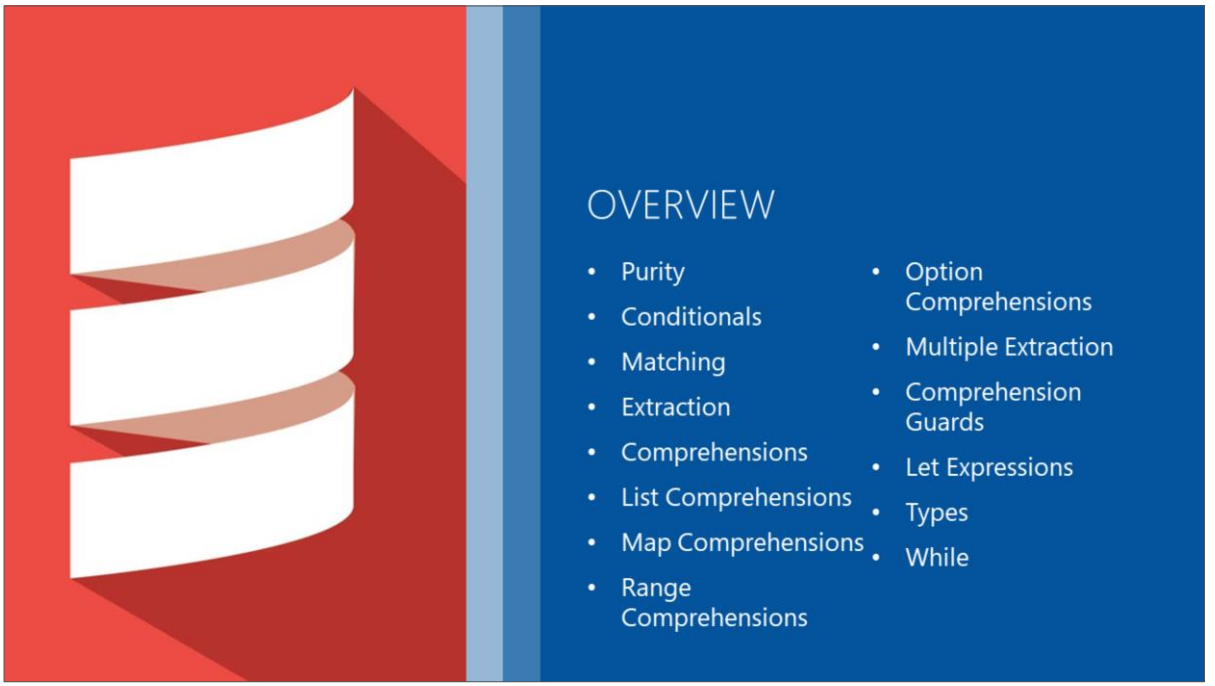
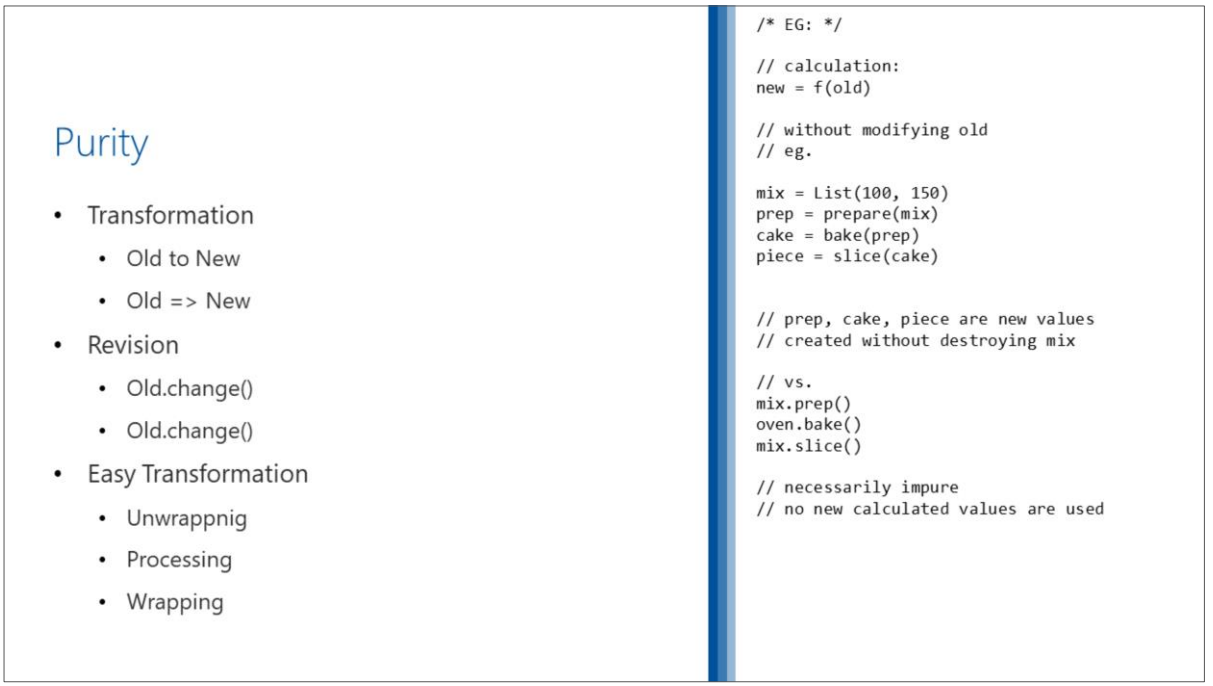
Transformation

SCALA PROGRAMMING
A DATA SCIENCE AND MACHINE LEARNING COURSE


OVERVIEW
Purity
Conditionals
Matching
Extraction
Comprehensions
List Comprehensions
Map Comprehensions
Range Comprehensions
Option Comprehensions
Multiple Extraction
Comprehension Guards
Let Expressions
Types
While



This chapter introduces two fundamental syntaxes of scala: for comprehensions and pattern matching.

These devices facillitate the phrasing of programs as pure transformations: making new values from old.

Comprehensions

- Extraction + Transform = Comprehension
  - fundamental to scala
- Transforming
  - Wrapper to Wrapper
    - change contents
    - new = f(old)
    - yield
  - Wrapper to Unit
    - eg., print() contents
- content <- datasource

```
/*
  for ( part <- source ) f(part)
  for ( part <- source ) yield f(part)
*/

//statement (unitary expression):

//actionS
for(c <- "Hello") println(c)

for( letter <- "Michael") {
  println(letter)
}

//expression:

//transformation
val letters = for ( letter <- "Michael")
  yield letter + 1


// OUTPUT :
H
e
l
l
O

letters = Vector(78, 106, 100, 105, 98, 102,
109)
```

The for comprehension is fundamental in scala

It works on general data structures

Extracting data from them and applying operations

yield wraps the data back into the original structure

After the for keyword, the arrow (<-) has two sides

on the RHS a data source object is given

on the LHS a label for the inner data is defined

The body of the comprehension provides the operations to be applied to the data

yield keyword precedes the value that will be produced

without a yield keyword the whole expression is Unit

List Comprehensions

- Loops
- Extract + Transform
  - an element
  - List[A] to List[B]
  - yield wraps in List[]

```
val names = List(
  "Sherlock", "Jefferson", "Washington"
)

for (n <- names) println(n)


//making a new list, from an old list

val uNames = for (n <- names )
                                yield
n.toUpperCase

println(uNames)


// OUTPUT :

Sherlock
Jefferson
Washington

List(SHERLOCK, JEFFERSON, WASHINGTON)
```

With lists comprehensions extract data by looping

Use a comprehension to extract an element

the body defines what to do with each element

Yielding wraps the elements back up into a List

List[String] becomes a List[Int] when yielding an int

Wrapping-up can instead be seen as a transformation

List[String] becomes List[Int]

when yielding the length of each element

For comprehensions are therefore just like functions:

new = f(old)

Map Comprehensions

- Loops
- Element == (Key, Value)
  - (k, v) <- element
- yield wraps
  - in List[] for single
  - in Map[] for pair
- Transform
  - Map[String, String] to a List[Int]

```
val people_locations = Map(
 "Sherlock" -> "London",
 "Jefferson" -> "Virginia"
)

for( (n, l) <- people_locations)
 println(s"${n} was born in ${l}")

val locations = for {
 (n, l) <- people_locations
} yield l

println(locations)


// OUTPUT:

Sherlock was born in London

Jefferson was born in Virginia

List(London, Virginia)
```

Comprehensions over Maps extract one pair at a time

Using a tuple pattern extracts the key and value separately

Often when creating a new variable, an extraction pattern may be used

Yielding wraps values, however:

when yielding only one value, a List is made

A Map comprehension might transform

Map[String, String] to a List[Int]

by yielding the length of the keys

Range Comprehensions

- Range is not a collection
  - generates on-demand
- Int, Double
  - .to
  - .until
- Looping Idiom
  - loop n times

```
val range = 1 to 10

val numbers = for(i <- 1 to 5) yield i

for(i <- 1 to 3) println(i)

for(j <- 10 until 30 by 10) println(j)

for(z <- 100.to(200).by(50)) println(z)


val ages = for(
         age <- 18 to 19
         if age % 2 == 0
) yield age


// OUTPUT :
1
2
3

10
20

100
150
200

ages: IndexedSeq[Int] = Vector(18)
```

A range is not a collection

However it behaves as-if it is one

Rather than store many elements in memory

A range calculates an element on demand

Ranges are most easily created using the methods

.to or .until available on integers, doubles, etc.

They are convenient when looping for a fixed number of iterations

1 to 3 provides the elements 1, 2, 3

a comprehension would therefore run three times

Option Comprehensions

- Extract + Transform = Comprehension
- Extract + Loop = Collections Comprehension
- : Option
  - None
  - Some(v)
- : Option[A] to Option[B]
  - None[A] to None[B]
  - Some[A] to Some[B]

```
// Any = String | Int | ...
val x: Any = 5
val y: Any = "X"

// Option[String] = None | Some("")
val name: Option[String] = None

val location: Option[String] = Some("UK")

for {
  l <- location
  n <- name
} println( l + n )

for {
  l <- location
} println(l)


println(x)
println(y)

// OUTPUT :

UK
5
X
x: Any = 5
y: Any = X
```

Comprehensions are for extracting and transforming data

they are not necessarily loops

and may be used with any sort of data structure

The data type Option is describes objects created with either:

None which is an empty option

Some(v) which is an option containing a v

Options therefore contain, at most, one value

A for-yield on an option

extracts this value if it exists, operates on it, and wraps it

if there is no value, you get the Option, None

Multiple Extraction

- Extract per Extract
- Collections
  - loop per loop
- Option
  - unpack given unpack

```
val names = List(
  "Michael Burgess".split(" "),
  "Michael Holmes".split(" "),
  "Sherlock Holmes".split(" ")
)

println(
  for {
    parts <- names
    part <- parts
  } yield part
)

for(x <- "ABCDEF"; y <- 0 until 6)
         yield x + y.toString

// OUTPUT :
List(Michael, Burgess, Michael, Holmes,
Sherlock, Holmes)

res15: IndexedSeq[String] = Vector(
A0, A1, A2, A3, A4, A5,
B0, B1, B2, B3, B4, B5,
C0, C1, C2, C3, C4, C5,
D0, D1, D2, D3, D4, D5,
E0, E1, E2, E3, E4, E5,
F0, F1, F2, F3, F4, F5)
```

Comprehensions become especially useful when operating on multiple data structures

the specific data structure in question defines what happens upon extraction

with collections, every subsequent extraction is an inner loop

therefore the second extract runs for every one of the first

Comprehension Guards

- Extract + Filter + Transform
  - filter source before transform
- if
  - include when true
  - reject when false

```
val names = List(
  "Michael Burgess".split(" "),
  "Michael Holmes".split(" "),
  "Sherlock Holmes".split(" ")
)

println( for {
  parts <- names
  part <- parts
  if parts(1) == "Holmes"
} yield part )

val myInts = for(
  x <- 1 to 100
  if x % 2 == 0
  if x < 50
) yield x


// OUTPUT :

List(Michael, Holmes, Sherlock, Holmes)

myInts: IndexedSeq[Int] = Vector(
2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24,
26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46,
48)
```

After extracting, a for-comprehension can filter elements

those which pass a test are forwarded on to the body

those which do not are thrown away

The if keyword is used to specify a condition

if the condition is true the element will be included

Let Expressions

- <-
  - Extract
  - Wrapper[T] to T
- <- then <-
  - Wrapper[Wrapper[T]] to T
- =
  - Label
  - A to A
  - scoped to comprehension

```
val people = List(
 "Michael Burgess",
 "John Doe",
 "Jane Doe"
)

val doeNames = for {
  person <- people
  lastName = person.split(' ')(1)
  if lastName == "Doe"
} yield person

println(doeNames)

// OUTPUT :
List(John Doe, Jane Doe)
```

It is sometimes convenient to label a piece of the extracted data without performing an additional extraction

use =

this creates a label that may be used in the body

it does not persist as a variable beyond the comprehension

because it is a reference to part of the extraction

Types

- Logic is in types
  - considered later in detail
- <-
  - unwrap
- yield
  - wrap

```
val str = for(c <- "Hello World") yield c
val chrs = for(c <- "Hello") yield c + ""

println(str)
println(chrs)
```

In general, comprehensions can only be understood in terms of types

The final type of the comprehension is the same as the source type

Each <- extraction is an unwrapping of the type

The yield is a wrapping-back up

While

- While
  - imperative
  - impure
  - not n = f(o)
- More efficient
  - provide pure API

```
var population = 0

while(population < 10) {
  population += 1
}

println(population)

var line = ""
while(line != "q") {
  line = io.StdIn.readLine("Message? ")
  if(line == "damn") {
   line = "q"
  } else {
    println(line)
  }
}

// OUTPUT :
10
Message? Hi
Message? q
```

While loops exist in scala

- they are fundamentally imperative
- do not interoperate with functional code well

A while loop is a repetition of an action while some condition is true

- it cannot be understood as an input -> output transformation

While loops are more efficient than for comprehensions over collections

- use when efficiency is needed
- within methods that are as-if pure

Extracting

- Encapsulation (vs. Extraction)
  - tames impurity
  - hamstrings purity
- SELECT * …
  - pure
  - extractable
- Extraction
  - Compiler type check
    - LHS pattern
  - Runtime structure check
    - RHS object

```
val (y, (_, z)) =
  ("File System", ("NTFS", 1024))

case class OperatingSystem(
  val name: (String, String),
  val addressSize: Int
)

val OperatingSystem((publisher, title), _) =
  OperatingSystem(
    ("Microsoft", "Windows 10"), 64
  )

println(y)
println(publisher)

// OUTPUT:


File System
Microsoft
```

object-oriented programmers promote encapsulation

> the hiding of data structure behind interfaces which resist revealing that structure
>
> this is a solution to tracking state change (i.e., make it difficult and predictable)

pure functional programmers have no problem with the state change so this form of encapsulation is very counter productive (in, SQL, imagine all tables being encapsulated!)

rather wherever possible data structures should be de-structurable

the extraction syntax enables this

> places an extract pattern on the LHS of an assignment and an object with a matching structure on the RHS
>
> the extraction pattern should have the same type
>
> And the object should have the correct runtime structure
>
> > Type failure can be caught at compile-time
> >
> > A Data structure failure causes a runtime exception

Extraction

- Assigning a piece
- Type Match
  - compile time
  - compiler error on failure
- Structure Match
  - runtime
  - exception on failure
- Matching + Extracting = Pattern Matching
  - very powerful
  - fundamental to scala
  - later topic

```
val List(wname, wlocation) =
    List("Winston", "London")

val (jname, jlocation) =
    ("Jefferson", "Virginia")

val deets =
  ("JFK", ("New York", 1970))

val (n, (l, y)) = deets

//preview:
case class Person(name: String, age: Int, l:
(String, String))

val me = Person("Michael", 27,
                    ("I st", "Bentonville"))

val Person(name, age, loc) = me

println(wname)
println(wlocation)
println(jname)
println(n)
println(loc)

// OUTPUT :
Winston
London
Jefferson
JFK
(I st,Bentonville)
```

When assigning, a pattern may be given on the left

Scala test the type of both sides

And that the pattern matches the structure of the object

object structure is in-memory so is a runtime test

throwing an exception (runtime error) if it fails

The pattern extracts data from the object

Assigning several variables at once

Extracting and matching may be combined into Pattern Matching

a very powerful technique considered later

Matching

- Matching is selecting
  - more detailed conditions
- match { }
  - case
  - condition
  - fat arrow (=>)
- Conditions
  - eg., literals
  - _

```
val name = "Sherlock"
val location = "London"

val message = name match {
 case "Sherlock" => "Evening, Holmes!"
 case "Watson" => "Hello, Dr. Watson"
 case _ => "Bonjour!"
}

println(message)

println(location match {
  case "UK" => "Welcome to the UK"
  case _ => "I do not speak it!"
})

// OUTPUT :

Good Evening, Holmes!

I do not speak it!
```

Selecting one value among many is a very common functional pattern

The match-case expression provides a powerful mechanism to described detailed conditions

The object to be tested is followed by the match keyword

Then a series of cases relating to a value to chose

The fat arrow (=>) relates a case to a value

In the simplest case, a case is just a literal

opposite the variable name is tested

The _ case provides the default value

If none is chosen in prior cases

Pattern Matching

- Match
- Extract
- Select

- Fundamental Syntax

```
val ingredients = List("Flour", "Sugar")
val List(f, s) = ingredients

val people =
 "Michael" :: "Sherlock" :: List()

val pets =
 "Spot" :: "Fluffy" :: Nil

people match {
  case h :: t => println(s"H: ${h} R: ${t}")
  case Nil => println("X")
}

println(people)

// OUTPUT :

H: Michael R: List(Sherlock)
List(Michael, Sherlock)
```

Pattern matching combines extraction patterns with match-case comparisons

While testing a value, extract data from it

Pattern matching is one of the fundamental syntactical devices of scala (and pure functional programming generally)

It is the technique which most conveniently provides polymorphism

Where each case specifics a different type of object

And so provides type-dependent behaviour without inheritance

case

- variable match …
  - selected case => selected value
- Expression
  - (super) type of branches

```
def day(name: String) =
  name slice (0, 2) match {
          case "Mo" => 1
          case "Tu" => 2
          case "We" => 3
          case "Th" => 4
          case "Fr" => 5
          case "Sa" => 6
          case "Su" => 7
          case _ => 0
}

println(day("Monday"))

// OUTPUT :

1
```

The match-case expression compares the value before the match keyword with one of several cases following it

The whole expression evaluates to the case branch which is chosen

The type of a match expression is the most specific type compatible with all its cases

If every cases maps to a Int, the expression has type Int

If one happens to map to a String, the expression has type Any

as the run-time value could be either a string or an int and these have nothing (other than Any's .toString) in common

Matching Tuples

- Left-hand side pattern
  - variables extract
  - literals match
  - _ means anything

```
val points = List((0,0), (1,0), (1,1))

val d = points(0) match {
 case (0, y) => s"V $y"
 case (x, 0) => s"H $x"
 case (1, _) => "D"
 case _ => "?"
}

println(d)


// OUTPUT :

V 0
```

Tuples demonstrate the basic capabilities of pattern matching

If the pattern on the LHS of the case matches that branch will be chosen

In the scope of the return expression any variables defined by extraction will be available

Variabels defined with the name _ are ignored

The _ in the case position is the default case when no other matches

Guards

- Filter after extraction
  - if
  - complex tests

```
val people = List(
  ("Michael", 26),
  ("Sarah", 15),
  ("John", 18)
)

for(person <- people) println(
  person match {
    case (name, age) if age >= 18 =>
      s"$name is eligible for work "

    case (name, _) =>
      s"$name is not eligible for work"
  }
)


// OUTPUT :

Michael is eligible for work

Sarah is not eligible for work

John is eligible for work
```

Case extraction matches only when the structure is the same

Additional filters on the specific value extracted can be added

The if keyword after the pattern may introduce a test that is applied to the value extracted

The case return expression (after =>) is only chosen if this test passes

Sequences

- Extraction
  - Fixed number of extractables (eg. tuple)
  - Variable number (sequence extraction)
- Sequences
  - ::
  - (element : T) :: (list : List[T])

```
val names = "Michael" :: List()

val more = "John" :: names

val evenMore = "Mark" :: more

println(names)
println(more)
println(evenMore)


println(more match {
  case f :: s :: rest => s
  case _ => ""
})
```

Many common types have their own extraction capabilities

In general you can add two forms of extraction capability to an object: tuple-based extraction called (unapply) and sequence-based extraction called unapplySeq

With sequence extraction novel patters may be used

Each time, the pattern must match the way that value could have been built

The :: operator can be used to create lists and destructure them

When creating the syntax is (element : T) :: (list : List[T])

With Nil being the empty list all lists can be expressed as

e_1 :: e_2 :: e_n :: Nil

So when extracting head :: tail peals off the first element into head and leaves the remaining list in tail

for yield match

- For comprehensions and pattern matching
  - foundations of scala programming
  - powerfully describe pure immutable transformations
- Comprehensions unwraps
- Pattern match extracts
  - defines a new value with a new structure to rewrap
- It is trivial to transform
  - List[(String, String, String)] to List[String]
  - Future[List[Option[(Int,Int)]]] to Future[List[String]]

```
val tupleList = List(
 ("A", "B", "C"), (1, 2, 3), (false, 1, "B")
)

val eachStartsWith = for (triple <- tupleList)
  yield triple match {
    case ("A", _, _) => "Starts with A"
    case (1, _, _) => "Starts with 1"
    case (_, _, _) => "Unknown!"
  }

println(eachStartsWith)


// OUTPUT :
List(
 Starts with A, Starts with 1, Unknown!
)
```

For comprehensions and pattern matching are the foundations of scala programming

Together they are a powerful device for spelling out pure immutable transformations of data

The for comprehensions unwraps the incoming source type

The pattern match extracts the wrapped value

And defines a new value with a new structure to rewrap

It is trivial to transform from List[(String, String, String)] to List[String]

But also more complex structures

Eg., Future[List[Option[(Int,Int)]]] to Future[List[String]]

Aside: Regex

- Regex Objects
  - .r method on String
- Regex Patterns
  - Match
  - Extraction

```
val people = List("Richard Dawkins",
  "Richard Feynman", "Lewis Carol",
  "David Lewis", "Carol Richards"
)

val Richards =
  """Richard (\w+)""".r

val Lewises =
  """(?:\w+ )?Lewis(?: \w+)?""".r

println(
  for (person <- people) yield person
  match {
    case Richards(name) =>
      "a scientist named: " + name
    case Lewises() => "a logician"
    case _         => "?"
  }
)

// OUTPUT :
List(
  a scientist named: Dawkins,
  a scientist named: Feynman,
  a logician, a logician, ?
)
```

It is possible to define an extractor on any object (considered in the OO chapter) by defining an .unapply method

This method is free to behave however it wishes in what data it provides for extraction

Allowing an extremely neat interface with regular expressions

The .r method on strings creates a regex object

Regex objects can be put on the LHS of an extraction (eg. After a case)

Matching the pattern occurs when the regex matches

Extracting the pattern occurs when the regex defines groups, the variables extracted correspond to the groups, in order, in the regex

Aside: Recursion

- Recursion
  - solved problem: .map .flatMap et al.
- Matching useful for explicit
  - One branch provides the terminating case
  - One branch provides the recursive case

```
val names = List(
  "michael", "sherlock", "watson"
)

println( names match {
    case head :: tail => s"Mr. $head"
    case Nil => "GUEST"
})


def breakUp(seq: List[String]): String =
  seq match {
    case head +: tail =>
       s"$head +: " + breakUp(tail)

    case Nil => "END"
  }

println(breakUp(List(
  "Michael", "John","Burgess"
)))


// OUTPUT :
Mr. michael
Michael +: John +: Burgess +: END
```

While recursion is an essential of pure functional programming to express loops, in scala it is rarely explicitly used

> Looping and data structure processing is often done with combinators (.map, etc.), for-comprehensions, etc.

Recursive algorithms are particularly clear when written with a match-case

> One branch provides the terminating case
>
> One branch provides the recursive case

A list can be analysed head :: tail by recursing until tail is the empty list (Nil)

Aside: Casting

- Pattern Matching for casting
  - Avoid
  - Suggests failure to type correctly
  - Run-time failure possible
- : Any
  - almost always a sign of a mistake
  - effectively means "do not check"

```
/* ERROR: */

val objs = List(1, false, "Hello", 1.0)

for(o <- objs) println( o match {
  case i : Int => s"An int: $i!"
  case b : Boolean => s"A bool: $b!"
  case s : String => s"A string: $s"
})


// OUTPUT :

An int: 1!
A bool: false!
A string: Hello


scala.MatchError: 1.0 (of class
java.lang.Double)
  at .$anonfun$res1$1(<pastie>:18)
... 36 elided
```

Pattern matching can be used to cast values

> This is to be avoided. Any occurs in situations where the compiler has failed to understand the types of your program correctly
>
> This is rarely a fault of the compiler and often the programmer can fix the issue by adding type annotations
>
> Or can change the design of the program so type checking is still possible
>
> The type 'Any' is almost always a sign of a mistake as this type renders it impossible for the compiler to give you meaningful information about type failures

In the case of a List[Any] the programmer has typically chosen the wrong data structure, mixed disparate lists together or failure to introduce a class definition

Exercise