Scripts, applications, and real-world workflows

INTRODUCTION TO SCALA

David Venturi
Curriculum Manager, DataCamp

DataCamp

The Scala interpreter

```
$ scala
```

```
Welcome to Scala 2.12.7.
Type in expressions for evaluation. Or try :help.
```

```
scala> 2 + 3
```

```
res0: Int = 5
```

Scala scripts

- A sequence of instructions in a file, executed sequentially
- Useful for smaller projects
- At a command prompt, the `scala` command executes a script by wrapping it in a template and then compiling and executing the resulting program

Run Code Submit Answer

Scala scripts

If we put this code into a file named `game.scala` :

```
// Start game
println("Let's play Twenty-One!")
```

Then run:

```
$ scala game.scala
```

```
Let's play Twenty-One!
```

Interpreted language vs. compiled language

Interpreter: a program that directly executes instructions written in a programming language, without requiring them previously to have been compiled into machine code.

Compiler: a program that translates source code from a high-level programming language to a lower level language (e.g., machine code) to create an executable program.

Scala applications

- Compiled explicitly then run explicitly
- Consist of many source files that can be compiled individually
- Useful for larger programs
- No lag time since applications are precompiled

Scala applications

If we put this code into a file named `Game.scala` :

```
object Game extends App {
    println("Let's play Twenty-One!")
}
```

First, **compile** with `scalac` :

```
$ scalac Game.scala
```

Second, **run** with `scala` :

```
$ scala Game
```

Scala applications

If we put this code into a file named `Game.scala` :

```
object Game extends App {
    println("Let's play Twenty-One!")
}
```

First, **compile** with `scalac` :

```
$ scalac Game.scala
```

Second, **run** with `scala` :

```
$ scala Game
```

```
Let's play Twenty-One!
```

Pros and cons of compiled languages

Pros

- Increased performance once compiled

Cons

- It takes time to compile code

Scala workflows

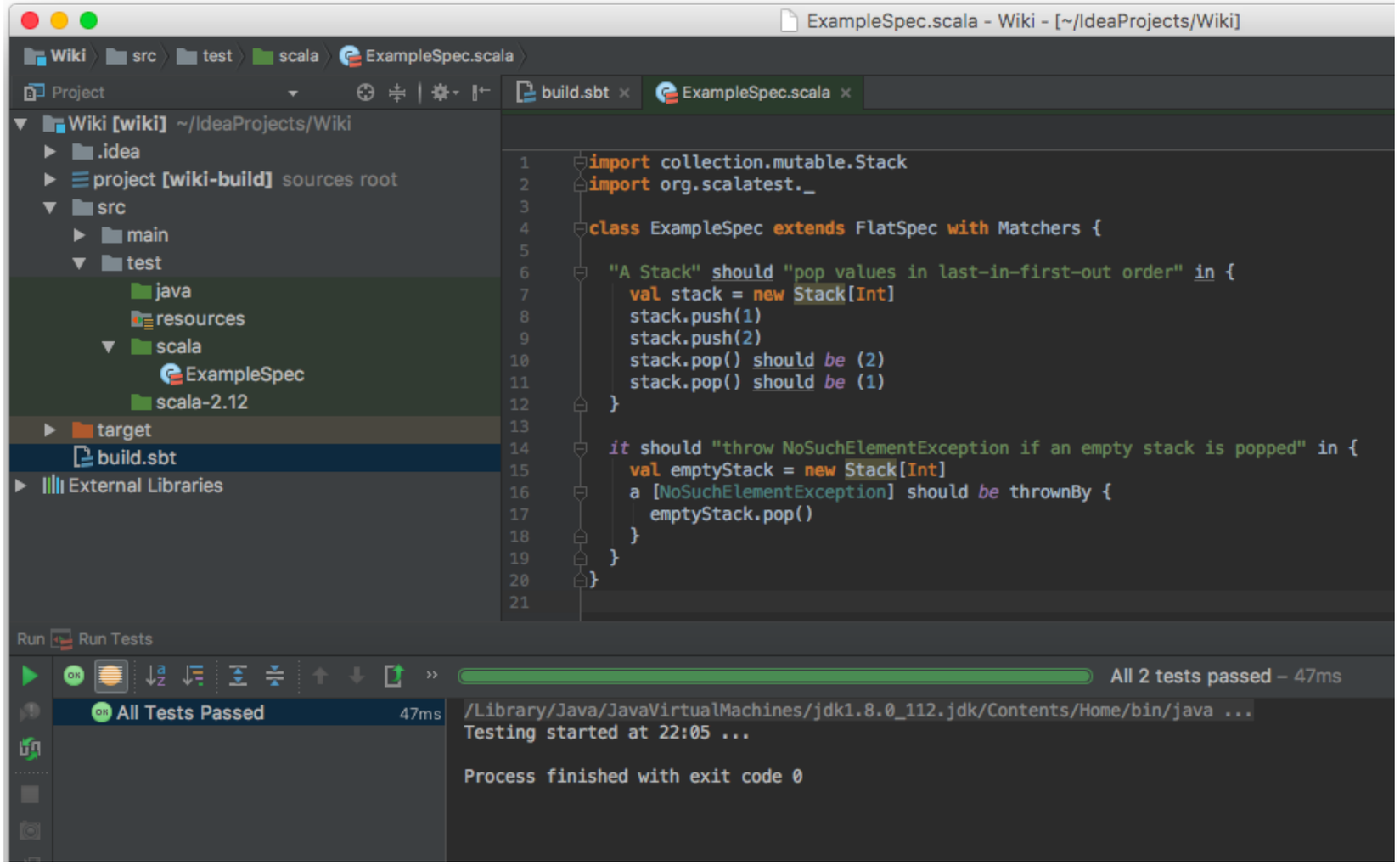
There are two main ways people prefer to work in Scala:

- Using the command line
- Using an IDE (integrated development environment)

IDE

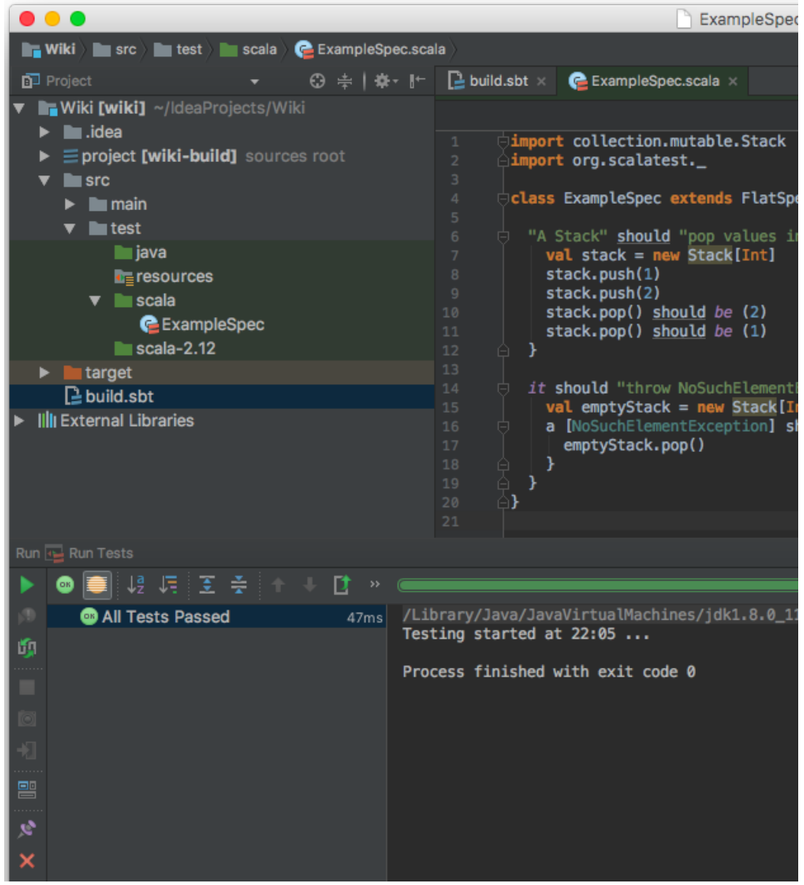
- Especially useful for larger projects with many files
- **IntelliJ IDEA**: most commonly-used IDE by Scala developers

IntelliJ IDEA



sbt

IntelliJ IDEA

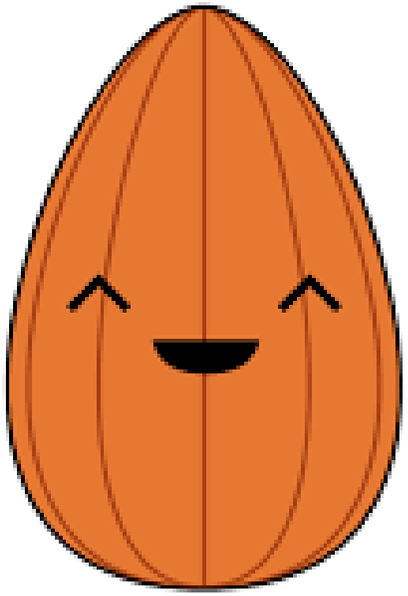


sbt

- "simple build tool"
- Compiles, runs, and tests

Scala applications

Scala kernel for Jupyter



[1] https://almond.sh/

Let's practice!

INTRODUCTION TO SCALA

DataCamp

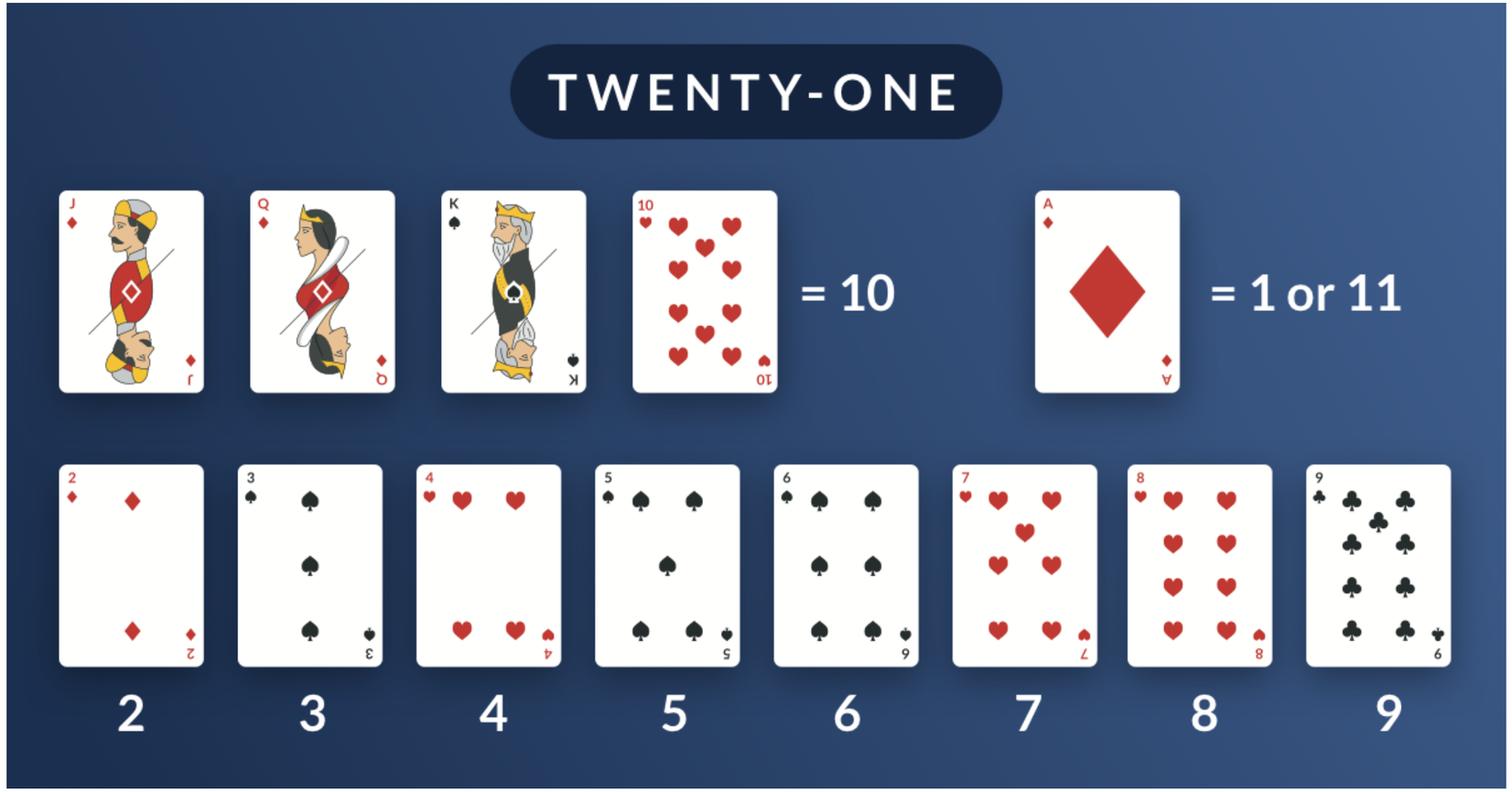
Functions

INTRODUCTION TO SCALA

David Venturi
Curriculum Manager, DataCamp

DataCamp

Twenty-One



[1] http://bit.ly/twenty_one_wikipedia

Functions

In this course

- Understand what functions are
- Call a function

In following courses

- Understand the anatomy of a function
- Define a function
- More...

What is a function?

What do functions do?

- Functions are invoked with a list of arguments to produce a result

What are the parts of a function?

1. Parameter list
2. Body
3. Result type

What is a function?

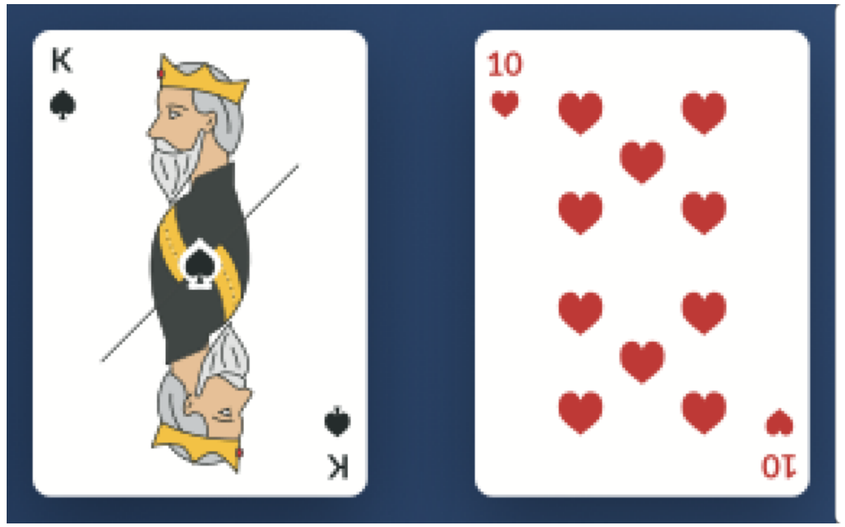
What do functions do?

- Functions are invoked with a list of arguments to produce a result

What are the parts of a function?

1. Parameter list
2. **Body**
3. Result type

A specific question



Hand value

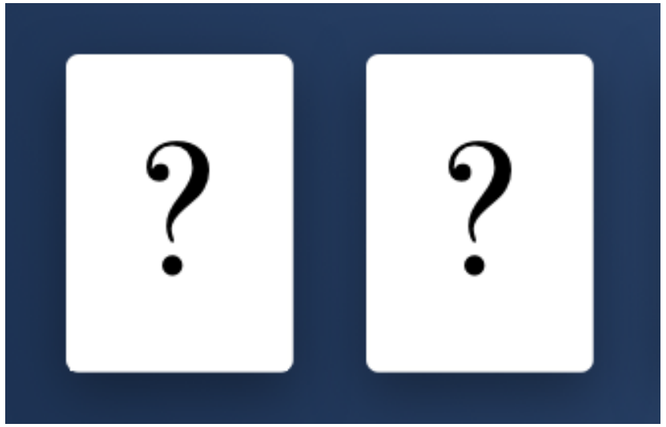
=

20

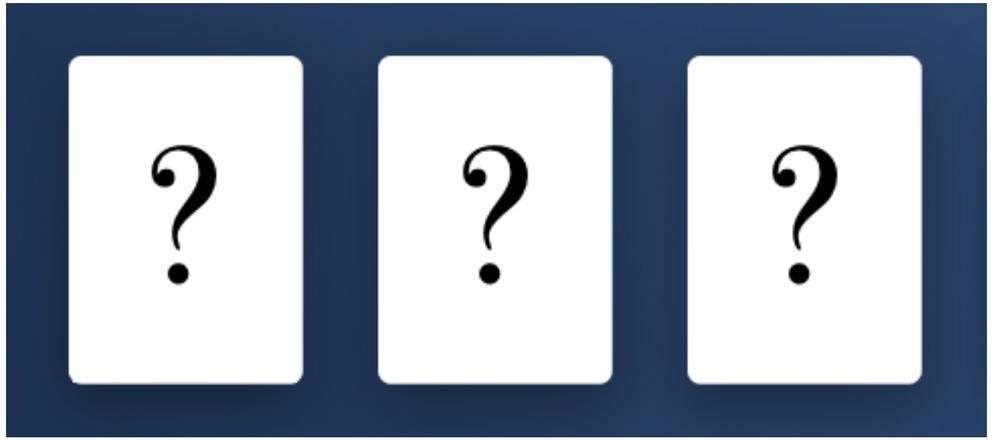
```
scala> 20 > 21
```

```
false
```

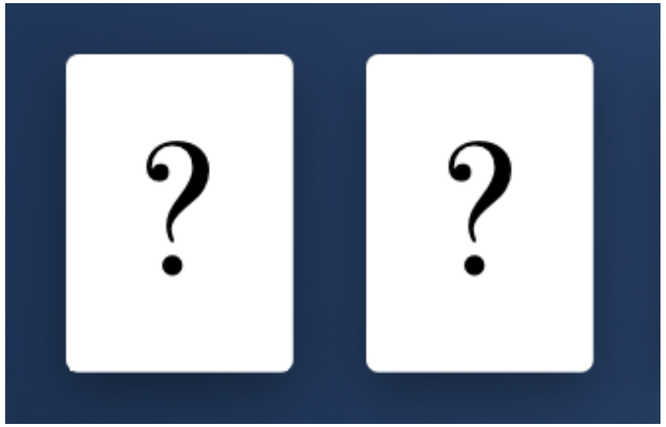
Generalizing that question



Generalizing that question

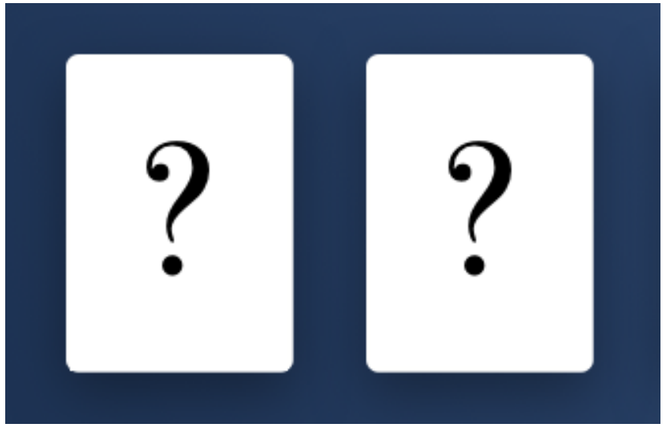


Generalizing that question



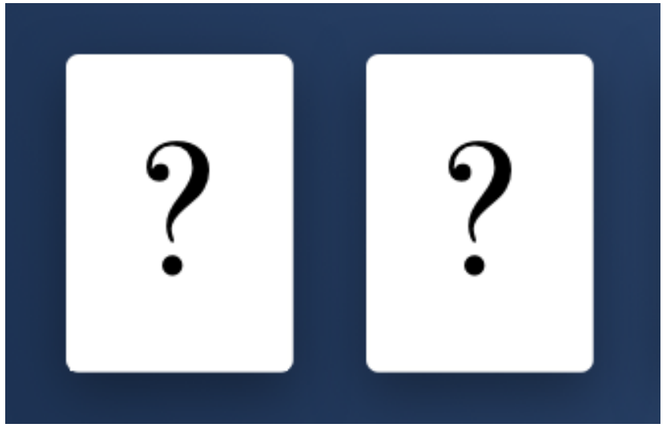

```
scala> 20 > 21
```

Generalizing that question




```
scala> hand > 21
```

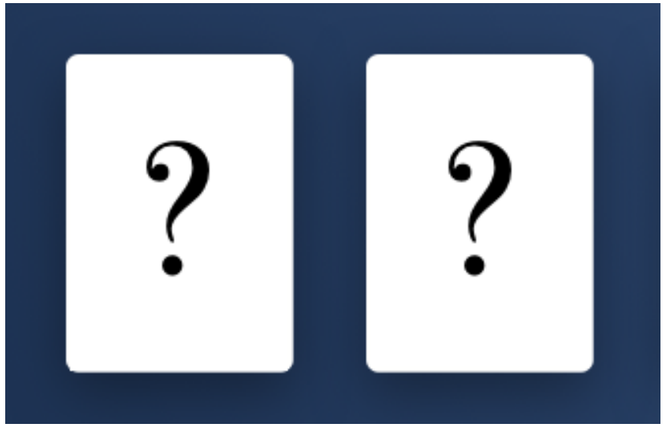
The bust function



```scala
// Define a function to determine if hand busts
def bust(hand: Int): Boolean = {
  hand > 21
}
```

- Function body: follows equals sign `=` in curly braces `{}`

The bust function



```scala
// Define a function to determine if hand busts
def bust(hand: Int) = {
  hand > 21
}
```

- Function body: follows equals sign `=` in curly braces `{}`

What do functions do again?

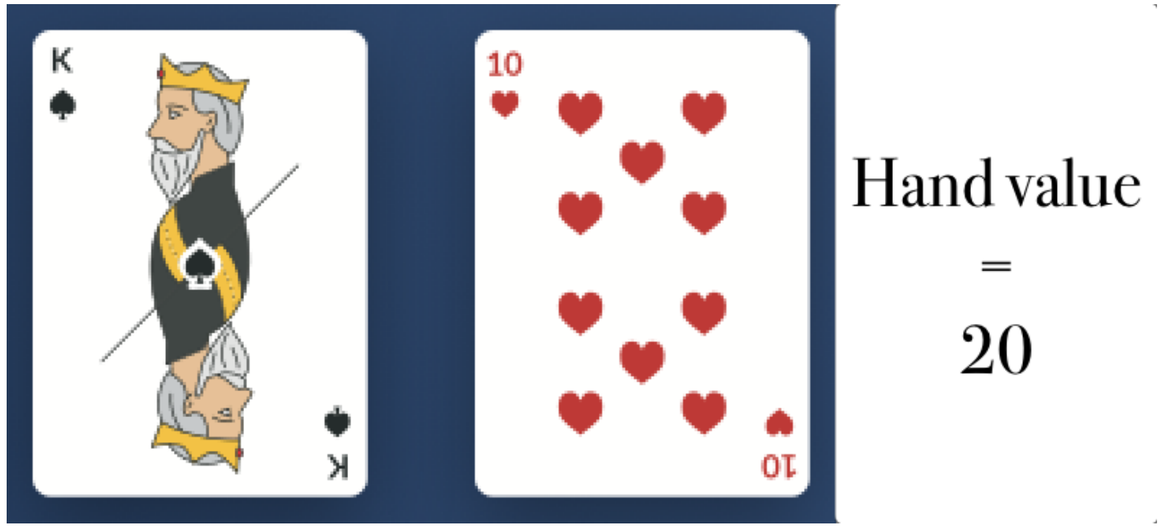
- Functions are invoked with a list of arguments to **produce a result**
- Functions are first-class **values**

```
// Define a function to determine if hand busts
def bust(hand: Int) = {
  hand > 21
}

println(bust(20))
println(bust(22))
```

```
false
true
```

Call a function with variables




```
println(bust(kingSpades + tenHearts))
```

```
false
```

Sneak peek at future courses

Kinds of functions

- **Method**: functions that are members of a class, trait, or singleton object
- **Local function**: functions that are defined inside other functions
- **Procedure**: functions with the result type of `Unit`
- **Function literal**: anonymous functions in source code (at run time, function literals are instantiated into objects called *function values*)

Let's practice!

INTRODUCTION TO SCALA

DataCamp

Arrays

INTRODUCTION TO SCALA

David Venturi
Curriculum Manager, DataCamp

DataCamp

Collections

- Mutable collections
  - can be updated or extended in place
- Immutable collections
  - never change

Array

- Mutable sequence of objects that share the same type
- **Parameterize an array**: configure its types and parameter values
- **Initialize elements of an array**: give the array data

```
scala> val players = Array("Alex", "Chen", "Marta")
```

```
players: Array[String] = Array(Alex, Chen, Marta)
```

Array

- **Parameterize an array**: configure its types and parameter values

```
scala> val players = new Array[String](3)
```

```
players: Array[String] = Array(null, null, null)
```

  - Type parameter: `String`
  - Value parameter: `length` which is 3

Array

- **Parameterize an array**: configure its types and parameter values

```
scala> val players: Array[String] = new Array[String](3)
```

```
players: Array[String] = Array(null, null, null)
```

  - Type parameter: `String`
  - Value parameter: `length` which is 3

Array

- **Parameterize an array**: configure its types and parameter values
- **Initialize elements of an array**: give the array data

```
scala> players(0) = "Alex"
scala> players(1) = "Chen"
scala> players(2) = "Marta"
scala> players
```

```
res3: Array[String] = Array(Alex, Chen, Marta)
```

Arrays are mutable

```
scala> val players = Array("Alex", "Chen", "Marta")
```

```
players: Array[String] = Array(Alex, Chen, Marta)
```

```
scala> players(0) = "Sindhu"
```

```
res5: Array[String] = Array(Sindhu, Chen, Marta)
```

Arrays are mutable

```
scala> val players = Array("Alex", "Chen", "Marta")
```

```
players: Array[String] = Array(Alex, Chen, Marta)
```

```
scala> players(0) = 500
```

```
<console>:13: error: type mismatch;
 found   : Int(500)
 required: String
       players(0) = 500
```

Recommendation: use val with Array

```
scala> var players = Array("Alex", "Chen", "Marta")
```

```
players: Array[String] = Array(Alex, Chen, Marta)
```

Elements can change

```
scala> players(0) = "Sindhu"
```

`players` can be reassigned

```
scala> players = new Array[String](5)
scala> players
```

```
res2: Array[String] = Array(null, null, null, null, null)
```

Scala nudges us towards immutability

immutability

The Any supertype

```
scala> val mixedTypes = new Array[Any](3)
```

```
mixedTypes: Array[Any] = Array(null, null, null)
```

```
scala> mixedTypes(0) = "I like turtles"
scala> mixedTypes(1) = 5000
scala> mixedTypes(2) = true
scala> mixedTypes
```

```
res3: Array[Any] = Array(I like turtles, 5000, true)
```

Let's practice!

INTRODUCTION TO SCALA

DataCamp

Lists

INTRODUCTION TO SCALA

David Venturi
Curriculum Manager, DataCamp

DataCamp

Collections

- Mutable collections: can be updated or extended in place
  - `Array` : mutable sequence of objects with the same type
- Immutable collections: never change
  - `List` : immutable sequence of objects with the same type

Lists have a type parameter

`Array`

```
scala> val players = Array("Alex", "Chen", "Marta")
```

```
players: Array[String] = Array(Alex, Chen, Marta)
```

`List`

```
scala> val players = List("Alex", "Chen", "Marta")
```

```
players: List[String] = List(Alex, Chen, Marta)
```

How Lists are useful while immutable

- `List` has methods, like all of Scala collections
  - *Method: a function that belongs to an object*
- There are **many** `List` methods
  - `myList.drop()`
  - `myList.mkString(", ")`
  - `myList.length`
  - `myList.reverse`

[1] http://bit.ly/scala_list_documentation

How Lists are useful while immutable

```
scala> val players = List("Alex", "Chen", "Marta")
```

```
players: List[String] = List(Alex, Chen, Marta)
```

```
scala> val newPlayers = "Sindhu" :: players
```

```
newPlayers: List[String] = List(Sindhu, Alex, Chen, Marta)
```

How Lists are useful while immutable

```
scala> var players = List("Alex", "Chen", "Marta")
```

```
players: List[String] = List(Alex, Chen, Marta)
```

```
scala> players  = "Sindhu" :: players
```

```
players: List[String] = List(Sindhu, Alex, Chen, Marta)
```

cons (::)

- Prepends a new element to the ***beginning*** of an existing `List` and returns the resulting `List`

```
scala> val players = List("Alex", "Chen", "Marta")
```

```
players: List[String] = List(Alex, Chen, Marta)
```

```
scala> val newPlayers = "Sindhu" :: players
```

```
newPlayers: List[String] = List(Sindhu, Alex, Chen, Marta)
```

  - An append operation exists, but its rarely used

[1] http://bit.ly/append_list_inefficient

Nil

- `Nil` is an empty list

```
scala> Nil
```

```
res0: scala.collection.immutable.Nil.type = List()
```

Nil

- A common way to initialize new lists combines `Nil` and `::`

```
scala> val players = "Alex" :: "Chen" :: "Marta" :: Nil
```

```
players: List[String] = List(Alex, Chen, Marta)
```

```
scala> val playersError = "Alex" :: "Chen" :: "Marta"
```

```
<console>:11: error: value :: is not a member of String
       val playersError = "Alex" :: "Chen" :: "Marta"
```

Concatenating Lists

- `:::` for concatenation

```
val playersA = List("Sindhu", "Alex")
val playersB = List("Chen", "Marta")

val allPlayers = playersA ::: playersB

println(playersA + " and " + playersB + " were not mutated,")
println("which means " + allPlayers + " is a new List.")
```

```
List(Sindhu, Alex) and List(Chen, Marta) were not mutated,
which means List(Sindhu, Alex, Chen, Marta) is a new List.
```

Scala nudges us towards immutability

immutability

Pros and cons of immutability

Pros

- Your data won't be changed inadvertently
- Your code is easier to reason about
- You have to write fewer tests

Cons

- More memory required due to data copying

Let's practice!

INTRODUCTION TO SCALA

DataCamp