Imperative Programming 3

Polymorphism

Peter Jeavons

Trinity Term 2019

Recall: Polymorphism

- Literally means “many shapes”

- The idea is that some constructs in a programming language can process objects of different data types in appropriate ways.

Polymorphism

- For example, the same operator or method can be explicitly defined for several different argument datatypes

  (“overloading” or “ad hoc polymorphism”)

- Code can be written to *inherit* an interface (or implementation) from other code so that it can be used interchangeably

  (“subtyping” or “inclusion polymorphism”)

Polymorphism

- Some code can be written *generically* so that it can handle argument values *identically* without depending on their type

  (“parametric polymorphism” or “generics”)

C. Strachey, *Fundamental concepts in programming languages*. Lecture notes for International Summer School in Computer Programming, Copenhagen, August 1967

Love your compile-time errors

- *Compile-time* errors can be detected early on and fixed (relatively) easily

- *Run-time* errors are much harder to track down and fix

Generics add stability to your code by making more errors *detectable at compile time*

Example: Cons-Lists

Immutable linked list

- constructed from two building blocks:

  `Nil` the empty list

  `Cons` a cell containing an element and the rest of the list

A list is either

- an empty list: `new Nil`
- a list consisting of a head element `x` and a tail list `xs`: `new Cons(x, xs)`

Defining a Cons-List

```
trait ListOfInt {
  def isEmpty: Boolean
  def head: Int
  def tail: ListOfInt
}
```

```
class Nil extends ListOfInt {
 def isEmpty = true
 def head = throw new NoSuchElementException(“Nil.head”)
 def tail = throw new NoSuchElementException(“Nil.tail”)
}
```

```
class Cons(val head:Int, val tail:ListOfInt) extends ListOfInt {
  def isEmpty = false
}
```

List of What?

- It seems too narrow to define only lists with `Int` elements
  - We would need another class hierarchy for list of `Double`, list of `String` and so on, one for each possible element type

- Copy-paste problem: code duplication, error propagation
- But using a list of `Any` objects creates a type-safety problem

```
val l: ListOfAny = ...
val h: Any = l.head
```

  - do not know at compile time what `h` might be – no type checking
  - need to downcast => common cause of errors

Fails *at runtime* with a `ClassCastException`

```
val l: ListOfAny = ListOfAny(2, “ssk”)
val s: String = l.head.asInstanceOf[String]
```

Defining a Cons-List

```
trait ListOfInt {
  def isEmpty: Boolean
  def head: Int
  def tail: ListOfInt
}
```

```
class Nil extends ListOfInt {
 def isEmpty = true
 def head = throw new NoSuchElementException(“Nil.head”)
 def tail = throw new NoSuchElementException(“Nil.tail”)
}
```

```
class Cons(val head:Int, val tail:ListOfInt) extends ListOfInt {
  def isEmpty = false
}
```

Defining a Generic Cons-List

```
trait List[T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}
```

Type parameters are written in square brackets [T1, T2]

(Java uses angle brackets <T1, T2, ...>)

```
class Nil[T] extends List[T] {
 def isEmpty = true
 def head = throw new NoSuchElementException(“Nil.head”)
 def tail = throw new NoSuchElementException(“Nil.tail”)
}
```

```
class Cons[T](val head:T, val tail:List[T]) extends List[T] {
  def isEmpty = false
}
```

Generic Functions

- Like classes, *functions* can also have type parameters
  - E.g.: here is a function creating a list with a single element

```
def singleton[T](elem: T) = new Cons[T](elem, new Nil[T])
```

```
singleton[Int](1)
singleton[Boolean](true)
```

- Type inference:
  - In fact, the Scala compiler can usually deduce the correct type parameters from the value arguments of a function call
  - So, in most cases, type parameters can be left out in function calls

```
singleton(1)
singleton(true)
```

Types and Evaluation

- Type parameters do not affect evaluation in Scala
- We can assume that all type parameters and type arguments are removed before evaluating the program
- This is also called type erasure
- Languages that use type erasure include Java, Scala, Haskell, ML, Ocaml
- Some other languages keep the type parameters around at run time, these include C++, C#, F#

Type Erasure

```
class Container[T](val obj: T)
```

Compiles into

```
class Container(val obj: Any)
```

T is replaced with Any

OK: String can be assigned to Any

```
val cs = new Container[String](“foo”)
val s: String = cs.obj
```

```
val c = new Container(“foo”)
val s = cs.obj.asInstanceOf[String]
```

Compiler inserts a downcast to the appropriate type

Type erasure:

- … compiler checks typing (at compile time)
- … compiler erases the type arguments
- … compiler adds safe downcasts where needed

Aside: C++ Generics

- C++ implements generics differently: templates

```
Container<String>
```

Compiles into

```
public class Container_String {
   protected String object;

   public Container(String o) {
      object = o;
   }
   public String get() {
      return object;
   }
}
```

```
public class Container<E> {
   protected E object;

   public Container(E o) {
      object = o;
   }
   public E get() {
      return object;
   }
}
```

```
Container<Integer>
```

Compiles into

```
public class Container_Integer {
   protected Integer object;

   public Container(Integer o) {
      object = o;
   }
   public Integer get() {
      return object;
   }
}
```

Aside: Templates vs. Type Erasure

- Templates are a kind of macros
  - class is *recompiled* for each concrete type parameter
  - no problem with object creation
    - type is known at runtime
- Problems with templates
  - it is not possible to compile the class alone
    - cannot check typing without knowing the type arguments
    - a class can work for some arguments, but not for others
  - bloated compiled code

Assumptions on the Type Parameter

- Consider a `sort` method which creates a new list with all elements sorted

```
def sort[T](list: List[T]) = {
  // at some point we need to compare two list elements
  ...
}
```

- `T` only known to have methods inherited from `Any`…
  - cannot know in advance what `T` will be substituted with
  - ... but we need objects of type `T` that are comparable

- Solution 1: One could change the signature of `sort`

```
def sort[T](list: List[T], lt: (T,T) => Boolean) = { ... }
```

Comparator: returns `true` if `l < r`

Type Bounds

- Solution 2: Require that `T` is a type that implements `Ordered[T]`

```
trait Ordered[T] {
  def compare(a: T): Boolean
}
```

```
def sort[T <: Ordered[T]](list: List[T]) = { ... }
```

  - "`<: Ordered[T]`" is an upper bound of the type parameter `T`
  - It means that `T` can be instantiated only to types that conform to `Ordered[T]`

Alternatively, `S <% T` means `S` *can be seen as* a subtype of `T`. This is called a "view bound"

- Generally, the notation
  - `S <: T` means: `S` is a *subtype* of `T`, and
  - `S >: T` means: `S` is a *supertype* of `T`, or `T` is a *subtype* of `S`

Covariance

Does `A <: B` imply `List[A] <: List[B]` ?

- Intuitively, this makes sense: a list of `A` objects is a special case of a list of `B` objects

The Liskov Substitution Principle

If `A <: B`, then everything one can do with a value of type `B`, one should also be able to do with a value of type `A`.

Covariance

Does `A <: B` imply `List[A] <: List[B]` ?

- Intuitively, this makes sense: a list of `A` objects is a special case of a list of `B` objects

The Liskov Substitution Principle

If `A <: B`, then everything one can do with a value of type `B`, one should also be able to do with a value of type `A`.

- We call types for which this relationship holds covariant
  - their subtyping relationship varies with the type parameter
- In Scala, lists are covariant but arrays are invariant

See Chapter 19 of “Programming in Scala” – especially 19.3-19.6

Scala Collection Framework

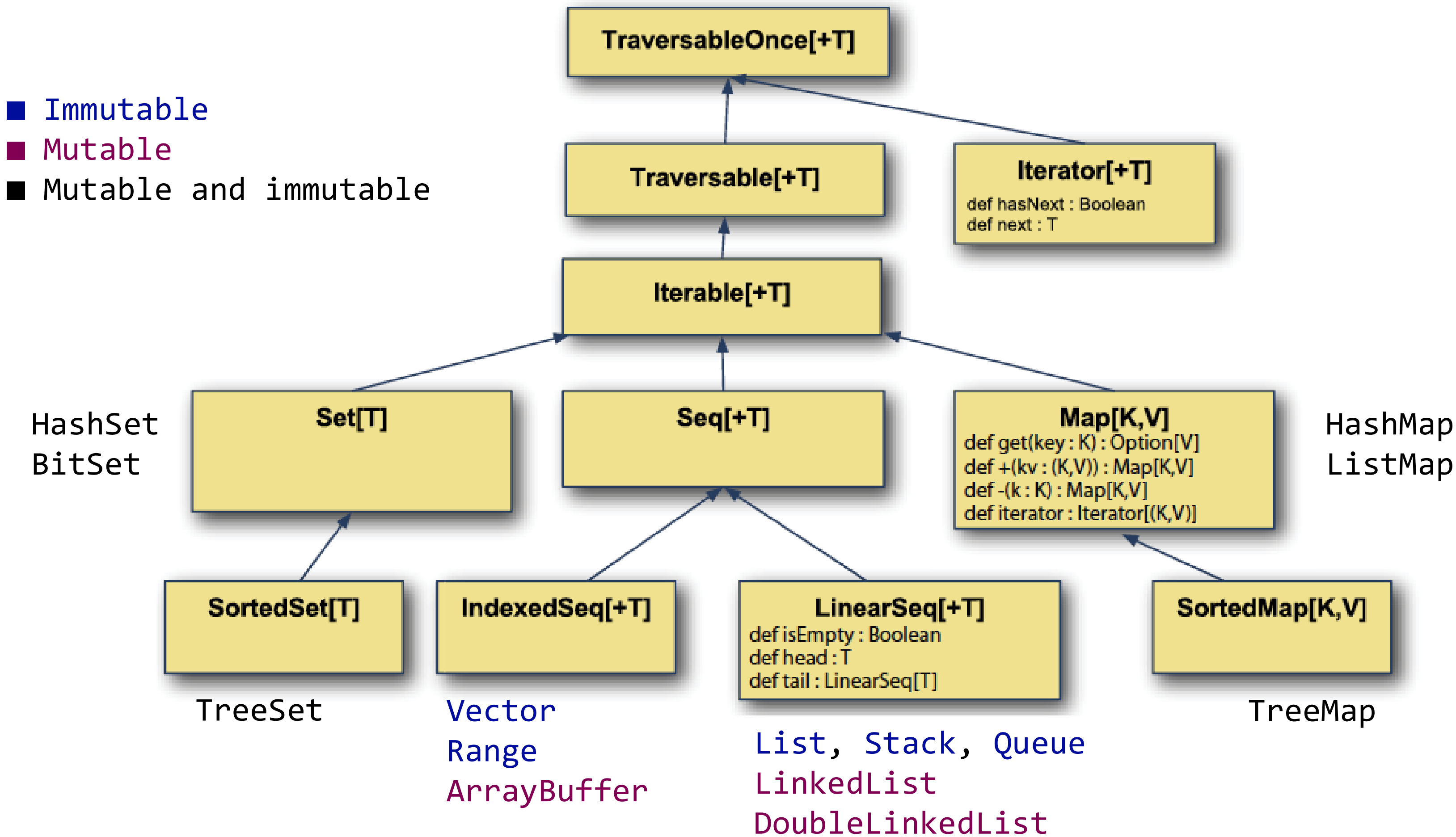
Scala Collection Framework

- Easy to use: 20-50 methods solve most collection problems
- Concise: functional-style syntax
- Safe: majority of programmer errors manifest as compile-time errors
- Fast: hand-tuned data structures and operations
- Universal: collections provide the same operations on any type where it makes sense to do so

Mutable and Immutable Collections

- Mutable collections can be updated or extended in place
  - `scala.collection.mutable` package
- Immutable collections are never changed
  - `scala.collection.immutable` package
  - additions, removals, and updates always return a new collection and leave the old collection unchanged
  - no interference between iterators and collection updates
- By default, Scala collections are immutable
  - `Set` (without any prefix) refers to `collection.immutable.Set`
  - For mutable versions, write explicitly `collection.mutable.Set`

Scala Collection Hierarchy



More info: docs.scala-lang.org/overviews/collections/introduction.html

Collections API

- Uniform syntax

```
Traversable(1, 2, 3)
Iterable(“x”, “y”, “z”)
Map(“x” -> 24, “y” -> 25, “z” -> 26)
Set(Color.red, Color.green, Color.blue)
```

- Trait `Traversable` has only one abstract operation

```
def foreach[U](f: Elem => U): Unit
```

and implements the behavior common to all collections

Map operations	`map, flatMap, collect`
Conversions	`toArray, toList, toSeq, toSet, toMap`
Size info	`isEmpty, nonEmpty, size, hasDefiniteSize`
Element retrieval	`head, headOption, last, lastOption`
Sub-collections	`tail, take, drop, takeWhile, dropWhile, filter`
Element conditions	`forall, exists, count`
Folds	`foldLeft, foldRight, reduceLeft, reduceRight`

Collections API

- Uniform return type principle: collections override the `Traversable` methods to change their result types wherever this makes sense
  - e.g., the `map` method in `Traversable` returns another `Traversable`, but calling `map` on a `List` yields a `List`
- Trait `Iterable` implements `foreach` in terms of an abstract method `iterator` (remember the Iterator pattern?)

```
def foreach[U](f: Elem => U): Unit = {
  val it = iterator
  while (it.hasNext) f(it.next())
}
```

Lists

- `List(x1, …, xn)` has `x1, …, xn` as elements

```
val fruit  = List(”apple”, ”banana”, ”pear”)  // List[String]
val nums   = 3 :: 5 :: 6 :: Nil               // List[Int]
val empty  = List()                           // List[Nothing]
val foo    = List(“bar”, 4, List(‘c’))        // List[Any]
```

- Like arrays, lists are homogeneous: all elements share the same static type (but their dynamic types can be different)

- Two important differences between lists and arrays
  - Lists are immutable – the elements of a list cannot be changed
  - Lists are recursive, while arrays are flat

List Patterns

- It is also possible to decompose lists with pattern matching

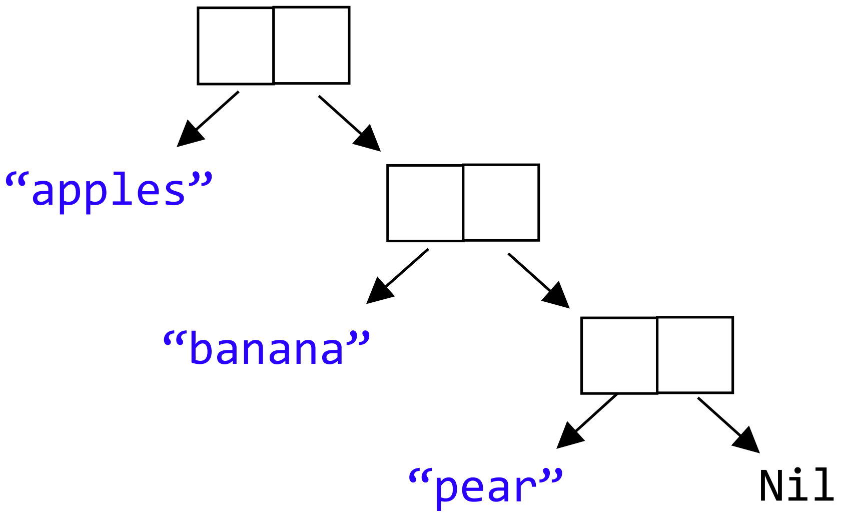
`Nil`	The `Nil` constant
`p :: ps`	A pattern that matches a list with a head matching `p` and a tail matching `ps`
`List(p1, …, pn)`	Same as `p1 :: … :: pn :: Nil`

- Example

`1 :: 2 :: xs`	Lists that start with 1 and then 2
`x :: Nil`	Lists of length 1
`List(x)`	Same as `x :: Nil`
`List()`	Same as `Nil`
`List(2 :: xs)`	A list that contains as only element another list that starts with 2

Lists vs. Arrays

- Lists are *linear* data structures



`head`
`tail`
`isEmpty`

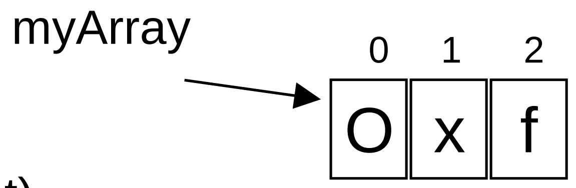
fast operations

All other operations on lists can be expressed using these three ops

```
def printList(x: List[Int]) = {
   for (i <- 0 until x.length) println(x(i))
}
```

indexing is O(n), loop is $O(n^2)$

- Arrays have fixed length and occupy sequential locations in memory
  - *O(1)* random access (e.g., getting the 5th element)



Arrays and Strings in Scala

- Arrays and Strings support the same operations as `Seq`
- However, they cannot be subclasses of `Seq` because they actually come directly from Java
- The Scala compiler *implicitly converts them* to sequences where needed

```
val xs: Array[Int] = Array(1,2,3)
xs map (x => 2 * x)

val ys: String = ”Hello World”
ys filter (_.isUpper)
```

Ranges

- A range represents a sequence of evenly spaced integers
- Three operators:
  - `to` (inclusive), `until` (exclusive), `by` (to determine step value)

```
val r: Range = 1 until 5                // 1, 2, 3, 4
val r: Range = 1 to 5                   // 1, 2, 3, 4, 5
1 to 10 by 3                            // 1, 4, 7, 10
6 to 1 by -2                            // 6, 4, 2
```

- Ranges represented as single objects with three fields
  - lower bound, upper bound, step value

Sets

- Sets are another abstraction in the Scala collections

```
val fruit = Set(”apple”, ”banana”, ”pear”)
val s = (1 to 6).toSet
```

- Most operations on sequences are also available on sets
- The principal differences between sets and sequences:
  - sets are unordered; elements have no predefined order in which they appear in the set
  - sets do not have duplicate elements

```
s map (_ % 3)            // 2, 0, 1
```

  - the fundamental operation on sets is contains

```
s contains 3             // true
```

Maps

- A map of type `Map[Key, Value]` is a data structure that associates keys of type `Key` with values of type `Value`

```
val capitalOfCountry= Map("UK" -> "London", "US" -> "Washington")
```

The syntax `key -> value` is just another way to write the pair `(key, value)`

- `Map[Key, Value]` extends `Iterable[(Key, Value)]`
  - Maps support the same collection operations as other iterables do

```
val countryOfCapital = capitalOfCountry map {
  case (x, y) => (y, x)
}      // Map("London" -> "UK", "Washington" -> "US")
```

Querying Maps

```
capitalOfCountry(“UK”)                    // “London”
capitalOfCountry(“Andorra”)
   // java.util.NoSuchElementException: key not found: Andorra
```

- Maps with default values

```
val cap1 = capitalOfCountry withDefaultValue “<unknown>”
cap1(“Andorra”)                             // “<unknown>”
```

- To query a map without knowing beforehand whether it contains a given key, you can use the `get` operation

```
capitalOfCountry get “UK”           // Some(“London”)
capitalOfCountry get “Andorra”      // None
```

The result of a `get` operation is an `Option` value...

The `Option` Type

```
trait Option[+A]
case class Some[+A](value: A) extends Option[A]
case object None extends Option[Nothing]
```

- Decomposing `Option`

```
def showCapital(country: String) =
  capitalOfCountry.get(country) match {
    case Some(capital) => capital
    case None => “missing data”
  }

showCapital(“UK”)              // “London”
showCapital(“Andorra”)         // “missing data”
```

- `Option`s support quite a few operations of the other collections (see Scala documenation)

Summary

- Generic types maximize code reuse and type safety
- Type erasure removes type information at compile time, which imposes certain limitations
  - Note that there are ways of retaining type information at runtime (e.g., see `ClassTag`, `TypeTag`, and context bounds in Scala)
- Scala provides immutable (e.g., lists) and mutable (e.g., arrays) collections and a powerful collection API

See also *Programming in Scala*: Chapters 19 & 24

- Next: GUI programming

[Optional] Arrays in Java

- For perspective, let's look at arrays in Java (and C#)
- Reminder:
  - An array of `T` elements is written `T[]` in Java
  - In Scala we use parameterized type syntax `Array[T]` to refer to the same type
- Arrays in Java are covariant, so one would have:

```
If  A <: B  then  A[] <: B[]
```

[Optional] Array Typing Problem in Java

- But covariant array typing causes problems
- To see why, consider the Java code below

```
A[] arrA = new A[] { new A() };
B[] arrB = arrA;
arrB[0] = new B();
A a = arrA[0];
```

Java throws an `ArrayStoreException`

- It looks like we assigned in the last line a B object to a variable of type A
- Scala arrays are invariant, so step 2 would fail to compile