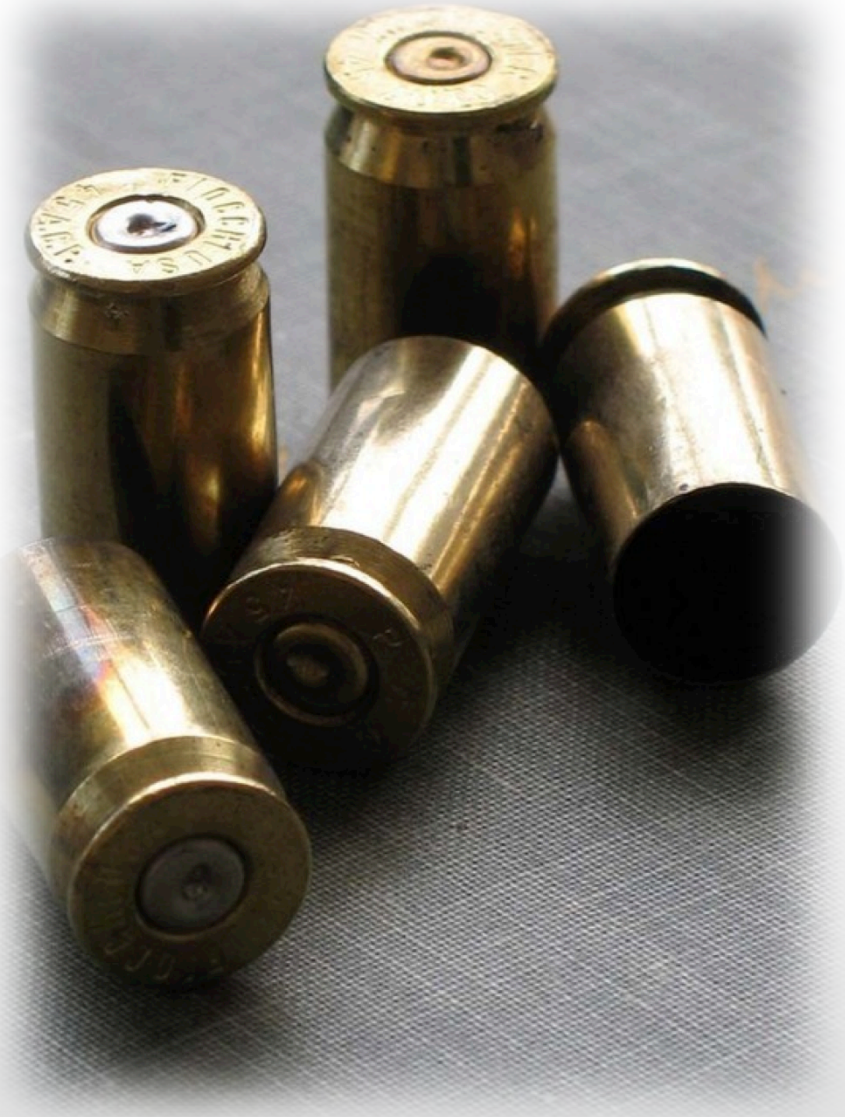
5 Bullets to Scala Adoption



Tomer Gabel, Wix

January 2015

WARMUP: THE BASICS

This is Spar– I mean, Java

```java
public class Person {
    private String name;
    private String surname;
    private int age;

    public Person( String name, String surname, int age ) {
        this.name = name;
        this.surname = surname;
        this.age = age;
    }

    public String getName() { return name; }
    public String getSurname() { return surname; }
    public int getAge() { return age; }

    @Override
    public String toString() { /* … */ }
}
```

Let’s make some people!

```
public class SimplePersonGenerator {

    // Prepare a bunch of useful constants…

    private static List<String> names;
    private static List<String> surnames;

    static {
        names = new ArrayList<>();
        names.add( "Jeffrey" );
        names.add( "Walter" );
        names.add( "Donald" );

        surnames = new ArrayList<>();
        surnames.add( "Lebowsky" );
        surnames.add( "Sobchak" );
        surnames.add( "Kerabatsos" );
    }
```

IT'S MADE OF PEOPLE!

```
// Continued from previous slide

private static Random random = new Random();

public static Person generatePerson( int age ) {
    String name = names.get( random.nextInt( names.size() ) );
    String surname = surnames.get( random.nextInt( surnames.size() ) );
    return new Person( name, surname, age );
}

public List<Person> generatePeople( int count, int age ) {
    List<Person> people = new ArrayList<>( count );
    for ( int i = 0; i < count; i++ )
        people.add( generatePerson( age ) );
    return Collections.unmodifiableList( people );
}
```

Scala syntax 101

Java

```
public class Person {
    private String name;
    private String surname;
    private int age;

    public Person( String name, String surname,
                   int age ) {
        this.name = name;
        this.surname = surname;
        this.age = age;
    }

    public String getName() { return name; }
    public String getSurname() { return surname; }
    public int getAge() { return age; }

    @Override
    public String toString() { /* … */ }

}
```

Scala syntax 101

Java

```java
public class Person {
    private String name;
    private String surname;
    private int age;

    public Person( String name, String surname,
                   int age ) {
        this.name = name;
        this.surname = surname;
        this.age = age;
    }

    public String getName() { return name; }
    public String getSurname() { return surname; }
    public int getAge() { return age; }

    @Override
    public String toString() { /* … */ }

}
```

Scala

```scala
case class Person(
  name: String,
  surname: String,
  age: Int )
```

Scala syntax 101

Java

```java
public class Person {
    private String name;
    private String surname;
    private int age;

    public Person( String name, String surname,
                   int age ) {
        this.name = name;
        this.surname = surname;
        this.age = age;
    }

    public String getName() { return name; }
    public String getSurname() { return surname; }
    public int getAge() { return age; }

    @Override
    public String toString() { /* … */ }

}
```

Scala

```scala
case class Person(
  name: String,
  surname: String,
  age: Int )
```

Provides:

- Constructor
- Property getters
- hashCode/equals
- toString
- … and more

Onwards…

Java

```
public class SimplePersonGenerator {

    // Prepare a bunch of useful constants…

    private static List<String> names;
    private static List<String> surnames;

    static {
        names = new ArrayList<>();
        names.add( "Jeffrey" );
        names.add( "Walter" );
        names.add( "Donald" );

        surnames = new ArrayList<>();
        surnames.add( "Lebowsky" );
        surnames.add( "Sobchak" );
        surnames.add( "Kerabatsos" );
    }
```

Onwards…

Java

```java
public class SimplePersonGenerator {

    // Prepare a bunch of useful constants…

    private static List<String> names;
    private static List<String> surnames;

    static {
        names = new ArrayList<>();
        names.add( "Jeffrey" );
        names.add( "Walter" );
        names.add( "Donald" );

        surnames = new ArrayList<>();
        surnames.add( "Lebowsky" );
        surnames.add( "Sobchak" );
        surnames.add( "Kerabatsos" );
    }
```

Scala

```scala
object NaiveScalaPersonGenerator {

  private val names: List[ String ] =
    List(
      "Jeffrey",
      "Walter",
      "Donald"
    )

  private val surnames: List[ String ] =
    List(
      "Lebowsky",
      "Sobchak",
      "Kerabatsos"
    )
```

... and finally:

Java

```
private static Random random = new Random();

public static Person generatePerson( int age ) {
  String name = names.get(
    random.nextInt( names.size() ) );
  String surname = surnames.get(
    random.nextInt( surnames.size() ) );
  return new Person( name, surname, age );
}

public List<Person> generatePeople(
    int count, int age ) {
  List<Person> people = new ArrayList<>( count );
  for ( int i = 0; i < count; i++ )
    people.add( generatePerson( age ) );
  return Collections.unmodifiableList( people );
}
```

... and finally:

Java

```java
private static Random random = new Random();

public static Person generatePerson( int age ) {
  String name = names.get(
    random.nextInt( names.size() ) );
  String surname = surnames.get(
    random.nextInt( surnames.size() ) );
  return new Person( name, surname, age );
}

public List<Person> generatePeople(
    int count, int age ) {
  List<Person> people = new ArrayList<>( count );
  for ( int i = 0; i < count; i++ )
    people.add( generatePerson( age ) );
  return Collections.unmodifiableList( people );
}
```

Scala

```scala
private val random: Random = new Random

def generatePerson( age: Int ): Person = {
  val name: String =
    names( random.nextInt( names.size ) )
  val surname: String =
    surnames( random.nextInt( surnames.size ) )
  new Person( name, surname, age )
}
```

... and finally:

Java

```java
private static Random random = new Random();

public static Person generatePerson( int age ) {
  String name = names.get(
    random.nextInt( names.size() ) );
  String surname = surnames.get(
    random.nextInt( surnames.size() ) );
  return new Person( name, surname, age );
}

public List<Person> generatePeople(
    int count, int age ) {
  List<Person> people = new ArrayList<>( count );
  for ( int i = 0; i < count; i++ )
    people.add( generatePerson( age ) );
  return Collections.unmodifiableList( people );
}
```

Scala

```scala
private val random: Random = new Random

def generatePerson( age: Int ): Person = {
  val name: String =
    names( random.nextInt( names.size ) )
  val surname: String =
    surnames( random.nextInt( surnames.size ) )
  new Person( name, surname, age )
}

def generatePeople( count: Int, age: Int ):
    List[ Person ] = {
  val people: mutable.ListBuffer[ Person ] =
    mutable.ListBuffer.empty
  for ( i <- 0 until count ) {
    people.append( generatePerson( age ) )
  }
  people.result()
}
```

1. TYPE INFERENCE

We started off with:

Java

```java
private static Random random = new Random();

public static Person generatePerson( int age ) {
  String name = names.get(
    random.nextInt( names.size() ) );
  String surname = surnames.get(
    random.nextInt( surnames.size() ) );
  return new Person( name, surname, age );
}

public static List<Person> generatePeople(
    int count, int age ) {
  List<Person> people = new ArrayList<>( count );
  for ( int i = 0; i < count; i++ )
    people.add( generatePerson( age ) );
  return Collections.unmodifiableList( people );
}
```

Scala

```scala
private val random: Random = new Random

def generatePerson( age: Int ): Person = {
  val name: String =
    names( random.nextInt( names.size ) )
  val surname: String =
    surnames( random.nextInt( surnames.size ) )
  new Person( name, surname, age )
}

def generatePeople( count: Int, age: Int ):
    List[ Person ] = {
  val people: mutable.ListBuffer[ Person ] =
    mutable.ListBuffer.empty
  for ( i <- 0 until count ) {
    people.append( generatePerson( age ) )
  }
  people.result()
}
```

Let's simplify!

Java

```java
private static Random random = new Random();

public static Person generatePerson( int age ) {
  String name = names.get(
    random.nextInt( names.size() ) );
  String surname = surnames.get(
    random.nextInt( surnames.size() ) );
  return new Person( name, surname, age );
}

public static List<Person> generatePeople(
    int count, int age ) {
  List<Person> people = new ArrayList<>( count );
  for ( int i = 0; i < count; i++ )
    people.add( generatePerson( age ) );
  return Collections.unmodifiableList( people );
}
```

Scala

```scala
private val random = new Random

def generatePerson( age: Int ): Person = {
  val name =
    names( random.nextInt( names.size ) )
  val surname =
    surnames( random.nextInt( surnames.size ) )
  new Person( name, surname, age )
}

def generatePeople( count: Int, age: Int ):
    List[ Person ] = {
  val people =
    mutable.ListBuffer.empty[ Person ]
  for ( i <- 0 until count ) {
    people.append( generatePerson( age ) )
  }
  people.result()
}
```

Also works for result types:

Java

```java
private static Random random = new Random();

public static Person generatePerson( int age ) {
  String name = names.get(
    random.nextInt( names.size() ) );
  String surname = surnames.get(
    random.nextInt( surnames.size() ) );
  return new Person( name, surname, age );
}

public static List<Person> generatePeople(
    int count, int age ) {
  List<Person> people = new ArrayList<>( count );
  for ( int i = 0; i < count; i++ )
    people.add( generatePerson( age ) );
  return Collections.unmodifiableList( people );
}
```

Scala

```scala
private val random = new Random

def generatePerson( age: Int ) = {
  val name =
    names( random.nextInt( names.size ) )
  val surname =
    surnames( random.nextInt( surnames.size ) )
  new Person( name, surname, age )
}

def generatePeople( count: Int, age: Int ) = {
  val people =
    mutable.ListBuffer.empty[ Person ]
  for ( i <- 0 until count ) {
    people.append( generatePerson( age ) )
  }
  people.result()
}
```

Also works for result types:

Java

```
private static Random random = new Random();

public static Person generatePerson( int age ) {
  String name = names.get(
    random.nextInt( names.size() ) );
  String surname = surnames.get(
    random.nextInt( surnames.size() ) );
  return new Person( name, surname, age );
}

public List<Person> generatePeople(
    int count, int age ) {
  List<Person> people = new ArrayList<>( count );
  for ( int i = 0; i < count; i++ )
    people.add( generatePerson( age ) );
  return Collections.unmodifiableList( people );
}
```

Scala

```
def generatePerson( age: Int ) = {
  val name =
    names( random.nextInt( names.size ) )
  val surname =
    surnames( random.nextInt( surnames.size ) )
  Person( name, surname, age )
}

def generatePeople( count: Int, age: Int ) = {
  val people =
    ArrayBuffer.empty[ Person ]
  for ( i <- 0 until count ) {
    people.append( generatePerson( age ) )
  }
  people.result()
}
```

Not necessarily a good idea.

- Code readability
- Public APIs
- Compilation times

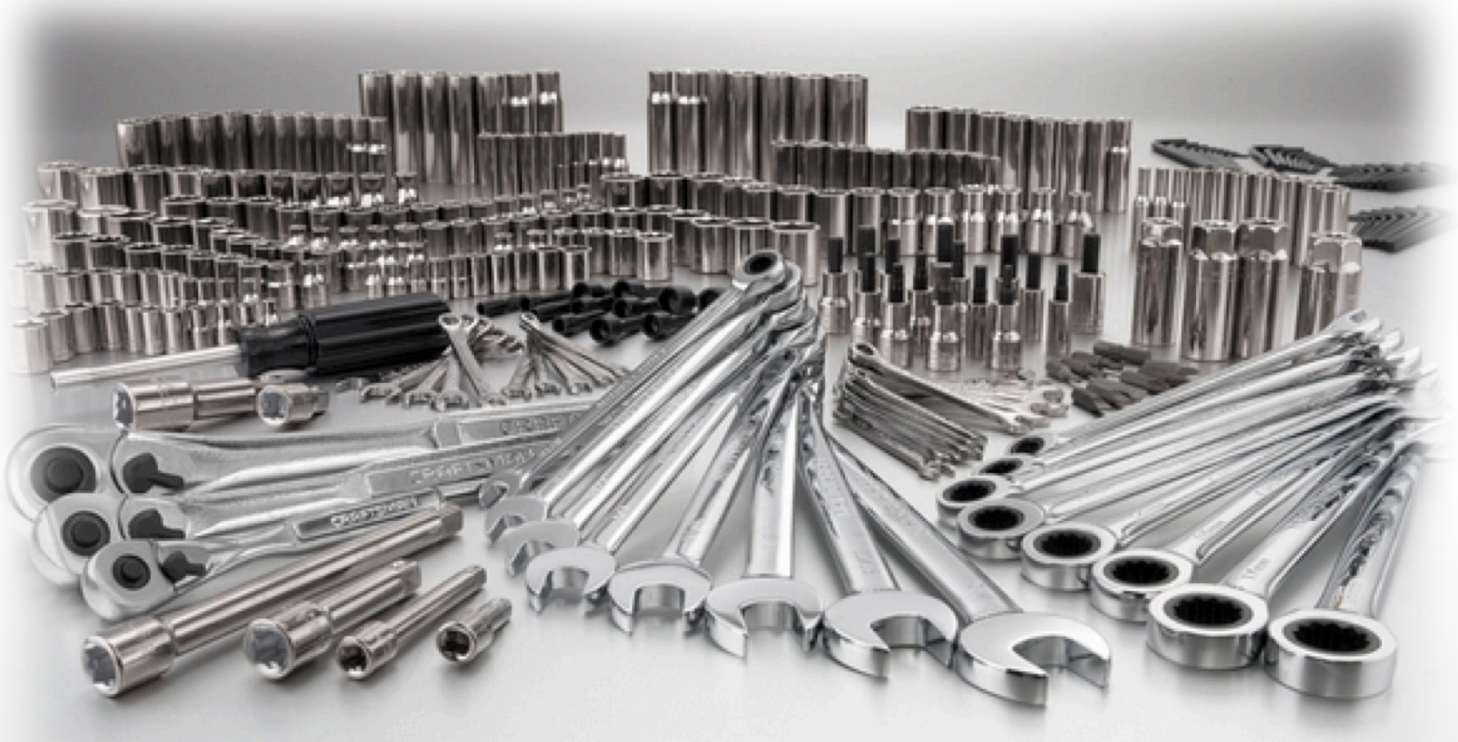
Bonus points: Named arguments

Java

```java
private static Random random = new Random();

public static Person generatePerson( int age ) {
  String name = names.get(
    random.nextInt( names.size() ) );
  String surname = surnames.get(
    random.nextInt( surnames.size() ) );
  return new Person( name, surname, age );
}

public static List<Person> generatePeople(
    int count, int age ) {
  List<Person> people = new ArrayList<>( count );
  for ( int i = 0; i < count; i++ )
    people.add( generatePerson( age ) );
  return Collections.unmodifiableList( people );
}
```

Scala

```scala
private val random = new Random

def generatePerson( age: Int ) = Person(
  name = names( random.nextInt( names.size ) )
  surname =
    surnames( random.nextInt( surnames.size ) )
  age = age
)

def generatePeople( count: Int, age: Int ) = {
  val people =
    mutable.ListBuffer.empty[ Person ]
  for ( i <- 0 until count ) {
    people.append( generatePerson( age ) )
  }
  people.result()
}
```



2. THE COLLECTION FRAMEWORK

Quick example

Java

```java
private static Random random = new Random();

public static Person generatePerson( int age ) {
  String name = names.get(
    random.nextInt( names.size() ) );
  String surname = surnames.get(
    random.nextInt( surnames.size() ) );
  return new Person( name, surname, age );
}

public static List<Person> generatePeople(
    int count, int age ) {
  List<Person> people = new ArrayList<>( count );
  for ( int i = 0; i < count; i++ )
    people.add( generatePerson( age ) );
  return Collections.unmodifiableList( people );
}
```

Scala

```scala
private val random = new Random

def generatePerson( age: Int ) = Person(
  name = names( random.nextInt( names.size ) )
  surname =
    surnames( random.nextInt( surnames.size ) )
  age = age
)

def generatePeople( count: Int, age: Int ) = {
  val people =
    mutable.ListBuffer.empty[ Person ]
  for ( i <- 0 until count ) {
    people.append( generatePerson( age ) )
  }
  people.result()
}
```

Quick example

Java

```java
private static Random random = new Random();

public static Person generatePerson( int age ) {
  String name = names.get(
    random.nextInt( names.size() ) );
  String surname = surnames.get(
    random.nextInt( surnames.size() ) );
  return new Person( name, surname, age );
}

public static List<Person> generatePeople(
    int count, int age ) {
  List<Person> people = new ArrayList<>( count );
  for ( int i = 0; i < count; i++ )
    people.add( generatePerson( age ) );
  return Collections.unmodifiableList( people );
}
```

Scala

```scala
private val random = new Random

def generatePerson( age: Int ) = Person(
  name = names( random.nextInt( names.size ) )
  surname =
    surnames( random.nextInt( surnames.size ) )
  age = age
)

def generatePeople( count: Int, age: Int ) =
 List.fill( count ) { generatePerson( age ) }
```

Scala collections primer

```
// Some data to start with…
val people = generatePeople( 5 )

val names = people.map( p => p.name )

val adults = people.filter( p => p.age >= 18 )

val averageAge =
  people.map( p => p.age ).sum / people.size

// Functions are 1st class citizens
def isAdult( p: Person ) = p.age >= 18
val adults2 = people.filter( isAdult )
assert( adults2 == adults )
```

Maps, too

```
val trivial: Map[ String, String ] =
  Map( "name"  -> "Jeffrey Lebowsky",
       "alias" -> "The Dude" )

val directory: Map[ Char, List[ Person ] ] =
  people
    .groupBy( p => p.surname.head.toUpper )
    .withDefaultValue( List.empty )

val beginningWithK: List[ Person ] =
  directory( 'K' )

val countByFirstLetter: Map[ Char, Int ] =
  directory.mapValues( list => list.size )
```

And much, much, much more

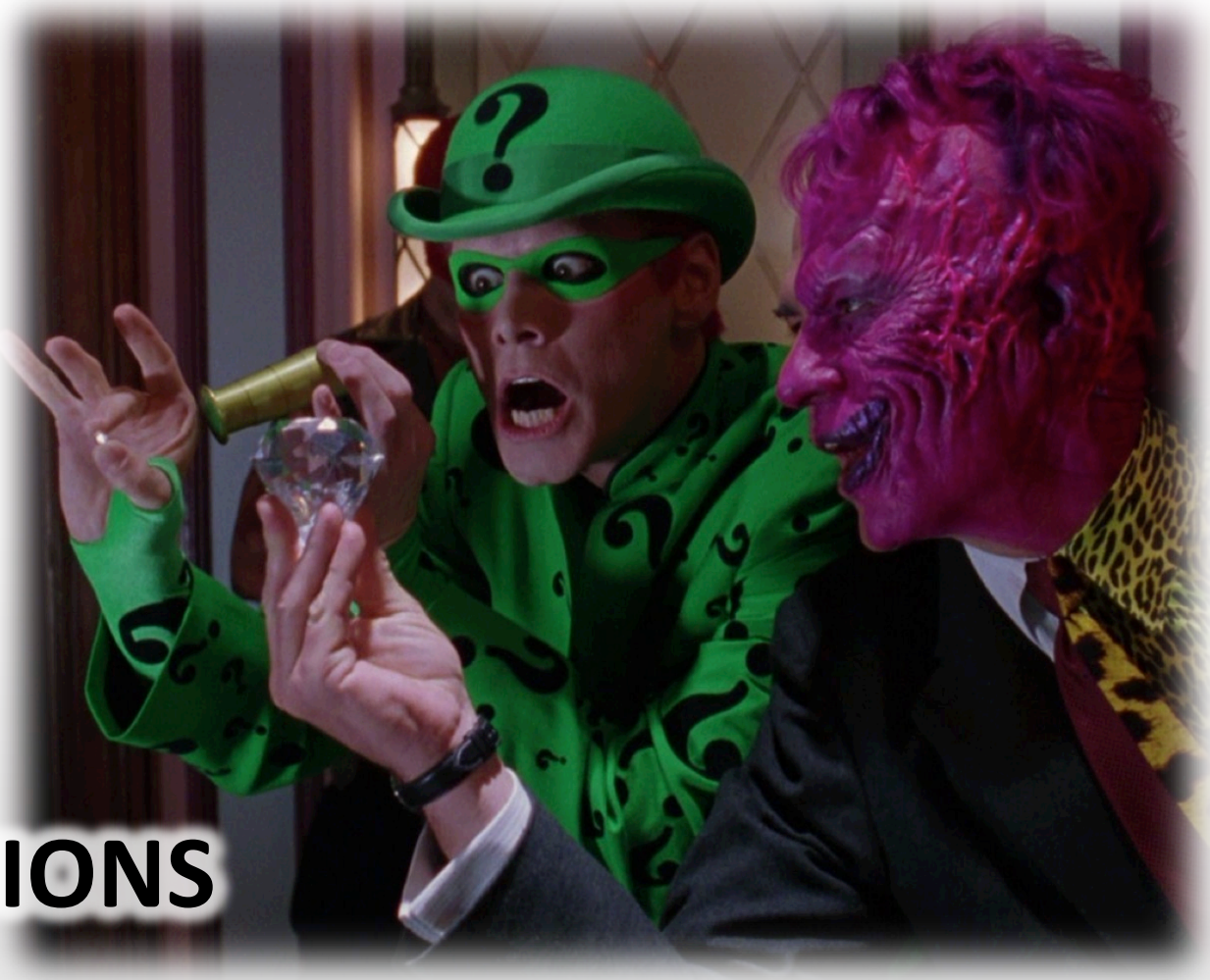
```
val ( adults, minors ) = people.partition( p => p.age >= 18 )

val randomPairs = Random.shuffle( people ).grouped( 2 )

val youngest = people.minBy( p => p.age )

val oldest = people.maxBy( p => p.age )

val hasSeniorCitizens = people.exists( p => p.age >= 65 )
```

Caveat emptor

- Scala is flexible
  - You can do the same thing in multiple ways:

```
people.foreach( person => println( person ) )     // "Normal"
people.foreach { person => println( person ) }    // Block-style
people.foreach { println(_) }                     // Placeholder syntax
people.foreach( println )                         // With function
people foreach println                            // Infix notation
```

- Stay sane. Stay safe.
  - Pick a style and stick with it



3. OPTIONS

Back in Java-Land

```
public class Person {
    private String name;
    private String surname;
    private int age;

    public Person( String name, String surname, int age ) {
        this.name = name;
        this.surname = surname;
        this.age = age;
    }

    public String getName() { return name; }
    public String getSurname() { return surname; }
    public int getAge() { return age; }

    @Override
    public String toString() { /* … */ }
}
```

We want to make these optional.

Typical solution in Java

```
public class PartiallyKnownPerson {
    private String name;
    private String surname;
    private Integer age;

    public PartiallyKnownPerson( String name, String surname, Integer age ) {
        this.name = name;
        this.surname = surname;
        this.age = age;
    }

    public String getName() { return name; }
    public String getSurname() { return surname; }
    public Integer getAge() { return age; }

    @Override
    public String toString() { /* … */ }
}
```

This sucks

```
public class PartiallyKnownPerson {
    private String name;
    private String surname;
    private Integer age;

    public PartiallyKnownPerson( String name, String surname, Integer age ) {
        this.name = name;
        this.surname = surname;
        this.age = age;
    }

    public String getName() { return name; }
    public String getSurname() { return surname; }
    public Integer getAge() { return age; }

    @Override
    public String toString() { /* … */ }
}
```

Implicit; does not document intent.

Returned values *always* need to be checked.

Falling back to JavaDocs

```
/**
 * Creates a new person.
 * @param name The person's name (or {@literal null} if unknown).
 * @param surname The person's surname (or {@literal null} if unknown).
 * @param age The person's age (or {@literal null} if unknown).
 */
public PartiallyKnownPerson( String name, String surname, Integer age ) {
  this.name = name;
  this.surname = surname;
  this.age = age;
}
```

Falling back to JavaDocs

```
/**
 * Creates a new person.
 * @param name The person's name (or {@literal null} if unknown).
 * @param surname The person's surname (or {@literal null} if unknown).
 * @param age The person's age (or {@literal null} if unknown).
 */
public PartiallyKnownPerson( String name, String surname, Integer age ) {
  this.name = name;
  this.surname = surname;
  this.age = age;
}
```

This still sucks.

Falling back to JavaDocs

```
/**
 * Creates a new person.
 * @param name The person's name (or {@literal null} if unknown).
 * @param surname The person's surname (or {@literal null} if unknown).
 * @param age The person's age (or {@literal null} if unknown).
 */
public PartialRecord( String name, String surname, Integer age ) {
  this.name = name;
  this.surname = surname;
  this.age = age;
}
```

This still sucks.

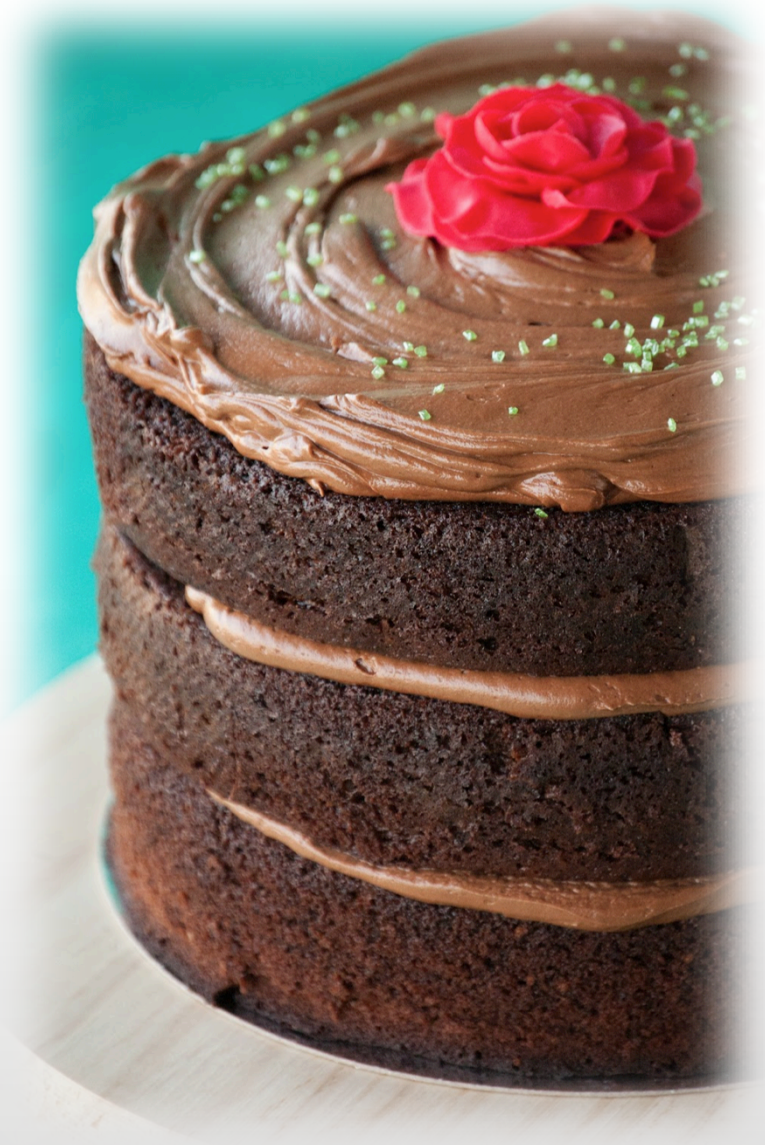
Because:

```
boolean isAdult( Person p ) {
  return p.getAge() >= 18;    // I love the smell of NPEs in the morning
}
```

Optional arguments, the Scala way

```
case class PartiallyKnownPerson( name:    Option[ String ] = None,
                                 surname: Option[ String ] = None,
                                 age:     Option[ Int ]    = None ) {

  def isAdult = age.map( _ > 18 )
}

val person = PartiallyKnownPerson( name = Some( "Brandt" ) )

val explicitAccess = person.age.get   // <-- NoSuchElementException thrown here

val withDefault = person.name.getOrElse( "Anonymous" )
println( "Hello " + withDefault + "!" )

// Options are also zero- or single-element collections!
def greet( whom: Iterable[ String ] ) =
  whom.foreach { name => println( "hello" + name ) }
greet( person.name )
```

4. TRAITS



Back in Java land…

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class ClassWithLogs {
    private static Logger log = LoggerFactory.getLogger( ClassWithLogs.class );

    public String getNormalizedName( Person person ) {
        log.info( "getNormalizedName called" );
        log.debug( "Normalizing " + person.toString() );
        String normalizedName = person.getName().toUpperCase().trim();
        log.debug( "Normalized name is: " + normalizedName );
        return normalizedName;
    }
}
```

… boilerplate is king

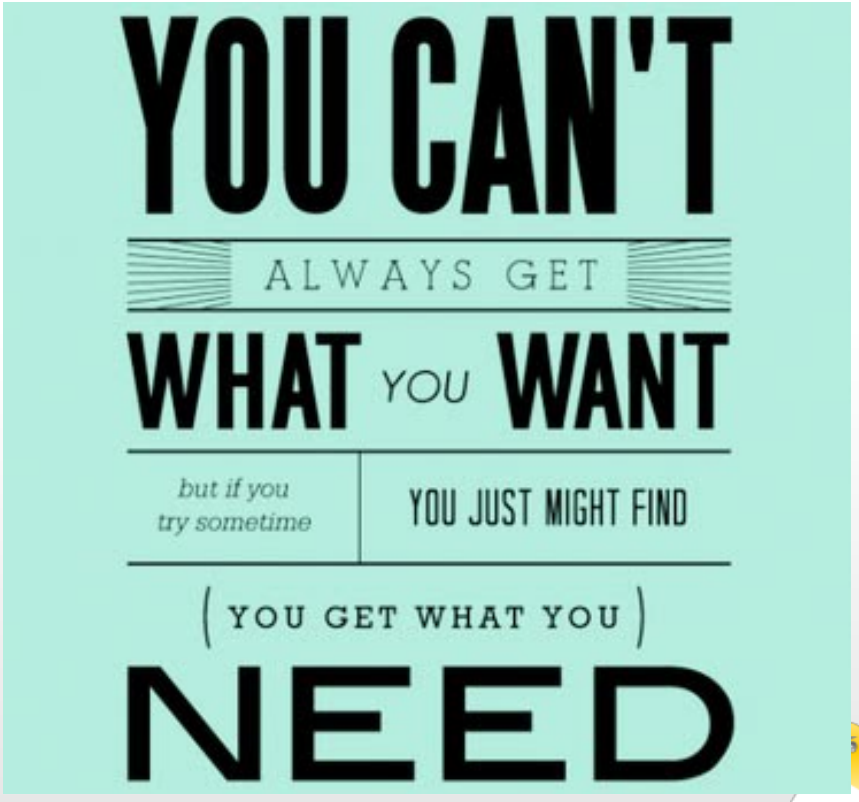
```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class ClassWithLogs {
    private static Logger log = LoggerFactory.getLogger( ClassWithLogs.class );

    public String getNormalizedName( Person person ) {
        log.info( "getNormalizedName called" );
        log.debug( "Normalizing " + person.toString() );
        String normalizedName = person.getName().toUpperCase().trim();
        log.debug( "Normalized name is: " + normalizedName );
        return normalizedName;
    }
}
```

x1000 classes…

Eager evaluation

What can we do about it?

- A solution should:
  - Require minimal boilerplate
  - Cause few or no side-effects
  - Be composable
  - Perform well
- An abstract base class is useless
  - Only one base class allowed
  - No way to do lazy evaluation



Enter: Scala traits

```
trait Logging {
  private val log = LoggerFactory.getLogger( this.getClass )

  protected def logInfo( message: => String ) =
    if ( log.isInfoEnabled ) log.info ( message )

  protected def logDebug( message: => String ) =
    if ( log.isDebugEnabled ) log.debug( message )

  protected def logWarn( message: => String ) =
    if ( log.isWarnEnabled ) log.warn ( message )

  protected def logError( message: => String ) =
    if ( log.isErrorEnabled ) log.error( message )
}
```

Boilerplate is king

Java

```java
public class ClassWithLogs {
  private static Logger log =
    LoggerFactory.getLogger(
      ClassWithLogs.class );

  public String getNormalizedName(
      Person person ) {
    log.info( "getNormalizedName called" );
    log.debug( "Normalizing " +
      person.toString() );
    String normalizedName =
      person.getName().toUpperCase().trim();
    log.debug( "Normalized name is: " +
      normalizedName );
    return normalizedName;
  }
}
```

Scala

```scala
class ClassWithLogs extends Logging {

  def getNormalizedName( person: Person ) = {
    logInfo( "getNormalizedName called" )
    logDebug( "Normalizing " +
      person.toString )
    val normalizedName =
      person.name.toUpperCase.trim
    logDebug( "Normalized name is: " +
      normalizedName )
    normalizedName
  }
}
```

Not convinced?

- Consider:

```
trait Iterator[ T ] {
   def hasNext: Boolean
   def next: T                          // Or throw NoSuchElementException
 }

 trait Iterable[ T ] {
   def getIterator: Iterator[ T ]
   def first: T                         // Or throw NoSuchElementException
   def firstOption: Option[ T ]
   def last: T                          // Or throw NoSuchElementException
   def lastOption: Option[ T ]
   def size: Int
 }
```

- What if you need to implement ArrayList, HashSet, LinkedList…?

A saner option

```
trait BaseIterable[ T ] extends Iterable[ T ] {

  def firstOption = {
    val it = getIterator
    if ( it.hasNext ) Some( it.next ) else None
  }

  def first = firstOption.getOrElse( throw new NoSuchElementException )

  def size = {
    val it = getIterator
    var count = 0
    while ( it.hasNext ) { count += 1; it.next }
    count
  }

  // …
}
```

Massive win!

```
case class ArrayList[ T ]( array: Array[ T ] )
  extends BaseIterable[ T ] {

  def getIterator = new Iterator[ T ] {
    var index = 0

    def hasNext = index < array.length

    def next =
      if ( hasNext ) {
        val value = array( index )
        index += 1
        value
      } else
        throw new NoSuchElementException
  }
}
```

- No boilerplate
- Implement only “missing bits”
- Mix and match multiple traits!
- Need features?
  - Add to your concrete class
- Need performance?
  - Override the default implementation

5. PATTERN MATCHING

Enhancing our domain model

Java

```
enum MaritalStatus {
  single, married, divorced, widowed
}

enum Gender {
  male, female
}

class Person {
    private String name;
    private String surname;
    private int age;
    private MaritalStatus maritalStatus;
    private Gender gender;
    // …
}
```

Enhancing our domain model

Java

```
enum MaritalStatus {
  single, married, divorced, widowed
}

enum Gender {
  male, female
}

class Person {
    private String name;
    private String surname;
    private int age;
    private MaritalStatus maritalStatus;
    private Gender gender;
    // …
}
```

Scala

```
sealed trait MaritalStatus
case object Single extends MaritalStatus
case object Married extends MaritalStatus
case object Divorced extends MaritalStatus
case object Widowed extends MaritalStatus

sealed trait Gender
case object Male extends Gender
case object Female extends Gender

case class Person(
  name: String, surname: String, age: Int,
  maritalStatus: Option[ MaritalStatus ],
  gender: Option[ Gender ] )
```

Salutation generation

Java

```
public String getSalutation() {
  if ( gender == null ) return null;
  switch( gender ) {
    case male:
      return "Mr.";

    case female:
      if ( maritalStatus == null )
        return "Ms.";
      switch( maritalStatus ) {
        case single:
          return "Miss";

        case married:
        case divorced:
        case widowed:
          return "Mrs.";
      }
  }
```

Salutation generation

Java

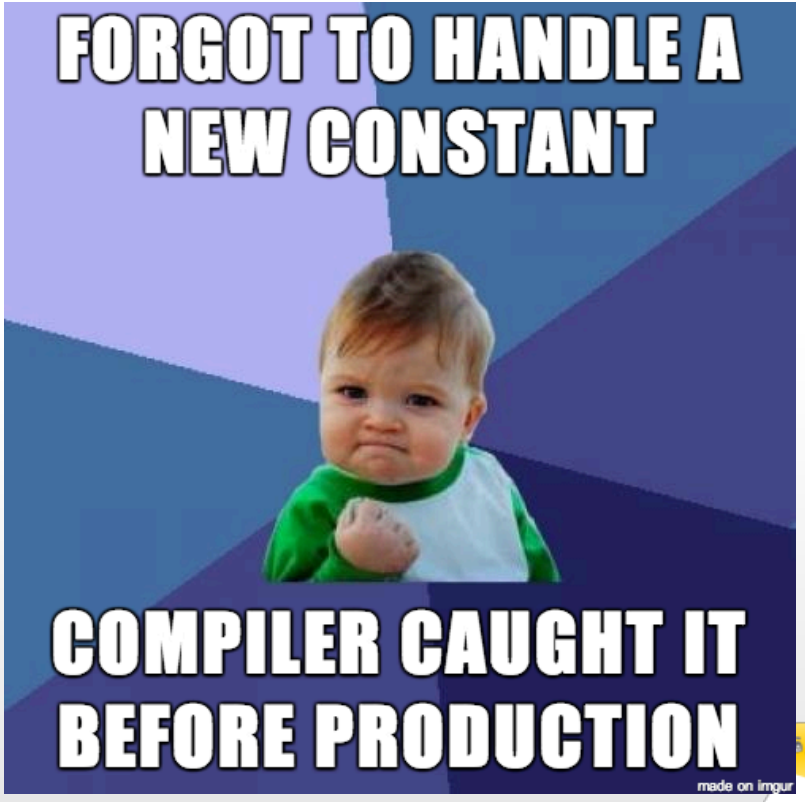
```
public String getSalutation() {
  if ( gender == null ) return null;
  switch( gender ) {
    case male:
      return "Mr.";

    case female:
      if ( maritalStatus == null )
        return "Ms.";
      switch( maritalStatus ) {
        case single:
          return "Miss";

        case married:
        case divorced:
        case widowed:
          return "Mrs.";
      }
  }
```

Scala

```
def salutation: Option[ String ] =
  ( gender, maritalStatus ) match {
    case (Some(Male   ), _             ) => Some("Mr.")
    case (Some(Female), Some(Single)) => Some("Miss")
    case (Some(Female), None           ) => Some("Ms.")
    case (Some(Female), _              ) => Some("Mrs.")
    case (None         , _             ) => None
  }
```

Why is this awesome?

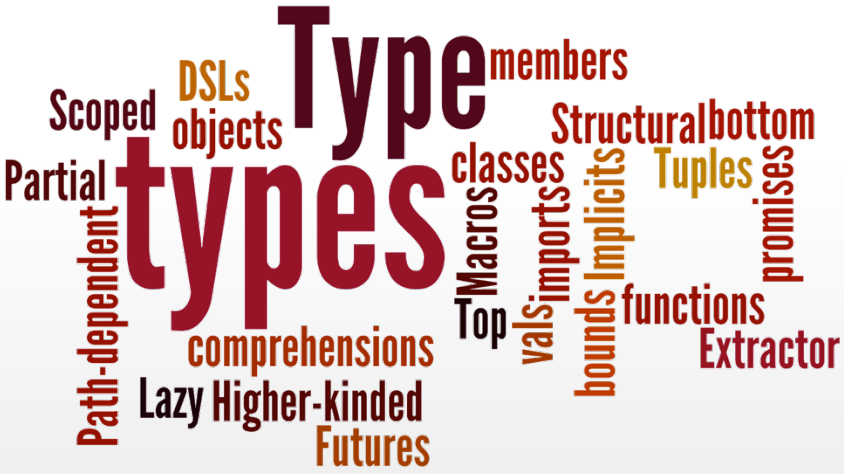
- Other use cases include:
  - Reacting to objects of unknown type
  - Decomposing structures
- Extensible via extractor objects
- ***Compiler performs exhaustiveness check!***



WE'RE ALMOST DONE!

What about Java 8, you ask?

- Java 8 is awesome
- … it validates Scala!
- But it's not up to par
- Closures/lambdas:
  - Syntax is limited
  - Functions aren't 1st class
- Default methods aren't a substitute for traits
  - No self-type annotation
  - No visibility modifiers (e.g. protected members)



Note that we haven't covered…

- For comprehensions
- Lazy vals
- Tuples
- Generics
  - Type bounds
  - Variance
  - Higher-kinded types
- Implicits
  - Type classes
  - Conversions
  - Parameters
- Functions and partial functions
- Top and bottom types
- Scoped imports
- Extractor objects (unapply/-seq)
- Structural types
- Type members
  - … and path-dependent types
- Futures and promises
- Macros
- DSLs

Thanks you for listening!
Questions?

Code is on GitHub

I am on Twitter @tomerg

Scala is on the internetz