Case Classes and Pattern Matching

Case Classes

Making a class a *case class* automatically adds conveniences.

```
case class Var(name: String)
case class BinOp(operator: String, left: Var, right: Var)
```

- Defines a factory method so you don't need `new Var(...)`
- Makes all constructor parameters `val` fields
- Defines structural `equals` and `hashCode` methods
- Defines a `copy` method with default parameters for each field

```
val x = Var("x")
x.name                           // x
x == Var("x")                    // true
x != Var("y")                    // true
x.hashCode == Var("x").hashCode  // true
val plus = BinOp("+", x, Var("y"))
val minus = plus.copy(operator = "-")
minus == BinOp("-", x, Var("y"))    // true
```

Case Classes for Models

Because of their conveniences, case classes are often used for model objects. From play-scala-forms-example:

```scala
package models

/**
 * Presentation object used for displaying data in a template.
 *
 * Note that it's a good practice to keep the presentation DTO,
 * which are used for reads, distinct from the form processing DTO,
 * which are used for writes.
 */
case class Widget(name: String, price: Int)
```

Pattern Matching

Case classes are powerful when combined with pattern matching. Given this family of case classes representing arithmetic expressions:

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
```

we can simplify expressions easily with pattern matching:

```
def simplifyTop(expr: Expr): Expr = expr match {
  case UnOp("-", UnOp("-", e)) => e // Double negation
  case BinOp("+", e, Number(0)) => e // Adding zero
  case BinOp("*", e, Number(1)) => e // Multiplying by one
  case _ => expr
}
val doubleNegX = simplifyTop(UnOp("-", UnOp("-", x)))
x == doubleNegX // true
```

Imagine doing this with the visitor pattern (which we'll learn later).

`match` Expressions with Case Classes

```scala
def simplify(expr: Expr): Expr = expr match {
  case UnOp("-", UnOp("-", e)) => e // Double negation
  case BinOp("+", e, Number(0)) => e // Adding zero
  case BinOp("*", e, Number(1)) => e // Multiplying by one
  case _ => expr
}
```

- General form: *selector* `match` { *alternatives* }
- Alternatives: pattern => expression
- Selector is matched against each pattern sequentially until a match is found.
- Expression corresponding to matched pattern is evaluated and returned as value of the match expression
- No fall through to subsequent alternatives
- `_` is used as a default if no other patterns match

Kinds of Patterns

The next few slides will summarize the kinds of patterns that may appear in alternatives:

- Wildcard patterns
- Constant patterns
- Variable patterns
- Constructor patterns
- Sequence patterns
- Tuple patterns
- Typed patterns
- Variable binding

Wildcard Patterns

Wildcard pattern matches any object. Can be used for defaults:

```
expr match {
  case BinOp(op, left, right) => println(expr + " is a BinOp")
  case _ => // handle the default case
}
```

... or to ignore parts of patterns:

```
expr match {
  case BinOp(_, _, _) => println(expr + " is a BinOp")
  case _ => println("It's something else")
}
```

Constant Patterns

Constant patterns match their values:

```
def describe(x: Any) = x match {
  case 5 => "five"
  case true => "truth"
  case "hello" => "hi!"
  case Nil => "the empty list"
  case _ => "something else"
}
describe(5)          // five
describe(true)       // truth
describe("hello")    // hi!
describe(Nil)        // the empty list
describe(List(1,2,3)) // something else
```

Georgia Tech

Variable Patterns

Variable patterns match any object, like a widlcard, but bind the variable name to the object:

```
expr match {
  case 0 => "zero"
  case somethingElse => "not zero: " + somethingElse
}
```

Some constants look like variables but aren't.

```
import math.{E, Pi}

val res = E match {
  case Pi => "strange math? Pi = " + Pi
  case _ => "OK"
}
res == "OK" // true
```

Because `Pi` in the first pattern is a constant, not a variable.

Variable-Constant Disambiguation

Simple names starting with lowercase letters treated as variable patterns. Here pi is a variable pattern, not a constant:

```
val pi = math.Pi
val strange = E match {
  case pi => "E is " + pi
}
strange.substring(0,10) == "E is 2.718" // true
```

In fact, with a variable pattern like this you can't even add a default alternative because the variable pattern is exhaustive:

```
val strange = E match {
  case pi => "E is " + pi
  case _ => "OK"
}
```

would result in an "unreachable code" error.

Constructor Patterns

```scala
expr match {
  case BinOp("+", e, Number(0)) => println("a deep match")
  case _ =>
}
```

- A constructor pattern consists of a name and patterns within parentheses
- Name should be the name of a case class, the names in parentheses can be any kind of pattern (including other case classes!)
- Nesting permits powerful deep matches

Sequence Patterns

Match a list of length three with 0 as first element and return second element as the value of the match expression:

```
val xs = List(0,2,4)

val two = xs match {
  case List(0, e, _) => e
  case _ => null
}
two == 2 // true
```

Match a list of any length greater than 1 with 0 as first element and return second element as the value of the match expression:

```
expr match {
  case List(0, e, _*) => e
  case _ => null
}
```

Tuple Patterns

```
def tupleDemo(expr: Any) = expr match {
  case (a, b, c) => "matched " + a + b + c
  case _ =>
}
val threeTuple = tupleDemo(("ein ", 3, "-Tupel"))
val nichts = tupleDemo((2, "-Tupel"))
```

Typed Patterns

```scala
def generalSize(x: Any) = x match {
  case s: String => s.length
  case m: Map[_, _] => m.size
  case _ => -1
}
generalSize("abc")                    // 3
generalSize(Map(1 -> 'a', 2 -> 'b')) // 2
generalSize(math.Pi)                  // -1
```

Patterns can't inspect type arguments because they are erased. So `Map[_,_]` just means any `Map`, but you still need the `Map[_,_]` because `Map` has type parameters (no "raw" collections in Scala).

Arrays are different ...

Matching Array Types

```
def arrayTest(a: Any) = a match {
  case ints: Array[Int] => "ints"
  case strs: Array[String] => "strs"
  case _ =>
}
arrayTest(Array(1,2,3))    // ints
arrayTest(Array("a","b","c")) // strs
```

Note that the parameter type of `arrayTest` must be `Any`, not `Array[Any]` becuase arrays are invariant. We'll learn what that means in a few lectures.

Variable Binding

In addition to simple variable binding, you can bind a variable to a matched nested pattern using variable @ before the pattern:

```
expr match {
  case UnOp("abs", e @ UnOp("abs", _)) => e
  case _ =>
}
```

The code above matches double applications of the abs operator and simplifies them by returning an equivalent single aplication (which is just the inner pattern).

Pattern Guards

What if we wanted to convert an addition of a number to itself to a multiplication of the number by two? Can't do it with only syntactic pattern matching:

```
def simplifyAdd(e: Expr) = e match {
  case BinOp("+", x, x) => BinOp("*", x, Number(2))
  case _ => e
}
```

- Above fails because `x` is defined twice.

Pattern guards allow us to add simple semantic checks to patterns:

```
def simplifyAdd(e: Expr) = e match {
  case BinOp("+", x, y) if x == y => BinOp("*", x, Number(2))
  case _ => e
}
```

Georgia Tech

Match Errors

Given our current `Expr` classes, this code produces a `scala.MatchError` at run-time:

```scala
def describe(e: Expr): String = e match {
  case Number(_) => "a number"
  case Var(_)   => "a variable"
}
describe(BinOp("+", Var("x"), Number(1)))
```

We can turn that into a compile-time warning by *sealing* our `Expr` classes.

Sealed Case Classes

Sealed case classes must all be defined in the same source file. Simply add `sealed` in front of superclass:

```scala
sealed abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
```

Now simply defining this function:

```scala
def describe(e: Expr): String = e match {
  case Number(_) => "a number"
  case Var(_)   => "a variable"
}
```

results in a `Warning: match may not be exhaustive`. If you know for sure that `describe` will only ever be called with `Number` or `Var`, you can shut compiler up with:

```scala
def describe(e: Expr): String = (e: @unchecked) match { ... }
```

The `Option` Type

Takes the form Option[T] and has two values:

- Some(x) where x is a value of type T, or
- None, an object which represents a missing value.

Typically used with pattern matching. The get method on Map returns an Option[T]:

```
scala> val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo")
scala> def show(x: Option[String]) = x match {
  case Some(s) => s
  case None => "?"
}
scala> show(capitals get "Japan")
res25: String = Tokyo
scala> show(capitals get "North Pole")
res27: String = ?
```

Better than returning `null`. For example, Java's collections, you have to remember which methods may return `null`'s, where in Scala this is made explicit and checked by the compiler.

Destructuring Binds

Similar to "tuple unpacking assignment" in Python:

```
scala> val (number, string) = (123, "abc")
number: Int = 123
string: String = abc
```

But more general:

```
scala> val exp = new BinOp("*", Number(5), Number(1))
exp: BinOp = BinOp(*,Number(5.0),Number(1.0))

scala> val BinOp(op, left, right) = exp
op: String = *
left: Expr = Number(5.0)
right: Expr = Number(1.0)
```

Patterns in `for` Expressions

Can use a destructuring bind in a `for` expression:

```
scala> for ((country, city) <- capitals)
         println(s"The capital of $country is $city")
The capital of France is Paris
The capital of Japan is Tokyo
```

Constructor patterns provide simple filtering:

```
scala> val results = List(Some("apple"), None, Some("orange"))
results: List[Option[String]] = List(Some(apple), None, Some(orange))

scala> for (Some(fruit) <- results) println(fruit)
apple
orange
```

Imagine writing that loop with explicit `null` checks.

Georgia Tech

Defining Functions with Case Sequences

A sequence of cases can be used anywhere a function literal can be used because a case sequence is a special kind of function literal.

- Each case is an entry point with its own list of parameters specified by the pattern.
- The body of each entry point is the right-hand side of the case.

```scala
val withDefault: Option[Int] => Int = {
    case Some(x) => x
    case None => 0
}
```

`withDefault` is a `val` of type `Option[Int] => Int` – a function type – and its value is a sequence of cases. This is a *total function* because an `Option` is a sealed abstract class with only `Some` or a `None` as concrete subclasses.

Partial Functions

A function is *total* if is defined for every element of its domain. A partial function can be defined with case sequences:

```
val second: List[Int] => Int = {
  case x :: y :: _ => y
}
```

- is defined only for `List`s with length 2 or greater.

Note that the static type of `second` is total – its partialness manifests only at runtime. You can use a static type annotation that tells the compiler that the function is partial, which allows you to test whether the function is defined for particular elements of the its domain:

```
val second: PartialFunction[List[Int],Int] = {
  case x :: y :: _ => y
}
scala> second.isDefinedAt(List(5,6,7))
res30: Boolean = true

scala> second.isDefinedAt(List())
res31: Boolean = false
```

Georgia Tech

Uses of Partial Functions

The Akka actors library uses partial functions to define the messages that an actor will handle:

```scala
var sum = 0

def receive = {
  case Data(byte) =>
    sum += byte

  case GetChecksum(requester) =>
    val checksum = ~(sum & 0xFF) + 1
    requester ! checksum
}
```

We'll learn actors later.

Conclusion

Case classes and pattern matching are frequently used in Scala

- Case classes give you convenience (parametric fields, `equals`, `hashCode`, `copy`) "for free"
- But case classes are most powerful when used together with pattern matching
- Pattern matching is also useful for destructuring binds