Comp 311
Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC

Some Language Features You Might Find Useful for The Homework

Requires Clauses on Class Constructors

```
case class Name(field1: Type1, …, fieldN: TypeN)
    require (boolean-expression)
```

- Checked on every constructor call
- Because case class instances are immutable, these ensures the property holds for the lifetime of an instance

Equals on Case Classes

- The equals method on a case class instance checks for structural equality with its argument:

```
Rational(4,6).equals(Rational(4,6)) ↦

true
```

Equals on Case Classes

- Note that equals is a binary method, and so we can also write this expression as:

```
Rational(4,6) equals Rational(4,6) ↦

true
```

Equals on Case Classes

- Of course, the built in equals method does not check for mathematical equality:

```
Rational(4,6) equals Rational(2,3) ↦

false
```

Equals on Case Classes

- Why is this definition of equality acceptable on case classes?
- What other definition is available to us?

```
Rational(4,6) equals Rational(2,3) ↦

false
```

Short-Circuiting And and Or Operators

- Just as we have defined a short-circuiting if-then-else operator, we can define short-circuiting and/or operators:

&& ||

- How do we define the static and dynamic semantics of these operators?

- When are they useful?

Calling and Defining Parameterless Methods Without Parentheses

```
def toString() = { … }
```

vs.

```
def toString = { … }
```

Calling and Defining Parameterless Methods Without Parentheses

```
Rational(4,6).toString()
```

vs.

```
Rational(4,6).toString
```

The Uniform Access Principle

- Client code should not be affected by whether an attribute is defined as a field or a method
- Only applies to immutable methods
- Can be strange even for some immutable methods (consider `reduce`)

Block Expressions

- The syntactic form

```
{e1
…
eN}
```

- is also an expression.
- Evaluate and remove top to bottom until the last expression
- Reduce to the value of the last expression

Block Expressions

- The syntactic form

```
{ 1 == 1
  2 == 3
  1 + 2 }
```

- is also an expression.
- Evaluate and remove top to bottom until the last expression
- Reduce to the value of the last expression

Block Expressions

- The syntactic form

```
{  true
   2 == 3
   1 + 2 }
```

- is also an expression.
- Evaluate and remove top to bottom until the last expression
- Reduce to the value of the last expression

Block Expressions

- The syntactic form

```
{ 2 == 3
  1 + 2 }
```

- is also an expression.
- Evaluate and remove top to bottom until the last expression
- Reduce to the value of the last expression

Block Expressions

- The syntactic form

```
{ false
1 + 2 }
```

- is also an expression.
- Evaluate and remove top to bottom until the last expression
- Reduce to the value of the last expression

Block Expressions

- The syntactic form

```
{ 1 + 2 }
```

- is also an expression.
- Evaluate and remove top to bottom until the last expression
- Reduce to the value of the last expression

Block Expressions

- The syntactic form

```
{ 3 }
```

- is also an expression.
- Evaluate and remove top to bottom until the last expression
- Reduce to the value of the last expression

Block Expressions

- The syntactic form

- is also an expression.
- Evaluate and remove top to bottom until the last expression
- Reduce to the value of the last expression

Block Expressions

- The syntactic form

```
{
  e1
  …
  eN
}
```

- is also an expression.
- Evaluate and remove top to bottom until the last expression
- Reduce to the value of the last expression

Block Expressions

- The syntactic form

```
{e1;…;eN}
```

- is also an expression.
- Evaluate and remove top to bottom until the last expression
- Reduce to the value of the last expression

Val Expressions

The syntactic form

```
val x = e
```

is also an expression with static type `Unit`

Val Expressions

To reduce

```
val x = e
```

in a block expression:

Reduce **e** to value **v**

Remove the binding expression and replace all free occurrences of **x** in the remainder of the block expression with **v**

Otherwise, Please Restrict Your Homework Submission to Features Covered in Class

- These should be the only import statements in your program:

```
import junit.framework.TestCase

import junit.framework.Assert._
```

Abstract Datatypes

Abstract Datatypes

- Often, we wish to abstract over a collection of compound datatypes that share common properties

- For example, we might wish to define an abstract datatype for shapes, with separate case classes for each of several shapes

- For this purpose, we define an *abstract class* and use *subclassing*

Abstract Datatypes

```
abstract class Shape
case class Circle(radius: Double) extends Shape
case class Square(side: Double) extends Shape
case class Rectangle(height: Double, width: Double) extends Shape
```

Recall Our Design Recipe

- **Analysis**: What are the objects in the problem domain? What data types we will use to represent them?

- **Contract**: What is name of our functions and their parameters? What are the requirements of the data they consume and produce? What is the meaning of what our program computes?

- **Repeat** until we are confident in our program's correctness

  - Write some **tests**

  - Sketch a function **template**

  - **Define** the function

Recall Our Design Recipe

- **Analysis**: This is the stage where we would discover we wish to model our problem domain with functions over an abstract datatype

- **Contract**: What contract holds for each function? Do additional constraints and assurances hold for specific subclasses?

- **Repeat** until we are confident in our program's correctness

  - Write some **tests:** Same as before

  - Sketch a function **template**: This needs re-examination

  - **Define** the function

The Design Recipe for Abstract Datatypes

- Our Function Template for computing with abstract datatypes depends on answering the following questions:
  - Do I expect to eventually add more subclasses?
  - Do I expect to eventually add more functions?

Case 1

We Expect Few New Functions
But Many New Variants

Case 1: We Expect Few New Functions But Many New Variants

- This is a case that object-oriented programming handles well
- Classic example domains: GUI Programming, Productivity Apps, Graphics, Games
- Declare an abstract method in our superclass and provide a concrete definition for each sub-class

  a.k.a.,

  The Union Pattern (for the datatype definitions)

  The Template Method Pattern (for the function definitions)

Abstract Datatypes

```
abstract class Shape {
  def area(): Double
}
```

Abstract Datatypes

```
case class Circle(radius: Double) extends Shape {
  val pi = 3.14

  def area() = pi * radius * radius

}
```

Abstract Datatypes

```
case class Square(side: Double) extends Shape {
  def area() = side * side

}
```

Abstract Datatypes

```
case class Rectangle(length: Double, width: Double)
extends Shape {

  def area() = length * width

}
```

How Do Abstract Classes Affect Our Type Checking Rules?

- When type checking a class definition, ensure that all abstract methods declared in the superclass are actually defined, with *compatible* method types

- When type checking a collection of class definitions, ensure that there are no cycles in the class hierarchy!

How Do Abstract Classes Affect Our Type Checking Rules?

- If a method is called on a receiver whose static type is an abstract class, extract an arrow type from the declaration (just as with a definition in a concrete class)

```
expr.area() ↦

Shape.area() ↦

() → Double
```

Type Checking Arguments to a Method Call

- The static types of an argument might no longer be an exact match:

```
abstract class Shape {
  def area(): Double

  def makeLikeMe(that: Shape): Shape
}
```

(Let us set aside the concrete definitions of `makeLikeMe` for awhile)

Now Consider a Call to Matcher With Concrete Types

```
Circle(1).makeLikeMe(Circle(2)) ⇒

Circle.makeLikeMe(Circle) ⇒

(Shape → Shape)(Circle)
```

And now we are stuck…

Recall The Substitution Model of Type Checking

- To type check the application of a function to arguments:
  - Reduce the function to an arrow type
  - Reduce the arguments, left to right, to static types
  - If the argument types match the corresponding parameter types, reduce the application to the return type

Subtyping

- We need to widen our definition of matching a type to include subtyping:

- A class is a subtype of the class it extends

- Subtyping is Reflexive:

`A <: A`

- Subtyping is Transitive:

If `A <: B` and `B <: C` then `A <: C`

Subtyping

- All types are a subtype of type **Any**
- Type `Nothing` is a subtype of all types
  - There is no value with value type `Nothing`

Recall The Substitution Model of Type Checking

- To type check the application of a function to arguments:
  - Reduce the function to an arrow type
  - Reduce the arguments, left to right, to static types
  - If the argument types **are subtypes of** the corresponding parameter types, reduce the application to the return type

Applying a Class Method Revisited

- To reduce the application of a method:

```
C(v1, …, vk).m(arg1, …, argN)
```

  - Reduce the receiver and arguments, left to right
  - Reduce the body of m, replacing constructor parameters with constructor arguments and method parameters with method arguments

Applying a Class Method Revisited

- To reduce the application of a method:

```
C(v1, …, vk).m(arg1, …, argN)
```

  - Reduce the receiver and arguments, left to right

  - **Find the body of m in C and reduce to that,** replacing constructor parameters with constructor arguments and method parameters with method arguments

The Body of *m*

- To find the body of method `m` in type `C`:
  - Find the definition of m in the body of `C`, if it exists
  - Otherwise, find the body of `m` in the immediate superclass of `C`

Overriding Methods

- Our new rules also handle method overriding!
- Use overriding when:
  - Factoring out a method definition common to several variants
    - Suppose several shapes compute their area in the same way
  - Augmenting the behavior of classes we do not maintain

Overriding Methods

- Scala requires that overriding method definitions include the keyword **`overrides`**

- Why require this extra keyword?

The Fragile Base Class Problem

- Suppose I define a base class `Shape`
- Later a client extends `Shape` with class `Triangle` and defines a private method `position` to record the position of one point of a triangle
- Yet later, I release a new version of my class `Shape` with a private method `position` to record the position of the center of the shape

The Fragile Base Class Problem

- This is an example of *accidental overriding*
- The `overrides` keyword catches the problem when the subclass `Triangle` is recompiled against the new version of `Shape`

Two Occasions to Consider Overriding

- The default equals methods on case classes:

```
Rational(4,6) equals Rational(2,3)
```

Two Occasions to Consider Overriding

- The default toString methods on case classes:

```
Rational(4,6) + Rational(2,3) ↦

Rational(4,3)
```

What is printed during Interactions is determined by toString

Two Occasions to Consider Overriding

- The default toString methods on case classes:

```
Rational(4,6) + Rational(2,3) ↦

4/3
```

What is printed during Interactions is determined by toString

Defining and Overriding Methods

- Recall our rule for abstract methods
  - When type checking a class definition, ensure that all abstract methods declared in the superclass are actually defined, with *compatible* method types
- We need to:
  - Augment our rule to mention overriding (this is easy)
  - Clarify “compatible method types”

Defining and Overriding Methods

- When type checking a class definition, ensure that:
  - All abstract methods declared in the superclass are actually defined, with compatible method types
  - The types of all overriding methods are compatible with the types of the methods they override

Defining and Overriding Methods

- When type checking a class definition, ensure that:
  - All abstract methods declared in the superclass are actually defined, and their types are subtypes of the method types in the corresponding declarations
  - The types of all overriding methods are subtypes of the method types they override

Arrow Types and Subtyping

- How do we define subtyping on arrow types?

- Historically this has been a painful source of bugs in object-oriented languages

Arrow Types and Subtyping

- The substitution principle of arrow typing:
- If a function **f** has type **S→T**

  and **S→T <: U→V**

  then **f** can be safely used in any context requiring a function of type **U→V**

Consider an Example

```
abstract class Shape {
  def area(): Double
  def makeLikeMe(that: Shape): Shape
}
```

- So, `makeLikeMe` has type `Shape → Shape`
- We are required to define it in all subclasses of `Shape`

Consider a Calling Context

```
def matcher(shape1: Shape, shape2: Shape) = {
  shape1.makeLikeMe(shape2).area
}
```

- What are some parameter types we can safely declare for `makeLikeMe` when defining it in class `Circle`?

- What are some return types we could safely declare?

Consider a Calling Context

```
def matcher(shape1: Shape, shape2: Shape) = {
  shape1.makeLikeMe(shape2).area
}
```

- Could class `Circle` define the parameter type of `makeLikeMe` to be `Circle`?

```
// NOT ALLOWED
def makeLikeMe(that: Circle): Shape = that
```

Consider a Calling Context

```
matcher(Circle(1), Square(1)) ↦
```

```
Circle(1).makeLikeMe(Square(1)).area ↦
```

And now we are stuck…

Consider a Calling Context

```
def matcher(shape1: Shape, shape2: Shape) = {
  shape1.makeLikeMe(shape2).area
}
```

- Could class `Circle` define the parameter type of `makeLikeMe` to be `Any`?

```
// This abides by our substitution principle
def makeLikeMe(that: Any): Shape = this
```

Consider a Calling Context

```
matcher(Circle(1), Square(1)) ↦

Circle(1).makeLikeMe(Square(1)).area ↦

Circle(1).area ↦

3.14
```

Consider a Calling Context

```
def matcher(shape1: Shape, shape2: Shape) = {
  shape1.makeLikeMe(shape2).area
}
```

- Could class `Circle` define the return type of `makeLikeMe` to be `Any`?

```
// NOT ALLOWED
def makeLikeMe(that: Any): Any = “what’s up?”
```

Consider a Calling Context

```
matcher(Circle(1), Square(1)) ↦
```

```
Circle(1).makeLikeMe(Square(1)).area ↦
```

```
“what’s up?”.area ↦
```

And now we are stuck…

Consider a Calling Context

```
def matcher(shape1: Shape, shape2: Shape) = {
  shape1.makeLikeMe(shape2).area
}
```

- Could class `Circle` define the return type of `makeLikeMe` to be `Circle`?

```
// This abides by our substitution principle
def isSimilarTo(that: Any): Circle = this
```

Consider a Calling Context

```
matcher(Circle(1), Square(1)) ↦

Circle(1).makeLikeMe(Square(1)).area ↦

Circle(1).area ↦

3.14
```

Subtyping for Arrow Types

- A type **S→T** is a subtype of **U→V** iff
  - **U** is a subtype of **S**
  - **T** is a subtype of **V**
- We say that arrow types are *contravariant* in their parameter type and *covariant* in their return type

A Limitation on Subtyping of Method Types in Scala

- Parameter types of overriding methods must match exactly in Scala

- This restriction is shared with Java and is a limitation of the JVM

- We will see other uses of arrow types in Scala where this restriction is not in place

Why Methods?

- Remember we are in Case 1: We Expect Few New Functions But Many New Variants

- How do methods help with this case?

    - All functions we support are declared in our abstract class

    - New variants can be added without changing old code:

        - Simply implement all the declared methods

Disadvantages of Methods

- If new functionality is added, every class definition must be modified to include it

Throwing And Catching Exceptions

We Can Throw and Catch Exceptions as in Java

```
def assertConstructorFail(m:Int, n:Int) = {
  try {
    Rational(m,n)
    fail()
  }
  catch {
    case e: IllegalArgumentException => {
    }
  }
}
```

Syntax For Try/Catch

```
try expr
 catch {
   case Pattern => expr
   …
}
```

Syntax For Throw

```
throw expr
```

Static Semantics For Throw

If `e` has static type `T` and

```
T <: Throwable
```

then

```
throw e
```

has static type

```
Nothing
```

Static Semantics For Try/ Catch

- Given an expression `e`:

```
try expr0
 catch {
   case Pattern => expr1
   …
   case Pattern => exprN
}
```

- Where `expr0: T0, expr1: T1, …, exprN: TN,`

- The type of `e` is the least type `T` such that:

```
T <: T0, T <: T1,…,T <: TN
```

Static Semantics For Try/ Catch

- The type of **e** is the least type **T** such that:

`T <: T0, T <: T1,…,T <: TN`

- Note that we can now use this approach to go back and define better static typing rules for `if-else` and `match` expressions

Dynamic Semantics For Throw

- To explain the semantics of throw, we must introduce new terminology

- Let the *continuation* of an expression **e** refer to all that remains to be done in a computation after **e** is reduced

- We can think of a continuation as an expression with a "hole" in it, corresponding to **e**

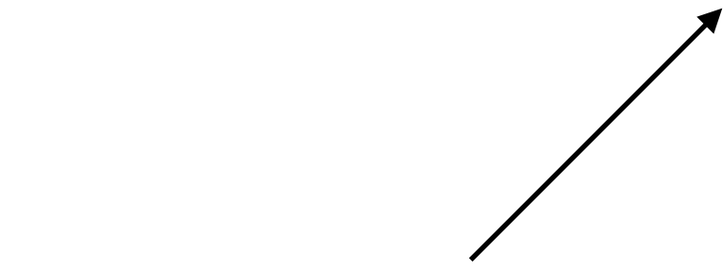
- Equivalently, we can think of a continuation as function that takes a parameter, corresponding to the result of evaluating **e**

Example Continuation

```
matcher(Circle(1), Square(1))
```

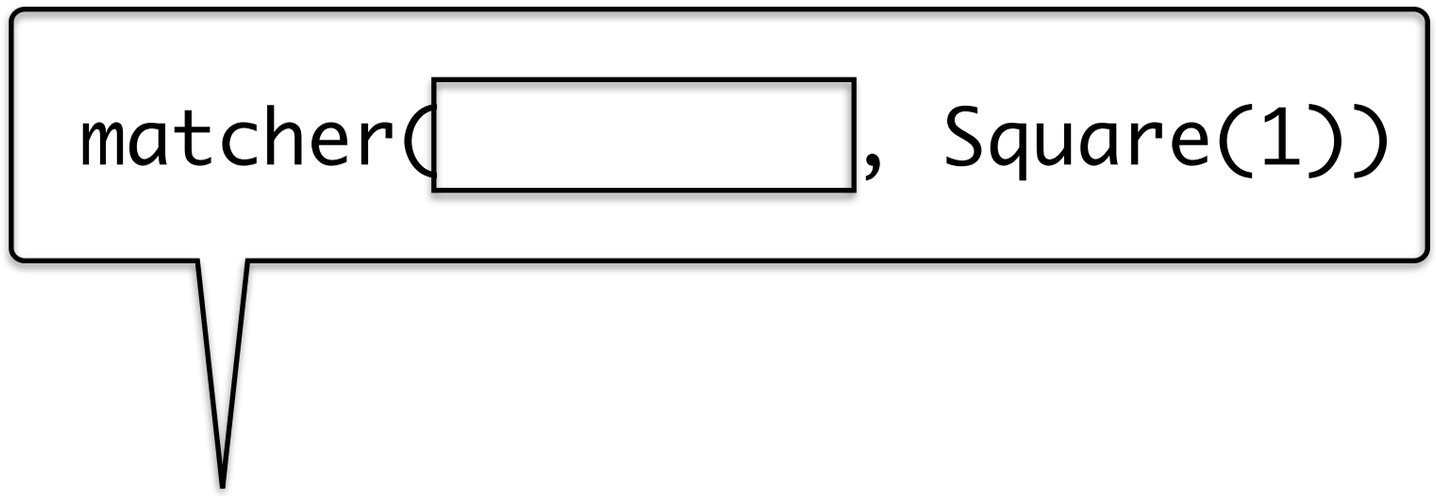
Example Continuation

```
matcher(Circle(1), Square(1))
```



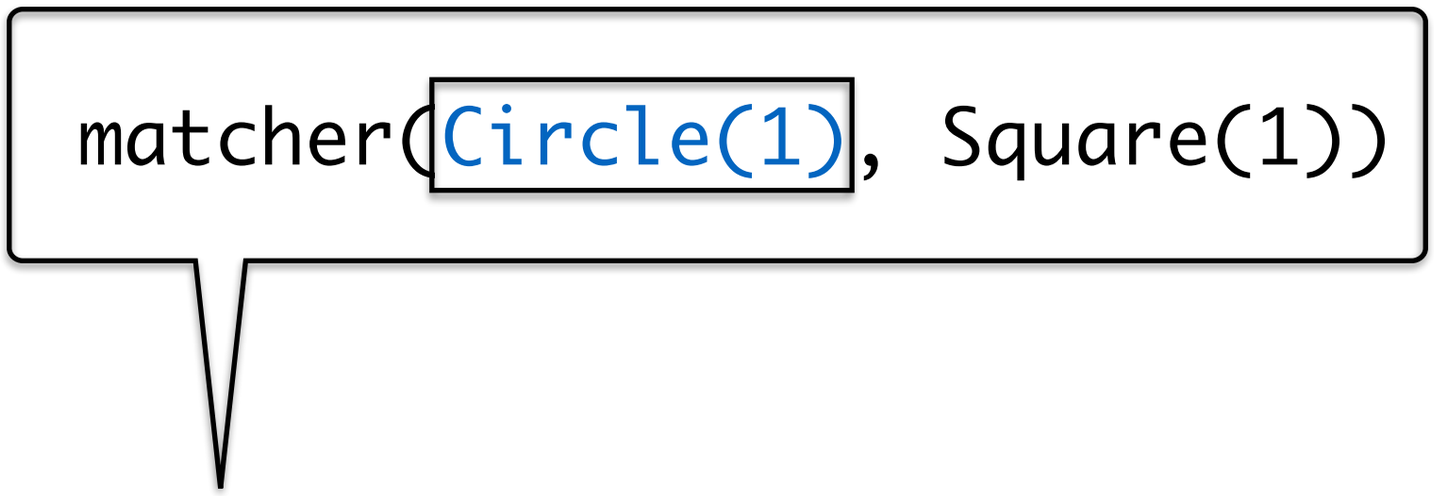
Let this be our expression **e**

Example Continuation



Then this is the continuation of **e**

Example Continuation



Once e is reduce to a value, the box is filled in, and the continuation can be reduced

Reducing a Throw Expression

- To reduce a throw expression:

`throw e`

  - Reduce **`e`** to a value **`v`**
  - Replace the continuation of the throw expression with the special expression **`throw v`**

Reducing a Try/Catch

- To reduce a try/catch expression:

```
try expr0
 catch {
   case Pattern => expr1
    …
   case Pattern => exprN
}
```

Reducing a Try/Catch

- Set aside the continuation `C` of the `try/catch`
- Reduce the body of the `try` in a special continuation `D`
- If `D` reduces to `throw v`:
  - Restore the continuation `C`
  - Try matching `v` against each pattern in the `catch` clause
  - If a match is found, evaluate the body of the matching case
  - Otherwise, reduce to `throw v`

Case 2

We Expect Many New Functions But Few New Variants

Case 2: We Expect Many New Functions But Few New Variants

- This is a case the traditional functional programming handles well

- Classic example domains: Compilers, theorem provers, numeric algorithms, machine learning

- Declare a top-level function with cases for each data variant

a.k.a., The Visitor Pattern

Again We Turn to Pattern Matching

```
val pi = 3.14

def area(shape: Shape) = {
  shape match {
    case Circle(r) => pi * r * r
    case Square(x) => x * x
    case Rectangle(x,y) => x * y
  }
}
```

We Can Define Arbitrary Functions Without Modifying Data Definitions

```
def makeLikeFirst(shape0: Shape, shape1: Shape) = {
  (shape0, shape1) match {
    case (Circle(r), Square(s)) => Circle(s)
    case (Circle(r), Rectangle(l,w)) => Circle((l+w)/2)

    case (Square(s), Circle(r)) => Square(r)
    case (Square(s), Rectangle(l,w)) => Square((l+w)/2)

    case (Rectangle(l,w), Circle(r)) => Rectangle(r,r)
    case (Rectangle(l,w), Square(s)) => Rectangle(s,s)

    case _ => shape1
  }
}
```

But A New Data Variant Requires Us To Modify All Functions Over the Datatype

```
val pi = 3.14

def area(shape: Shape) = {
  shape match {
    case Circle(r) => pi * r * r
    case Square(x) => x * x
    case Rectangle(x,y) => x * y
    case Triangle(b,h) => b*h/2
  }
}
```

But A New Data Variant Requires Us To Modify All Functions Over the Datatype

```
def makeLikeFirst(shape0: Shape, shape1: Shape) = {
  (shape0, shape1) match {
    case (Circle(r), Square(s)) => Circle(s)
    case (Circle(r), Rectangle(l,w)) => Circle((l+w)/2)
    case (Circle(r), Triangle(b,h)) => Circle(b)

    case (Square(s), Circle(r)) => Square(r)
    case (Square(s), Rectangle(l,w)) => Square((l+w)/2)
    case (Square(s), Triangle(b,h)) => Square(b+h/2)

    case (Rectangle(l,w), Circle(r)) => Rectangle(r,r)
    case (Rectangle(l,w), Square(s)) => Rectangle(s,s)
    case (Rectangle(l,w), Triangle(b,h)) => Rectangle(b,h)

    case _ => shape1
  }
}
```