# EECS 390 – Lecture 13

## Object-Oriented Programming



3/10/24

# Review: Data Abstraction

- Abstraction separates what something is from how it works
- ***Abstract data types (ADTs)*** separate the interface of a data type from its implementation
- ***Encapsulation*** is an important, though not universal, property of an ADT, bundling the data of the ADT along with the functions that operate on that data
- We built a hierarchy of ADTs, beginning with immutable pairs all the way up to an abstraction similar to that provided by object orientation

```
>>> a = account(33)
>>> a('deposit')(4)
37
>>> a('withdraw')(7)
30
```

```
>>> a = account(33)
>>> a.deposit(4)
37
>>> a.withdraw(7)
30
```

# Object-Oriented Programming

- Object-oriented languages provide a systematic mechanism for defining abstract data types
- Fundamental features:
  - ***Encapsulation***: bundling together data of an ADT along with the functions that operate on the data
  - ***Information hiding***: restricting access to the implementation details of an ADT
  - ***Inheritance***: reusing code of an existing ADT when defining a new one
  - ***Subtype polymorphism***: using an instance of a derived ADT where a base ADT is expected
    - Requires some form of ***dynamic binding***, where the derived functionality is used at runtime

The term "encapsulation" is often used to encompass information hiding as well.

# Terminology

- A ***class*** defines a pattern for the instances of an ADT
  - Specifies the data included and the functions that operate on that data
- An ***object*** is an instance of a class
- The individual data items and functions that comprise a class are its ***members***
- Data members are also called ***fields*** or ***attributes***
- Member functions are usually called ***methods***

```
struct Foo {
  int x;
  Foo(int x_);
  int bar(int y);
};
```

Field

Constructor

Method

# Static Fields

- Each object has its own set of instance fields
- ***Static fields*** are associated with a class, and there is only one copy shared by all instances of the class
  - Can generally be accessed directly through class or indirectly through an instance
- Example in Java:

```
class Foo {
  static int bar = 3;
}

class Main {
  public static void main(String[] args) {
    System.out.println(Foo.bar);
    System.out.println(new Foo().bar);
  }
}
```

Access through class

Access through instance

# Static Fields in Python

- In Python, variables defined directly within the class definition are automatically static fields

```
class Foo:
    bar = 3

print(Foo.bar)
print(Foo().bar)
```

- Instance fields have to be defined through `self`

```
class Baz:
    def __init__(self):
        self.bar = 3
```

# Access Control

- Information hiding requires ability to restrict access to members of a class
- Access modifiers, in languages that have them, allow the programmer to specify what code has access

| | `public` | `private` | `protected` | | `internal` in C#, Java default | Python |
|---|---|---|---|---|---|---|
| | | | C++, C# | Java | | |
| Same instance | X | X | X | X | X | X |
| Same class | X | X | X | X | X | X |
| Derived classes | X | | X | X | | X |
| Code in same package | X | | | X | X | X |
| Global access | X | | | | | X |

In Ruby, field access is restricted to the same instance.

# Instance Methods

- Instance methods take in the instance on which to operate as a parameter
  - Often named `self` or `this`
  - Usually an implicit parameter
- Example in C++:

```
class Foo {
  int x;
public:
  Foo(int x_) : x(x_) {}
  int get_x() { return this->x; }
};

Foo f(3);
f.get_x();
```

**this-> can be elided if x not hidden by local variable**

**Object that receives method call**

**Address of object implicitly passed as this**

# Static Methods

- ***Static methods*** do not operate on an instance, so they do not have access to instance members
- In many languages, the `static` keyword denotes a static method
- In Python, the `@staticmethod` decorator must be used to enable access through both a class and instance

```
class Baz:
    @staticmethod
    def name():
        return 'Baz'

print(Baz.name())
print(Baz().name())
```

# Property Methods

- Some languages enable ***property methods*** to be defined, which have the syntax of field access but invoke methods
  - Abstract the interface of a field from its implementation

```
>>> c = Complex(1, math.sqrt(3))
>>> c.magnitude
2.0
>>> c.angle / math.pi
0.3333333333333333
```

- Example in Python:

```
class Complex(object):
    def __init__(self, real, imag):
        self.real, self.imag = real, imag

    @property
    def magnitude(self):
        return (self.real ** 2 +
                self.imag ** 2) ** 0.5

    @property
    def angle(self):
        return math.atan2(self.imag, self.real)
```

# OOP and Message Passing

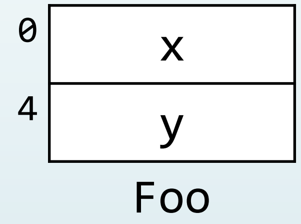
- Conceptually, object-oriented programming consists of passing messages to objects, which then respond to the message
  - Member access on an object can be thought of as sending a message to the object
- Languages differ in:
  - Whether the set of messages an object responds to (i.e. its members) is fixed at compile time
  - Whether the actual message to be sent to an object must be known at compile time

# Record[1]-Based Implementation

- In languages that prioritize efficiency, the members of an object are known at compile time
- Fields of an object are stored directly within the memory of the object, at offsets that can be computed at compile time
- Field access can be translated by the compiler to an offset into the object

```
class Foo {
public:
  int x, y;
  Foo(int x_, int y_);
};

Foo f(3, 4);
cout << (f.x + f.y);
```




[1]Records are called ***structs*** in C++.

# Dictionary-Based Implementation

- In languages that allow members to be added to an object at runtime, an object's members are usually stored in a dictionary
  - Similar to our message-passing implementation using functions
- A well-defined lookup process specifies how to look up a member
  - In Python, check instance dictionary first, then class

```
class Foo:
    y = 2
    def __init__(self, x):
        self.x = x

f = Foo(3)
print(f.x, f.y, Foo.y)   # prints 3 2 2
f.y = 4
print(f.x, f.y, Foo.y)   # prints 3 4 2
```

Adds binding to instance dictionary

# Dynamic Messages

- Dictionary-based languages often provide a way to construct and send messages to an object at runtime

```
>>> x = [1, 2, 3]
>>> getattr(x, 'append')(4)
>>> x
[1, 2, 3, 4]
```

- Some record-based languages do so as well (e.g. Java's ***reflection*** API)

```
class Main {
  public static void main(String[] args)
    throws Exception {
    String s = "Hello World";
    java.lang.reflect.Method m =
      String.class.getMethod("length", null);
    System.out.println(m.invoke(s)); // prints 11
  }
}
```

# Types of Inheritance in C++

- C++ supports private, protected, and public inheritance
  - Determine the set of code that has access to the fact that a derived class has a specific base class
  - Most languages only support public inheritance
- Example:

```
struct A {
  void a() {
    cout << "A::a()" << endl;
  }
};

struct B : private A {
  void b() {
    A *a = this;
    a->a();
  }
};
```

B knows that A is its base class

```
int main() {
  B b;
  b.b();
  b.a();          ✖
  A *ap = &b;     ✖
}
```

The outside world does not

# Abstract Methods

- A method is ***abstract*** if it doesn't have an implementation
  - Pure virtual functions in C++
- A class is abstract if it has at least one abstract method
- Used for interface inheritance, as well as polymorphism
- Example in Java:

```
abstract class A {
  abstract void foo();
}
```

**Abstract class must be qualified by abstract keyword**

**Abstract method denoted by abstract keyword**

# Interfaces

- A class that only has abstract methods is often called an ***interface***
- Java has a special mechanism for defining and implementing interfaces

```
interface I {
  void bar();
}

class C extends A implements I {
  void foo() {
    System.out.println("foo() in C");
  }
  public void bar() {
    System.out.println("bar() in C");
  }
}
```

**Only one base class is allowed**

**Any number of interfaces can be implemented**

# Mixins

- Some languages decouple inheritance from polymorphism by allowing code to be inherited without establishing a parent-child relationship
- Example in Ruby:

```
class Counter
  include Comparable
  attr_accessor :count
  def initialize()
    @count = 0
  end
  def increment()
    @count += 1
  end
  def <=>(other)
    @count <=> other.count
  end
end
```

**Includes comparsion operators that call <=>**

```
> c1 = Counter.new()
> c2 = Counter.new()
> c1.increment()
=> 1
> c1 == c2
=> false
> c1 < c2
=> false
> c1 > c2
=> true
```

# Root Class

- In some languages, every object eventually derives from some root class
  - `Object` in Java, `object` in Python
- Example of code that uses the root class:

```
Vector<Object> unique(Vector<Object> items) {
  Vector<Object> result = new Vector<>();
  for (Object item : items) {
    if (!result.contains(item)) {
      result.add(item);
    }
  }
  return result;
}
```

Calls equals() method on item