Object-oriented Programming

- Programs are structured in terms of **objects**: collections of variables ("**fields**") and functions ("**methods**")
  - Implicitly associates variable data with functions
  - Invented to model discrete entities or processes, e.g.
    - Simulations (object = real-world object)
    - Graphical user interfaces (object = desktop item)
  - But can be used for any programming task
OOP Characteristics

- OOP languages usually support
  - Dynamic dispatch
  - Encapsulation
  - Inheritance
  - Subtyping

...but there is no precise definition of OOP
## Some important OOP languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Static types?</th>
<th>Class-based?</th>
<th>All values are objects?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simula67</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>Smalltalk</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>C++</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>Java/C#</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>JavaScript</td>
<td></td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td></td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>Ruby</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
Procedures vs. Methods

Fundamental OOP control structure is method invocation

Similar to function call in a procedural language

But each method takes the object itself as an implicit argument, and this receiver object also helps resolve method name:

```java
s.add(x) ; OO style
Set.add(s,x) ; procedural style
```

Change in metaphor: instead of applying functions to values, we “send messages to object.”
Dynamic Dispatch

In most OOP languages, the receiving object itself controls how each message is processed.

This is a form of dynamic overloading (i.e., a certain kind of polymorphism)

Example:

s1 = new ordered-list-set
s2 = new balanced-tree-set
if ... then s = s1 else s = s2
s.add(42)

The implementation of the add method is completely different in s1 and s2; the choice of which one runs is determined at run time.
Classes

- In OOP, we typically want to create multiple objects having the same structure (field names) and method definitions.

- In most OO languages this is done by defining a class, which is a kind of template from which new objects can be created.

- Different instances of the class will typically have different field values, but all will share the same method implementations.

- Classes are not essential; one alternative (used by JavaScript, e.g.) is to create new objects by cloning existing prototype objects.
Encapsulation

- Objects are often (though not always) designed so that their data fields can only be accessed by the object’s own methods.

- This kind of encapsulation is just what is needed to implement an Abstract Data Type (ADT).

  - It allows the representation or implementation of an object to change without affecting client code that interacts with the object only via method send.

- However, OO programmers often violate encapsulation. For example, object fields may be public, allowing them to be accessed from code outside the object’s methods.
Subtyping in general

- Recall that types can often be naturally refined into subtypes
  - type = set of values supporting certain operations
  - subtype = subset of values supporting those operations and more

- Example: Type “Animal” supports operation “eat.” Subtype “Bird” supports operation “eat” and also operation “fly.”

- For type soundness, say B is a subtype of A if we can use a B value wherever an A value is expected.

- Informally, makes sense to say B is a subtype of A if every value of B “is a” value of A.
Subclasses

In class-based OO languages, it is common to declare immediate subtypes by defining subclasses.

- e.g. in Scala: class Bird extends Animal

The overall subtyping relation is automatically taken to be the reflexive, transitive closure of the immediate subclass declarations.

- e.g. if we also define class Crow extends Bird then automatically Crow is a subtype of Animal

This leads to a subtyping hierarchy
Heterogenous Collections

Subtyping and dynamic dispatch combine to give powerful support for manipulating heterogeneous collections:

```scala
abstract class Animal {
  def name() : String
  def eat() : String
}

class Bird extends Animal {
  def name() = "bird"
  def eat() = "chomp on insects"
}

class Bee extends Animal {
  def name() = "bee"
  def eat() = "suck nectar"
}

object Main {
  val animals : List[Animal] = List(new Bird(), new Bee())
  for (animal <- animals)
    println(animal.name() + ":" + animal.eat())
}
```

Output:

```
bird:chomp on insects
bee:suck nectar
```
Inheritance

- Classes may also be related because their implementations are similar.

- To avoid writing duplicate code, we might like to inherit most of the implementation of one class from another.
  - possibly overriding some aspects of the implementation in the subclass

- This works nicely when the inheriting class is also a subtype of the providing class
Using Inheritance

abstract class Animal {
    def name() : String
    def eat() : String
}

class Bird extends Animal {
    def name() = "bird"
    def eat() = "chomp on insects"
}

class Bee extends Animal {
    def name() = "bee"
    def eat() = "suck nectar"
}

object Main {
    val animals : List[Animal] = List(new Bird(), new Crow(), new Bee())
    for (animal <- animals)
        println(animal.name() + "":" + animal.eat())
}

class Crow extends Bird {
    override def name() = "crow"
}

definition of eat() is inherited from Bird
definition of name() from Bird is overridden

Output:
bird:chomp on insects
crow:chomp on insects
bee:suck nectar
Flexibility of Dynamic Dispatch

• Method calls are always dispatched to the original receiving object, so superclass code can access functionality in subclasses.

```scala
abstract class Animal {
  def name() : String
  def eat() : String
}

class Bird extends Animal {
  def name() = "bird"
  def eat() = "chomp on " + food()
  def food() = "insects"
}

class Crow extends Bird {
  override def name() = "crow"
  override def food() = "anything"
}

class Robin extends Bird {
  override def name() = "robin"
  override def food() = "worms"
}

object Main {
  val animals : List[Animal] = List(new Bird(), new Crow(), new Robin())
  for (animal <- animals)
    println(animal.name() + ":") + animal.eat()
}
```

Output:

bird:chomp on insects
crow:chomp on anything
robin:chomp on worms
Subtyping vs. Inheritance

Sometimes we’d like to use inheritance even when subtyping is not appropriate, e.g.:

- Suppose we add bat as a new kind of animal. Since bats also eat insects, we might be tempted to make bat a subclass of bird so that it would inherit the implementation of eat().

- But it is not the case that a bat “is a” bird! That is, it does not suffice to provide a bat when a bird is expected. (E.g. if bird had a method lay_eggs(), we would not be able to give an appropriate implementation of that method for bat.)

- Inheritance concerns can warp design of subtyping hierarchy
Beyond Single Inheritance

- More flexible inheritance mechanisms can help.

- Some languages (e.g. C++) let a class inherit from multiple super-classes. (Semantics of field inheritance can be messy.)

- Java supports subtyping through multiple interfaces, which are like classes without fields.

- Scala supports both inheritance and subtyping through traits, which are like partial class definitions that can be “mixed in” together.
abstract class Named {
    def name() : String
}

trait Flies extends Named {
    def fly() : String
}

trait Eats extends Named {
    def eat() : String
}

object Main {
    val fliers : List[Flies] = List(new Bird(), new Airplane())
    val eaters: List[Eats] = List(new Bird(), new Lion())
    for (flier <- fliers) println(flier.name() + "::" + flier.fly())
    for (eater <- eaters) println(eater.name() + "::" + eater.eat())
}

class Airplane extends Flies {
    def name() = "plane"
    def fly() = "using jets"
}

class Lion extends Eats {
    def name() = "lion"
    def eat() = "chomps on red meat"
}

class Bird extends Flies with Eats {
    def name() = "bird"
    def fly() = "by flapping wings"
    def eat() = "chomps on insects"
}
abstract class Animal {
    def name() : String
    def eat() : String
    def reproduce() : String
}

trait Insectivorous {
    def eat() = "chomp on insects"
}

trait Oviparous {
    def reproduce() = "lay eggs"
}

class Bird extends Animal with Insectivorous with Oviparous {
    def name() = "bird"
}

class Bat extends Animal
    with Insectivorous {
    def name() = "bat"
    def reproduce() = "bear live young"
}

class Bee extends Animal
    with Oviparous {
    def name() = "bee"
    def eat() = "suck nectar"
}

object Main {
    val animals : List[Animal] = List(new Bird(), new Bat(), new Bee())
    for (animal <- animals)
        println(animal.name() + ":" + animal.eat() + ":" + animal.reproduce())
}
Representation of Objects

- An object is essentially a record (usually heap-allocated to support unlimited lifetime) containing values for the fields and code pointers for methods.

- In a dynamically-typed language, the object record fields/methods have labels that can be searched at run time (at least in a naive implementation).

- In a statically-typed language, we can (usually) compute the offset of each field/method statically, avoiding run-time search.

- In a class-based language, we usually factor the representation so that method pointers are stored in a separate class record.
Example without inheritance

class A {
    int x;
    int y;
    f() = x+y;
    g(z) = f()+z;
}
val a1:A = ...
val a2:A = ...
val w = a1.g(10)

Generated code in C-like notation

\[
f(a) = a->x + a->y \\
g(a,z) = a->class->f(a) + z \\
w = a1->class->g(a1,10)
\]
Implementing Sub-classes

- A key observation is that sub-classes only add (or override) fields and methods to the super-class.

- So the list of contents for super-class representation is always a prefix of list for sub-class representation.

- All classes in the hierarchy can share the same offsets for the fields and methods they have in common, making run-time search unnecessary.

- (This is just an implementation trick, but the resulting efficiency was historically important for the adoption of OOP.)

- Only works for single inheritance; multiple inheritance, interfaces, or traits require more work.
Example with inheritance

class A {
    int x;
    int y;
    f() = x+y;
}
class B extends A {
    int z;
    g() = x+z;
}
val a:A = …
val b:B = …
val w = b.f()+b.g()