- Scala is both an object-oriented programming (OOP) and a functional programming (FP) language.

Using Function Literals (Anonymous Functions)

- **anonymous function**–also known as a **function literal**
- You can pass it into a method that takes a function, or to assign it to a variable

```
val x = List.range(1, 10)
val evens = x.filter((i: Int) => i % 2 == 0) //example of anonymous function (explicit)
val evens = x.filter(_ % 2 == 0) // other way to shorten the function
```

- Function literals

```
(i: Int) => i % 2 == 0
```

- the => symbol as a transformer.
- Because the expression transforms the parameter list on the left side of the symbol (an Int named i) into a new result using the algorithm on the right side of the symbol (in this case, an expression that results in a Boolean).
- Different ways

```
val evens = x.filter((i: Int) => i % 2 == 0) //most explicit
val evens = x.filter(i => i % 2 == 0) // dropping the int declaration (scala can infer it)
val evens = x.filter(_ % 2 == 0) // Scala lets you use the _ wildcard instead of a variable name when the parameter
//appears only once in your function
```

- Other similar example

```
x.foreach((i:Int) => println(i)) //  most explicit form
x.foreach((i) => println(i)) // Int declaration isn't required
x.foreach(i => println(i)) //remove parentheses
x.foreach(println(_)) // _ wildcard
x.foreach(println) //  if a function literal consists of one statement that takes a single argument
```

Using Function Literals (Anonymous Functions)

- pass a function around like a variable, just like you pass String, Int

```
val multiply = (i: Int) => { i * 2 }// it's an instance of a function. known as a function value
multiply (2) // 4
multiply (3) // 6
```

- you can also pass it to any method (or function) that takes a function parameter with its signature.

```
val list = List.range(1, 5)
list.map(double)
```

- Different approach

```
// implicit approach
val add = (x: Int, y: Int) => { x + y }
val add = (x: Int, y: Int) => x + y
```

```
// explicit approach
val add: (Int, Int) => Int = (x,y) => { x + y }
val add: (Int, Int) => Int = (x,y) => x + y
```

Defining a Method That Accepts a Simple Function

- Define a function as below; the method will take one parameter named callback, which is a function. That function must have no input parameters and must return nothing:

```
def executeFunction(f:() => Unit) {
        f()
}
```

- The f:() **[In place of f you can use any name**]syntax defines a function that has no parameters. If the function had parameters, the types would be listed inside the parentheses.
- The => Unit (similar to void in Java) portion of the code indicates that this method returns nothing.
- Next, define a function that matches this signature. The following function named **sayHello** takes no input parameters and returns nothing:

```
val sayHello = () => { println("Hello") }
```

- pass the sayHello function to the executeFunction method:

```
executeFunction(sayHello)
```

- To define a function that takes a String and returns an Int, use one of these two signatures:

```
executeFunction(f:String => Int)
executeFunction(f:(String) => Int)
```

- Next, define a method that takes this function as a parameter and also takes a second Int parameter.

```
def executeXTimes(callback:() => Unit, numTimes: Int) {
for (i <- 1 to numTimes) callback()
}
```

- Use this function

```
executeXTimes(sayHello, 3)
```

- Another example

```
def executeAndPrint(f:(Int, Int) => Int, x: Int, y: Int) {
val result = f(x, y)
println(result)
}

val sum = (x: Int, y: Int) => x + y
val multiply = (x: Int, y: Int) => x * y

executeAndPrint(sum, 2, 9) // prints 11
executeAndPrint(multiply, 3, 9) // prints 27
```