Imperative Programming 2:
Polymorphism I

Hongseok Yang
University of Oxford

Polymorphism

- A language feature that enables a program expression to have multiple types.
- It enables succinct code.

Polymorphism is used everywhere

```
scala> println(List(1, 2, 3) map (4 + _))
List(5, 6, 7)

scala> println(List(0.1, 0.2, 0.3) map (4 + _))
List(4.1, 4.2, 4.3)
```

Polymorphism is used everywhere

```
scala> println(List(1, 2, 3) map (4 + _))
List(5, 6, 7)

scala> println(List(0.1, 0.2, 0.3) map (4 + _))
List(4.1, 4.2, 4.3)
```

`println` expects an argument of type `Any`.
But the dynamic type of arg. is `List[Double]`.

Polymorphism is used everywhere

Int-type argument

```
scala> println(List(1, 2, 3) map (4 + _))
List(5, 6, 7)

scala> println(List(0.1, 0.2, 0.3) map (4 + _))
List(4.1, 4.2, 4.3)
```

Double-type argument

Polymorphism is used everywhere

`map` is used for a list of integers

```
scala> println(List(1, 2, 3) map (4 + _))
List(5, 6, 7)

scala> println(List(0.1, 0.2, 0.3) map (4 + _))
List(4.1, 4.2, 4.3)
```

`map` is used for a list of doubles

Learning outcome

- Can write polymorphic methods in Scala, using generics, subtyping and overloading.
- Can explain how overloading interacts with subtyping.
- Can use double dispatch to overcome the limitation of Scala's overloading mechanism.

Defining a polymorphic method

- A method is polymorphic if it works for arguments of different types.
- Examples: `map`, `println`, `+`.

Defining a polymorphic method

- A method is polymorphic if it works for arguments of different types.
- Examples: `map`, `println`, `+`.

1. Define one method that does not depend on the specifics of arguments' types.

Defining a polymorphic method

- A method is polymorphic if it works for arguments of different types.
- Examples: `map`, `println`, `+`.

1. Define one method that does not depend on the specifics of arguments' types.
2. Define multiple methods with the same name, which take arguments with different types.

One def'n for many types I: Generic method

- A method that takes type parameters.
- Typical syntax:

```
def head[T](l : List[T]): T = ...

head[Int](List(1,2)),    head(List(1,2))
```

- The body of a generic method does not use any information about its actual type parameters.

One def'n for many types 1: Generic method

- A method that takes type parameters.
- Typical syntax:

```
def head[T](l : List[T]): T = ...

head[Int](List(1,2)),    head(List(1,2))
```

- The body of a generic method does not use any information about its actual type parameters.

[Q] Complete the definition of `head`

One def'n for many types 1: Generic method

- A method that takes type parameters.
- Typical syntax:

```
def head[T](l : List[T]): T = ...
```

```
head[Int](List(1,2)),    head(List(1,2))
```

- The body of a generic method does not use any information about its actual type parameters.

[Q] Complete the definition of head

```
def head[T](l: List[T]): T =
  l match {
    case x::_ => x
  }
```

One def'n for many types 2: Subtyping polymorphism

- Just a usual definition of a method.
- A method "`f(x:A):B`" can take an argument of any subtype of `A`.
- The body of `f` can use information about `A`, but not the dynamic type of its argument.

One def'n for many types 2: Subtyping polymorphism

- Just a usual definition of a method.
- A method "`f(x:A):B`" can take an argument of any subtype of `A`.
- The body of `f` can use information about `A`, but not the dynamic type of its argument.

[Q] Define:
`toStr2(x:Any): String`

```
scala>toStr2(List(1,2))
res0: String = List(1,2)List(1,2)

scala>toStr2("IP2")
res1: String = IP2IP2
```

One def'n for many types 2: Subtyping polymorphism

- Just a usual definition of a method.
- A method "`f(x:A):B`" can take an argument of any subtype of `A`.
- The body of `f` can use information about `A`, but not the dynamic type of its argument.

[Q] Define:
`toStr2(x:Any): String`

```
def toStr2(x: Any) =
  x.toString + x.toString
```

```
scala>toStr2(List(1,2))
res0: String = List(1,2)List(1,2)

scala>toStr2("IP2")
res1: String = IP2IP2
```

Multiple def'n for different types: Operator overloading

- Define multiple methods with the same name `f` in a class, but with different argument types.
- For each call to such `f`, the compiler picks a right definition based on the arguments' types.

Multiple def'n for different types: Operator overloading

- Define multiple methods with the same name `f` in a class, but with different argument types.
- For each call to such `f`, the compiler picks a right definition based on the arguments' types.

```
class Measure {
  def size(x: List[Int]) = x.length
  def size(x: Int) = math.log(x).ceil.toInt
  def size(x: Traversable[Any]) = 1
  def size(x: Any) = 0
}

val m = new Measure
println(m.size(4))
println(m.size(List(1,2,3,4)))
println(m.size(Set(2,3)))
```

Output:

```
2
4
1
```

Multiple def'n for different types: Operator overloading

- Define multiple methods with the same name `f` in a
- For ea
right d

Subtyping used.
The best fit is chosen:
`Set[Int] <: Traversable[Any] <: Any`

```
class Measure {
  def size(x: List[Int]) = x.length
  def size(x: Int) = math.log(x).ceil.toInt
  def size(x: Traversable[Any]) = 1
  def size(x: Any) = 0
}

val m = new Measure
println(m.size(4))
println(m.size(List(1,2,3,4)))
println(m.size(Set(2,3)))
```

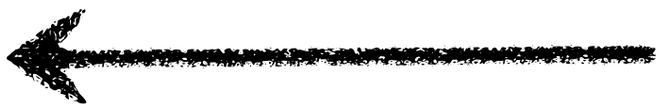
Output:

```
2
4
1
```

Three kinds of polymorphism

1. Generic methods.
2. Subtyping polymorphism.
3. Overloading.

One uniform definition.



Multiple ad-hoc definitions.

Three kinds of polymorphism

1. Generic methods. ← One uniform definition.
2. Subtyping polymorphism.
3. Overloading. ← Multiple ad-hoc definitions.

Nontrivial interaction between overloading and subtyping. Focus of the rest of the lecture.

```
class Flower
class Rose extends Flower

class Human {
  def identify(r: Rose) { println(“Rose“) }
  def identify(f: Flower) { println(“Flower“) }
}

val (r: Rose) = new Rose
val (f: Flower) = r
val h = new Human
h.identify(r)
h.identify(f)
```

[Q] What is the output?

1)	2)	3)
Flower	Rose	Rose
Flower	Rose	Flower

```
class Flower
class Rose extends Flower

class Human {
  def identify(r: Rose) { println("Rose") }
  def identify(f: Flower) { println("Flower") }
}

val (r: Rose) = new Rose
val (f: Flower) = r
val h = new Human
h.identify(r)
h.identify(f)
```

[Q] What is the output?

1)	2)	3)
Flower	Rose	Rose
Flower	Rose	Flower

- The static type of `f` is `Flower`, and its dynamic type is `Rose`.
- The static types of arguments are used when overloading is resolved.

```
class Flower
class Rose extends Flower

class Human {
  def identify(r: Rose) { println(“Rose”) }
  def identify(f: Flower) { println(“Flower”) }
}

class Mathematician extends Human {
  override def identify(r: Rose) { println(“Red plant”) }
  override def identify(f: Flower) { println(“Plant”) }
}

val (f: Flower) = new Rose
val m = new Mathematician
val (h: Human) = m
m.identify(f)
h.identify(f)
```

[Q] What is the output?

1)	2)	3)
Red plant	Plant	Plant
Rose	Flower	Plant

```
class Flower
class Rose extends Flower

class Human {
  def identify(r: Rose) { println(“Rose”) }
  def identify(f: Flower) { println(“Flower”) }
}

class Mathematician extends Human {
  override def identify(r: Rose) { println(“Red plant”) }
  override def identify(f: Flower) { println(“Plant”) }
}

val (f: Flower) = new Rose
val m = new Mathematician
val (h: Human) = m
m.identify(f)
h.identify(f)
```

[Q] What is the output?

1)	2)	3)
Red plant	Plant	Plant
Rose	Flower	Plant

- Dynamic dispatch.
- The dynamic type of a receiver object `h` is used.

Resolving a call to an overloaded method

```
obj.meth(arg)
```

- Suppose that the method `meth` is overloaded, and is defined in multiple classes.

- The dynamic type of `obj` and the static type of `arg` is used to choose a method to run.
- The same mechanism is used in Java.
- Can we make the choice of a method depend on the dynamic type of its argument?

Double dispatch

- An idiom for making the choice of a method depend on the dynamic type of its arguments.
- Key idea: Introduce one further call where arg & receiver are switched, i.e., `def f(a:A)=a.g(this)`.

```
class A
class B extends A

class X {
  def f(a:A){ println(“XA“) }
  def f(b:B){ println(“XB“) }
}
```

```
scala> val (a:A)=new B; val x=new X; x.f(a)
XA
```

Double dispatch

- An idiom for making the choice of a method depend on the dynamic type of its arguments.
- Key idea: Introduce one further call where arg & receiver are switched, i.e., `def f(a:A)=a.g(this)`.

```
class A
class B extends A

class X {
  def f(a:A){ println("XA") }
  def f(b:B){ println("XB") }
}
```

```
class A {
  def g(x:X){ println("XA") }
}
class B extends A {
  override def g(x:X){
     println("XB") }
}
class X {
  def f(a: A) = a.g(this)
  def f(b: B) = b.g(this)
}
```

```
scala> val (a:A)=new B; val x=new X; x.f(a)
XA
XB
```

Double dispatch

- An idiom for making the choice of a method depend on the dynamic type of its arguments.
- Key idea: Introduce one further call where arg & receiver are switched, i.e., `def f(a:A)=a.g(this)`.

```
class A
class B extends A

class X {
  def f(a:A){ println("XA") }
  def f(b:B){ println("XB") }
}
```

```
class A {
  def g(x:X){ println("XA") }
}
class B extends A {
  override def g(x:X){
    println("XB") }
}
class X {
  def f(a: A) = a.g(this)
  def f(b: B) = b.g(this)
}
```

```
scala> val (a:A)=new B; val x=new X; x.f(a)
XA
XB
```

```
class Flower
class Rose extends Flower

class Human {
  def identify(r: Rose) { println(“Rose”) }
  def identify(f: Flower) { println(“Flower”) }
}

class Mathematician extends Human {
  override def identify(r: Rose) { println(“Red plant”) }
  override def identify(f: Flower) { println(“Plant”) }
}

val (f: Flower) = new Rose
val (h: Human) = new Mathematician
h.identify(f)
```

The output of this program is:

```
Plant
```

[Q] Change the program using double dispatch so that the output becomes:

```
Red Plant
```

```
class Flower {
  def print(h: Human) {                          }
  def print(m: Mathematician) { println("Plant") }
}
class Rose extends Flower {
  override def print(h: Human) { println("Rose") }
  override def print(m: Mathematician) {
                                                 }
}
class Human {
  def identify(r: Rose) { r.print(this) }
  def identify(f: Flower) {                      }
}
class Mathematician extends Human {
  override def identify(r: Rose) {                }
  override def identify(f: Flower) { f.print(this) }
}

val (f: Flower) = new Rose
val (h: Human) = new Mathematician
h.identify(f)
```

[Q] Fill in the blank so that the output is:

```
Red Plant
```

```
class Flower {
  def print(h: Human) { println("Flower") }
  def print(m: Mathematician) { println("Plant") }
}
class Rose extends Flower {
  override def print(h: Human) { println("Rose") }
  override def print(m: Mathematician) {
    println("Red Plant")
  }
}
class Human {
  def identify(r: Rose) { r.print(this) }
  def identify(f: Flower) { f.print(this) }
}
class Mathematician extends Human {
  override def identify(r: Rose) { r.print(this) }
  override def identify(f: Flower) { f.print(this) }
}

val (f: Flower) = new Rose
val (h: Human) = new Mathematician
h.identify(f)
```

[Q] Fill in the blank so that the output is:

```
Red Plant
```

```
class Flower {
  def print(h: Human) { println(“Flower”) }
  def print(m: Mathematician) { println(“Plant”) }
}
class Rose extends Flower {
  override def print(h: Human) { println(“Rose”) }
  override def print(m: Mathematician) {
    println(“Red Plant”)
  }
}
class Human {
  def identify(r: Rose) { r.print(this) }
  def identify(f: Flower) { f.print(this) }
}
class Mathematician extends Human {
  override def identify(r: Rose) { r.print(this) }
  override def identify(f: Flower) { f.print(this) }
}

val (f: Flower) = new Rose
val (h: Human) = new Mathematician
h.identify(f)
```

[Q] Which methods are executed?

```
class Flower {
  def print(h: Human) { println(“Flower”) }
  def print(m: Mathematician) { println(“Plant”) }
}
class Rose extends Flower {
  override def print(h: Human) { println(“Rose”) }
  override def print(m: Mathematician) {
    println(“Red Plant”)
  }
}
class Human {
  def identify(r: Rose) { r.print(this) }
  def identify(f: Flower) { f.print(this) }
}
class Mathematician extends Human {
  override def identify(r: Rose) { r.print(this) }
  override def identify(f: Flower) { f.print(this) }
}

val (f: Flower) = new Rose
val (h: Human) = new Mathematician
h.identify(f)
```

[Q] Which methods are executed?

Summary

- Three kinds of polymorphisms:
  - Generic methods.
  - Subtyping polymorphism.
  - Operator overloading.
- Overloading and subtyping -- dynamic type of a receiver and static type of arguments.
- Double dispatch.
- Read Chapter 19.