Polymorphism

Nancy Hitschfeld
Matías Toro

Polymorphism, revisited

Polymorphism is crucial to write *extensible* and *generic* programs. It comes in many flavors:

- Subtype Polymorphism
- Ad-hoc Polymorphism
- Parametric Polymorphism

Subtype Polymorphism

Subtyping

- Type B is a subtype of type A if any context expecting an expression of type A may take an expression of type B without errors.
- Subtyping is about substitutability
- Corresponds directly to the containment relation on object

Subtype polymorphism allows a **single** term to have **many types**

Implicit Subtype Polymorphism

In dynamically-typed languages, subtype polymorphism is **implicit**

Python

```
def foo(o):
  o.m1()
  print(o.m2())
  return o.m3()
```

any object that understands
AT LEAST
m1,m2,m3 will do

“If it walks like a duck and quacks like a duck, then it’s a duck!”

Explicit Subtype Polymorphism

In most statically-typed languages, the subtype relation has to be **explicitly** declared.

If class A defines all methods of class B, A is not a subtype of B!!

```
trait Animal {
  def talk(): Unit
}
class Dog extends Animal {
  def talk(): Unit = println("Guau!")
}
class Cat extends Animal {
  def talk(): Unit = println("Miau!")
}
class Robot {
  def talk(): Unit = println("BMO!")
}
```

```
def present(a: Animal){
    a.talk();
}
```

Single dispatch!

```
present(new Cat());
present(new Dog());
present(new Robot());
```

Ad-hoc Polymorphism (overloading)

An object can define several methods with the **same name**, but **different parameters**

```
trait List {
  def add(o: Any): Unit

  def add(i: Int, o: Any): Unit
}

val l: List = ArrayList()

l.add(new Point(1, 2))
l.add(3, new Point(1, 2))
```

different arity

```
class Display {
  def paint(p: Point): Unit = {
    print("Point")
  }

  def paint(p: Triangle): Unit = {
    print("Triangle")
  }
}

val disp = new Display()

disp.paint(new Point(1, 2))
disp.paint(new Point(5, 3, 2))
```

different types

Ad-hoc Polymorphism (overloading)

In Java, overloading is resolved **STATICALLY**

```
class CollectionClassifier {
  def classify(s: Set): String = {
    "Set";
  }
  def classify(l: List): String = {
    "List";
  }
  def classify(c: Collection): String = {
    "Unknown Collection";
  }
}
val cl = new CollectionClassifier();
val cols: List[Collection] = { new HashSet()
                               new ArrayList()
                               new HashMap().values() }

for(c <- cols)
  println(c.classify(c))
```

c has declared type Collection

The actual (runtime) type of an object is not used!!

It usage may lead to bugs

Use overloading Judiciously

Beware of overloading

- Avoid “confusing” uses of overloading
- Not confusing:
  - Different arity
  - Types are “unrelated” (none can be seen as a subtype of the other)

Parametric Polymorphism: problem

Suppose we create a class to store an object of **any** type.

```
class Cell {
  var element: Option[Any] = None

  def getContent(): Option[Any] = element

  def setContent(o: Any): Unit = {
    element = Some(o)
  }
}
```

Parametric Polymorphism: problem

Suppose we create a class to store an object of **any** type.

```
class Cell {
  var element: Option[Any] = None

  def getContent(): Option[Any] = element

  def setContent(o: Any): Unit = {
    element = Some(o)
  }
}
```

```
val c: Cell = new Cell()
val p: Point = new Point(1,2)
c.setContent(p)
…
val a: Point = c.getContent().get
a.moveBy(10,10)
```

Error! The type system does not “know” that the content is a Point

Parametric Polymorphism: problem

Suppose we create a class to store an object of **any** type.

```
class Cell {
  var element: Option[Any] = None

  def getContent(): Option[Any] = element

  def setContent(o: Any): Unit = {
    element = Some(o)
  }
}
```

```
val c: Cell = new Cell()
val p: Point = new Point(1,2)
c.setContent(p)
…
val a: Point = c.getContent().get.asInstanceOf[Point]

a.moveBy(10,10)
```

Fix: introduce a **cast**
(TRUST ME!)

Parametric Polymorphism: problem

Trust you????

```
def someObscureCall(cell: Cell)
{
    cell.setContent(2);
}
```

```
val c: Cell = new Cell()
val p: Point = new Point(1,2)
c.setContent(p)
…
someObscureCall(c)
…
val a: Point = c.getContent().get.asInstanceOf[Point]

a.moveBy(10,10)
```

Exception in thread "main"
java.lang.ClassCastException:
java.lang.Integer cannot be cast to shapes.Point
at Test.main(Test.java:6)

Problem: losing type information

When we put a value in a a container like Cell, we lose information about its type.

The only way to regain information is by deferring to runtime: fragile!

```
class Cell {
  var element: Option[Any] = None

  def getContent(): Option[Any] = element

  def setContent(o: Any): Unit = {
    element = Some(o)
  }
}
```

general-purpose because it uses Any (relies on subtype polymorphism)

Type Parameters

Better solution: **parametric polymorphism**

Use **type parameters**

In Java, this is known as “Generics” (Java 5)

introduce type
parameter T

```
class Cell {
  var element: Option[Any] = None

  def getContent(): Option[Any] = element

  def setContent(o: Any): Unit = {
    element = Some(o)
  }
}
```

```
class Cell[T] {
  var element: Option[T] = None

  def getContent(): Option[T] = element

  def setContent(o: T): Unit = {
    element = Some(o)
  }
}
```

Generics

```
val c: Cell[Point] = new Cell[Point]()
val p: Point = new Point(1, 2)
c.setContent(p)
…
someObscureCall(c);
…
val a: Point = c.getContent().get
a.moveBy(10, 10)
```

Cannot do damage anymore!
(Well, shouldn't…)

OK! c has type Cell[Point] so getContent() returns a Option[Point] and getContent().get returns a Point

Generics

scala.collection.immutable

List

Companion object List

```
sealed abstract class List[+A] extends AbstractSeq[A] with LinearSeq[A] with LinearSeqOps[A, List,
List[A]] with StrictOptimizedLinearSeqOps[A, List, List[A]] with StrictOptimizedSeqOps[A, List,
List[A]] with IterableFactoryDefaults[A, List] with DefaultSerializable
```

scala.collection.immutable

Seq

Companion object Seq

```
trait Seq[+A] extends Iterable[A] with collection.Seq[A] with SeqOps[A, Seq, Seq[A]] with
IterableFactoryDefaults[A, Seq]
```

Generics

My first design pattern:
The curiously recurring template pattern

```
class Foo(val n:Int) extends Ordered[Foo] {
  def compare(that: Foo) = this.n - that.n
}
```

```
val l = Array(
          new Foo(5),
          new Foo(1),
          new Foo(3))

println(l(0).n)
scala.util.Sorting.quickSort(l)
println(l(0).n)
```

Generics

A class or interface can have more than one type parameter

scala.collection

Map

```
trait Map[K, +V] extends Iterable[(K, V)]
```

Abstract Value Members

```
abstract def get(key: K): Option[V]
```
Optionally returns the value associated with a key.

```
abstract def iterator: Iterator[(K, V)]
```
Creates a new iterator over all key/value pairs of this map

Concrete Value Members

```
def +(kvs: (K, V)*): Map[K, V]
```
[use case] Adds key/value pairs to this map, returning a new map.

```
abstract def +(kv: (K, V)): Map[K, V]
```
[use case] Adds a key/value pair to this map, returning a new map.

```
def ++(xs: Traversable[(K, V)]): Map[K, V]
```
[use case] Adds all key/value pairs in a traversable collection to this map, returning a new map.

```
def ++[B >: (K, V), That](that: GenTraversableOnce[B])(implicit bf: CanBuildFrom[Map[K, V], B, That]): That
```
Returns a new traversable collection containing the elements from the left hand operand followed by the elements from the right hand operand.

```
def ++:[B >: (K, V), That](that: Traversable[B])(implicit bf: CanBuildFrom[Map[K, V], B, That]): That
```
As with ++, returns a new collection containing the elements from the left operand followed by the elements from the right operand.

```
def ++:[B](that: TraversableOnce[B]): Map[B]
```
[use case] As with ++, returns a new collection containing the elements from the left operand followed by the elements from the right operand.

```
def -(elem1: K, elem2: K, elems: K*): Map[K, V]
```
Creates a new collection from this collection with some elements removed.

```
abstract def -(key: K): Map[K, V]
```
[use case] Removes a key from this map, returning a new map.

Generics

Generic **methods** and **constructors**

```
def process[T](o: T): T = …

process[Point](new Point(1, 2))
```

Exercise 1:

Implement a generic class to represent tuples of values. Both values do not necessarily have to be of the same type. Include methods for retrieving the left and right elements, and provide a usage example.

Exercise 1:

Implement a generic class to represent tuples of values. Both values do not necessarily have to be of the same type. Include methods for retrieving the left and right elements, and provide a usage example.

```
class Tuple[A,B](left: A, right: B){
  def _1: A = left
  def _2: B = right
}
```

```
val t = new Tuple[Int, String](1, "hola")
t._1 + 1
t._2.substring(1)
```

Exercise 2:

Reimplement Options. An Option can be Some value or None. Provide methods to retrieve the underlying value, check if the option is empty, and obtain the underlying value or fail with a default value.

```
trait Option[T] {
  def get: T
  def isEmpty: Boolean
  def getOrElse(default: T): T
}
```

Exercise 2, try #1:

```
trait Option[T] {
  def get: T
  def isEmpty: Boolean
  def getOrElse(default: T): T
}

class Some[T](t: T) extends Option[T] {
  def get: T = t
  def isEmpty: Boolean = false
  def getOrElse(default: T): T = t
}

class None extends Option[Nothing] {
  def get: Nothing = throw new NoSuchElementException("None.get")
  def isEmpty: Boolean = true
  def getOrElse(default: T): T = default
}
```

Do not compile!! What is T there?

Bounds to the rescue

Upper and Lower bounds

```
def foo[T<:A](x: T){…}
def foo[T>:A](x: T){…}
```

```
trait Option[T]{
 …
 def getOrElse[S>:T](default: S): S
}
```

Exercise 2, try #2:

```scala
trait Option[T]{
  def get: T
  def isEmpty: Boolean
  def getOrElse[S>:T](default: S): S
}

class Some[T](t: T) extends Option[T]{
  def get: T = t
  def isEmpty: Boolean = false
  def getOrElse[S>:T](default: S): S = t
}

class None extends Option[Nothing] {
  def get: Nothing = throw new NoSuchElementException("None.get")
  def isEmpty: Boolean = true
  def getOrElse[S >: Nothing](default: S): S = default
}
```

Exercise 3:

Implement a generic Tree library. A tree is parameterized by a comparable trait and must implement methods to find an element (returning a boolean) and compute the maximum element of the tree using a 'max' method. A Tree can be either a Node or a Leaf

Exercise 3:

Implement a generic Tree library. A tree is parameterized by a comparable trait and must implement methods to find an element (returning a boolean) and compute the maximum element of the tree using a 'max' method. A Tree can be either a Node or a Leaf

```
trait Comparable[T] {
  def compareTo(o: T): Int
}
trait Tree[T <: Comparable[T]] {
  def find(x: T): Boolean
  def max: T
}
```

Exercise 3:

```scala
trait Comparable[T] {
  def compareTo(o: T): Int
}
trait Tree[T <: Comparable[T]] {
  def find(x: T): Boolean
  def max: T
}
```

```scala
class Node[T <: Comparable[T]](
    val value: T,
    val left: Tree[T],
    val right: Tree[T]
) extends Tree[T] {
  def max: T = {
    val innerMax =
      if (left.max.compareTo(right.max) > 0) left.max
      else right.max
    if (value.compareTo(innerMax) > 0) value else innerMax
  }
  def find(x: T): Boolean =
    (value.compareTo(x) == 0) || left.find(x) || right.find(x)
}

class Leaf[T <: Comparable[T]](val value: T) extends Tree[T] {
  def max: T = value
  def find(x: T): Boolean = (value.compareTo(x) == 0)
}
```

Exercise 3:

Write a use example:

```
class Person(val name: String, val age: Int) extends Comparable[Person] {
  override def compareTo(o: Person): Int = age.compareTo(o.age)
  override def toString() = s"Person($name, $age)"}

val tree: Tree[Person] = new Node[Person](
  new Person("Juan", 20),
  new Leaf[Person](new Person("Pedro", 30)),
  new Leaf[Person](new Person("Maria", 40))
)

println(tree.max)
println(tree.find(new Person("Pedro", 30)))
println(tree.find(new Person("Pedro", 31)))
```

```
Person(Maria, 40)
true
false
```

License

Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

- Share: copy and redistribute the material in any medium or format
- Adapt: remix, transform, and build upon the material for any purpose, even commercially

The licensor cannot revoke these freedoms as long as you follow the license terms

Attribution: you must give appropriate credit

ShareAlike: if you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original

Complete license: https://creativecommons.org/licenses/by-sa/4.0/