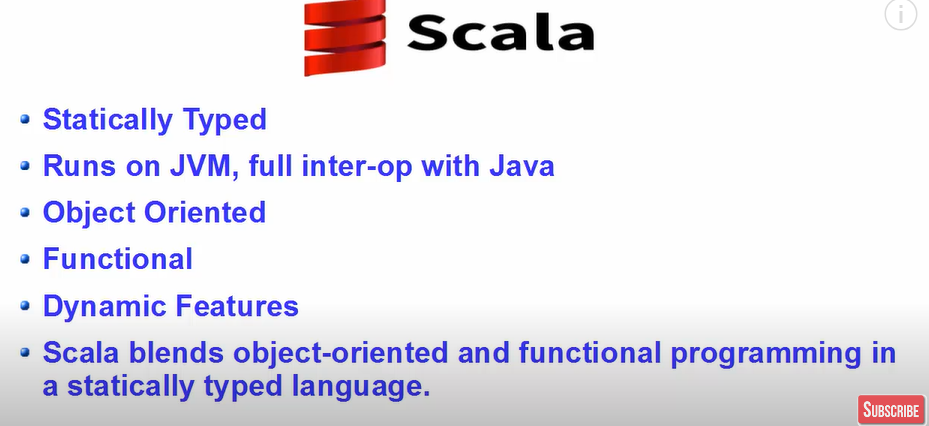
A white background with blue text

Description automatically generated

Scala is a statically typed language, means a variable declared of a type will hold the value of only that type.

Scala is as fast as java, advantage of shorter code.

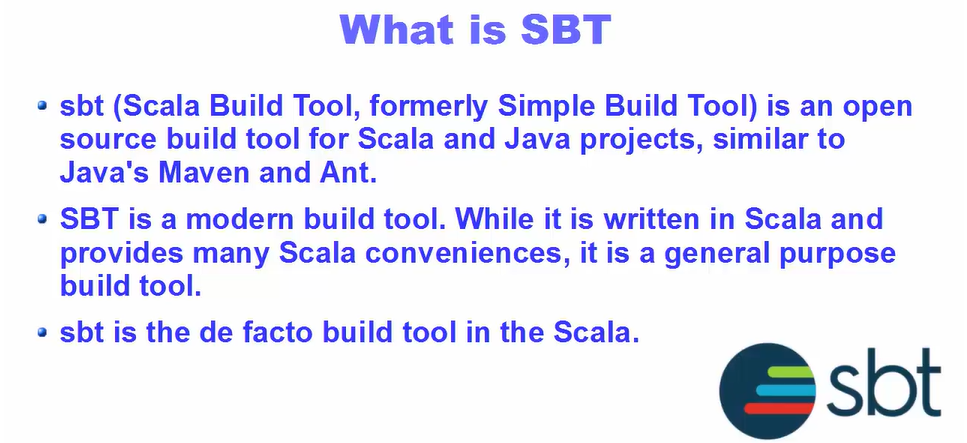


A white background with blue text

Description automatically generated

**Boilerplate** refers to sections of code that are repeated in multiple places with little to no variation. It's often necessary to include this code to meet a language's or framework's requirements, but it doesn't contribute to the core logic or functionality of the program.

Boilerplate code can make programs verbose and harder to read, as it clutters the codebase with repetitive and often non-essential details.

****

**REPL:** read, eval, print, loop

**DOWNLOAD JAVA**

Java version 1.8 and scala version 2.13

1. Install java 8 , after that add them to the path by :

A screenshot of a computer

Description automatically generated

Add both the variables to the path as:

1. To the system variables add:

A screenshot of a computer

Description automatically generated

1. To the user variables add:

JAVA\_HOME

C:\Program Files\Java\jdk1.8.0\_202

Save the variables and click ok and exit

A screenshot of a computer

Description automatically generated

**DOWNLOAD SCALA BUILD TOOL**

Download scala 2.13 msi

A screenshot of a computer program

Description automatically generated

Copy the bin folder path and set it in the path in the system variables.

A screenshot of a computer

Description automatically generated

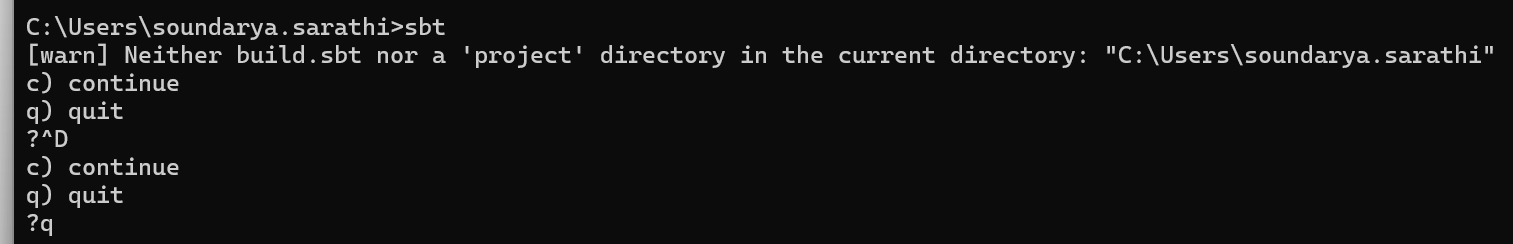
A screen shot of a computer

Description automatically generated

A black background with white text

Description automatically generated

Install sbt from the scala website.



To build using sbt:

A screenshot of a computer program

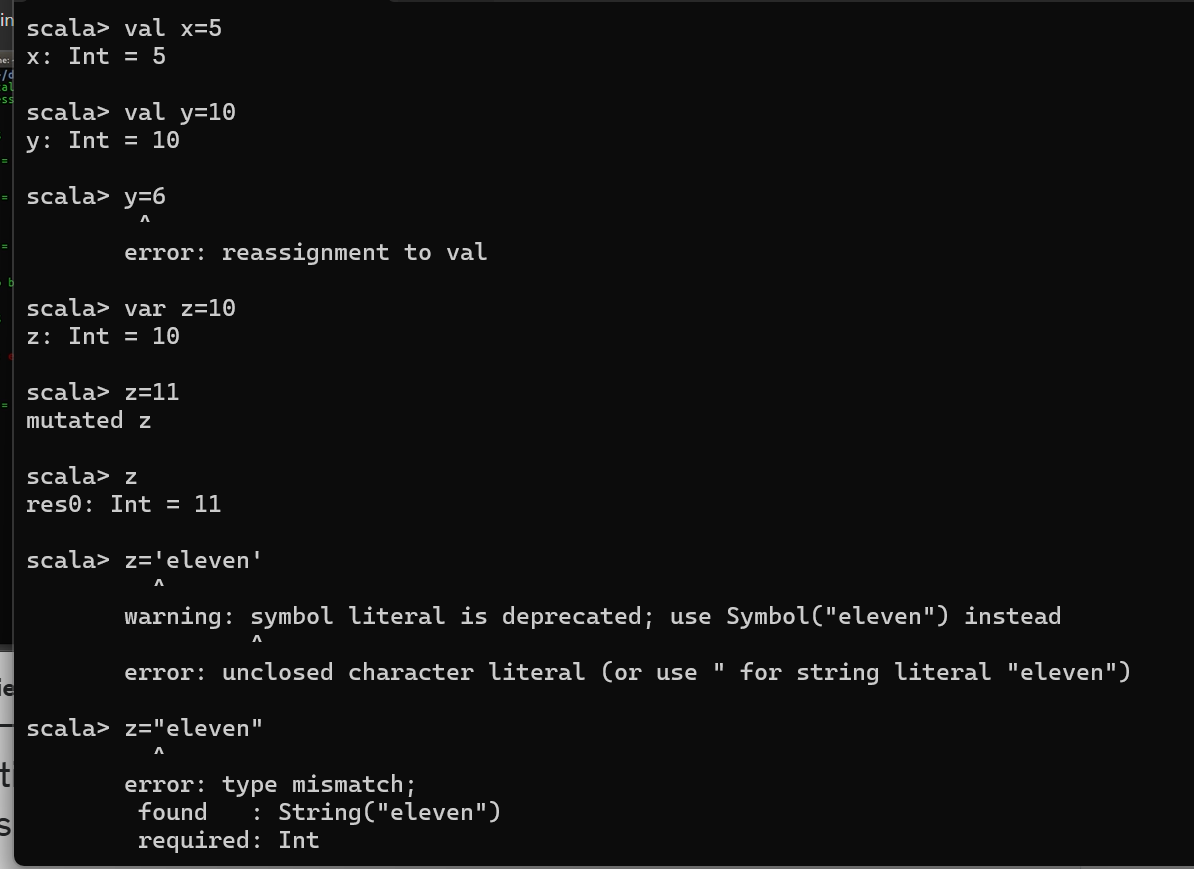
Description automatically generated

**SCALA CONSOLE**

1. Val vs vars

A screenshot of a computer

Description automatically generated

****

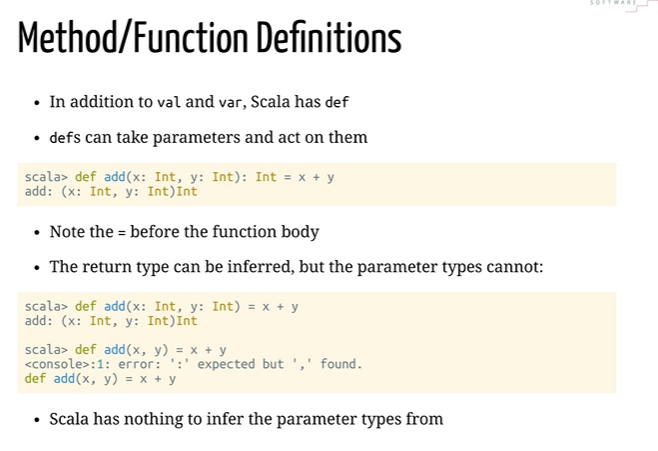
1. Paste mode

A screenshot of a computer code

Description automatically generated A screenshot of a computer program

Description automatically generated

1. **Methods/Functions**



A black screen with white text

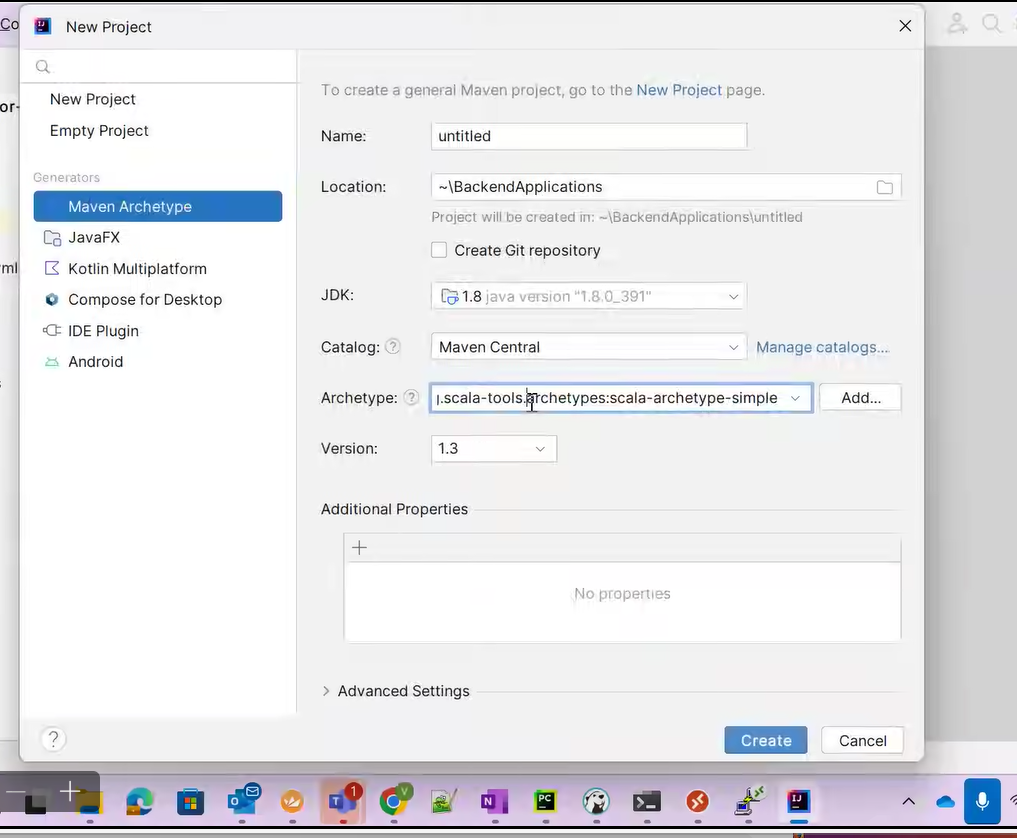
Description automatically generated

1. If statement

A screenshot of a math problem

Description automatically generated

**INTELLIJ**

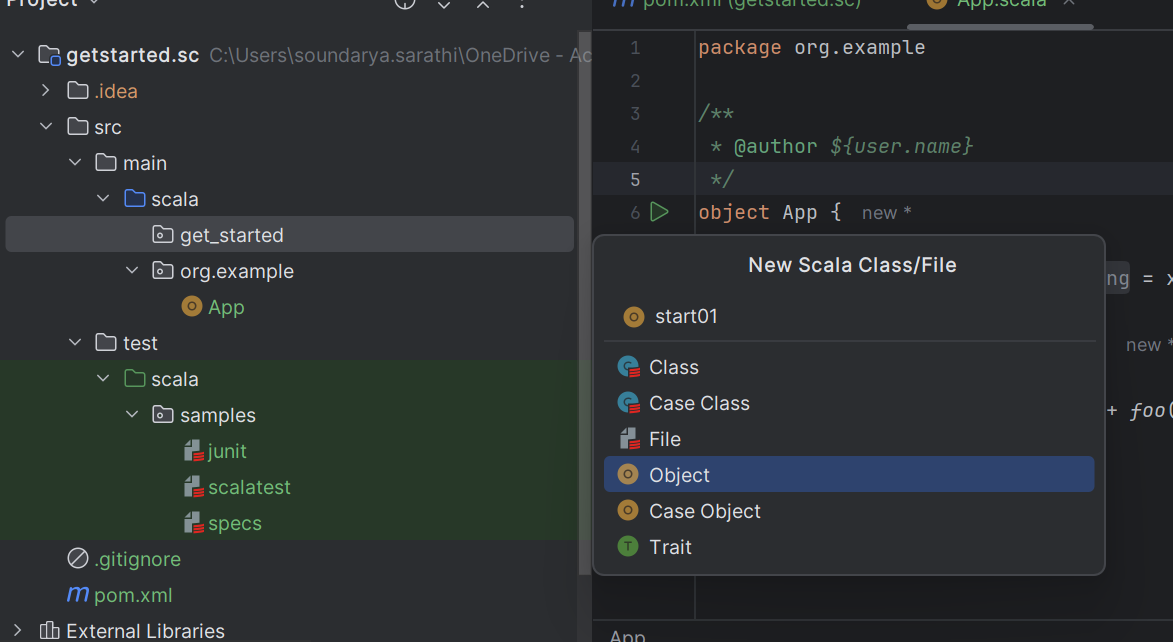


For scala in intellij, make sure jdk specific version and scala specific versionz is preinstalled. Install the scala plugin.

**Troubleshoot:**

* Build error: Ensure that the **Scala version (2.12.6)** is compatible with the **JDK version** you are using.
  + For Scala 2.12.x, use JDK 8 or 11. Avoid JDK 17 or higher as they may cause issues.
* If needed, install a compatible JDK version and configure IntelliJ to use it:
  + Go to **File > Project Structure > SDKs** and set the appropriate JDK.

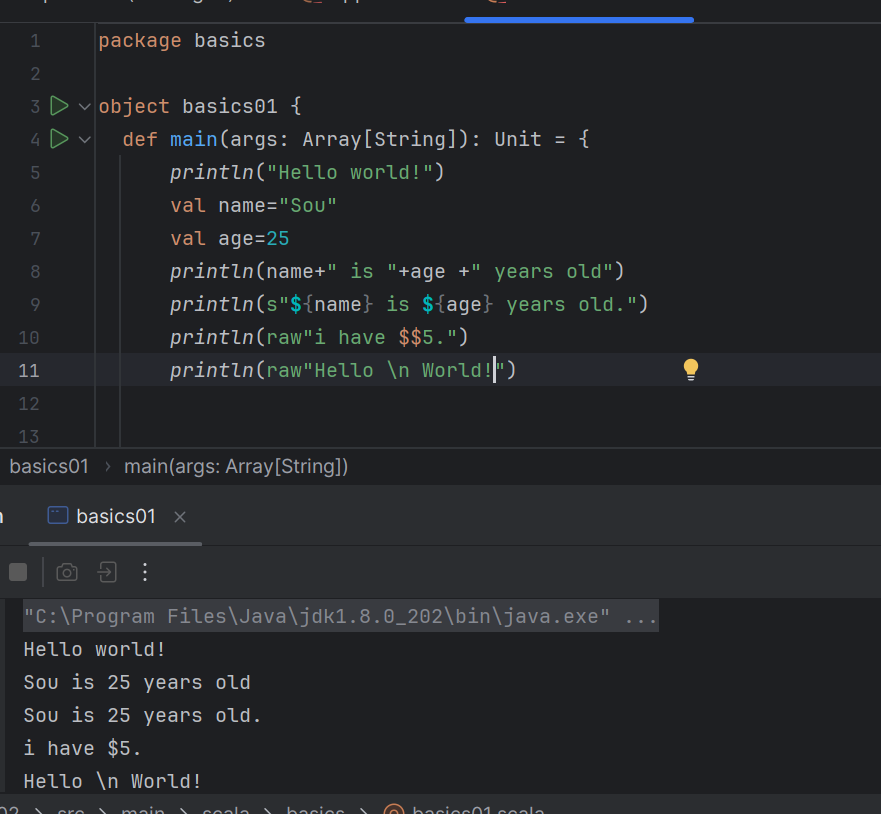
Create a package and then a scala file as:



A screen shot of a computer

Description automatically generated

1. **STRING INTERPOLATION**

****

1. **IF ELSE STATEMENTS**

A screen shot of a computer program

Description automatically generated

1. **LOOPS**

Cant write x++ in scala, not present in scala.

**While, DOWhile loop:**

**A computer screen shot of numbers

Description automatically generated**

**For Loop :**

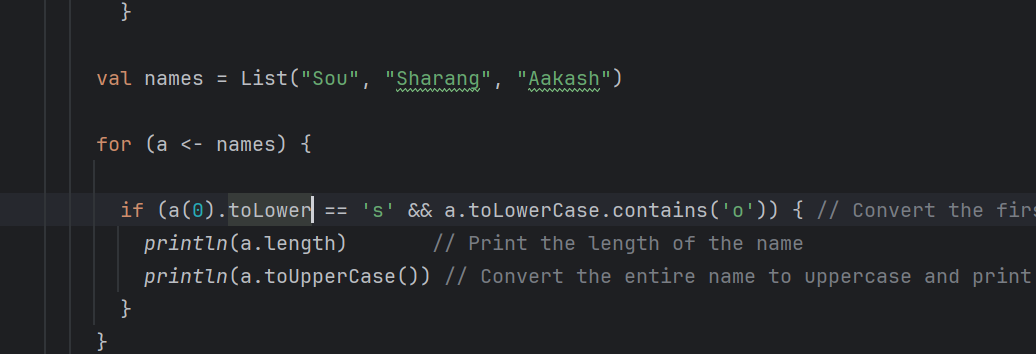
for (a <- 1 to 5)

if 2 loops, separate by a semi colon.

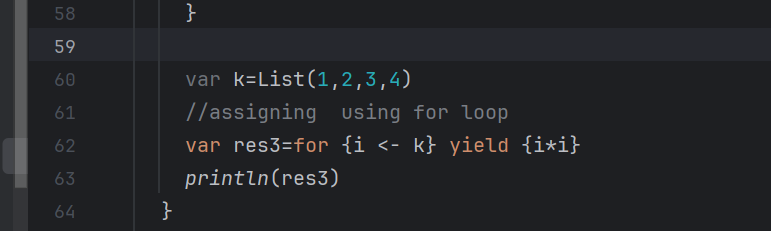
**A screen shot of a computer

Description automatically generated**

**Using List:**

****

**Assigning from for loop:**

****

**A number on a black background

Description automatically generated**

1. **MATCH EXPRESSIONS**

Like switch statements:

A screen shot of a computer program

Description automatically generated

1. **SCALA Function**

A screen shot of a computer screen

Description automatically generated

*println*(Math.*add*(13,12));

println(Math square 3)

When the function doesn’t return anything, return type is Unit(void)

A computer screen shot of a black screen with white text

Description automatically generated

*println*(Math.*print*(12));

Anonymous functions:ss

A black background with white text

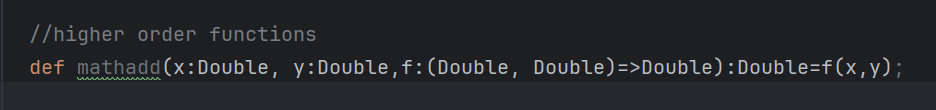
Description automatically generated

A screenshot of a computer program

Description automatically generated

1. **HIGHER ORDER FUNCTIONS**

Higher order functions are able to take functions as an argument and return functions as well.



Mathadd accepts x, y and a function “f” and mathadd returns a double value. f takes in value double and returns double.

val res=*mathadd*(5,2,(x,y)=>x+y);

with wildcards:

**def tripadd(x:Double, y:Double, z:Double, f:(Double, Double)=>Double):Double=f(f(x,y),z);**

*println*(*tripadd*(5,2,3,\_+\_))

*println*(*tripadd*(5,2,3,\_ max \_)) – to find max of 3 numbers.

1. **Partially Applied Functions**

A computer screen shot of a program

Description automatically generated

val date= new Date;  
val newlog=*log*(date,\_:String);  
newlog("mess 1");  
newlog("mess 2");

1. **Closures**

Closures are functions that use variables declared outside the function.

A computer screen shot of a number

Description automatically generated

*println*(*addd*(30));

As the value of number changes, the returned value also changes.

1. **Currying**

Currying is the technique of transforming a function that takes multiple arguments into a function that takes in a single argument.

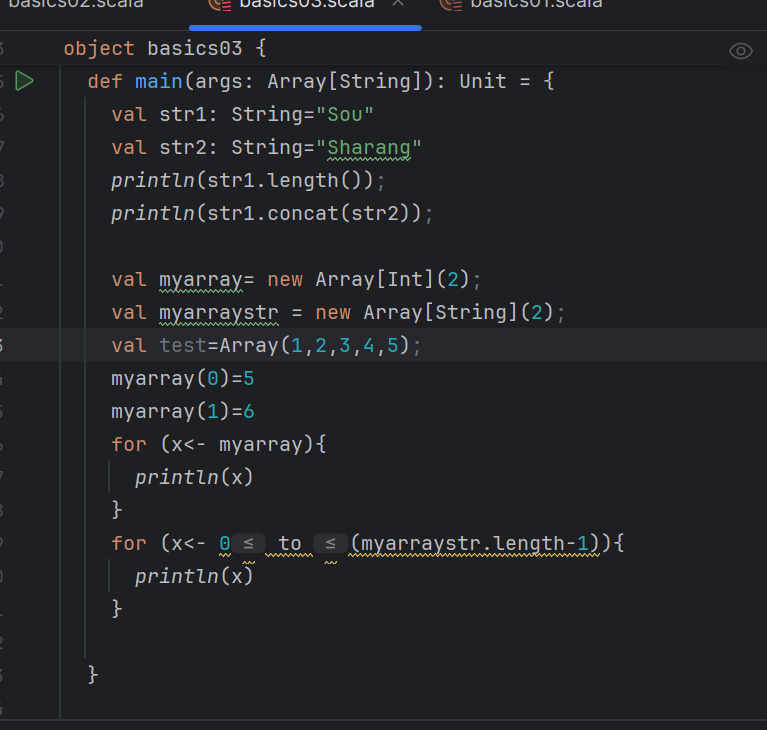
A black screen with white text

Description automatically generated

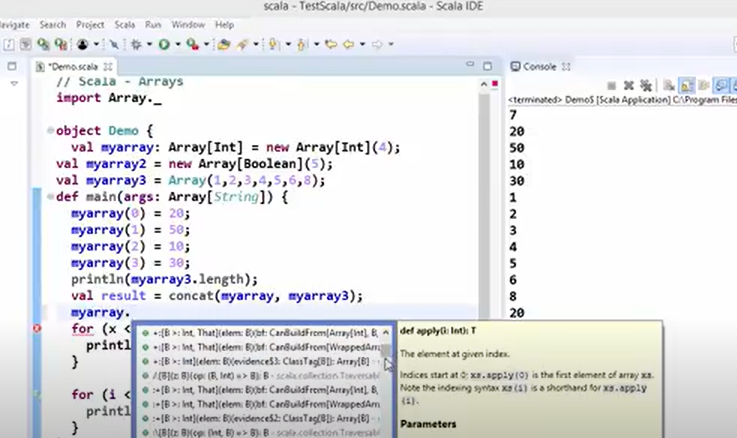
A screenshot of a computer code

Description automatically generated

1. **Strings and Arrays**

****

**Methods of arrays**

****

1. **Lists**

val myList: List[Any] = List("Scala", 42, true, 3.14) myList.foreach(println)

can store any kind of data.

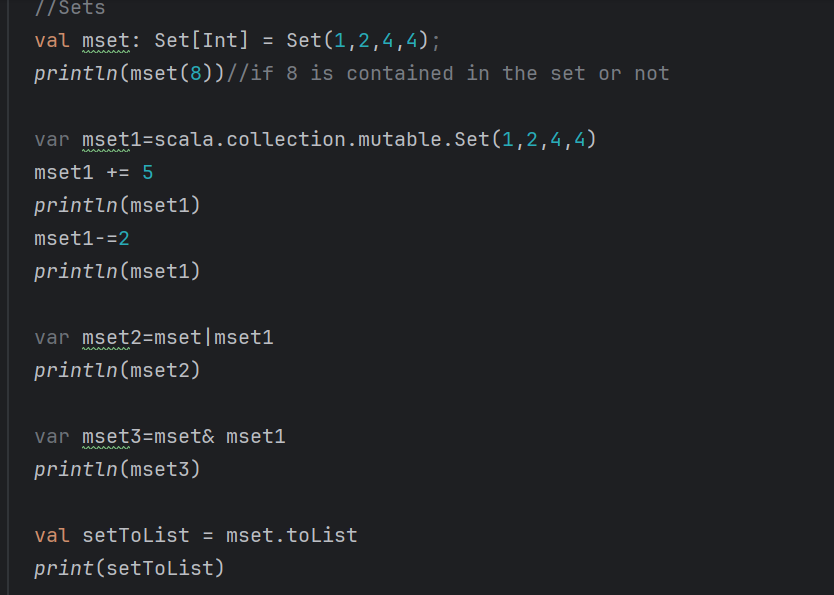
**A screen shot of a computer program

Description automatically generated**

var sum:Int=0;  
mylist.foreach(sum+=\_);  
*println*(sum)

1. **Sets**

Set cannot have duplicate values, are unordered.



1. **Maps**

**A screen shot of a computer program

Description automatically generated**

Mutable maps:

A screen shot of a computer

Description automatically generated

for ((step, inst)<-maptoriches){println(step, inst)}

1. **Tuples**

Scala supports 22 elements to be added in a tuple.

Tuples use . \_<index> to access elements, where <index> starts from 1 (not 0 as in lists or arrays).

A **tuple** in Scala is a collection of fixed-size elements of potentially different types. Unlike collections (e.g., lists or arrays), tuples are immutable and can hold a heterogeneous set of data.

A screen shot of a computer code

Description automatically generated

1. **Map, FlatMap, Flatten, Filter**

-> Map method will iterate over every element of the collection and applies some function on it.

A computer screen shot of a program code

Description automatically generated

->Flatten will flatten list inside lists.

A computer screen shot of text

Description automatically generated

->Flatmap

A screenshot of a computer code

Description automatically generated

->Filter Method

A screenshot of a computer

Description automatically generated

1. **REDUCE, FOLD AND SCAN**

* REDUCE

Reduces the elements of a collection to a single result by applying a binary operation.

Reduce method takes binary operator, and operates on the list.

Reduce left goes from left to right and reduce right applies from right to left.

A computer code with numbers and symbols

Description automatically generated

Reduce Left:

A screenshot of a computer

Description automatically generated

Reduce Right:

A screenshot of a computer

Description automatically generated

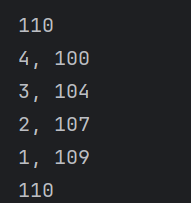
A,b=>3,4 so 3+4=7, then take next element from right which is 2 add 7 to it 2+7 which is 9 and so on.

* FOLD

Fold is similar to reduce but it accepts an additional initial argument which will be used as the initial value before applying the fold operation.

*println*(mlist1.foldLeft(100)(\_+\_))//becomes 110  
*println*(mlist1.foldRight(100)(\_+\_))

*println*(mlist1.foldRight(100)((a,b)=>{*println*(a+", "+b);a+b;}))



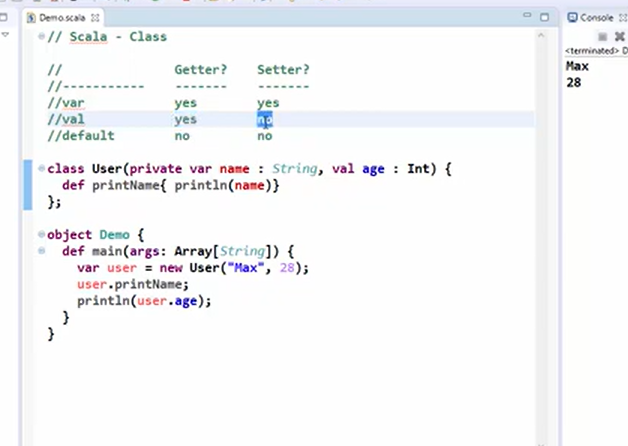
* SCAN

Scan will give intermediate results also, but works same as fold.

*println*(mlist1.scanLeft(100)(\_+\_))

List(100, 101, 103, 106, 110)

1. **CLASSES**

****

Private variables cant be accessed outside the class. So user.name will error out.

No public keyword in scala, so by default all the variables which are not explicitly defined private are public.

**In Scala, the return keyword is used to explicitly return a value from a method or function. However, it's rarely needed because Scala functions automatically return the last expression's value. This makes Scala more concise and aligns with its functional programming principles.**

**Constructors in Scala**

In Scala, a class can have two types of constructors:

1. **Primary Constructor**
   * The main constructor defined as part of the class definition.
   * Executes the class body.
   * Allows for default values and parameterization.
2. **Auxiliary Constructors**
   * Defined using the this keyword.
   * Provide alternative ways to initialize a class.
   * Must call the primary constructor directly or indirectly.

A computer screen shot of a program code

Description automatically generated

Calling the constructors:

A screen shot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

1. **PARAMETERS, FIELDS, PARAMETRIC FIELDS**

**1. Parameters**

* **Definition**: Parameters are variables declared in the parentheses of a method or constructor.
* **Scope**: Parameters are local to the constructor or method in which they are declared.
* **Access**: They cannot be accessed directly outside the constructor or method unless explicitly stored in a field.

**Example:**

scala

Copy code

class Example(param: String) {

// 'param' is a constructor parameter, not accessible outside the constructor

def printParam(): Unit = {

println(param) // Accessible here

}

}

val example = new Example("Hello")

// example.param // This will give a compilation error as 'param' is not a field.

**2. Fields**

* **Definition**: Fields are variables or constants defined at the class level.
* **Scope**: Fields are accessible throughout the class and, depending on access modifiers, may be accessed externally.
* **Access**: Fields are explicitly declared using var (mutable) or val (immutable).

**Example:**

class Example {

var mutableField: String = "Hello" // Mutable field

val immutableField: String = "World" // Immutable field

}

val example = new Example()

println(example.mutableField) // Accessible

example.mutableField = "Hi" // Can modify as it's mutable

println(example.immutableField) // Accessible, but cannot modify

**3. Parametric Fields**

* **Definition**: A parametric field is a class parameter declared with a val, var, or private keyword in the primary constructor. This converts the parameter into a field that is accessible throughout the class and possibly outside, depending on access modifiers.
* **Scope**: They are stored as fields in the class and accessible like regular fields.

**Example:**

scala

Copy code

class Example(val name: String, var age: Int, private var secret: String) {

def printFields(): Unit = {

println(s"Name: $name, Age: $age")

println(s"Secret: $secret")

}

}

val example = new Example("Alice", 25, "Scala")

println(example.name) // Accessible due to 'val'

println(example.age) // Accessible and modifiable due to 'var'

example.age = 30 // Modifiable

// println(example.secret) // Compilation error: 'secret' is private

Var=mutable(value is changeable)

Val=immutable(value not changeable)

1. **TOString, Preconditions in scala**

**toString:**

In Scala, **hashCode** is a method defined in the Any class (the superclass of all classes in Scala) that provides an integer value associated with an object. ClassName@HashCode.

var rat= new Rational(2,3);  
*println*(rat.toString())

op: [basics.Rational@5a01ccaa](mailto:basics.Rational@5a01ccaa)

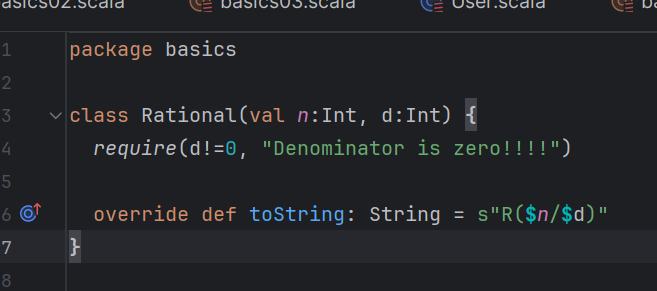
**require:**

It checks if a condition is true and throws an IllegalArgumentException if the condition is false.

require(condition: Boolean, message: => Any = "")

condition: A boolean expression that you want to validate.

message: An optional message that describes the problem if the condition is false. It is lazily evaluated.



var rat= new Rational(2,3);  
*println*(rat.toString())  
var rat1=new Rational(2,0)

gives:

R(2/3)

requirement failed: Denominator is zero!!!!

**Referencing self(using this)**

**Symbolic method**

**A computer screen shot of a program

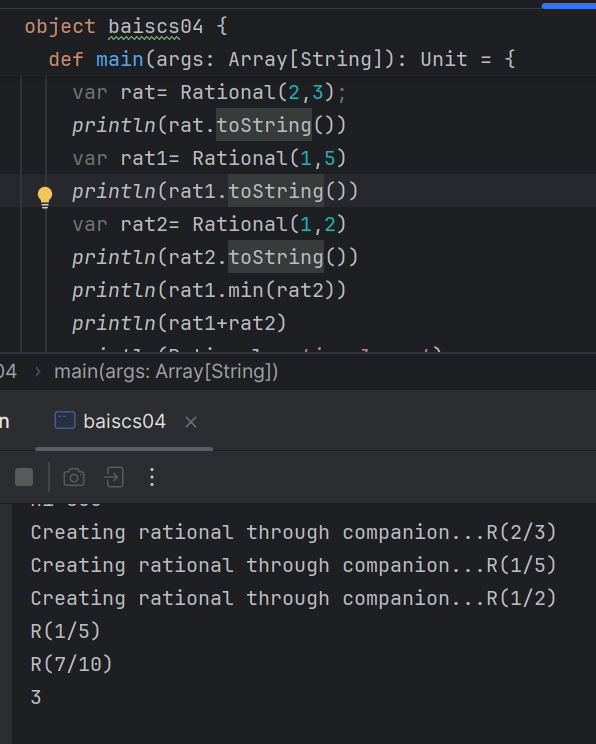
Description automatically generated** ****

1. **Companion Objects**

A companion object in Scala is an object that is associated with a class. The object and the class share the same name and are defined in the same source file. The main purpose of a companion object is to provide functionality that does not belong to individual instances of the class but is still related to that class.

Remember to drop the “new” keyword!

A screen shot of a computer program

Description automatically generated 

In order to be a true companion object, it must bear the same name as a class or trait, be in the same package, and be defined in the same source file.

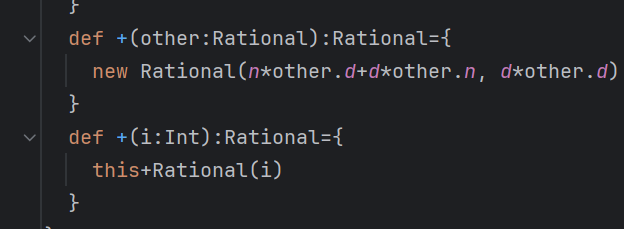
Declaring the constructor as private:

A computer screen with text

Description automatically generated

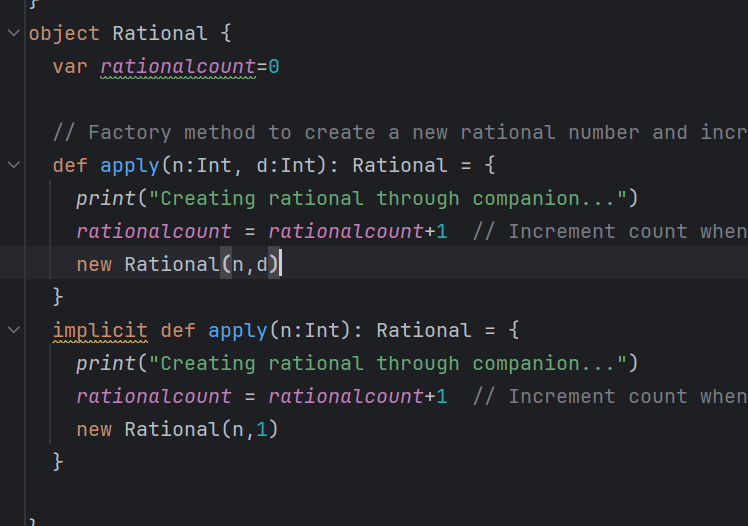
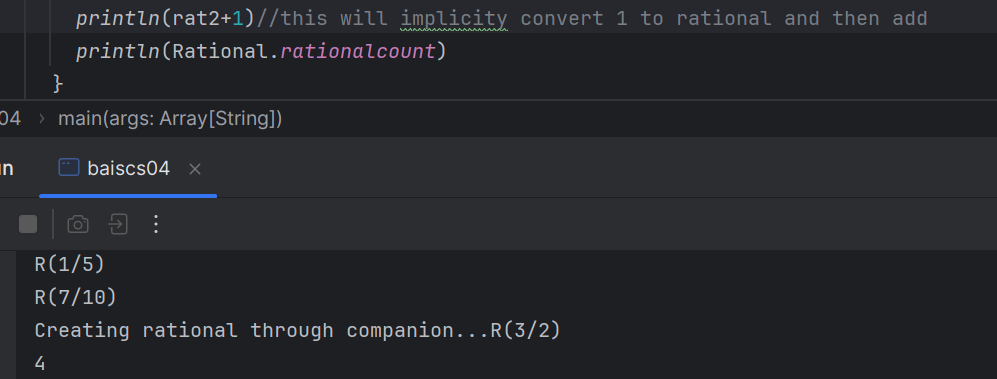
After this we cant use the “new” method to create the objects, instead have to use companion objects as above.

**Function Overloading:**

****A screenshot of a computer program

Description automatically generated

**Implicit conversion:**

1 is implicitly converted to rational.

Import language.implicitConversions to avoid any warnings.

1. **INHERITANCE**

In Scala, **inheritance** allows a class to inherit the characteristics (methods and properties) of another class.

//superclass Animal

A screen shot of a computer program

Description automatically generated

//subclass Cat with overridden method

A screen shot of a computer

Description automatically generated

//subclass Dog with overridden method plus accessing the makesound of superclass as well.

A screen shot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generated

Super is used to access the superclass methods/variables.

Correct, private and protected are keywords in the language and restrict access either to the class or to the class and its subtypes. While there is a public access type in Scala, it is the default for vals and defs, and there is no explicit keyword for it.s

1. **FINAL KEYWORD IN SCALA**

The final keyword in Scala is used to prevent further inheritance or modification. It can be applied to classes, methods, and variables to restrict changes.

1. Final Class cant be inherited:

final class Animal {

def makeSound(): Unit = {

println("Animal makes a sound")

}

}

// This will give a compile-time error:

// class Dog extends Animal // Error: cannot inherit from final class Animal

1. Final Method cannot be overridden.

class Animal {

// Final method cannot be overridden

final def makeSound(): Unit = {

println("Animal makes a sound")

}

}

class Dog extends Animal {

// This will give a compile-time error:

// override def makeSound(): Unit = {

// println("Dog barks")

// }

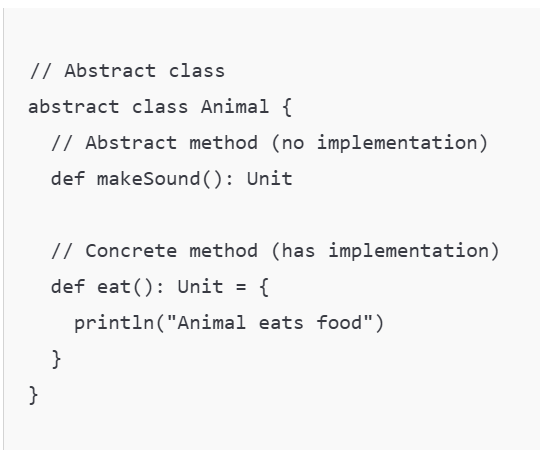
}

1. A final variable cannot be reassigned after initialization
2. **ABSTRACT CLASSES**

An **abstract class** in Scala is a class that cannot be instantiated directly and is used as a blueprint for other classes. Abstract classes can contain both **abstract methods** (methods without implementation) and **concrete methods** (methods with implementation).

Abstract Method: A method in an abstract class that does not have a body (i.e., it is not implemented).

Concrete Method: A method in an abstract class that has a body (i.e., it is implemented).

 A screenshot of a computer code

Description automatically generated

**Key Differences Between final and abstract:**

1. **final Class**:
   * Cannot be extended by other classes.
   * Ensures immutability in inheritance.
2. **Abstract Class**:
   * Cannot be instantiated directly.
   * Can contain abstract methods (without implementation) that must be implemented in subclasses.
   * Can contain concrete methods (with implementation).
3. **LAZY EVALUATION**

: A lazy val is a value that will not be computed until it is accessed for the first time. Lazy evaluation can improve performance by avoiding the evaluation of expensive computations unless absolutely necessary.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

Value e of class lazyv is assigned only when we access it like y.e.

1. **TRAITS**

In Scala, a **trait** is a fundamental building block for defining reusable and modular functionality. Traits are similar to interfaces in Java but more powerful, as they can contain both abstract methods and concrete methods with implementations. A class can extend multiple traits, allowing for a form of multiple inheritance.

When two traits in Scala have methods with the same name, the behavior depends on how the methods are implemented and in what order the traits are mixed into the class. Here's how Scala resolves such conflicts:

**Case 1: Both Traits Have Concrete Methods**

If both traits provide concrete implementations of the same method, Scala requires you to explicitly override the method in the class that mixes in the traits. This ensures that there is no ambiguity about which implementation to use.

**Example:**

trait TraitA {

def greet(): Unit = println("Hello from TraitA!")

}

trait TraitB {

def greet(): Unit = println("Hello from TraitB!")

}

class MyClass extends TraitA with TraitB {

override def greet(): Unit = {

println("Resolving conflict...")

super[TraitA].greet() // Explicitly call TraitA's greet

super[TraitB].greet() // Explicitly call TraitB's greet

}

}

object Main {

def main(args: Array[String]): Unit = {

val obj = new MyClass

obj.greet()

}

}

**Output:**

Resolving conflict...

Hello from TraitA!

Hello from TraitB!

**Case 2: One Trait Has a Concrete Method, and the Other Has an Abstract Method**

If one trait provides a concrete implementation and the other defines the method as abstract, the concrete implementation is used by default unless explicitly overridden in the class.

**Example:**

trait TraitA {

def greet(): Unit = println("Hello from TraitA!")

}

trait TraitB {

def greet(): Unit // Abstract method

}

class MyClass extends TraitA with TraitB

object Main {

def main(args: Array[String]): Unit = {

val obj = new MyClass

obj.greet()

}

}

**Output:**

Hello from TraitA!

**Case 3: Both Traits Have Abstract Methods**

If both traits define abstract methods with the same name and signature, the class or object that mixes them must provide an implementation.

**Example:**

trait TraitA {

def greet(): Unit

}

trait TraitB {

def greet(): Unit

}

class MyClass extends TraitA with TraitB {

override def greet(): Unit = println("Concrete implementation in MyClass")

}

object Main {

def main(args: Array[String]): Unit = {

val obj = new MyClass

obj.greet()

}

}

**Output:**

Concrete implementation in MyClass

**Explanation:**

* Since both TraitA and TraitB only define abstract greet methods, MyClass must provide a concrete implementation.

**Case 4: Diamond Problem (Shared Parent Trait)**

If multiple traits that are mixed into a class inherit from the same parent trait and override its methods, Scala resolves the method calls using the **linearization order** (left-to-right in the inheritance chain).

**Example:**

trait Parent {

def greet(): Unit = println("Hello from Parent!")

}

trait TraitA extends Parent {

override def greet(): Unit = {

println("Hello from TraitA!")

super.greet()

}

}

trait TraitB extends Parent {

override def greet(): Unit = {

println("Hello from TraitB!")

super.greet()

}

}

class MyClass extends TraitA with TraitB

object Main {

def main(args: Array[String]): Unit = {

val obj = new MyClass

obj.greet()

}

}

**Output:**

Hello from TraitB!

Hello from TraitA!

Hello from Parent!

**Explanation:**

1. **Linearization Order**:
   * The order of the traits determines how super calls are resolved.
   * TraitB is mixed in after TraitA, so its greet method is called first.
2. **super.greet() Resolution**:
   * TraitB calls super.greet(), which resolves to TraitA.
   * TraitA calls super.greet(), which resolves to Parent.

**Key Points:**

1. If two traits have methods with the same name:
   * Scala requires explicit overriding in the class if both methods are concrete.
   * Abstract methods can coexist, but the implementing class must provide a concrete implementation.
2. You can use super[TraitName].method to call a specific trait's method in case of conflicts.
3. Linearization order (left-to-right) resolves method calls in a class with multiple mixed-in traits.

This ensures that Scala handles method conflicts in a clear and predictable manner.

Array=mutable

List,Set,Mutable=immutable(default),