Imperative Programming 2:
Inheritance 1

Hongseok Yang
University of Oxford

Motivation

- Scala programs consist of classes, traits and singleton objects.
- Inheritance is a fundamental building block for relating and organising them.

Plan

- Today: Inheritance 1 (Chap10).
  - Basic concepts.
- Tomorrow: Inheritance 2 (Chap 10).
  - Slightly bigger example.

What is the output?

```
class Person {
  def work = ""
  def greet = "Hi. "+ work
}

class Student extends Person {
  override def work = "Party. "
  def cv = greet + "Scala expert. "
}

class FirstYear extends Student {
  override def work = "Study. "
  override def greet = "Busy. "
}

def f(s: Student) { println(s.work); println(s.cv) }

val yr1: FirstYear = new FirstYear; f(yr1)
```

What is the output?

```
class Person {
  def work = ""
  def greet = "Hi. "+ work
}

class Student extends Person {
  override def work = "Party. "
  def cv = greet + "Scala expert. "
}

class FirstYear extends Student {
  override def work = "Study. "
  override def greet = "Busy. "
}

def f(s: Student) { println(s.work); println(s.cv) }

val yr1: FirstYear = new FirstYear; f(yr1)
```

```
(1) Compilation error

(2) Party.
    Hi. Scala expert.

(3) Study.
    Busy. Scala expert.
```

What is the output?

```
class Person {
  def work = ""
  def greet = "Hi. "+ work
}

class Student extends Person {
  override def work = "Party. "
  def cv = greet + "Scala expert. "
}

class FirstYear extends Student {
  override def work = "Study. "
  override def greet = "Busy. "
}

def f(s: Student) { println(s.work); println(s.cv) }

val yr1: FirstYear = new FirstYear; f(yr1)
```

(1) Compilation error

(2) Party.
    Hi. Scala expert.

(3) Study.
    Busy. Scala expert.

Type mismatch?

What is the output?

```
class Person {
  def work = ""
  def greet = "Hi. "+ work
}

class Student extends Person {
  override def work = "Party. "
  def cv = greet + "Scala expert. "
}

class FirstYear extends Student {
  override def work = "Study. "
  override def greet = "Busy. "
}

def f(s: Student) { println(s.work); println(s.cv) }

val yr1: FirstYear = new FirstYear; f(yr1)
```

(1) Compilation error

(2) Party.
    Hi. Scala expert.

(3) Study.
    Busy. Scala expert.

Static binding

What is the output?

```
class Person {
  def work = ""
  def greet = "Hi. "+ work
}

class Student extends Person {
  override def work = "Party. "
  def cv = greet + "Scala expert. "
}

class FirstYear extends Student {
  override def work = "Study. "
  override def greet = "Busy. "
}

def f(s: Student) { println(s.work); println(s.cv) }

val yr1: FirstYear = new FirstYear; f(yr1)
```

(1) Compilation error

(2) Party.
    Hi. Scala expert.

(3) Study.
    Busy. Scala expert.

Static binding

What is the output?

```
class Person {
  def work = ""
  def greet = "Hi. "+ work
}

class Student extends Person {
  override def work = "Party. "
  def cv = greet + "Scala expert. "
}

class FirstYear extends Student {
  override def work = "Study. "
  override def greet = "Busy. "
}

def f(s: Student) { println(s.work); println(s.cv) }

val yr1: FirstYear = new FirstYear; f(yr1)
```

(1) Compilation error

(2) Party.
Hi. Scala expert.

(3) Study.
Busy. Scala expert.

Dynamic binding

What is the output?

```
class Person {
  def work = ""
  def greet = "Hi. "+ work
}

class Student extends Person {
  override def work = "Party. "
  def cv = greet + "Scala expert. "
}

class FirstYear extends Student {
  override def work = "Study. "
  override def greet = "Busy. "
}

def f(s: Student) { println(s.work); println(s.cv) }

val yr1: FirstYear = new FirstYear; f(yr1)
```

(1) Compilation error

(2) Party.
    Hi. Scala expert.

(3) Study.
    Busy. Scala expert.

Dynamic binding

What is the output?

```
class Person {
  def work = ""
  def greet = "Hi. "+ work
}

class Student extends Person {
  override def work = "Party. "
  def cv = greet + "Scala expert. "
}

class FirstYear extends Student {
  override def work = "Study. "
  override def greet = "Busy. "
}

def f(s: Student) { println(s.work); println(s.cv) }

val yr1: FirstYear = new FirstYear; f(yr1)
```

```
(1) Compilation error

(2) Party.
    Hi. Scala expert.

(3) Study.
    Busy. Scala expert.
```

Learning outcome

- Can write classes using inheritance.
- Can answer the question in the previous slide, and explain the answer.
- Can explain key principles behind inheritance - code reuse, dynamic binding and subtyping.

Syntax: "class D extends C"

- C is called a superclass of D, and D a subclass of C.

```
class Person { .. }
class Student extends Person { .. }
```

- If class parameters are expected for a superclass, they should be specified in the extends clause.

```
class Company(name: String, type: String) { .. }
class Univ(name: String) extends Company(name, "Edu.") { .. }
```

- "extends" can be used with singleton objects and traits.

```
object OxfordUni extends Univ("U. Oxford") { .. }
class C {..}; trait D extends C {..}
trait X {..}; trait Y extends X {..}; class Z extends X {..}
```

Consequence 1: Inheriting methods & fields

- Methods and fields of a superclass become available to a subclass (unless they are declared private).
- This enables code reuse.

```
class Person {

  def work = ""
  def greet = "Hi. "+ work
}

class Student extends Person {

  def cv = greet + "Scala expert. "
}

val st = new Student; println(st.greet)
```

Consequence 1: Inheriting methods & fields

- Methods and fields of a superclass become available to a subclass (unless they are declared private).
- This enables code reuse.

```
class Person {
  val location = “UK“
  def work = ““
  def greet = “Hi. “+ work
}

class Student extends Person {

  def cv = greet + “Scala expert. “ + location
}

val st = new Student; println(st.greet)
```

Consequence 1: Inheriting methods & fields

- Methods and fields of a superclass become available to a subclass (unless they are declared private).
- This enables code reuse.

```
class Person {
  private val location = “UK“
  def work = ““
  def greet = “Hi. “+ work
}

class Student extends Person {

  def cv = greet + “Scala expert. “ + location
}

val st = new Student; println(st.greet)
```

Consequence 1: Inheriting methods & fields

- Methods and fields of a superclass become available to a subclass (unless they are declared private).
- This enables code reuse.

```
class Person {
  private val location = “UK“
  def work = ““
  def greet = “Hi. “+ work
}

class Student extends Person {

  def cv = greet + “Scala expert. “ + location
}

val st = new Student; println(st.greet)
```

Compilation error. Private fields & methods cannot be accessed in a subclass.

Consequence 2: Subtyping

- A subclass becomes a subtype of a superclass.
- A subtype is a transitive relation among classes and traits.

- If A is a subtype of B (denoted A <: B), an object of type A can be used as an object of type B.

```
class Person { ... }
class Student extends Person { ... }
class FirstYear extends Student { ... }

def f(s: Student) { println(s.work); println(s.cv) }

val yr1: FirstYear = new FirstYear; f(yr1)
```

Consequence 2: Subtyping

- A subclass becomes a subtype of a superclass.
- A subtype is a transitive relation among classes and traits.

- If A is a subtype of B (denoted A <: B), an object of type A can be used as an object of type B.

```
class Person { ... }
class Student extends Person { ... }
class FirstYear extends Student { ... }

def f(s: Student) { println(s.work); println(s.cv) }
def g(s: Person) { println(s.work) }
val yr1: FirstYear = new FirstYear; f(yr1); g(yr1)
```

No complaint from the Scala compiler.
FirstYear <: Student and Student <: Person.
Hence, FirstYear <: Person, by transitivity.

- A subclass becomes a subtype of a superclass.
- A subtype is a transitive relation among classes and traits.

- If A is a subtype of B (denoted A <: B), an object of type A can be used as an object of type B.

```
class Person { ... }
class Student extends Person { ... }
class FirstYear extends Student { ... }

def f(s: Student) { println(s.work); println(s.cv) }
def g(s: Person) { println(s.work) }
val yr1: FirstYear = new FirstYear; f(yr1); g(yr1)
```

Consequence 3: Overriding and specialisation

- A method of a superclass can be redefined in a subclass. This is called method overriding.
- Changes the meaning of inherited methods. They use overriden methods, not original ones.
- Frequently used for specialising a superclass.

```
class Person {
  def work = ""
  def greet = "Hi. "+ work
}
class Student extends Person { override def work = "Party. " }
val s = new Student; println(s.work); println(s.greet)
```

Consequence 3: Overriding and specialisation

- A method of a superclass can be redefined in a subclass. This is called method overriding.
- Changes the meaning of inherited methods. They use overriden methods, not original ones.
- Frequently used for specialising a superclass.

```
class Person {
  def work = ""
  def greet = "Hi. "+ work
}
class Student extends Person { override def work = "Party. " }
val s = new Student; println(s.work); println(s.greet)
class Tutor extends Person { override def work = "Sleep. " }
val t = new Tutor; println(t.greet)
```

Dynamic binding

- For each method call obj.meth, a compiler chooses code to run based on the dynamic type of obj.

```
class Person {
  def work = ""
  def greet = "Hi. "+ work
}

class Student extends Person {
  override def work = "Party. "
  def cv = greet + "Scala expert. "
}

val s: Student = new Student; println(s.cv)
val p: Person = s; println(p.work)
```

Dynamic binding

- For each method call obj.meth, a compiler chooses code to run based on the dynamic type of obj.

```
class Person {
  def work = ""
  def greet = "Hi. "+ this.work
}

class Student extends Person {
  override def work = "Party. "
  def cv = this.greet + "Scala expert. "
}

val s: Student = new Student; println(s.cv)
val p: Person = s; println(p.work)
```

What is the output?

```
class Person {
  def work = ""
  def greet = "Hi. "+ work
}

class Student extends Person {
  override def work = "Party. "
  def cv = greet + "Scala expert. "
}

class FirstYear extends Student {
  override def work = "Study. "
  override def greet = "Busy. "
}

def f(s: Student) { println(s.work); println(s.cv) }

val yr1: FirstYear = new FirstYear; f(yr1)
```

(1) Compilation error

(2) Party.
    Hi. Scala expert.

(3) Study.
    Busy. Scala expert.

Follow-up 1: Can we bind `greet` statically?

```
class Person {
  def work = ""
  def greet = "Hi. "+ work
}

class Student extends Person {
  override def work = "Party. "
  def cv = greet + "Scala expert. "
}

class FirstYear extends Student {
  override def work = "Study. "
  override def greet = "Busy. "
}

def f(s: Student) { println(s.work); println(s.cv) }

val yr1: FirstYear = new FirstYear; f(yr1)
```

Expected output:

```
Study.
Hi. Study. Scala expert.
```

Follow-up 1: Can we bind `greet` statically?

```
class Person {
  def work = ""
  def greet = "Hi. "+ work
}

class Student extends Person {
  override def work = "Party. "
  def cv = super.greet + "Scala expert. "
}

class FirstYear extends Student {
  override def work = "Study. "
  override def greet = "Busy. "
}

def f(s: Student) { println(s.work); println(s.cv) }

val yr1: FirstYear = new FirstYear; f(yr1)
```

Expected output:

```
Study.
Hi. Study. Scala expert.
```

super.meth

- Always calls the method meth of the superclass.
- Binds a method call to an implementation statically.
- Stops the influence of overriding and dynamic binding.

Follow-up 2: Make parameterisation explicit

```
class Person {
  def work = ""
  def greet = "Hi. "+ work
}

class Student extends Person {
  override def work = "Party. "
  def cv = super.greet + "Scala expert. "
}

class FirstYear extends Student {
  override def work = "Study. "
  override def greet = "Busy. "
}

def f(s: Student) { println(s.work); println(s.cv) }

val yr1: FirstYear = new FirstYear; f(yr1)
```

Intention:
greet in Person is a method parameterised by work.

Follow-up 2: Make parameterisation explicit

```
abstract class Person {
  def work: String
  def greet = “Hi. “+ work
}

class Student extends Person {
  override def work = “Party. “
  def cv = super.greet + “Scala expert. “
}

class FirstYear extends Student {
  override def work = “Study. “
  override def greet = “Busy. “
}

def f(s: Student) { println(s.work); println(s.cv) }

val yr1: FirstYear = new FirstYear; f(yr1)
```

Intention:
greet in Person is a method parameterised by work.

Abstract class

```
abstract class Person { def work: String; ... }
class Student extends Person {
  override def work = “Party. “
  ... }
class FirstYear extends Student {
  override def work = “Study. “
  ... }
```

- A class is abstract if it declares methods (or types) without giving their implementations.
- Such a class should be declared “abstract”.
- Missing implementations are provided by subclasses.
- Intuition: View an abstract class as a parameterised class, and its subclass as an instantiation.

Follow-up 3: Which one compiles?

```
class Person
class Student extends Person
class FirstYear extends Student

abstract class Univ { def teach(x: Student): Student }
```

1

```
class Oxford1 extends Univ {
  override def teach(x: Student): FirstYear = new FirstYear
}
```

2

```
class Oxford2 extends Univ {
  override def teach(x: Person): Student = new Student
}
```

3

```
class Oxford3 extends Univ {
  override def teach(x: FirstYear): Student = new Student
}
```

Which one compiles?

Rule for overriding: Keep the types of the arguments.
But we can replace the result's type by a subtype.

```
class Person
class Student extends Person
class FirstYear extends Student

abstract class Univ { def teach(x: Student): Student }
```

1

```
class Oxford1 extends Univ {
  override def teach(x: Student): FirstYear = new FirstYear
}
```

2

```
class Oxford2 extends Univ {
  override def teach(x: Person): Student = new Student
}
```

3

```
class Oxford3 extends Univ {
  override def teach(x: FirstYear): Student = new Student
}
```

Which one compiles?

Rule for overriding: Keep the types of the arguments.
But we can replace the result's type by a subtype.

```
class Person
class Student extends Person
class FirstYear extends Student { def f() { println("FY") } }

abstract class Univ { def teach(x: Student): Student }
```

1

```
class Oxford1 extends Univ {
  override def teach(x: Student): FirstYear = new FirstYear
}
```

2

```
class Oxford2 extends Univ {
  override def teach(x: Person): Student = new Student
}
```

3

```
class Oxford3 extends Univ {
  override def teach(x: FirstYear): Student =
    x.f(); new Student
}
```

When one compiles:

Rule for overriding: Keep the types of the arguments.
But we can replace the result's type by a subtype.

```
class Person
class Student extends Person
class FirstYear extends Student

abstract class Univ { def teach(x: Student): Student }
```

1

```
class Oxford1 extends Univ {
  override def teach(x: Student): FirstYear = new FirstYear
}
```

2

```
class Oxford2 extends Univ {
  override def teach(x: Person): Student = new Student
}
```

Again, we lose the guarantee of the type system.

Exercise:
What is the output?

```
abstract class A {
  def f = "A.f calls "+ g +" and "+ h
  def g: String
  def h: String
}

class B extends A {
  override def g = "B.g"
  override def h = "B.h calls " + g
}

class C extends B {
  override def f = "C.f calls " + super.f
  override def h = "C.h calls " + super.h
}

val c = new C; println(c.f)
```

Exercise:
What is the output?

```
abstract class A {
  def f = "A.f calls "+ g +" and "+ h
  def g: String
  def h: String
}

class B extends A {
  override def g = "B.g"
  override def h = "B.h calls " + g
}

class C extends B {
  override def f = "C.f calls " + super.f
  override def h = "C.h calls " + super.h
}

val c = new C; println(c.f)
```

```
C.f calls A.f calls B.g and C.h calls B.h calls B.g
```

Summary

- Consequences of inheritance.
  - Inheriting methods and fields -- code reuse.
  - Subtyping -- reuse of client programs of a class.
  - Overloading and dynamic binding -- parameterisation.
- Read: Chap 10.