Overview

Last time: “programming in the large”
- Programs, packages/namespaces, importing
- Modules and interfaces
- Mostly: using Scala for examples

Today: the elephant in the room:
- Objects and Classes
- A taste of “advanced” OOP constructs: inner classes, anonymous objects and mixins
- Illustrate using examples in Scala, and some comparisons with Java

Objects

- An object is a module with some additional properties:
  - Encapsulation: Access to an object’s components can be limited to the object itself (or to a subset of objects)
  - Self-reference: An object is a value and its methods can refer to the object’s fields and methods (via an implicit parameter, often called this or self)
  - Inheritance: An object can inherit behavior from another “parent” object

- Objects/inheritance are tied to classes in some (but not all) OO languages
- In Scala, the object keyword creates a singleton object (“class with only one instance”)
- (in Java, objects can only be created as instances of classes)

Self-Reference

- Inside an object definition, the this keyword refers to the object being defined.
- This provides another form of recursion:

```scala
object Fact {
  def fact (n: Int): Int = {
    if (n == 0) {1} else {n * this.fact(n-1)}
  }
}
```

- Moreover, as we'll see, the recursion is open: the method that is called by this.foo(x) depends on what this is at run time.
Encapsulation and Scope

- An object can place restrictions on the scope of its members.
- Typically used to prevent external interference with ‘internal state’ of object.
- For example: Java, C++, C# all support.
  - private keyword: “only visible to this object”
  - public keyword: “visible to all”
- Java: package scope (default): visible only to other components in the same package.
- Scala: `private[X]` allows qualified scope: “private to (class/object/trait/package) X”
- Python, Javascript: don’t have (enforced) private scope (relies on programmer goodwill).

Classes

- A class is an interface with some additional properties:
  - **Instantiation**: classes can describe how to construct associated objects. (instances of the class)
  - **Inheritance**: classes may inherit from zero or more parent classes as well as implement zero or more interfaces.
  - **Abstraction**: Classes may be abstract, that is, may name but not define some fields or methods.
  - **Dynamic dispatch**: The choice of which method is called is determined by the run-time type of a class instance, not the static type available at the call.

- Not all object-oriented languages have classes!
  - Smalltalk, JavaScript are well-known exceptions.
  - Such languages nevertheless often use prototypes, or commonly-used objects that play a similar role to classes.

Constructing instances

- Classes typically define special functions that create new instances, called constructors.
  - In C++/Java, constructors are defined explicitly and separately from the initialized data.
  - In Scala, there is usually one “default” constructor whose parameters are in scope in the whole class body.
  - (additional constructors can be defined as needed)
- Constructors called with the `new` keyword.

```scala
class C(x: Int, y: String) {
  val i = x
  val s = y
  def this(x: Int) = this(x,"default")
}
scala> val c1 = new C(1,"abc")
scala> val c2 = new C(1)
```

Inheritance

- An object can inherit from another.
- This means: the parent object, and its components, become “part of” the child object.
  - accessible using `super` keyword
  - (though some components may not be directly accessible)
- In Java (and Scala), a class extends exactly one superclass (Object, if not otherwise specified).
- In C++, a class can have multiple superclasses.
- Non-class-based languages, such as JavaScript and Smalltalk, support inheritance directly on objects via extension.
**Subtyping**

- As (briefly) mentioned last week, an object \( \text{Obj} \) that extends a trait \( \text{Tr} \) is automatically a subtype \( \text{Obj} <: \text{Tr} \).
- Likewise, a class \( \text{Cl} \) that extends a trait \( \text{Tr} \) is a subtype of \( \text{Tr} \) \( \text{Cl} <: \text{Tr} \).
- A class (or object) \( \text{Sub} \) that extends another class \( \text{Super} \) is a subtype of \( \text{Super} \) \( \text{Sub} <: \text{Super} \).
- However, subtyping and inheritance are distinct features:
  - As we’ve already seen, subtyping can exist without inheritance.
  - Moreover, subtyping is about types, whereas inheritance is about behavior (code).

**Inheritance and encapsulation**

- Inheritance complicates the picture for encapsulation somewhat.
- private keyword prevents access from outside the class (including any subclasses).
- protected keyword means “visible to instances of this object and its subclasses”.
- Scala: Both private and protected can be qualified with a scope \([X]\) where \(X\) is a package, class or object.

```scala
class A { private[A] val a = 1
protected[A] val b = 2 }
class B extends A {
def foo() = a + b
} // "a" not found
```

**Cross-instance sharing**

- Classes in Java can have static fields/members that are shared across all instances.
- Static methods can access private fields and methods.
- static is also allowed in interfaces (but only as of Java 8).
- Class with only static members ~ module.
- C++: friend keyword allows sharing between classes on a case-by-case basis.

**Companion Objects**

- Scala has no static keyword.
- Scala instead uses **companion objects**
  - Companion = object with the same name as the class and defined in the same scope.
  - Companions can access each others’ private components.

```scala
object Count { private var x = 1 }
class Count { def incr() {Count.x = Count.x+1} }
```

- Note: This can only be done in compiled code, not interactively.
Multiple inheritance and the **diamond problem**

- As noted, C++ allows *multiple inheritance*
- Suppose we did this (in Scala terms):
  ```scala
  class Win(val x: Int, val y: Int)
  class TextWin(...) extends Win
  class GraphicsWin(...) extends Win
  class TextGraphicsWin(...) extends TextWin and GraphicsWin
  ```
  - In C++, this means there are two copies of `Win` inside `TextGraphicsWin`
  - They can easily become out of sync, causing problems
  - Multiple inheritance is also difficult to implement (efficiently); many languages now avoid it

Abstraction

- A class may leave some components undefined
  - Such classes must be marked abstract in Java, C++ and Scala
  - To instantiate an abstract class, must provide definitions for the methods (e.g. in a subclass)
- Abstract classes can define common behavior to be inherited by subclasses
- In Scala, abstract classes can also have unknown type components
  - (optionally with subtype constraints)
  ```scala
  abstract class ConstantVal {
    type T <: AnyVal
    val c: T
  } // a constant of any value type
  ```

Dynamic dispatch

- An abstract method can be implemented in different ways by different subclasses
- When an abstract method is called on an instance, the corresponding implementation is determined by the *run-time type* of the instance.
- *(necessarily in this case, since the abstract class provides no implementation)*
  ```scala
  abstract class A { def foo(): String}
  class B extends A { def foo() = "B"}
  class C extends A { def foo() = "C" }
  scala> val b: A = new B
  scala> val c: A = new C
  scala> (b.foo(), c.foo())
  ```

Overriding

- An inherited method that is already defined by a superclass can be *overridden* in a subclass
- This means that the subclass’s version is called on that subclass’s instances using dynamic dispatch
- In Java, `@Override` annotation is optional, checked documentation that a method overrides an inherited method
- In Scala, must use `override` keyword to clarify intention to override a method
  ```scala
  class A { def foo() = "A"}
  class B extends A { override def foo() = "B" }
  scala> val b: A = new B
  scala> val c: A = new C
  scala> b.foo() // error
  scala> c.foo()
  ```
**Type tests and coercions**

- Given \( x: A \), Java/Scala allow us to test whether its run-time type is actually subclass \( B \)
  
  ```scala
  scala> b.isInstanceOf[B]
  ```

- and to *coerce* such a reference to \( y: B \)
  
  ```scala
  scala> val b2: B = b.asInstanceOf[B]
  ```

- Warning: these features can be used to violate type abstraction!

  ```scala
  def weird[A](x: A) = if (x.isInstanceOf[Int]) {
    (x.asInstanceOf[Int]+1).asInstanceOf[A]
  } else {x}
  ```

**Advanced constructs**

- So far, we’ve covered the “basic” OOP model (circa Java 1.0)

- Modern languages extend this in several ways
  - We can define a class/object inside another class:
    - As a member of the enclosing class (tied to a specific instance)
    - or as a static member (shared across all instances)
    - As a local definition inside a method
    - As an anonymous local definition

- Some languages also support *mixins* (e.g. Scala traits)

- Scala supports similar, somewhat more uniform composition of classes, objects, and traits

**Classes/objects as members**

- In Scala, classes and objects (and traits) can be nested arbitrarily
  
  ```scala
  class A { object B { var x = 1 } }
  scala> val a = new A
  ```

  ```scala
  object C {class D { var x = 1 } }
  scala> val d = new C.D
  ```

  ```scala
  class E { class F { var x = 1 } }
  scala> val e = new E
  scala> val f = new e.F
  ```

**Summary**

- Today
  - Objects, encapsulation, self-reference
  - Classes, inheritance, abstraction, dynamic dispatch

- This is only the tip of a very large iceberg...
  - there are almost as many “object-oriented” programming models as languages
  - the design space, and “right” formalisms, are still active areas of research

- Next time:
  - Inner classes, anonymous objects, mixins, parameterized types
  - Combining object-oriented and functional programming