز نام متغیر کلید را وارد نمایید.

```
print(persons( "yasin"))    => ۲۳
```

در ساختمان داده Map می‌توانید با استفاد از متد isEmpty خالی یا خالی نبودن لیست را بررسی کنید و متد contains برای بررسی وجود یا عدم وجود یک کلید خاص مورد استفاده قرار می‌گیرد. متدهای keys و values به منظور نمایش کلیدها و مقادیر هستند و برای اتصال دو Map همانند Set عمل کنید. در تصویر ۲۵ مثالی از کار با ساختمان داده Map را مشاهده می‌نمایید.

```scala
object MapOperation {
   def main(args: Array[String]) {
      val persons1: Map[String, Int] = Map("yasin" -> 23, "ali" -> 45)
      val persons2: Map[String, Int] = Map("amir" -> 63, "hasan" -> 55)
      val persons3 = Map()

      println(persons1.keys)
      println(persons1.values)
      println(persons1.isEmpty)
      println(persons3.isEmpty)

      println(persons1++persons2)
      println(persons1.++(persons2))

      println(persons1.contains("yasin"))
      println(persons2.contains("yasin"))

      persons1.keys.foreach { _ =>
        println("keys is = " + _)
        println("values is = " + persons1(_))
   }
}
```

تصویر ۲۵ کار با Map در اسکالا

Output:

Set(yasin, ali)

MapLike.DefaultValuesIterable(23, 45)

false

true

Map(yasin -> 23, ali -> 45, amir -> 63, hasan -> 55)

Map(yasin -> 23, ali -> 45, amir -> 63, hasan -> 55)

true

```
false
keys is = yasin
values is = 23
keys is = ali
values is = 45
```

5.5 Tuple

تاپل ساختمان داده‌ای که برخلاف لیست و آرایه می‌تواند عناصر با انـواع مختلـف را در خـود ذخیره نماید. الگوی تعریف تاپل به صورت زیر می‌باشد:

var/val نام متغیر = new Tuple[تعداد عناصر] (عناصر تاپل)

```
var tuple = (1, 3.14, "hello")
```

```
var tuple = new Tuple3(1, 3.14, "hello")
```

برای دسترسی به عناصر تاپل نیز به صورت زیر عمل کنید:

tuple._شماره عنصر

اندیس عناصر تاپل از یک شروع می‌شود. در حال حاضر تعداد عناصـر تاپـل از ۲۲ نمی‌توانـد بیشتر باشد و تعداد بیشتر از ۲۲ سبب ایجاد خطا می‌شود.

برای پیمایش تاپل یا تبدیل آن به Iterator از متد productIterator استفاده می‌شود.

```
tuple.productIterator.foreach{ item => println("Value = " + item ) }
```

برای جابجایی دو عنصر تاپل از متد swap استفاده می‌گردد.

```
tuple.swap
```

برای جمع‌بندی ساختمان داده تاپل به تصویر ۲۶ توجه نمایید.

```
object TupleOperation {
   def main(args: Array[String]) {
      val tuple = (1,2,3,4,5)
      val tuple2 = ("hello", "world")
      val sum = tuple._1 + tuple._2 + tuple._3 + tuple._4

      println( "Sum of elements: "  + sum )
      tuple.productIterator.foreach {
        item => println("item value is: "+ item)
      }
      println("Swapped Tuple: " + tuple2.swap)
   }
}
```

تصویر ۲۶ کار با تاپل در اسکالا

Output:

```
scala Sum of elements: 10
item value is: 1
item value is: 2
item value is: 3
item value is: 4
item value is: 5
Swapped Tuple: (world,hello)
```

5.6 Iterator

Iterator ساختمان داده نیست بلکه راهی برای دسترسی به عناصر یک ساختمان به صورت یک به یک است. دو متد اساسی در Iterator عملیات‌های next و hasNext هستند. متد next به عنصر بعدی ساختمان داده می‌رود و حالت Iterator را مقداردهی مجدد می‌کند و متد hasNext بررسی می‌کند که آیا عنصر بعدی وجود دارد یا خیر. برای آشنایی با تعدادی از عملیات‌ها و متدهای Iterator به تصویر ۲۷ توجه نمایید.

```scala
object IteratorOperation {
   def main(args: Array[String]) {
      var iterator = Iterator(1,2,3,4,5,6,7,8,9,10)

      println("Maximum valued element : " + iterator.max )
      iterator = Iterator(1,2,3,4,5,6,7,8,9,10)
      println("Minimum valued element : " + iterator.min )
      iterator = Iterator(1,2,3,4,5,6,7,8,9,10)
      println("Value of iterator size :  " + iterator.size )
      iterator = Iterator(1,2,3,4,5,6,7,8,9,10)
      println("Value of iterator length :  " + iterator.length )
   }
}
```

تصویر ۲۷ Iterator در اسکالا

Output:

Maximum valued element : 10

Minimum valued element : 1

Value of iterator size : 10

Value of iterator length : 10

علت تکرار هر بار مقداردهی به متغیر این است که متدهای اجرا شده چون هر بار Iterator عناصر را تا آخر پیمایش می‌کند، به آخر ساختمان داده رسیده و عنصر دیگری وجود نخواهد داشت و برنامه با خطا مواجه می‌شود. برای جلوگیری از این خطا هر بار متغیر را مقداردهی

کرده‌ایم. برای پیمایش عناصر Iterator نیز از متدهای next و hasNext V استفاده می‌کنیم (تصویر ۲۸).

```
object IteratorOperation {
   def main(args: Array[String]) {
      var iterator = Iterator(1,2,3,4,5,6,7,8,9,10)

      while(iterator.hasNext){
        print(iterator.next() + "\t")
      }
   }
}
```

تصویر ۲۸ پیمایش ساختمان داده با استفاده از Iterator

Output:

1 2 3 4 5 6 7 8 9 10

6 توابع

به مجموعه‌ای از کدها که هدف خاصی را دنبال و کاری را انجام می‌دهند، تابع گفته می‌شود. از توابع برای عدم تکرار دستورات در چندین جای برنامه استفاده می‌شود. در حقیقت به جای نوشتن یک مجموعه کد و تکرار آن در چندین جای برنامه، آن را در یک تابع واحد نوشته و چندین بار آن را فراخوانی می‌کنیم. همانطور که در ویژگی‌های اسکالا نیز به آن اشاره گردید، اسکالا یک زبان شی‌گرا است. پس در اسکالا هم تابع وجود دارد و هم متد.

به تابعی که داخل یک کلاس نوشته می‌شود، متد می‌گویند. خود تابع یک شی کامل است که می‌تواند به یک متغیر نیز انتساب شود.

فرمت کلی توابع در زبان برنامه اسکالا به ترتیب زیر می باشد (تصویر ۲۹).

```
Def نام تابع (پارامترهای ورودی): نوع خروجی = {
    بدنه تابع
}
```

```
object Function {
    def add(x:Int , y : Int ) : Int = {
        return (x+y)
    }

    def main(args : Array[String]){
        print(add(12 , 13))
    }
}
```

تصویر ۲۹ تعریف تابع جمع‌کننده اعداد در زبان اسکالا

Output:

25

در تصویر ۲۹ اگر در تابع add از کلمه کلیدی return هم استفاده نکنید، مقدار به تابع اصلی برگردانده می‌شود. به عبارتی گذاشتن کلمه کلید return در اسکالا اختیاری است و اگر گذاشته نشود، آخرین مقدار موجود در بدنه تابع به عنوان خروجی بازگردانده می‌شود (تصویر ۳۰).

```
object function {
    def add(x:Int , y : Int )  = {
        x+y
    }
    def main(args : Array[String]){
        print(add(12 , 13))
    }
}
```

تصویر ۳۰ تابع حاصل جمع اعداد بدون استفاده از return

برای توابعی که مقداری برنمی گردانند می‌توانید = را از تعریف تابع حذف نمایید مانند تابع اصلی برنامه یا نوع آن را Unit معرفی نمایید که معادل void دیگر زبان‌ها برنامه نویسی است. اسکالا در فراخوانی توابع از فراخوانی با مقدار[7] استفاده می‌نماید. یعنی پارامترهای فرستاده شده به تابع ابتدا مورد ارزیابی قرار می‌گیرند، سپس حاصل ارزیابی به تابع ارسال می‌گردد. مثلاً همان تابع اضافه کردن دو عدد در تصویر شماره را در نظر بگیرید که آن را به شکل زیر فراخوانی نماییم.

add(12+3, 14+1)

[7] Call by value

در اینصورت ابتدا جمع‌های درون فراخوانی تابع انجام شده و سپس ۱۵ و ۱۵ به تابع فرستاده می‌شوند. در روش فراخوانی با نام[8] این اتفاق صورت نمی‌گیرد و پارامترهای ارسال شده به تابع تا رسیدن به داخل تابع مورد ارزیابی قرار نمی‌گیرند. در مثال بالا به همان صورت نوشته شده، به تابع ارسال می‌شود و تنها زمانی که لازم باشد بروی آن محاسبه‌ای انجام شود، مورد ارزیابی قرار می‌گیرند. برای فراخوانی متدهای به صورت فراخوانی با نام در زبان برنامه‌نویسی اسکالا مکانیزم‌هایی که تعبیه شده است (تصویر ۳۱).

[8] Call by name

```scala
object Demo {
   def main(args: Array[String]) {
     println(mul(3, add(2, 1)))
   }

   def mul(x: Int, y: => Int) : Int = {
     println("mul")
     x * y
   }

   def add(x: Int, y: Int): Int = {
    println("add")
    x + y
   }
}
```

تصویر ۳۱ فراخوانی با نام در اسکالا

Output:

mul

add

9

در این مثال ابتدا تابع ضرب (mul) اجرا می‌گردد، اما چون مقدار y مشخص نمی‌باشد و باید مشخص گردد تابع جمع (add) فراخوانی شده و سپس به ضرب انجام می‌شود. همانگونه که مشاهده می‌کنید در مثال در هنگام اجرا ابتدا تابع mul اجرا می‌شود، در صورتی که در روش فراخوانی با مقدار باید ابتدا تابع جمع و مقدار add(2,1) محاسبه می‌شد و سپس تابع ضرب. این مثال نشان می‌دهد که می‌توانید با تغییر رویکرد برنامه از اسکالا برای کاربردهای فراخوانی با نام نیز استفاده نمایید (در این مثال y: => Int سبب فراخوانی با نام می‌شود). به این توابع که می‌توانند تابع دیگری را به عنوان ورودی بگیرند، high-order function گفته می‌شود (مانند تابع mul). به تابع (y: => Int) نیز توابع بی‌نام[9] گفته می‌شود. این توابع یکی از اصول اساسی برنامه‌نویسی تابعی می‌باشند. در تصویر ۳۲ چند نمونه از توابع بی‌نام را مشاهده می‌نمایید. ایده این توابع از اصل لامبدا حسابان آمده است.

[9] Anonymous

```
object  Functional {
   def main(args: Array[String]) {
      val mul = (x: Int, y: Int) => x*y
      println(mul(2,3))

      // تابع بی نام و بدون پارامتر ورودی
      val username = () => "linux"
      println(username())
   }
}
```

تصویر ۳۲ تابع بدون نام در اسکالا

Output:

6

linux

در تصویر ۳۳، نمونه‌ای از high-order function را مشاهده می‌نمایید.

```
object  Functional {
   def main(args: Array[String]) {
      println(s"2 + 3 = ${operation(add, 2,3)}")
      println(s"2 - 3 = ${operation(sub, 2,3)}")
      println(s"2 * 3 = ${operation(mul, 2,3)}")
      println(s"2 / 3 = ${operation(div, 2,3)}")
      println(s"2 % 3 = ${operation(mod, 2,3)}")
   }

  def operation(func: (Int, Int) => Int, x: Int, y: Int) = func(x, y)
  def add (x: Int, y: Int) = x+y
  def sub (x: Int, y: Int) = x-y
  def mul (x: Int, y: Int) = x*y
  def div (x: Int, y: Int) = x/y
  def mod (x: Int, y: Int) = x%y
}
```

تصویر ۳۳ high-order در اسکالا

Output:

5

-1

6

0

2

در این مثال پارامتر func که پارامتر ورودی operation است، پارامتری است که می‌توانید به آن تابع مورد نظر خود را فرستاده و جواب را دریافت نمایید. تابع operation در این مثال نقش تابع high-order را بازی می‌کند.

- **تابع با تعداد ورودی نامعین**

تابع جمع (add) در مثال‌های بالا برای دو ورودی تعریف شده است. اگر به جای دو ورودی سه ورودی برای تابع ارسال شود، چه اتفاقی می‌افتد؟ در صورت انجام این کار برنامه با خطا مواجه می‌شود. برای جلوگیری از این خطا، در توابعی که باید بتوانند به تعداد دلخواه متغیر بپذیرند به روش زیر عمل می‌شود (تصویر ۳۴).

```
object FlexibleFunction {
  def main(args: Array[String]){
    println(add(12,1,13))
  }

  def add(a: Int*): Int = {
    var sum: Int = 0
    for (i <- a){
      sum +=i
    }
    sum
  }
}
```

تصویر ۳۴ تابع با ورودی‌های نامعین

Output: 26

پس برای اینکه یک تابع بتواند چندین ورودی را بپذیرد از الگوی زیر استفاده نمایید.

نوع داده* => Int*, Long*, String*, ...

- توابع بازگشتی

یکی از مهم‌ترین اصول برنامه‌نویسی تابعی، توابع بازگشتی هستند. تابع بازگشتی به تابعی گفته می‌شود که خودش را فراخوانی می‌کند مانند تابع فاکتوریل یا فیبوناچی. در تصویر ۳۵، پیاده‌سازی تابع فاکتوریل با استفاده از تابع بازگشتی را مشاهده می‌نمایید.

```
object Factorial {
   def main(args: Array[String]) {
      for (i <- 1 to 10)
         println(i + "! = " + factorial(i) )
   }

   def factorial(n: BigInt): BigInt = {
      if (n <= 1)
         1
      else
        n * factorial(n - 1)
   }
}
```

تصویر ۳۵ تابع فاکتوریل

کلاس BigInt کلاسی در جاوا و اسکالا است که برای اعداد بزرگ به کار می‌رود. برای درک بهتر توابع بازگشتی برنامه سری فیبوناچی را در تصویر ۳۶ تحلیل نمایید.

```
object Fibonacci {
   def main(args: Array[String]) {
      for (i <- 1 to 10)
         println(i + "! = " + factorial(i) )
   }

   def fibonacci(num: BigInt): BigInt = {
      if (num==0 || num==1)
        1
      else
        fibonacci(num-1) + fibonacci(num-2)
   }
```

تصویر ۳۶ تابع فیبوناچی

➢ توابع تودرتو

به تابعی که درون یک تابع دیگر تعریف می‌شود، تابع تودرتو می‌گویند. توابع داخل یک تابع دیگر می‌توانند به متغیرهای تابع بیرونی دسترسی داشته باشند.

در زبان برنامه‌نویسی اسکالا امکان تعریف توابع تودرتو نیز وجود دارد. هدف از تعریف توابع تودرتو یکپارچه‌سازی بخشی از عملیات‌ها در یک تابع واحد است. به عنوان مثال می‌خواهیم فاصله اقلیدسی دو بردار را از یک یکدیگر به دست بیاوریم. برای محاسبه فاصله اقلیدسی دو بردار از فرمول تصویر ۳۷ استفاده می‌شود:

تصویر ۳۷ فرمول محاسبه فاصله اقلیدسی دو بردار

برنامه این فرمول در اسکالا مطابق تصویر ۳۸ خواهد بود.

```scala
object NestedFunction {
  def distance(x0: Int, y0: Int, x1: Int, y1: Int)= {

    def difPow(a: Int, b: Int)= {
      (a-b) * (a-b)
    }
    Math.sqrt(difPow(x0, x1) + difPow(y0, y1))
  }

  def main(args: Array[String]){
    println(distance(3,3,9,11))
  }
}
```

تصویر ۳۸ پیاده‌سازی فاصله اقلیدسی با استفاده از مفهوم توابع تودرتو

Output:

10

همانطور که در برنامه محاسبه فاصله اقلیدسی مشاهده نمودید، تابع توان اختلاف دو بردار تابعی است که در درون تابع دیگر قرار گرفته است. با قرار دادن تابع در بیرون این تابع نیز نتیجه همان خواهد بود، اما این کار سبب می‌شود که کدهای مرتبط با یک تابع خاص بهتر سازمان‌دهی شوند.

در کد بالا چندین نکته وجود دارد که در اینجا آن‌ها را مورد بررسی قرار می‌دهیم. در ابتدا کتابخانه Math کتابخانه‌ای است که برای انجام عملیات‌های ریاضی از آن استفاده می‌شود و

نیازی به افزودن آن به برنامه وجود ندارد. اما در صورت نیاز می‌توانید با دستور import آن را به برنامه خود اضافه کنید. سعی کنید که توابع بالا را بدون گذاشتن مساوی در جلوی تابع بازنویسی نمایید، آیا نتیجه اجرا همین خواهد بود؟

علامت مساوی در جلو توابع اسکالا نقش تعیین کننده در بازگرداندن یا عدم بازگرداندن مقدار از تابع دارد. یعنی اگر در جلو یک تابع = گذاشته نشود، این تابع مشابه با توابع void در سایر زبان‌های برنامه‌نویسی رفتار می‌کند.

در ادامه با معرفی چند چالش برنامه بالا را کامل‌تر می‌نماییم. یکی از چالش‌های که ما ممکن است، در حین پیاده‌سازی با آن روبرو شویم، این است که ترتیب آرگومان‌های ورودی را اشتباده وارد کنیم. در اینصورت نتیجه برگردانده شده اشتباه خواهد بود و ممکن است ما در درستی یا عدم درستی برنامه نوشته شده دچار تردید نماید. برای حل این چالش می‌توانیم در حین فراخوانی تابع نام هر یک از آرگومان‌های ورودی تابع را نیز مشخص نماییم. برای درک بهتر این مسأله نگاهی به کد تصویر ۳۹ بیندازید. تصور کنید ورودی ما به شکل زیر باشد:

```
distance(3,9,3,11)
```

در اینصورت خروجی تابع برابر ۲ خواهد بود که با نتیجه واقعی آن یعنی ۱۰ خیلی تفاوت دارد و این اشتباه فقط به دلیل خطای کاربر در وارد کردن ترتیب آرگومان‌های ورودی تابع روی داده است.

```
object NestedFunction {
  def distance(x0: Int, y0: Int, x1: Int, y1: Int)= {

    def difPow(a: Int, b: Int)= {
      (a-b) * (a-b)
    }
    Math.sqrt(difPow(x0, x1) + difPow(y0, y1))
  }

  def main(args: Array[String]){
    println(distance(x0 = 3, x1 = 9, y0 = 3, y1 = 11))
  }
}
```

تصویر ۴۰ برنامه محاسبه فاصله اقلیدسی با قابلیت ترتیب آرگومان‌های دلخواه

چالش بعدی که در هنگام کار با توابع ممکن است با آن روبرو شویم، این است که اگر در هنگام فراخوانی برنامه تصویر ۲۸، به تابع هیچ مقداری ندهیم تابع با خطای زیر مواجه می‌شود:

```
distance( )
```

```
error: not enough arguments for method distance: (x0: Int, y0: Int, x1: Int, y1: Int)Double.
```

برای رفع مشکل بالا نیز کافی است که به توابع یک مقدار اولیه اختصاص داده شود، در اینصورت با هیچ گونه خطایی مواجه نمی‌شویم. برنامه تصویر ۲۸ را به صورت، تصویر ۴۱ تغییر می‌دهیم.

```
object NestedFunction {
  def distance(x0: Int = 0, y0: Int = 0, x1: Int = 0, y1: Int = 0)= {

    def difPow(a: Int, b: Int)= {
      (a-b) * (a-b)
    }
    Math.sqrt(difPow(x0, x1) + difPow(y0, y1))
  }

  def main(args: Array[String]){
    println(distance(x0 = 3, x1 = 9, y0 = 3, y1 = 11))
  }
}
```

تصویر ۴۱ تنظیم مقادیر پیش‌فرض برای پارامترهای ورودی برای کارکرد بهتر برنامه

توجه داشته باشید که انتساب مقادیر پیش‌فرض به آرگومان‌های ورودی سبب می‌شود که برنامه ما ورودی‌هایی مانند ورودی زیر را نیز بپذیرد، ولی اگر همین ورودی را به توابع بالا بدهید، برنامه دچار خطا می شود.

distance(x0 = 3, x1 = 9, y0 = 3, y1 = 11)

Output:

6.0

- **توابع چند مقداری**

منظور از توابع چند مقداری توابعی است که چندین خروجی داشته باشند. در زبان‌هایی مثل جاوا و سی و سی پلاس پلاس همچنین امکانی وجود ندارد و باید از آرایه‌ها استفاده شود. اما در زبان برنامه‌نویسی اسکالا همچنین قابلیتی وجود دارد (تصویر ۴۲).

```
object MultipleReturn {
  def addMul(a:Int,b:Int):(Int,Int) = {
    (a+b , a*b)
    }
    def main(args: Array[String]) {
        var (mySum,myMul)=addMul(4,5)
        println(mySum)
        println(myMul)
    }
}
```

تصویر ۴۲ بازگرداندن چندین متغیر در تابع

➢ توابع بستار[10]

تابعی که مقدار برگردانده شده از تابع به مقادیر متغیرهای خارج از تابع بستگی دارد. به عنوان مثال تابع زیر را در نظر بگیرید که یک تابع بدون نام است که دو عدد خاص را با هم جمع می‌کند.

```
val add = (a: Int, b: Int) => a + b
```

در این تابع متغیرهای a و b متغیرهای رسمی یا متغیرهای تابع هستند. حال اگر برنامه را به شکل دیگری تغییر دهیم و یک متغیر خارج از تابع به برنامه اضافه کنیم، این تابع به یک تابع بستار تبدیل می‌شود.

```
val factor = 1
val add = (a: Int, b: Int) => a + b + factor
```

در این برنامه علاوه بر دو متغیر تابعی a و b، یک متغیر دیگر هم اضافه شده است که مقدار آن در خارج از تابع کنترل می‌شود. برنامه کامل مبتنی بر تابع بستار را در تصویر ۴۳ مشاهده نمایید.

[10] Closures

```
object Main {
  def main (args: Array[String]){
    println(add(12, 13))
  }

  val number = 1
  val add = (a: Int, b: Int) => a + b + number
}
```

تصویر ۴۳ تابع بستار در اسکالا

7 شی‌گرایی

شی‌گرایی یکی از روش‌های اصلی برنامه‌نویسی و توسعه نرم‌افزار می‌باشد. این روش به طور کامل از طبیعت اقتباس شده است. اگر به طبیعت نگاه بیندازید هر چیزی که در طبیعت وجود دارد یک شیء است که دارای یک سری خصوصیت و رفتار می‌باشد. اشیائی که ویژگی‌ها و رفتارهای مشابه داشته باشند، در یک گروه واحد قرار می‌گیرند. به این گروه واحد که در آن اشیاء با رفتارها و خصوصیات مشابه در آن قرار می‌گیرند، کلاس گفته می‌شود.

در برنامه‌نویسی شی‌گرا، فضای مسأله به مجموعه‌ای از اشیاء تقسیم می‌شود، که برای حل مسأله این مجموعه اشیاء باید با همدیگر تعامل و ارتباط داشته باشند. هر شی دارای

مجموعه‌ای از خصوصیات و رفتار می‌باشد که ما انتظار داریم، متناسب با آن خصوصیات بتوانیم با شی مورد نظر تعامل نماییم. کلاس را مانند یک نقشه ساختمان در نظر بگیرید که به تنهایی موجودیت ندارد. اما هنگامی که از این نقشه برای ساخت یک ساختمان استفاده می‌کنیم، ساختمان موجودیتی است که از نقشه پیروی کرده است (ساختمان یک نمونه از کلاس است). از هر نقشه می‌توان در چندین ساختمان استفاده کرد، یعنی از هر کلاس می‌توان چندین نمونه ایجاد کرد.

به عنوان مثال انسان را یک کلاس در نظر می‌گیریم، که مجموعه خصوصیت و رفتارهای آن را در جدول ۸ مشاهده می‌نمایید. در این مفهوم ویژگی‌ها نقش متغیرها و رفتارها نقش متدها را بازی می‌کنند. متد، تابعی است که در درون یک کلاس قرار می‌گیرد. خصوصیات و رفتارها این معنی را دارند که هر نمونه از این کلاس دارای این ویژگی‌ها و رفتارها (به گونه‌های مختلف) می‌باشد.

جدول ۸ خصوصیات و رفتار انسان

ویژگی	رفتار
نام	صحبت کردن
نام‌خانوادگی	راه رفتن
سن	گوش دادن
قد	لمس کردن

به هر شی‌ای که از یک کلاس ایجاد می‌گردد، موجودیت یا نمونه گفته می‌شود.

- **چهار اصل شی‌گرایی**

فلسفه بوجود آمدن مفهوم شی‌گرایی به پاسخ داده به این چهار اصل اساسی برمی‌گردد.

- انتزاع (تجرید)[11]

از یک کلاس خاص فقط خصوصیات و رفتارهایی که بنا به مسأله شما دارای اهمیت است را در نظر بگیرید و از ذکر بقیه موارد خودداری نمایید. مثلاً برای یک بانک نام و نام‌خانوادگی و آدرس و شماره تلفن معنادار است، اما اگر بانک در فرم ثبت نام از شما بخواهد که نام غذا مورد علاقه، یا فیلم مورد علاقه یا رنگ چشم و ... را ذکر نمایید، آیا کار منطقی‌ای است؟ بنابراین هرگاه که بخواهیم برخی از خصوصیات و رفتارهای یک کلاس خاص را نادیده بگیریم، از اصل انتزاع استفاده می‌نماییم.

در یک سطح بالاتر هدف از انتزاع پنهان‌سازی جزئیات غیرضروری از دید کاربر است. مثلاً در نظر بگیرید که می‌خواهید با استفاده از یک قهوه‌ساز برقی، قهوه درست کنید. کافی است که شما آب و قهوه را آماده کرده و در قهوه‌سازی بریزید و آن را روشن کنید. اما آیا نیازی به این دارید که از نحوه ساخت قهوه توسط قهوه‌سازی و فرآیندهای داخلی قهوه‌ساز آگاهی داشته باشید؟ یا در هنگام ارسال ایمیل شما کافی است که متن پیام خود را نوشته و آن را ارسال کنید، آیا نیازی به اطلاع از اینکه ایمیل

تصویر ۴۴ نمونه‌ای از انتزاع، هر روز از خودپردازها استفاده می‌کنیم اما آیا از جزئیات داخلی آن‌ها آگاهی داریم؟ منبع: Sitebay

[11] Abstraction

چگونه ارسال می‌شود یا از چه مسیرهایی می‌گذرد و ... دارید؟ برای درک بهتر به تصویر ۴۴ دقت نمایید.

- **محصورسازی (کپسوله‌سازی)**[12]

ویژگی‌ها و رفتارهای یک کلاس در درون کلاس محصورسازی می‌شوند. یعنی سطح دسترسی آن‌ها را به گونه‌ای تغییر می‌دهند که در بیرون از کلاس یا پکیج قابل تغییر نباشند. محصورسازی مفهومی است که همیشه با آن سروکار داریم مثل امنیت و حریم خصوصی، یا اینکه افراد چگونه فکر می‌کنند و ...

- **چندریختی**[13]

یک رفتار در بین اشیاء یک کلاس به شیوه‌های گوناگونی انجام می‌شود. مثلاً سگ و گربه هر دو یک زیرکلاس از کلاس حیوانات هستند. این حیوانات هر دو صدا تولید می‌کنند اما صدای تولید شده در آن‌ها با هم متفاوت است. یا همه موجودات برای زنده ماندن نیاز به تغذیه دارند، اما شیوه‌های غذا خوردن موجودات مختلف متفاوت است.

- **ارث‌بری**[14]

[12] Encapsulation

[13] Polymorphism

[14] Inheritance

زمانی که نقاط مشترک بین دو کلاس زیاد باشد، هر دو را به یک کلاس کلی‌تر تبدیل می‌کنیم که کلاس‌های کوچک‌تر از آن ارث‌بری می‌نمایند. مثلاً حیوان یک کلاس است که تعداد زیادی موجودات را شامل می‌شود. خود کلاس حیوان دارای چند زیرکلاس مانند پرندگان، پستانداران، خزندگان و ... است و خود این زیرکلاس‌ها نیز به زیر کلاس‌های کوچک‌تر تقسیم می‌شوند. به کلاسی که کلاس‌های دیگر از آن ارث‌بری می‌کنند، کلاس والد و به کلاسی که از کلاس والد ارث‌بری می‌کند کلاس فرزند یا زیرکلاس گفته می‌شود.

8 کلاس

اسکالا یک زبان برنامه‌نویسی شی‌گرای خالص است، یعنی همه چیز در اسکالا شی می‌باشد. به عنوان مثال می‌توانیم عبارت 1 + 2 را به صورت شی‌گرا، یعنی به صورت (1).+2 بنویسیم. این عبارت نشان می‌دهد که برخلاف جاوا که در آن تایپ‌های اولیه شی نبودند، در اسکالا تایپ‌های اولیه نیز شی می‌باشند. شیء یک موجودیت در دنیای واقعی است که دارای حالت یا خصوصیت و رفتار می‌باشد. حالت‌ها یا خصوصیت‌ها را با متغیرها و رفتارها را با متدها نمایش می‌دهیم.

برای شروع برنامه‌نویسی شی‌گرا به سراغ مثال اول در تصویر ۴۵ توجه نمایید.

```
class Student {
  var id = 1234
  var fName = "yasin"
  var lName = "amini"
}
object Main {
    def main(args: Array[String]) {
      var student = new Student()
      println("first name: %s and las name: %s".format(student.fName, student.lName))
    }
}
```

تصویر ۴۵ تعریف کلاس در اسکالا

برای تعریف کلاس در زبان اسکالا مانند بیشتر زبان‌ها از کلمه کلید class استفاده می‌شود. در حالت کلی فرم کلاس‌ها به صورت زیر خواهد بود:

```
class نام کلاس {
    بدنه کلاس
}
```

نکته: به خط ۱۰ توجه نمایید که برای نوشتن متغیر در درون رشته در اسکالا به این ترتیب عمل می‌شود.

حال مثال بالا را به کمک سازنده کلاس (constructor) دوباره نویسی می‌نماییم. اگر شما برنامه‌نویس زبان‌های جاوا یا سی‌شارپ باشید، احتمالاً کد شما مانند تصویر ۴۶ خواهد بود.

```
class Student {
  var id = 0
  var fname = null
  var lname = null

  def Student(id: Int, fname: String, lname: String){
    this.id = id
    this.fname = fname
    this.lname = lname
  }
}

object First {
  def main(args: Array[String]){
    var student = new Student()
    println("First name: %s and Last name: %s".format(student.fname, student.lastname))
  }
}
```

تصویر ۴۶ ایجاد سازنده مطابق با قواعد زبان‌های جاوا و سی‌شارپ

مثال تصویر ۴۶ را اجرا کنید، آیا بدون مشکل اجرا می‌شود؟ درواقع این قواعدی است که برای زبان‌های سی‌شارپ و جاوا صحیح بوده و به درستی کار می‌کند. اما زبان اسکالا برای سخت سازنده دارای قواعد منحصربه‌فردی می‌باشد که سبب می‌شود کدهای اسکالا بسیار مختصرتر از دیگر زبان‌ها باشد. نحوه نوشتن صحیح سازنده را در تصویر ۴۷ مشاهده می‌نمایید.

```
class Student(id: Int, fname: String, lname: String) {
  def echo(){
    println("First name: %s and Last name: %s".format(this.fname, this.lastname))
  }
}

object First {
  def main(args: Array[String]){
    var student = new Student()
    student.echo()
  }
}
```

تصویر ۴۷ نحوه تعریف سازنده کلاس در زبان اسکالا

برای آن دسته از برنامه‌نویسانی که قبلاً با زبان‌های جاوا و سی‌شارپ کار کرده‌اند این نحوه نوشتن، آن‌ها را یاد تابع می‌اندازد، اما باید توجه داشته باشید که پارامترهای موجود در تعریف کلاس نقش سازنده را برای زبان اسکالا بازی می‌کنند.

در مفهوم شی گرایی و کلاس، متغیرهای نقش خصوصیات و متدها نقش رفتار کلاس را بازی می‌کنند. به عنوان مثال در نظر بگیرید که یک کلاس انسان داشته باشیم. این کلاس دارای خصوصیاتی از قبیل نام، نام خانوادگی، سن، قد و وزن و ... است و رفتارهایی مانند سلام کردن، صحبت کردن، پیاده‌روی و ... می‌تواند انجام دهد. پیاده‌سازی مفاهیم گفته شده در بالا به ترتیب زیر خواهد بود (تصویر ۴۸).

```scala
class Human (val firstName: String, val lastName: String, val age: Int) {
    def sayHello(): String = {
      return "Hello"
    }

    def repeat(word: String): String = {
      return word
    }

   override def toString: String =
    s"$firstName  $lastName"
}
object Main {
    def main(args: Array[String]) {
      var human = new Human("yasin", "amini", 23)
      println(human)
      println(human.sayHello())
      println(human.repeat("salam"))
    }
}
```

تصویر ۴۸ پیاده‌سازی کلاس انسان در اسکالا

Output:

yasin amini

Hello

salam

همانگونه که مشاهده می‌کنید برای این کلاس سه خصوصیت نام، نام‌خانوادگی و سن و سه متد "سلام گفتن"، "تکرار حرف" و "toString" تعریف شده است. متد toString در همه کلاس‌ها وجود دارد و نحوه نمایش شی را نشان می‌دهد. اگر این متد را حذف کرده و برنامه را دوباره اجرا کنید متوجه می‌شوید که خط اول خروجی به ترتیب زیر خواهد بود.

Human@40e6dfe1

پیش از متد toString از کلمه کلیدی override استفاده شده است. اگر متد کلاس پدر در کلاس فرزند دوباره‌نویسی شود به آن override می‌گویند. اگر چند متـد دارای اسـم مشـترک باشند اما ورودی آن‌ها متفاوت باشد را overloading می‌گویند (مانند تصویر ۴۹).

```
class Human (val firstName: String, val lastName: String, val age: Int) {
    def sayHello(): String = {
      return "Hello"
    }

    def sayHello(name: String): String = {
      return s"Hello $name "
    }

    def repeat(word: String): Stirng = {
      return word
    }

   override def toString: String =
    s"$firstName  $lastName"
}
object Main {
    def main(args: Array[String]) {
      var human = Human("yasin", "amini", 23)
      println(human.sayHello())
      println(human.sayHello("world"))
      println(human.repeat("salam"))
    }
}
```

تصویر ۴۹ overloading در اسکالا

در این مثال یک trait به اسم Animal تعریف شده است. Trait ساختاری مشابه با کلاس است که در آن امضای متدها نوشته می‌شوند، اما بدنه داخلی آن پیاده‌سازی نمی‌گردد. دو کلاس dog و cat از کلاس Animal ارث‌بری کرده‌اند. برای ارث‌بری کردن در زبان اسکالا از کلمه کلیدی extends استفاده می‌شود. این دو کلاس هر دو دارای متدی به نام makeNoise هستند، این خاصیت چندریختی یا polymorphism نامیده می‌شود (تصویر شماره). به کلاسی که از آن ارث‌بری می‌شود کلاس والد (superclass) و به کلاسی که از کلاس پدر ارث‌بری می‌کند کلاس فرزند (subclass) گفته می‌شود. در اسکالا هر کلاس فقط می‌تواند از یک کلاس دیگر ارث‌بری داشته باشد (تصویر ۵۰).

```scala
trait Animal {
    def makeNoise()
}

class Cat extends Animal {
    override def makeNoise (): String = {
        "Meow"
    }
}

class Dog extends Animal {
    override def makeNoise (): String = {
        "Bark"
    }
}

object Main {
    def main(args: Array[String]) {
      var cat = new Cat()
      var dog = new Dog()

      println(cat.makeNoise())
      println(dog.makeNoise())
    }
}
```

تصویر ۵۰ چندریختی در اسکالا

9 سطوح دسترسی

اعضای یک کلاس یا تابع می‌توانند دارای سطوح دسترسی خصوصی (private)، محافظت‌شده (protected) یا عمومی (public) باشند. در اسکالا نیازی به گذاشتن سطح دسترسی عمومی وجود ندارد و سطح دسترسی به طور پیش‌فرض عمومی در نظر گرفته می‌شود. برای سطوح دسترسی سه کودک ۱۰، ۶ و ۴ را در نظر بگیرید. به کودک ۱۰ اجازه بازی در کوچه داده می‌شود و سطح دسترسی آن عمومی است. کودک ۶ ساله محدود به بازی در حیاط خانه است و نمی‌تواند بیرون برود و سطح دسترسی آن محافظت شده است. کودک چهار ساله حق رفتن به حیاط را نیز ندارند و سطح دسترسی آن خصوصی است.

- **سطح دسترسی خصوصی**

فقط اعضای کلاس یا شیء که از آن ساخته شده است توانایی دسترسی به آن را دارند. در تصویر شماره a یک متغیر خصوصی است که در داخل کلاس مقدار آن تغییر کرده است، اما اگر خارج از کلاس بخواهید مقدار آن را تغییر دهید برنامه با خطا مواجه می‌شود (تصویر ۵۱).

```
class Example {
    private var a:Int=7
    def show(){
        a=8
        println(a)
    }
}
object access {
  def main(args: Array[String]) {
    var e=new Example()
    e.show()
    //e.a=8
    //println(e.a)
  }
}
```

تصویر ۵۱ سطح دسترسی خصوصی

- سطح دسترسی محافظت‌شده

اگر سطح دسترسی یک جزء محافظت شده باشد، از طریق اعضای کلاسی که در آن قرار دارد و کلاس‌هایی که از آن ارث‌بری کرده‌اند قابل دسترس است (تصویر ۵۲).

```
class Example{
    protected var a:Int=7
    def show( ){
        a=8
        println(a)
    }
}

class Example1 extends Example{
    def show1( ){
        a=9
        println(a)
    }
}

object access {
  def main(args: Array[String]){
    var e=new Example( )
    e.show( )
    var e1=new Example1( )
    e1.show1( )
    //e.a=10
    println(e.a)
  }
}
```

تصویر ۵۲ سطح دسترسی محافظت شده

- **سطح دسترسی عمومی**

اجزایی که دارای سطح دسترسی عمومی هستند از تمام برنامه‌ها و کلاس‌ها قابل دسترس و قابل تغییر می‌باشند. اگر مثال‌های بالا سطح دسترسی خصوصی و محافظت شده را به عمومی تغییر دهید، مشاهده می‌کنید که برنامه بدون هیچ گونه مشکلی اجرا می‌گردد.

یکی دیگر از راه‌های تعیین سطح دسترسی برای متغیرها و اعضای برنامه از طریق پکیج‌ها می‌باشد. در واقع پکیج عبارتی کلی‌تر از کلاس است که می‌تواند چندین کلاس را شامل شود. سطح دسترسی به یک پکیج خاص از طریق private[package] یا protected[package] تعیین می‌گردد (تصویر ۵۳).

```
package society {
   package professional {
      class Executive {
         private[professional] var workDetails = null
         private[society] var friends = null
         private[this] var secrets = null

         def help(another : Executive) {
            println(another.workDetails)
            println(another.secrets) //ERROR
         }
      }
   }
}
```

تصویر ۵۳ تعیین سطح دسترسی به کمک استفاده از پکیج

کلمه کلیدی this به کلاس جاری اشاره دارد. همانگونه که در مثال مشاهده می‌کنید، متغیرهای دارای سطح دسترسی یک پکیج خاص فقط به متغیرهای آن پکیج و پکیج‌های زیرین دسترسی دارند و نمی‌توانند به متغیرهای پکیج‌های بالاتر دسترسی داشته باشند.

object نوعی کلاس است که به آن singleton گفته می‌شود بدین معنی که از آن تنها می‌توان یک شی ساخت. در اسکالا امکان تعریف استاتیک برای اعضا کلاس وجود ندارد و برای اینکه امکان معادلی وجود داشته باشد از object استفاده می نماییم. وقتی که کلاسی از این نوع است نمی‌توانیم با new از آن شی بسازیم.

10 مدیریت استثناءها

مدیریت استثناء به معنای انجام کار مناسب در وضعیت غیرمنتظره است. مثلاً در جریان اجرای یک برنامه ممکن است خطای غیرمنتظره‌ای مثل تقسیم بر صفر یا عدم وجود فایل رخ دهد. برای جلوگیری از متوقف شدن برنامه از مدیریت کننده استثناءها استفاده می‌گردد. میان استثناء و خطا تفاوت‌هایی وجود دارد. خطاهای موجود در برنامه باید توسط برنامه‌نویس رفع شوند، اما در استثناء خطایی وجود ندارد و دلیل وجود خطا به انتخاب‌های نادرست کاربر و ورودی اشتباه برمی‌گردد.

برای مدیریت استثناءها روش‌های گوناگونی وجود دارد. اولین روش استفاده از try {} ، {} catch است. در این روش دستوراتی که احتمال دارد با خطا مواجه شوند را در بخش try نوشته می‌شوند. در صورت بروز خطاهای پیش‌بینی شده، دستورات درون catch اجرا می‌شوند. برای بعضی از کاربردها نیاز داریم که در صورت اجرا دستورات یا استثناءها، بعضی از عملیات‌ها صورت بپذیرند. برای اینگونه کاربردها از کلمه کلید finally استفاده می‌شود (تصویر ۵۴).

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

object ExceptionHandling {
   def main(args: Array[String]) {
     //  در هنگام خواندن فایل امکان دارد با خطای عدم وجود فایل یا پایین بودن سطح دسترسی مواجهه
شوید  try {
         val f = new FileReader("input.txt")
      }
     // در اینجا چون احتمال عدم وجود فایل و خطای ورودی و خروجی وجود دارد، در صورت وجود هر کدام
گردد catch{ خطای مربوطه انجام
         case ex: FileNotFoundException => {
            println("Missing file exception")
         }

         case ex: IOException => {
            println("IO Exception")
         }
      }
     // در صورت اجرای دستورات یا استثناء ها این بخش اجرا میگردد
     finally {
         println("Exiting finally...")
      }
}
```

تصویر ۵۴ مدیریت استثناء

Output:

Missing file exception

Exiting finally…

در زبان برنامه‌نویسی جاوا برای رفع چندین استثناء از چند بلاک catch استفاده می‌گردید . اما همانگونه که در مثال بالا مشاهده کردید، در اسکالا این کار با استفاده از کلمه کلید case صورت می‌پذیرد. به عبارتی دیگر با استثناء به شکل pattern matching برخورد می‌نماید.

برای ساختن پیام‌های شخصی‌سازی شده در استثناء نیز از کلمه کلید throw استفاده می‌شود. یعنی در هر جا که احتمال خطا وجود دارد آن را بـه داخـل اسـتثناء پرتـاب می‌نمـاییم (تصـویر ۵۵).

```
object ExceptionHandling {
  def main(args: Array[String]){
    try{
            division(10,2)
            division(10,0)
    }
    catch {
          case ex: Exception =>  println("Division by Zero")
    }

    finally {
            println("finally call")
    }
    println("it work")

    }

    def division(a: Int, b:Int)  = {
        if (b==0)
          throw new Exception
        else {
          var result: Int = a/b
          println(result)
        }
    }
}
```

تصویر ۵۵ throw در اسکالا

Output:

5

Division by Zero

finally call

it work

می‌توانید برنامه بالا را به شکل ساده‌تر زیر بازنویسی نمایید (تصویر ۵۶).

```
object Main {
    def main(args: Array[String]) {
        division(2,10)
        division(10,4)
        division(4,0)
        division(0,2)
    }

    def division(a: Int, b: Int) = {
      if (b==0){
        throw new Exception("Divide by Zero")
      }
      else {
        println(a/b)
      }
    }
}
```

تصویر ۵۶ استفاده از throw

تفاوت دو برنامه بالا در چیست؟ آیا خروجی آن‌ها یکسان می‌باشد؟ در واقع از try-catch برای مدیریت استثناء و عدم توقف برنامه استفاده می‌شود، اما throw خطا را به عبارتی که آن را فراخوانده است، برمی‌گرداند.

کلمه کلیدی throws روش دیگری است که از آن نیز برای مدیریت استثناء استفاده می‌گردد. تفاوت آن با throw در این است :

- از throw در داخل توابع استفاده می‌گردد، اما throws همراه با امضای تابع یا به عنوان حاشیه نویسی[15] مورد استفاده قرار می‌گیرد.
- throw توانایی مدیریت یک خطا را دارد اما به کمک throws می‌توانید همزمان چند خطا را مدیریت نمایید.
- throw به نمونه‌ای از کلاس استثناء ارجاع داده می‌شود، اما در throws خود کلاس استثناء فراخوانی می‌گردد.
- throw برای تعریف خطاهای واضح به کار می‌رود، اما throws می‌تواند چندین خطا را مدیریت کند.
- throw برای استثناءهای بررسی نشده استفاده می‌گردد، اما throws برای استثناءهای بررسی شده.

تصور کنید در هنگام نوشتن تابعی برای تبدیل رشته به عدد هستید، برای این کار چون امکان دارد که کاربر رشته را تابع پاس دهد، باید از مدیریت استثناء استفاده نمایید (تصویر ۵۷).

[15] annotation

```
object Throws {
  def main(args: Array[String]){
    println(validate("12"))
    println(validate("ab"))
    println(validate("21"))
  }
  @throws(classOf[NumberFormatException])
  def validate(input: String)={
    input.toInt
  }

}
```

تصویر ۵۷ استفاده از throws

Output:

12

java.lang.NumberFormatException: For input string: "ab" ….

همانطور که مشاهده می‌کنید برنامه بالا بعد از اعلام خطا متوقف می‌شود. برای جلوگیری از توقف برنامه و مدیریت خطا چه راه‌کاری ارائه می‌دهید؟ برنامه بالا را به گونه‌ای بازنویسی نمایید که متوقف نشود و در خروجی ۲۱ را نیز چاپ کند (جواب در تصویر ۵۸).

```
object Throws {
  def main(args: Array[String]){
    println(validate("12"))
    println(validate("ab"))
    println(validate("21"))
  }
  @throws(classOf[NumberFormatException])
  def validate(input: String)={
    try{
      input.toInt
    }
    catch {
      case e :Exception  => print("invalid input")
    }
  }

}
```

تصویر ۵۸ استفاده از کلمه کلیدی throws

Output: 12

invalid input

21

برای مدیریت استثناءها دو روش کلی وجود دارد که در ادامه آن‌ها را مورد بررسی قرار می‌دهیم.

- بررسی شده (checked)

به استثناءهایی که در زمان کامپایل مورد بررسی قرار می‌گیرند، استثناءهای بررسی شده گفته می‌شود. اگر در برنامه خود از استثناءهای بررسی شده استفاده نمایید، باید حتماً برای آن خطاهای احتمالی را نیز تعریف نمایید. مثلاً در زبان جاوا در هنگام کار کردن با فایل‌ها یا پایگاه داده، تعریف استثناءها برای مدیریت عدم وجود فایل یا پایگاه داده الزامی است. در حقیقت در زبان‌هایی که از این روش استفاده می‌شود، تعریف بعضی از استثناءها توسط برنامه‌نویس ضروری است، در غیراینصورت برنامه کامپایل نمی‌گردد. برای درک بهتر این روش برنامه موجود در تصویر ۵۹ را که با زبان جاوا نوشته شده است بررسی نمایید. جاوا از هر دو روش بررسی شده و بررسی نشده به صورت ترکیبی استفاده می‌نماید (در فایل و پایگاه داده به صورت بررسی شده و در خطاهایی مانند تقسیم بر صفر به صورت بررسی نشده عمل می‌کند).

```java
import java.io.*;

class Main {
    public static void main(String[] args) {
        FileReader file = new FileReader("~/input.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // چاپ سه خط اول فایل
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```

تصویر ۵۹ مدیریت استثناء بررسی شده در جاوا

اگر برنامه بالا را اجرا نمایید، با خطای زیر مواجه می‌شوید که به شما یادآوری می‌کند که باید از مدیریت استثناء استفاده نمایید. برای رفع خطای پیش‌آمده کافی است که به متد main عبارت throws IOException را اضافه نمایید.

```
Main.java:6: error: unreported exception FileNotFoundException;
must be caught or declared to be thrown
        FileReader file = new FileReader("~/input.txt");
                          ^
```

```
Main.java:11: error: unreported exception IOException; must be caught or declared to be thrown
        System.out.println(fileInput.readLine());
                                    ^
Main.java:13: error: unreported exception IOException; must be caught or declared to be thrown
    fileInput.close();
              ^
3 errors
```

- بررسی نشده (unchecked)

این روش دقیقاً مخالف با روش بررسی شده است. یعنی در این روش استثناءها در زمان کامپایل بررسی نمی‌شوند. پس کامپایلر شما را مجبور به مدیریت استثناءها نمی‌نماید. در این روش وظیفه یافتن تمامی خطاهای احتمالی بر عهده برنامه‌نویس می‌باشد. برای مقایسه این دو روش همین مثال را با زبان اسکالا بازنویسی می‌گردد (تصویر ۶۰). اسکالا به طور کامل از روش بررسی نشده استفاده می‌نماید. روش‌های گفته شده هیچ مزیتی به هم ندارند و بسته به توانایی برنامه‌نویس می‌توانند متفاوت باشند. به عنوان مثال برای برنامه‌نویسان حرفه‌ای که قدرت تشخیص خطاهای احتمالی را دارند روش بررسی نشده و برای برنامه‌نویسان مبتدی روش بررسی شده مناسب‌تر است.

```scala
import scala.io.Source

object Main {
    def main(args: Array[String]) {
        val home = System.getProperty("user.home")
        val s = Source.fromFile(s"${home}/yasin.txt").foreach{ print }
    }
}
```

تصویر ۶۰ مدیریت استثناء بررسی نشده در اسکالا

11 کار با فایل‌ها

ورودی و خروجی‌های برای تعامل به کاربران استفاده می‌شوند. مثلاً دستور print، یک عبارت را به کاربر نشان می‌دهد. یکی دیگر از شکل‌های وارد کردن اطلاعات به برنامه از طریق فایل‌ها می‌باشد. برای کار کردن با فایل ابتدا باید فایل را وارد برنامه نمایید. سپس پردازش‌های لازم را روی آن انجام داده و سپس فایل را ببنیدید. پس برای کار کردن با فایل‌ها سه مرحله وجود دارد که شامل باز کردن فایل، پردازش و بستن فایل.

برای خواندن فایل در زبان اسکالا علاوه بر کتابخانه‌های جاوا می‌توانید از scala.io.Source نیز استفاده نمایید. تصویر ۶۱ نحوه خواندن و چاپ اطلاعات موجود در یک فایل را نشان می‌دهد.

```
import scala.io.Source

object Main {
    def main(args: Array[String]) {
        val home = System.getProperty("user.home")
        val s = Source.fromFile(s"${home}/input.txt").foreach{ print }
    }
}
```

تصویر ۶۱ خواندن از فایل

Outut:

hello

hi

اگر از سیستم عامل لینوکس استفاده می‌کنید اضافه کردن System.getProperty برای شناساندن home لازم است. برای خواندن یکباره متن می‌توانید از متد mkString استفاده نمایید. برنامه بالا را به گونه‌ای تغییر دهید که فقط خط اول فایل را چاپ نماید (جواب در تصویر ۶۲).

```scala
import scala.io.Source

object Main {
    def main(args: Array[String]) {
        val home = System.getProperty("user.home")
        val s = Source.fromFile(s"${home}/input.txt").getLines().take(1)
        println(s)
    }
}
```

تصویر ۶۲ خواندن از فایل

در برنامه تصویر شماره، تابع getLinse محتویات فایل را به صورت خط به خط می‌خوانـد و با استفاده از تابع take اعلام می‌کنیم که فقط خـط اول را برگردانـد. در حقیقت getLines یک iterator برمی‌گرداند و تا زمانی که پیمایش نشود، هیچ گونه سرباری بر سیسـتم اعمـال نمی‌کند. به این عملیات‌ها، سست یا Lazy گفته می‌شود. یعنی تـا زمـانی کـه لازم نباشـد هیچ گونه عملیاتی صورت نمی‌پذیرد. برای خواندن یک بازه خاص از فایل مثلاً خط اول تا چهارم از تابع slice استفاده می‌شود. این تابع دو ورودی دریافت می‌کند کـه ورودی اول بـرای نقطـه شروع و ورودی دوم برای نقطه پایان است.

slice(start, end).

برای نوشتن در فایل از کلاس java.io استفاده می‌شـود. برنامـه نوشـتن در فایـل را در تصـویر ۶۳ مشاهده می‌نمایید.

```scala
import java.io._

object FileWrite {
   def main(args: Array[String]) {
      val writer = new PrintWriter(new File("input.txt" ))

      writer.write("Scala Programming language :) ")
      writer.close( )
   }
}
```

تصویر ۶۳ نوشتن در فایل

در این برنامه ابتدا فایل باز و در متغیر writer ذخیره شده است. با استفاده از متد write محتویات مورد نظر در فایل نوشته می‌شود و در آخر با دستور close فایل بسته خواهد شد. توجه داشته باشید که در آخر کار با فایل‌ها حتماً باید با دستور close بسته شوند. توجه داشته باشید که استفاده از کلاس PrintWriter سبب می‌شود که کل محتویات فایل دوباره نویسی می‌شود (برای اضافه کردن ورودی‌های جدید به محتویات موجود در فایل به تصویر ۶۴ توجه نمایید). حال برنامه‌ای بنویسد که محتویات یک فایل را خوانده و آن را درون فایل دیگری بنویسد؟

```scala
import java.io._

object FileWrite {
   def main(args: Array[String]) {
      val writer = new FileWriter("input.txt", true )

      writer.write("Scala Programming language :) ")
      writer.close()
   }
}
```

تصویر ۶۴ نوشتن در فایل بدون پاک کردن محتویات قبلی

با انتساب true به پارامتر دوم FileWriter مشخص می‌کنید که محتویات وارد شده به محتویات قبلی افزوده شود و فایل بازنویسی نگردد. اگر این پارامتر را خالی بگذارید به صورت پیش‌فرض false در نظر گرفته شده و فایل را دوباره‌نویسی می‌کند.

12 نتیجه‌گیری

زبان برنامه‌نویسی اسکالا با توجه به مدت زمان کوتاهی که از ارائه آن می‌گذرد، توانسته جای خوبی را برای خود در برنامه‌نویسان و توسعه‌دهندگان دست و پا کند و در این سال‌ها به محبوبیت و مقبولیت قابل قبولی رسیده است. این زبان برنامه‌نویسی با توجه به ویژگی‌های خارق‌العاده‌ای که دارد می‌تواند یک جایگزین کامل و جامع برای زبان برنامه‌نویسی جاوا باشد. وارد شدن به این زبان و حرفه‌ای شدن در آن کار نسبتاً پیچیده‌ای است، چون این زبان شیب یادگیری سختی دارد. اما با حرفه‌ای شدن در آن امکان کار در زمینه‌های مختلف کامپیوتری برای شما ارائه می‌شود.

مطلب حاضر به هیچ عنوان آموزش کامل اسکالا نبوده و فقط قطره‌ای از آن دریای بی‌پایان را بیان کرده است و از خواننده عزیز خواهشمندم که از اکتفا به این مطلب خودداری کرده و به مستندات اصلی زبان اسکالا مراجعه نماید.

در آخر باید یادآوری شود که آموزش بالا برای کسانی مناسب است که قبلاً یک زبان برنامه نویسی به ویژه جاوا کارکرده باشند تا بتوانند یک آشنایی با نحو و دستورات اسکالا داشته باشند و برای آموزش بیشتر می‌توانند به خود مستندات اسکالا و یا منابع آموزشی ذکر شده در قسمت منابع و مراجع مراجعه فرمایند.

منابع و مراجع

[1] https://docs.scala-lang.org/tutorials/

[2] http://allaboutscala.com/

[3] http://datafliar/blog/scala

[4] https://github.com/scala

[5] https://www.tutorialspoint.com/scala/

[6] https://www.javatpoint.com/scala-tutorial/

[7] https://spark.apache.org/