

# Essentials of Scala

---

Eric Smith

@eric\_s\_smith

github: ericssmith

# Disclaimer

---

- This is **not** a survey of the features of Scala.
- There are many important features we will not cover.
- The goal **is** to make you sufficiently literate in Scala to understand a few key features -- features that are probably missing from your current programming language.

# Runs on Java Virtual Machine (JVM)

---

- JVM is a “virtual” processor that runs as a process on a host OS. It abstracts away hardware and OS implementation. Manages memory, threading, input/output.
- User programs are compiled to “byte code” which is executed by the virtual processor.
- “Just-in-time” (JIT) compilation -- compiled code can be optimized during execution to run almost as fast as “native” machine code.

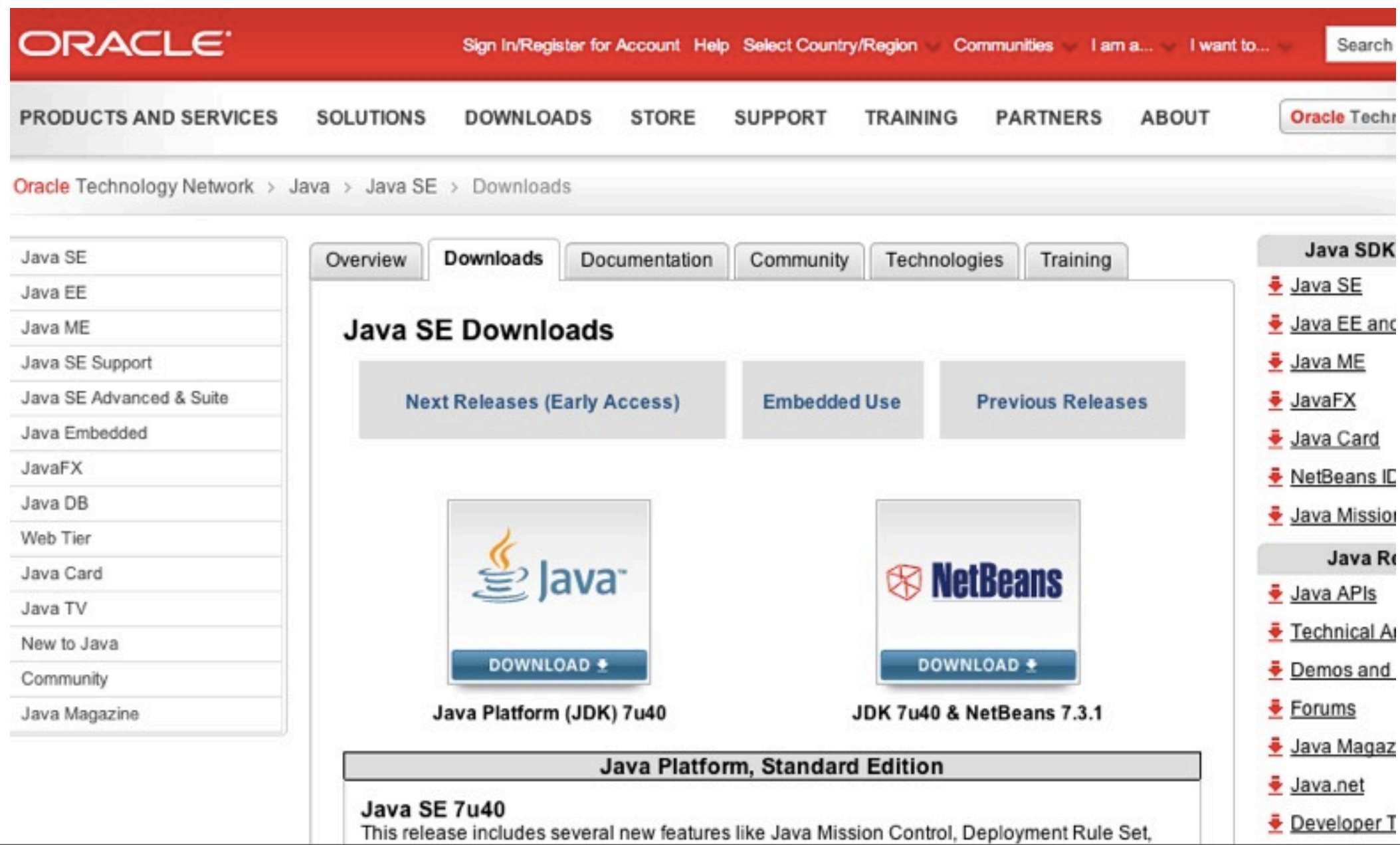
# Check if you have Java Development Kit (JDK)

---

```
~ $ java -version
java version "1.6.0_37"
Java(TM) SE Runtime Environment (build 1.6.0_37-b06-434-11M4406)
Java HotSpot(TM) 64-Bit Server VM (build 20.12-b01-434, mixed mode)
~ $ █
```

# Download Java Development Kit (JDK)

- <http://www.oracle.com/technetwork/java/javase/downloads/>
- Or search for “jdk downloads”



The screenshot shows the Oracle Java SE Downloads page. The top navigation bar includes the Oracle logo, links for Sign In/Register, Help, Select Country/Region, Communities, I am a..., I want to..., and a Search bar. Below this is a secondary navigation bar with links for PRODUCTS AND SERVICES, SOLUTIONS, DOWNLOADS, STORE, SUPPORT, TRAINING, PARTNERS, and ABOUT. The breadcrumb trail reads: Oracle Technology Network > Java > Java SE > Downloads.

On the left, a sidebar lists various Java products: Java SE, Java EE, Java ME, Java SE Support, Java SE Advanced & Suite, Java Embedded, JavaFX, Java DB, Web Tier, Java Card, Java TV, New to Java, Community, and Java Magazine.

The main content area has tabs for Overview, Downloads (selected), Documentation, Community, Technologies, and Training. The title is "Java SE Downloads". Below the title are three buttons: "Next Releases (Early Access)", "Embedded Use", and "Previous Releases".


















There are two main download options displayed as cards:

- Java Platform (JDK) 7u40**: Features the Java logo and a "DOWNLOAD" button.
- JDK 7u40 & NetBeans 7.3.1**: Features the NetBeans logo and a "DOWNLOAD" button.

Below these cards, a section titled "Java Platform, Standard Edition" highlights the "Java SE 7u40" release, noting that it includes several new features like Java Mission Control, Deployment Rule Set, etc.

On the right, a sidebar lists links for Java SDK (Java SE, Java EE and, Java ME, JavaFX, Java Card, NetBeans IDE, Java Mission Control) and Java Resources (Java APIs, Technical Articles, Demos and, Forums, Java Magazine, Java.net, Developer Tools).

# Choose JDK installer for OS

Java SE Development Kit 7u40		
You must accept the <a href="#">Oracle Binary Code License Agreement for Java SE</a> to download this software.		
<input type="radio"/> Accept License Agreement <input checked="" type="radio"/> Decline License Agreement		
Product / File Description	File Size	Download
Linux ARM v6/v7 VFP Hard Float ABI	67.62 MB	 <a href="#">jdk-7u40-linux-arm-vfp-hflt.tar.gz</a>
Linux ARM v6/v7 VFP Soft Float ABI	67.62 MB	 <a href="#">jdk-7u40-linux-arm-vfp-sflt.tar.gz</a>
Linux x86	115.55 MB	 <a href="#">jdk-7u40-linux-i586.rpm</a>
Linux x86	132.83 MB	 <a href="#">jdk-7u40-linux-i586.tar.gz</a>
Linux x64	116.83 MB	 <a href="#">jdk-7u40-linux-x64.rpm</a>
Linux x64	131.63 MB	 <a href="#">jdk-7u40-linux-x64.tar.gz</a>
Mac OS X x64	183.35 MB	 <a href="#">jdk-7u40-macosx-x64.dmg</a>
Solaris x86 (SVR4 package)	139.84 MB	 <a href="#">jdk-7u40-solaris-i586.tar.Z</a>
Solaris x86	95.29 MB	 <a href="#">jdk-7u40-solaris-i586.tar.gz</a>
Solaris x64 (SVR4 package)	24.43 MB	 <a href="#">jdk-7u40-solaris-x64.tar.Z</a>
Solaris x64	16.17 MB	 <a href="#">jdk-7u40-solaris-x64.tar.gz</a>
Solaris SPARC (SVR4 package)	139.06 MB	 <a href="#">jdk-7u40-solaris-sparc.tar.Z</a>
Solaris SPARC	98.07 MB	 <a href="#">jdk-7u40-solaris-sparc.tar.gz</a>
Solaris SPARC 64-bit (SVR4 package)	23.74 MB	 <a href="#">jdk-7u40-solaris-sparcv9.tar.Z</a>
Solaris SPARC 64-bit	18.18 MB	 <a href="#">jdk-7u40-solaris-sparcv9.tar.gz</a>
Windows x86	123.46 MB	 <a href="#">jdk-7u40-windows-i586.exe</a>
Windows x64	125.25 MB	 <a href="#">jdk-7u40-windows-x64.exe</a>

Java SE Development Kit 7u40 Demos and Sample Downloads

 [De](#)

 [Tu](#)

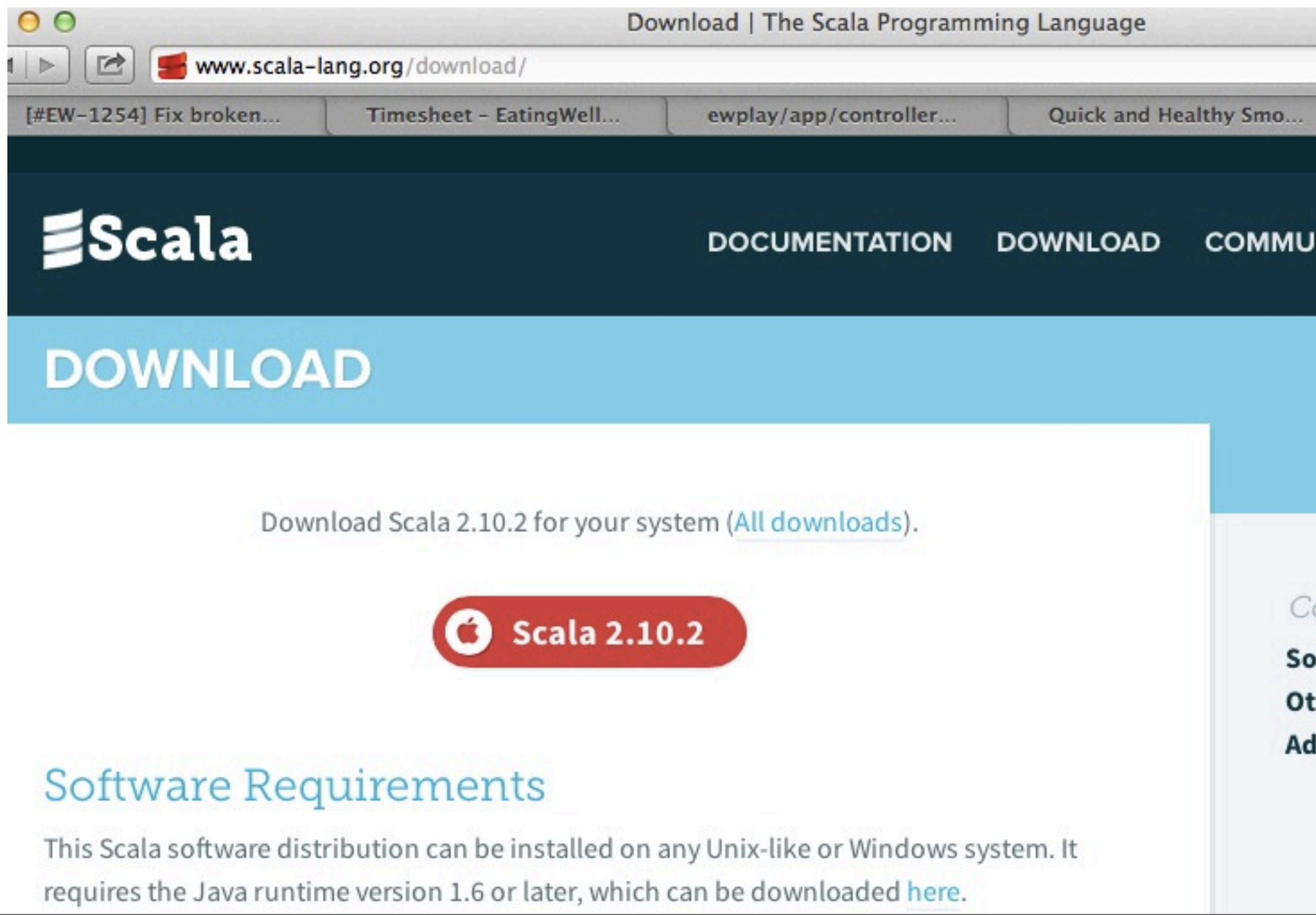
 [Ja](#)



# You could download Scala . . .

---

- <http://www.scala-lang.org/download>





. . . but we want the Scala Build Tool (SBT)

---

- <http://scala-sbt.org>

The screenshot shows the 'Setup' page of the Scala Build Tool (SBT) website. The page has a dark blue header with navigation links: 'Home', 'Documentation', and 'Download'. Below the header, the word 'Setup' is prominently displayed. A search bar is located in the top right corner. A version selector shows 'Version 0.13.0'. Breadcrumbs indicate the current location: 'Welcome! | Contents > Getting Started'. The main content area is titled 'Overview' and lists steps for creating an sbt project. A sidebar on the right contains a 'Contents' menu with links to 'Setup', 'Overview', 'Installing', 'Getting Started', 'Manual Installation', 'Using SBT', and 'Next'.

Home Documentation Download

# Setup

Version 0.13.0

« Welcome! | Contents > Getting Started

## Overview

To create an sbt project, you'll need to take these steps:

- Install sbt and create a script to launch it.
- Setup a simple *hello world* project
  - Create a project directory with source files in it.
  - Create your build definition.
- Move on to *running* to learn how to run sbt.
- Then move on to *.sbt build definition* to learn more about build definitions.

## Installing sbt

sbt provides several packages for different operating systems or you can do [Manual Installation](#).

Officially supported packages:

- [MSI for Windows](#)
- [ZIP or TGZ packages](#)
- [RPM package](#)
- [DEB package](#)

### Contents

- Setup
  - Overview
  - Installing
    - Mac
    - M
    - H
  - Getting Started
  - Manual Installation
    - U
    - W
- Tips and Tricks
- Next



# SBT

---

- MSI installer works great on Windows (sets PATH)
- ZIP/TGZ on Mac/BSD/Linux
  - Set PATH in .bashrc or .profile *to run from any folder*  

```
export PATH=$HOME/Downloads/sbt/bin/:$PATH
```
- First time you run it, downloads a boatload of dependencies

# SBT configuration

---

- Configure SBT in *build.sbt* in project folder
- Examples: by default, uses version of Scala used to build SBT itself
  - Set version of Scala explicitly:

```
scalaVersion := "2.10.2"
```

# SBT project folders

---

- SBT expects this directory structure.
- *We aren't going to use this until later.*

```
~/projects/sbtdemo $ mkdir -p src/{main,test}/{java,resources,scala}
~/projects/sbtdemo $ mkdir lib project target
~/projects/sbtdemo $
```

```
.
|-lib
|-project
|-src
|---main
|-----java
|-----resources
|-----scala
|---test
|-----java
|-----resources
|-----scala
|-target
```

# Run SBT in project folder

---

- Type 'sbt'
- SBT prompt is ">"

```
~/projects/sbtdemo $ sbt
[info] Loading global plugins from /Users/esmith/.sbt/plugins
[info] Set current project to default-38c8ae (in build file:/Users/esmith/projects/sbtdemo/)
> █
```

# Run Scala interpreter

---

- Type “console” at SBT prompt.
- “scala>” is the interpreter prompt.

```
~/projects/sbtdemo $ sbt
[info] Loading global plugins from /Users/esmith/.sbt/plugins
[info] Set current project to default-38c8ae (in build file:/Users/esmith/projects/sbtdemo/)
> console
[info] Updating {file:/Users/esmith/projects/sbtdemo/}default-38c8ae...
[info] Resolving org.scala-lang#scala-library;2.10.2 ...
[info] Done updating.
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.10.2 (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_37).
Type in expressions to have them evaluated.
Type :help for more information.

scala> █
```

# Scala REPL

---

- Read-Eval-Print-Loop
- Value of expression you enter is assigned to an auto-generated variable:
  - Auto-created variable for value. Eg. 'res2'
  - Type of value, separated by colon. Eg. 'res2: Int'
  - Value of expression, separated by '=', the assignment operator

```
scala> 42
res2: Int = 42

scala> 2 + 2
res3: Int = 4

scala> res2 + res3
res4: Int = 46

scala> █
```



# Variables

---

- Immutable variables with keyword “val”
- Mutable “variables” with keyword “var”

```
scala> val foo = 42
foo: Int = 42

scala> foo = 3
<console>:8: error: reassignment to val
      foo = 3
        ^

scala> var bar = 42
bar: Int = 42

scala> bar = 3
bar: Int = 3

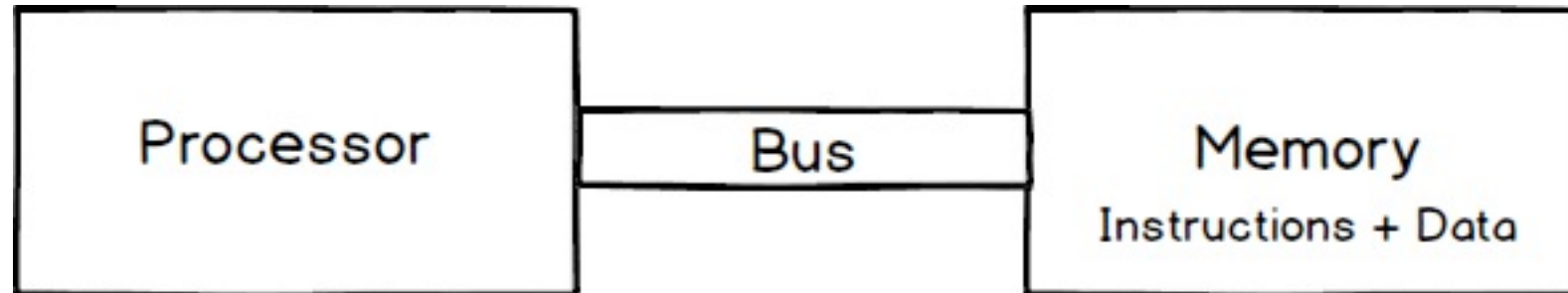
scala> bar
res5: Int = 3

scala> █
```

# ‘Mainstream’ programming: 1945 - 2015

---

- Imperative programming: thin abstraction layer over machine architecture
- “Von Neumann” computer:
  - Instructions & data stored as bits in volatile memory
  - Processor communicates with memory by fixed-width connection (“bus”)



- “Variables” in imperative programming imitate memory locations
- Assignment statements imitate fetch, store, and arithmetic
- Control structures imitate jump and test instructions
  - If/else, loop, break, return, etc.

# Imperative programming - examples

---

```
def is_palindrome(word):  
    i = 0  
    j = len(word)-1  
  
    while i<j:  
        if word[i] != word[j]:  
            return False  
        i = i + 1  
        j = j - 1  
  
    return True
```

```
is_palin:  
    mov dl, [ebx]  
    cmp [eax], dl  
    jne not_palin  
    inc ebx  
    dec eax  
    loop is_palin  
    push msg1  
    call puts  
    add esp,4  
    jmp done
```

```
not_palin:  
    push msg2  
    call puts  
    add esp,4
```

```
done:  
    ret
```

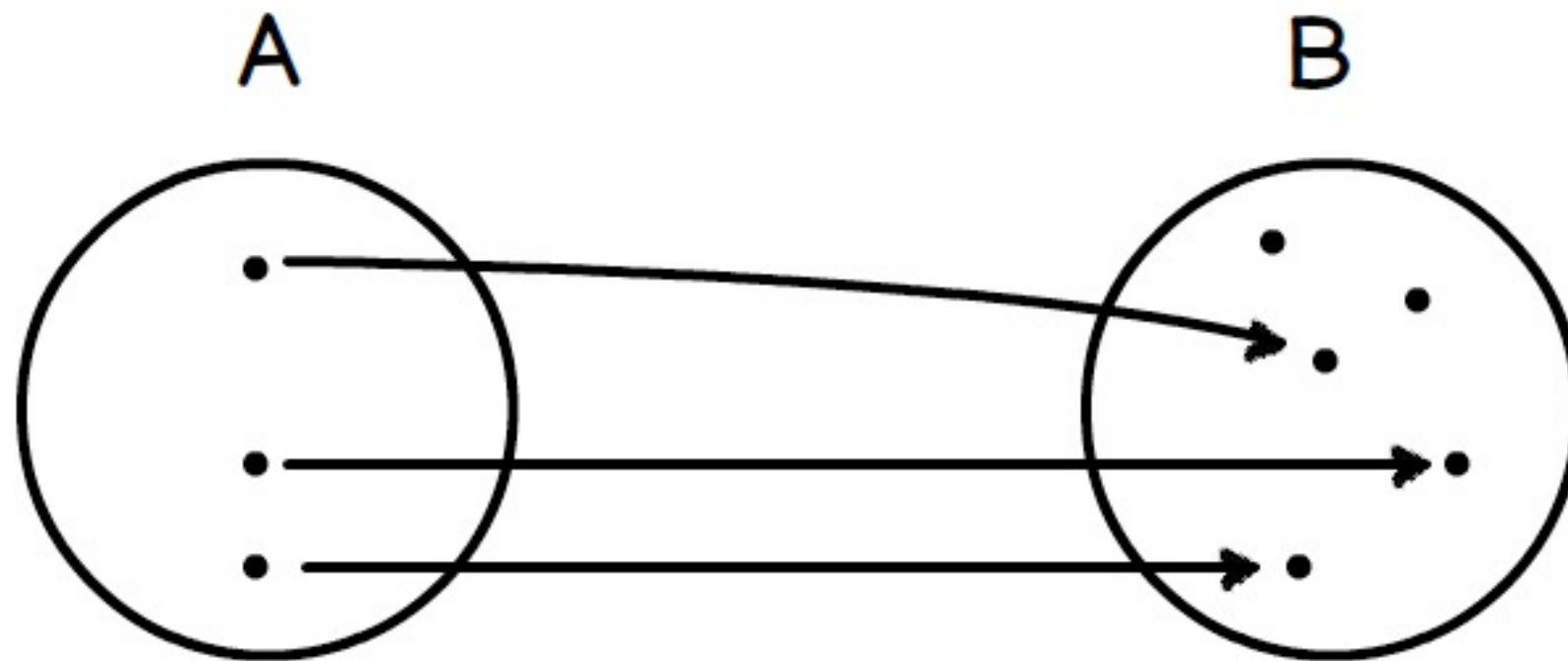
# Functional programming - example

---

```
isPalin w = w == reverse w
```

# Function

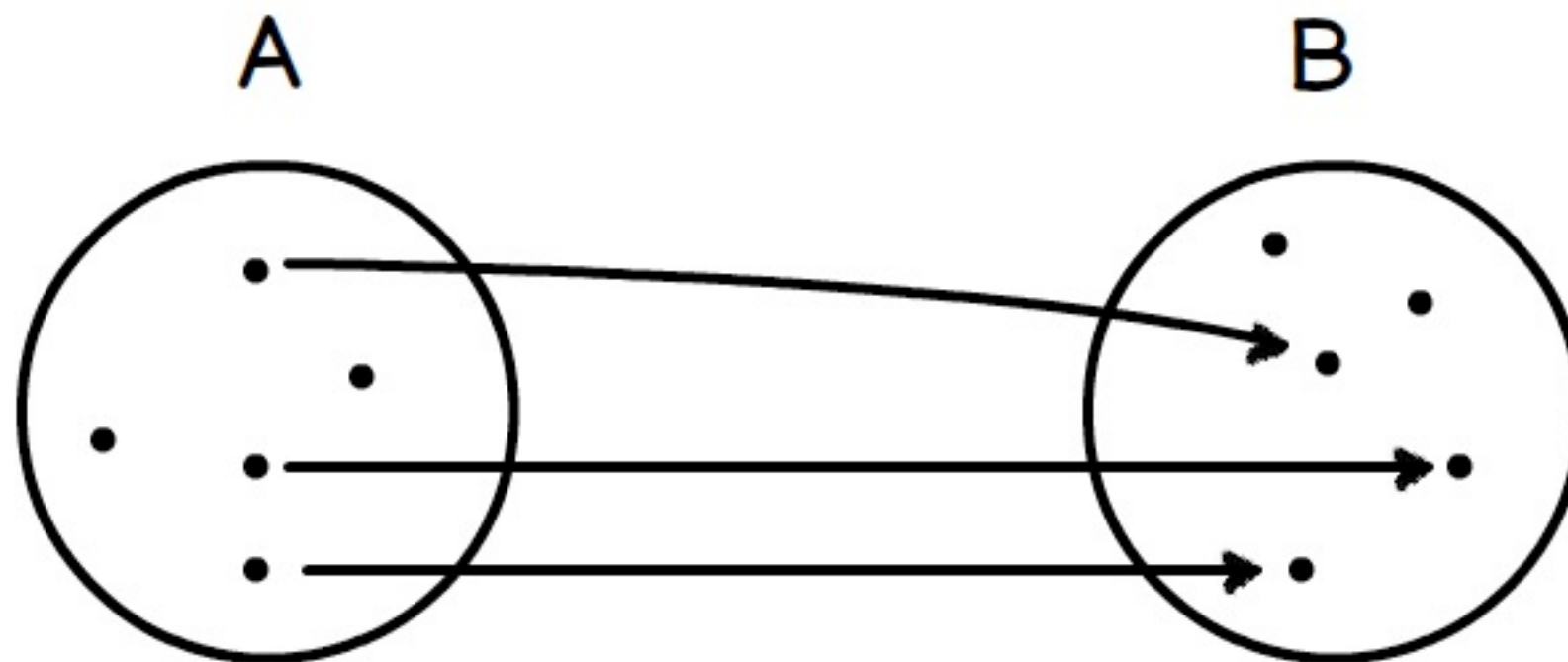
---



- “A function is a relation between two sets (or ‘types’) called the *domain* and *co-domain*. To each element in the domain the function assigns a unique element of the co-domain.”

Is this a function? No

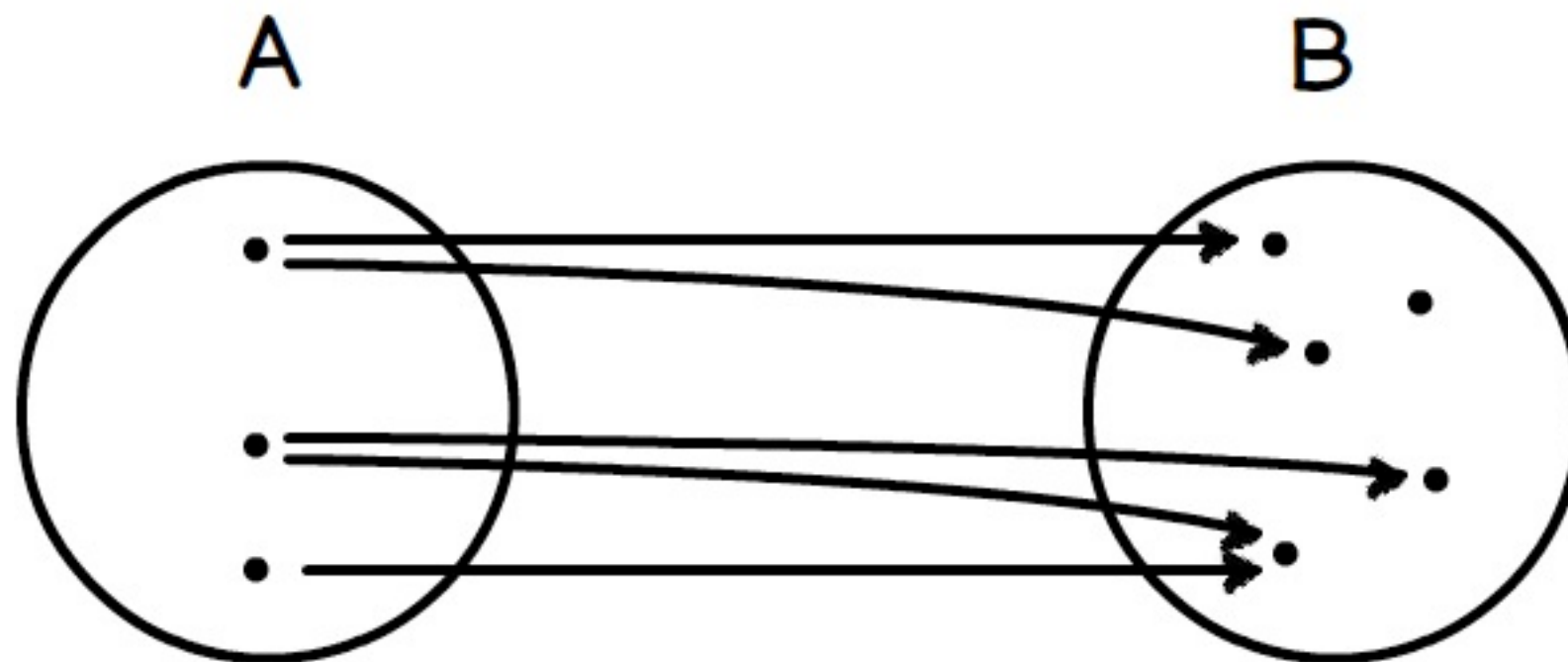
---





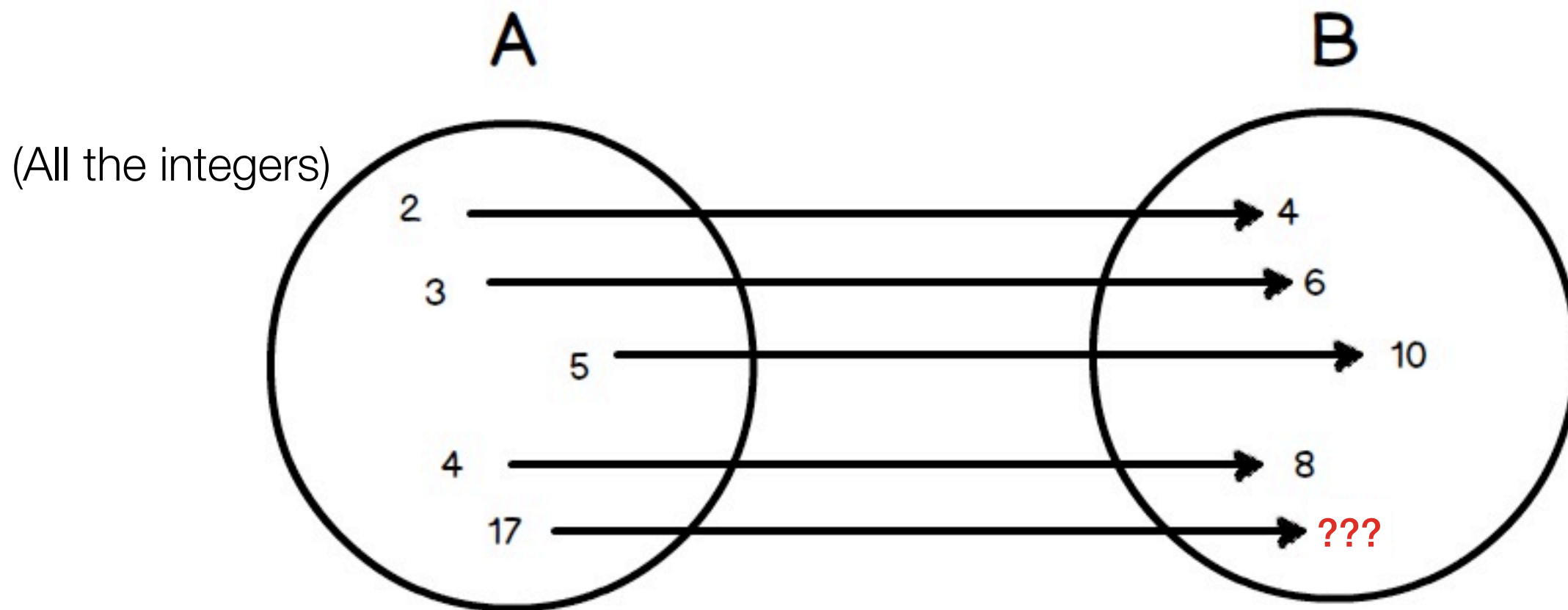
Is this a function? No

---



# Abstraction

---



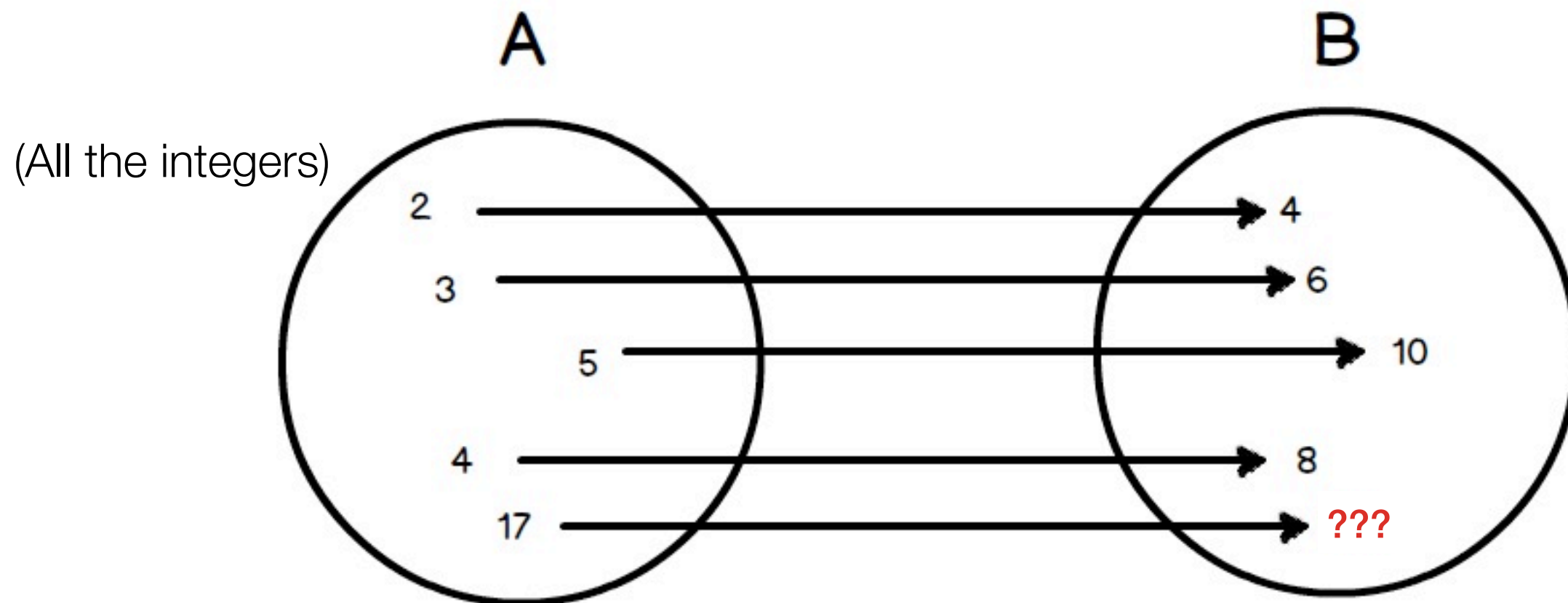
“twice”

$$f(x) = x + x$$

$$f(x) = 2 * x$$

# Abstraction & Application

---



“twice”

$$f(x) = x + x$$

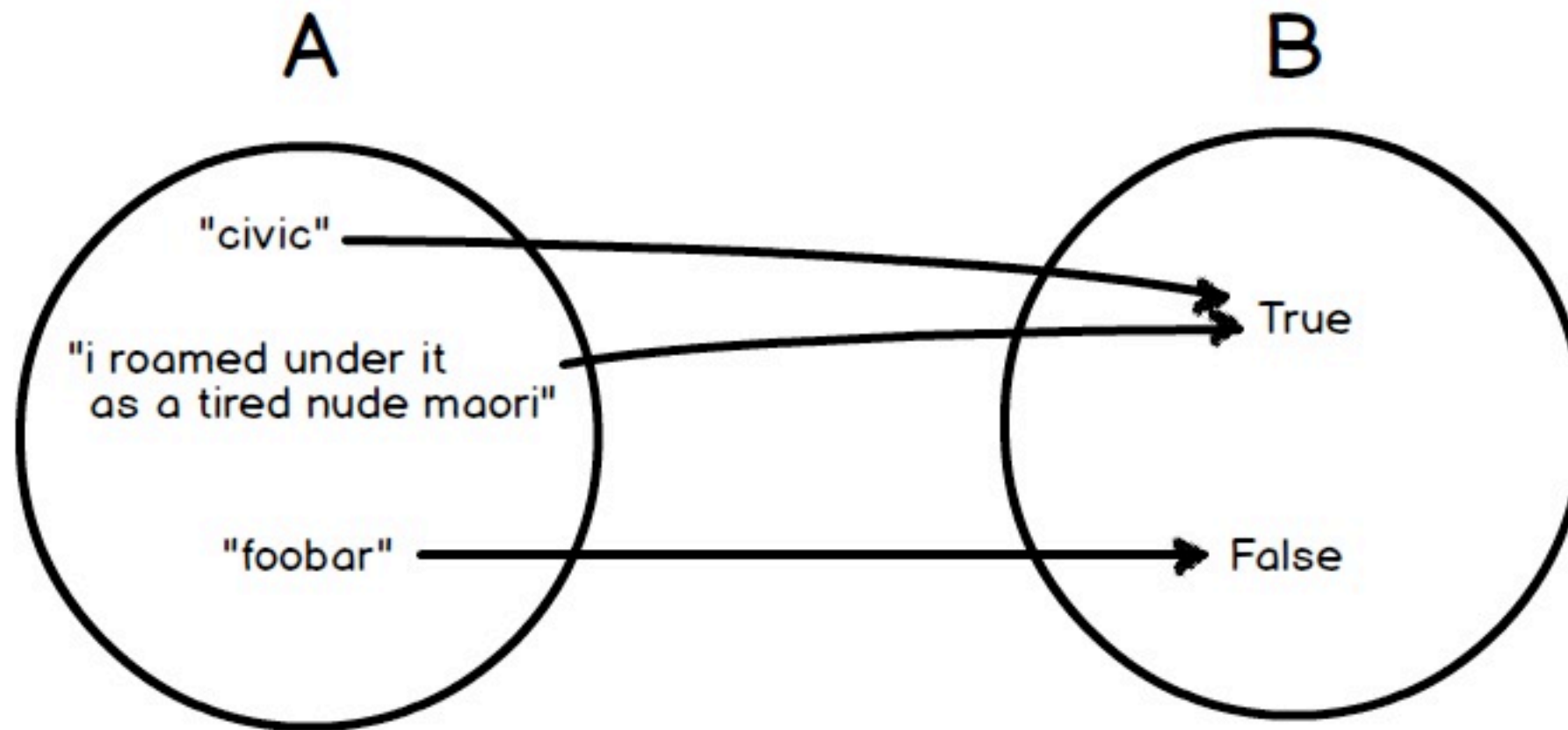
$$f(x) = 2 * x$$

$$f(17) = ???$$

$$f(17) = 34$$

# What is the relation?

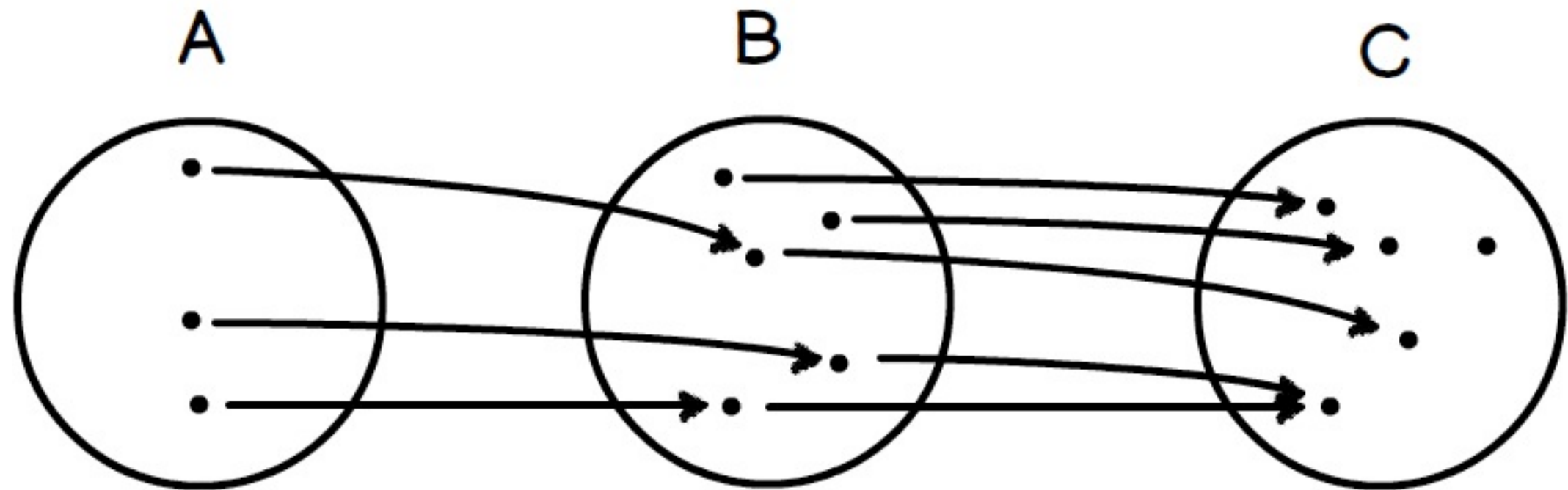
---



Is the inverse of the string the same?

# Why do we care?

---



- Does program match its specification?
- Should expect ‘mathematical’ rigor in our computations.
- Variables do not change in the middle of a computation.
- Functions do not have “secondary” input or results.

# Type inference

---

- The compiler 'figures out' the type based on the usage

```
scala> val myInt = 4
myInt: Int = 4

scala> val myDouble = 4.0
myDouble: Double = 4.0

scala> val myLong = 4L
myLong: Long = 4

scala> val myChar = '4'
myChar: Char = 4

scala> val myString = "4"
myString: String = 4
```



# Type annotations

---

- May want to explicitly declare the type.
- Type annotation **follows** variable, separated with colon.

```
scala> val myNum = 4  
myNum: Int = 4  
  
scala> val myNum: Double = 4  
myNum: Double = 4.0  
  
scala> val myNum: Long = 4  
myNum: Long = 4  
  
scala> val myNum: BigDecimal = 4  
myNum: BigDecimal = 4
```

# Function - must specify parameter types

---

```
def add(a: Int, b: Int) = a + b
```

- Functions start with 'def' keyword.
- Parameters in parentheses, separated by commas.
- Body set off with 'equals'.
- Last expression is returned; no 'return' keyword required.
- By convention, supply type of return value.
- Multi-line 'blocks' set off with curly braces.

```
def add(a: Int, b: Int): Int = {  
    a + b  
}
```

# Function application

---

- Apply function to arguments in parentheses, separated by commas.

```
scala> def add(a: Int, b: Int) = a + b  
add: (a: Int, b: Int)Int  
  
scala> add(42, 7)  
res9: Int = 49
```

# Multiple lines in REPL

---

- In REPL, multiple-line definitions are prefixed with ‘pipe’ characters.
- Can copy/paste code into REPL from editor.
- When pasting multiple definitions, use :paste in REPL.

```
scala> def add(a: Int, b: Int): Int = {  
    |   a + b  
    | }  
add: (a: Int, b: Int)Int  
scala> █
```

# Recursion

---

- Used for repeated computation.
- Functions calls 'itself' recursively toward a 'base case'.
- Based on 'inductive proof'.
- If/else is an expression and returns a value.

```
def sumSquare(a: Int, b: Int): Int = {  
  if (a > b) {  
    0  
  } else {  
    sumSquare(a+1, b) + (a * a)  
  }  
}
```

```
scala> sumSquare(1, 3)  
res28: Int = 14
```

# Recursion - step through

---

sumSquare(1, 3)

= sumSquare(1+1,3) + (1\*1)

= ((sumSquare(2+1,3) + (2\*2)) + (1\*1))

= (((sumSquare(3+1,3) + (3\*3)) + (2\*2)) + (1\*1))

= (((0 + (3\*3)) + (2\*2)) + (1\*1))

= ((0 + 9) + (2\*2)) + (1\*1)

= ((0 + 9) + 4) + (1\*1)

= ((0 + 9) + 4) + 1

= (9 + 4) + 1

```
def sumSquare(a: Int, b: Int): Int = {  
  if (a > b) {  
    0  
  } else {  
    sumSquare(a+1, b) + (a * a)  
  }  
}
```



# Recursion

---

- Sum of consecutive squares

```
def sumSquare(a: Int, b: Int): Int = {  
  if (a > b) {  
    0  
  } else {  
    sumSquare(a+1, b) + (a * a)  
  }  
}
```

- Sum of consecutive cubes

```
def sumCube(a: Int, b: Int): Int = {  
  if (a > b) {  
    0  
  } else {  
    sumCube(a+1, b) + (a * a * a)  
  }  
}
```

# Higher-order functions

---

- Functions that take other functions as arguments.
- Function type from input to output given with '=>'

```
def sumPower(fn: Int => Int, a: Int, b: Int): Int = {  
  if (a > b) {  
    0  
  } else {  
    sumPower(fn, a+1, b) + fn(a)  
  }  
}
```

# Higher-order functions

---

- Pass 'square' and 'cube' functions as arguments

```
scala> def square(n: Int): Int = n * n
square: (n: Int)Int

scala> def cube(n: Int): Int = n * n * n
cube: (n: Int)Int

scala> sumPower(square, 1, 3)
res12: Int = 14

scala> sumPower(cube, 1, 3)
res13: Int = 36
```

# Anonymous functions

---

- Don't have to assign name to functions.
- Also known as 'functional literal' or 'lambda'.
- Parameters set off from function body with '=>'

```
scala> sumPower(x => x * x, 1, 3)
res14: Int = 14

scala> sumPower(x => x * x * x, 1, 3)
res15: Int = 36
```

# Curried function

---

- Turn a function that takes two arguments into a function that takes one argument and returns a function that takes the other argument.

```
scala> def add(x: Int, y: Int) = x + y
add: (x: Int, y: Int)Int

scala> add(1, 2)
res42: Int = 3

scala> def curriedAdd(x: Int) = (y: Int) => x + y
curriedAdd: (x: Int)Int => Int

scala> curriedAdd(1)(2)
res43: Int = 3

scala> def curriedAddShort(x: Int)(y: Int) = x + y
curriedAddShort: (x: Int)(y: Int)Int

scala> curriedAddShort(1)(2)
res44: Int = 3
```

# Scala is a “pure” object-oriented language

---

- All values are objects.
- Objects are values that combine both data **and** operations on data.

# Class

---

- A “class” is a template for the creation of objects.
- The definition of a class creates a type, and at the same time, defines what it means to construct values of that type.
- To define a class, use keyword “class”.
- Create an object from a class with keyword “new”. Can assign to a variable.
- Variable’s value is of type ‘Elephant’ and is a reference to created object.

```
scala> class Elephant
defined class Elephant

scala> val elmer = new Elephant
elmer: Elephant = Elephant@1ae0f136
```

# Fields

---

- Classes can contain variables. Called “fields”.
- Access field in object using ‘dot’ and the name of the field.

```
class Sloth {  
    val numToes = 3  
}
```

```
scala> val sam = new Sloth  
sam: Sloth = Sloth@2369f54d  
  
scala> sam.numToes  
res16: Int = 3
```



# Methods

---

- Classes can contain “methods”. Methods are *practically* the same as functions.
- Access method in object with “dot” and the name of the method.

```
class Dog {  
  def speak(): String = "Woof!"  
}
```

```
scala> val fido = new Dog  
fido: Dog = Dog@2747b1c8  
  
scala> fido.speak  
res17: String = Woof!
```

# Uniform Access Principle

---

- The notation to access a feature of a class should not depend on whether it's implemented through storage (field) or computation (method).

```
class Dog {  
  def speak(): String = "Woof!"  
}
```

```
class Cat {  
  def speak = "Meow!"  
}
```

```
class Mouse {  
  val speak = "Squeak!"  
}
```

```
scala> val fido = new Dog  
fido: Dog = Dog@277663d6
```

```
scala> fido.speak  
res0: String = Woof!
```

```
scala> val kitty = new Cat  
kitty: Cat = Cat@67439515
```

```
scala> kitty.speak  
res1: String = Meow!
```

```
scala> val mickey = new Mouse  
mickey: Mouse = Mouse@3ba102ef
```

```
scala> mickey.speak  
res2: String = Squeak!
```

# Primary constructor

---

- Class definitions can have parameters.
- If prefixed with “val” or “var”, get automatic field.

```
scala> class Employee(val name: String, var salary: Int)
defined class Employee

scala> val emp1 = new Employee("Joe", 42000)
emp1: Employee = Employee@4424f1a9

scala> emp1.name
res3: String = Joe

scala> emp1.salary
res4: Int = 42000

scala> emp1.salary = 84000
emp1.salary: Int = 84000

scala> emp1.salary
res5: Int = 84000
```

# Class inheritance

---

- A class can “inherit” the fields and methods of another class using the keyword “extends”.
- Primary constructor parameters can be passed to the superclass.
- The subclass type is a “subtype” of the superclass type: Its type can be used where the type of the superclass is expected.

```
scala> class Person(val name: String)
defined class Person

scala> class Employee(name: String, var salary: Int) extends Person(name)
defined class Employee

scala> val emp1 = new Employee("Joe", 42000)
emp1: Employee = Employee@39d12be

scala> def sayName(p: Person): String = p.name
sayName: (p: Person)String

scala> sayName(emp1)
res7: String = Joe
```

# Abstract class

---

- If subclasses will have their own unique implementation of methods, make the superclass “abstract” and declare the method type signature without a body.
- Implement the methods in the subclasses.
- The abstract class cannot be instantiated into an object.

```
abstract class Animal {  
  def speak: String  
}  
  
class Dog extends Animal {  
  def speak = "Woof"  
}  
  
class Cat extends Animal {  
  def speak = "Meow"  
}
```

```
scala> val fido = new Dog  
fido: Dog = Dog@6faa3306  
  
scala> def saySomething(a: Animal): String = {  
  |   a.speak  
  | }  
saySomething: (a: Animal)String  
  
scala> saySomething(fido)  
res9: String = Woof
```



# Trait

---

- Traits are more common than abstract classes.
- Traits cannot have constructor parameters.

```
trait Animal {  
  def speak: String  
}  
  
class Dog extends Animal {  
  def speak = "Woof"  
}  
  
class Cat extends Animal {  
  def speak = "Meow"  
}
```

```
scala> val kitty = new Cat  
kitty: Cat = Cat@66be9fc1  
  
scala> def saySomething(a: Animal): String = {  
    |     a.speak  
    | }  
saySomething: (a: Animal)String  
  
scala> saySomething(kitty)  
res2: String = Meow
```

# Mixin

---

- Inherit from traits using “with”.

```
trait Vehicle {  
  def maxSpeed: String  
}
```

```
trait Motorcycle extends Vehicle {  
  val wheels = 2  
}
```

```
trait GasVehicle {  
  def noise() = "Vrooom"  
}
```

```
scala> class SportsBike extends Motorcycle with GasVehicle {  
      |   def maxSpeed = "150mph"  
      | }  
defined class SportsBike  
  
scala> val ninja = new SportsBike  
ninja: SportsBike = SportsBike@6d8f5c10  
  
scala> ninja.noise  
res2: String = Vrooom  
  
scala> ninja.wheels  
res3: Int = 2  
  
scala> ninja.maxSpeed  
res4: String = 150mph
```

# Mixin during instantiation

---

- Can also mixin while creating an object.

```
trait Vehicle {  
  def maxSpeed: String  
}
```

```
trait Motorcycle extends Vehicle {  
  val wheels = 2  
}
```

```
trait GasVehicle {  
  def noise() = "Vrooom"  
}
```

```
trait ElectricVehicle {  
  def noise() = "Whirrrr"  
}
```

```
scala> class Scooter extends Motorcycle {  
      |   def maxSpeed = "65mph"  
      | }  
defined class Scooter  
  
scala> val vespa = new Scooter with ElectricVehicle  
vespa: Scooter with ElectricVehicle = $anon$1@6fb05c54  
  
scala> vespa.maxSpeed  
res6: String = 65mph  
  
scala> vespa.noise  
res7: String = Whirrrr
```



# Singleton object

---

- Declares a class and its only instance. Use keyword “object” instead of “class”.
- Often used as placeholder for functionality (including entry point of programs)

```
object Demo {
```

```
  def main(args: Array[String]) = {  
    println(sayGreeting)  
  }
```

```
  def sayGreeting = "Guten Tag"
```

```
}
```

```
scala> Demo.sayGreeting  
res0: String = Guten Tag
```

```
> ~run  
[info] Running Demo  
Guten Tag  
[success] Total time: 0 s, completed Sep 19, 2013 9:38:16 AM  
1. Waiting for source changes... (press enter to interrupt)  
█
```

# Case class

---

- Prefix class (or object) definition with “case”.
- Constructor parameter automatically becomes a “val” (i.e, you get an immutable field).
- Don’t need “new” to create objects.
- Used in pattern matching.

```
scala> case class Person(fname: String, lname: String)
defined class Person

scala> val joe = Person("Joe", "Smith")
joe: Person = Person(Joe,Smith)
```

# Pattern matching

---

- A pattern is a syntactic expression that defines the structural properties of a value.
- Patterns can be compared against values.
- Patterns can be used to decompose complex values and bind the component values to variables.

# Types of patterns

---

- Literal pattern-- 0, "hello", Nil, True
- Variable pattern -- x, \_
- Type pattern -- x: Int (include a variable)
- Tuple pattern -- (42, x), (42, \_)
- Other extractor patterns -- Array(x, y), Array(42, \_\*), head :: tail

```
scala> val (x,y) = (1, 2)
x: Int = 1
y: Int = 2

scala> val Array(first, second, _) = Array(1,2,3,4,5)
first: Int = 1
second: Int = 2
```

# Pattern matching - *match* expression

---

- Used to select a branch of code based on pattern match.
- Keyword 'match' for pattern matching expression.
- A sequence of alternative patterns to compare against a value. Each begins with 'case'. Followed by '=>' and the block of code to execute.

```
scala> val foo = (42, 0)
foo: (Int, Int) = (42,0)

scala> foo match {
  |   case (0, _) => "Zero first"
  |   case (y, 0) => y + " then zero"
  |   case _    => "No zero"
  | }
res29: String = 42 then zero
```

# Pattern matching - function dispatch

---

- Patterns are compared against function arguments to decide among possible results of the same type.

```
def take(m: Int, ys: List[Any]): List[Any] = {  
  (m, ys) match {  
    case (0, _) => Nil  
    case (_, Nil) => Nil  
    case (n, head :: tail) => n :: take(n-1, tail)  
  }  
}
```

```
scala> take(2, List(1,2,3,4))  
res20: List[Any] = List(1, 2)  
  
scala> take(3, List('a','b','c'))  
res21: List[Any] = List(a, b, c)
```

# Pattern matching - case classes

---

- Case classes automatically get a hidden method (called “unapply”) that extracts the values from an object that were used to construct it.

```
scala> case class Droid(name: String, model: String)
defined class Droid

scala> val r2d2 = Droid("R2-D2", "Astromech")
r2d2: Droid = Droid(R2-D2,Astromech)

scala> val c3p0 = Droid("C-3P0", "Protocol")
c3p0: Droid = Droid(C-3P0,Protocol)

scala> r2d2 match {
  |   case Droid("C-3P0", m) => "Gotcha, you " + m + "droid"
  |   case Droid("R2-D2", m) => "Come with us " + m + " droid"
  |   case _ => "Not the droids we are looking for"
  | }
res23: String = Come with us Astromech droid
```



# Parametric polymorphism (generics)

---

- Classes, traits, methods, functions can have type parameters.
- Type parameters go after the name, enclosed in square brackets.
- Type parameters are used in definitions to specify the types of parameters and return values.

```
scala> case class Pair[T, S] (fst: T, snd: S)
defined class Pair

scala> val p = Pair('b', 42)
p: Pair[Char,Int] = Pair(b,42)

scala> p.fst
res26: Char = b

scala> p.snd
res27: Int = 42
```



# Covariance annotation

---

- Used to make subtype relationship explicit in parametric polymorphism.
- Can `Pair[Student]` be used where `Pair[Person]` is expected when `Student` is a subclass of `Person`?
- Annotate type parameters with “+” to mean “covariant in T”.

```
scala> class Person(val name: String)
defined class Person

scala> case class Student(fullName: String) extends Person(fullName)
defined class Student

scala> case class Pair[+T](fst: T, snd: T)
defined class Pair

scala> def makeFriends(p: Pair[Person]) = p.fst.name + " likes " + p.snd.name
makeFriends: (p: Pair[Person])String

scala> val bff = Pair[Student](Student("Kathy"), Student("Jane"))
bff: Pair[Student] = Pair(Student(Kathy),Student(Jane))

scala> makeFriends(bff)
res0: String = Kathy likes Jane
```

# Collections (immutable) hierarchy

---



# Combinators - examples

---

- *Combinator* is an informal term referring to (often higher-order) functions that create, manipulate, and combine values, and themselves are combined to produce more complex behavior.

```
scala> List(1,2,3).map(x => x * x).sum
res33: Int = 14

scala> List(1,2,3).map(x => x * x).reduce((x,y) => x + y)
res34: Int = 14

scala> List(1,2,3).map(x => x * x).reduce(_ + _)
res35: Int = 14

scala> List(1,2,3).map(x => x * x).foldLeft(0)(_ + _)
res36: Int = 14

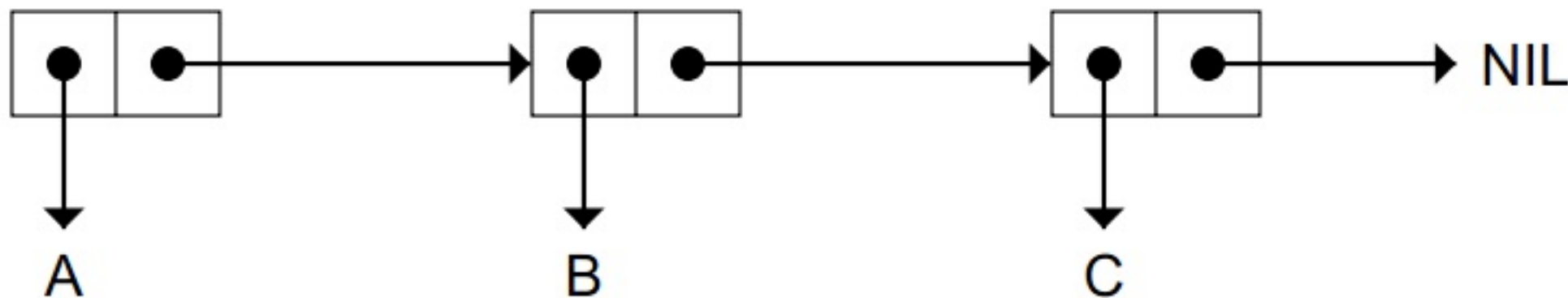
scala> List(1,2,3).map(x => x * x).foldRight(0)(_ + _)
res37: Int = 14

scala> (1 to 3).map(x => x * x).sum
res38: Int = 14
```

# Cons - making a singly linked list, functionally

---

- *Cons* is a two-argument function that constructs a list. The second argument is a list or Nil (ie, the empty list).
- The values in a list must be of the same type.



`Cons(A, Cons(B, Cons(C, Nil)))`

# Combinator library example

---

```
trait ZList[+T]
case object Empty extends ZList[Nothing]
case class Cons[+T](first: T, rest: ZList[T]) extends ZList[T]

object ZList {

  def zmap[T,S](fn: T => S, lst: ZList[T]): ZList[S] = {
    (fn, lst) match {
      case (fn, Empty) => Empty
      case (fn, Cons(x,xs)) => Cons(fn(x), zmap(fn, xs))
    }
  }

  def zfoldRight[T,S](lst: ZList[T], z: S)(fn: (T, S) => S): S = {
    lst match {
      case Empty => z
      case Cons(x, xs) => fn(x, zfoldRight(xs, z)(fn))
    }
  }

  def apply[T](xs: T*): ZList[T] = {
    if (xs.isEmpty)
      Empty
    else
      Cons(xs.head, apply(xs.tail: _*))
  }
}
```

# Combinator library - usage

---

```
scala> import ZList._  
import ZList._  
  
scala> val foo = ZList(1,2,3)  
foo: ZList[Int] = Cons(1,Cons(2,Cons(3,Empty)))  
  
scala> val bar = zmap((x: Int) => x * x, foo)  
bar: ZList[Int] = Cons(1,Cons(4,Cons(9,Empty)))  
  
scala> zfoldRight(bar, 0)((x,y) => x+y)  
res1: Int = 14
```



# “Transition” books

---

- Most accessible when coming from an ‘old-school’ (i.e., imperative) language.



<http://typesafe.com/resources/free-books>

---

Thank you!