OBJECT-ORIENTED PROGRAMMING

Instructors: Crista Lopes

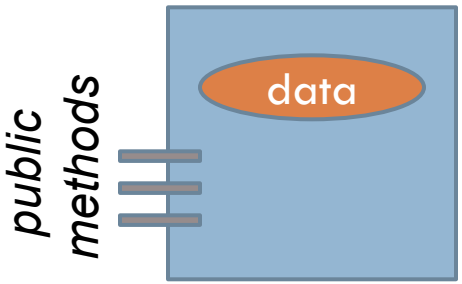
Goal of this lecture

- OOP...
  - Encapsulation
  - Classes vs.
    - Prototypes
    - Singletons
    - Abstract Data Types
  - Code reuse: inheritance vs. delegation vs. mixins
  - Dynamic dispatch
  - Self reference
- ... in different flavors

Encapsulation

Encapsulation

- Previously, in structured programming:
  - data, which is passive
  - functions, which manipulate data
- An object contains both data and behavior (methods) that manipulate that data
  - An object *does* things
  - An object is *responsible* for its own data
    - But: it can *expose* that data to other objects



Example: A "Rabbit" object

- You could (in a game, for example) create an object representing a rabbit
- It would have data:
  - How hungry it is
  - How frightened it is
  - Where it is
- And methods:
  - eat, hide, run, dig

Advice: Restrict access

- Always, *always* strive for a narrow interface
- Follow the principle of information hiding:
  - the caller should know as little as possible about how the method does its job
  - the method should know little or nothing about where or why it is being called
- Make as much as possible private
- Your class is responsible for it's own data; don't allow other classes to screw it up!

Advice: Use setters and getters

```
class Employee extends Person {
   private double salary;
   private boolean male;
   public void setSalary (double newSalary) {
      salary = newSalary;
   }
   public double getSalary () { return salary; }
   public boolean isMale() { return male; }
}
```

- This way the object maintains control
- Setters and getters have conventional names: set***DataName***, get***DataName***, is***DataName*** (booleans only)

Kinds of access

- Java provides four levels of access:
  - public: available everywhere
  - protected: available within the package (in the same subdirectory) and to all subclasses
  - [default]: available within the package
  - private: only available within the class itself
- The default is called package visibility
- In small programs this isn't important...right?

Smalltalk-style

Chapter 12 of the book

- Objects:
  - Wrap around data
  - Receive messages
    - messages are labels with optional data
  - Execute methods in reaction to those messages
    - they need to lookup methods given the message labels
    - lookup is dynamic

Stateful vs. Stateless Objects

- Typically, objects are stateful
  - The data represents the state of the object
  - State changes over time, with assignments

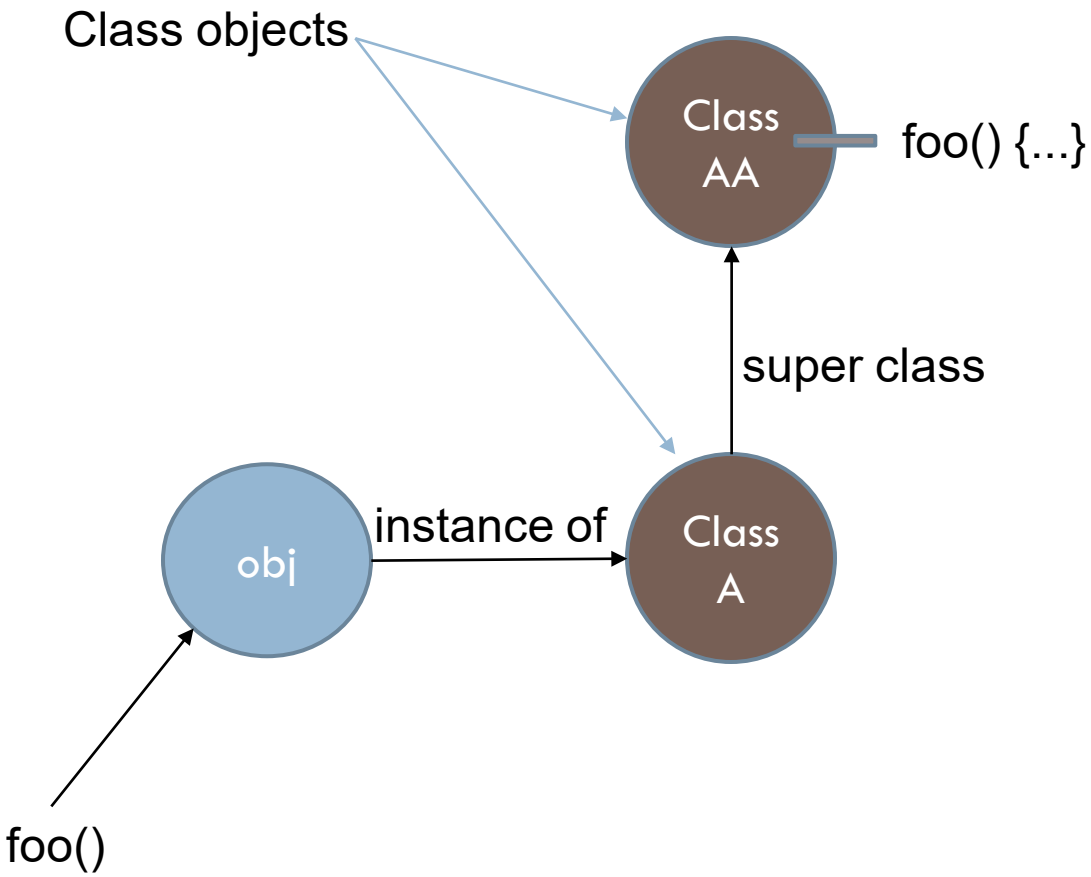
- Object can be stateless
  - When data never changes after the object is created

Classes vs. ...

Class = object factory + type + repo

- A class is like a template, or cookie cutter, or a factory
  - The class describes fields and methods
  - You use the class's constructor to make objects
- A class is also a type
  - Every object belongs to (is an instance of) a class
- A class is also a runtime repository of methods
  - obj.foo() looks up method foo in the class of obj

Class as **repository of methods**



Prototypes: objects without classes

Chapter 13 of the book

- E.g. JavaScript
- Prototypes are objects that contain their own methods

```
var apple = {
    type: "macintosh",
    color: "red",
    getInfo: function () {
        return this.color + ' ' + this.type + ' apple';
    }
}

apple.color = "reddish";
alert(apple.getInfo());
```

Abstract Data Types (aka Interfaces)

Chapter 14 of the book

- Abstractions of behavior without committing to any implementation
- ADT → multiple possible implementations

Specifying ADTs in Java

```
public interface Queue<E> extends Collection<E> {
        boolean add(E e);
        E element();
        boolean offer(E e);
        E peek();
        E poll();
        E remove();
}
```

Implementing ADTs in Java

```
public class LinkedList<E> implements Queue<E> {
        boolean add(E e) {…}
        E element() {…}
        boolean offer(E e) {…}
        E peek() {…}
        E poll() {…}
        E remove() {…}
}
```

```
public class PriorityQueue<E> implements Queue<E> {
        boolean add(E e) {…}
        E element();
        boolean offer(E e) {…}
        E peek() {…}
        E poll() {…}
        E remove() {…}
}
```

Are Classes ADTs?

- No, unless they are marked abstract.

Singletons: single instances of a class

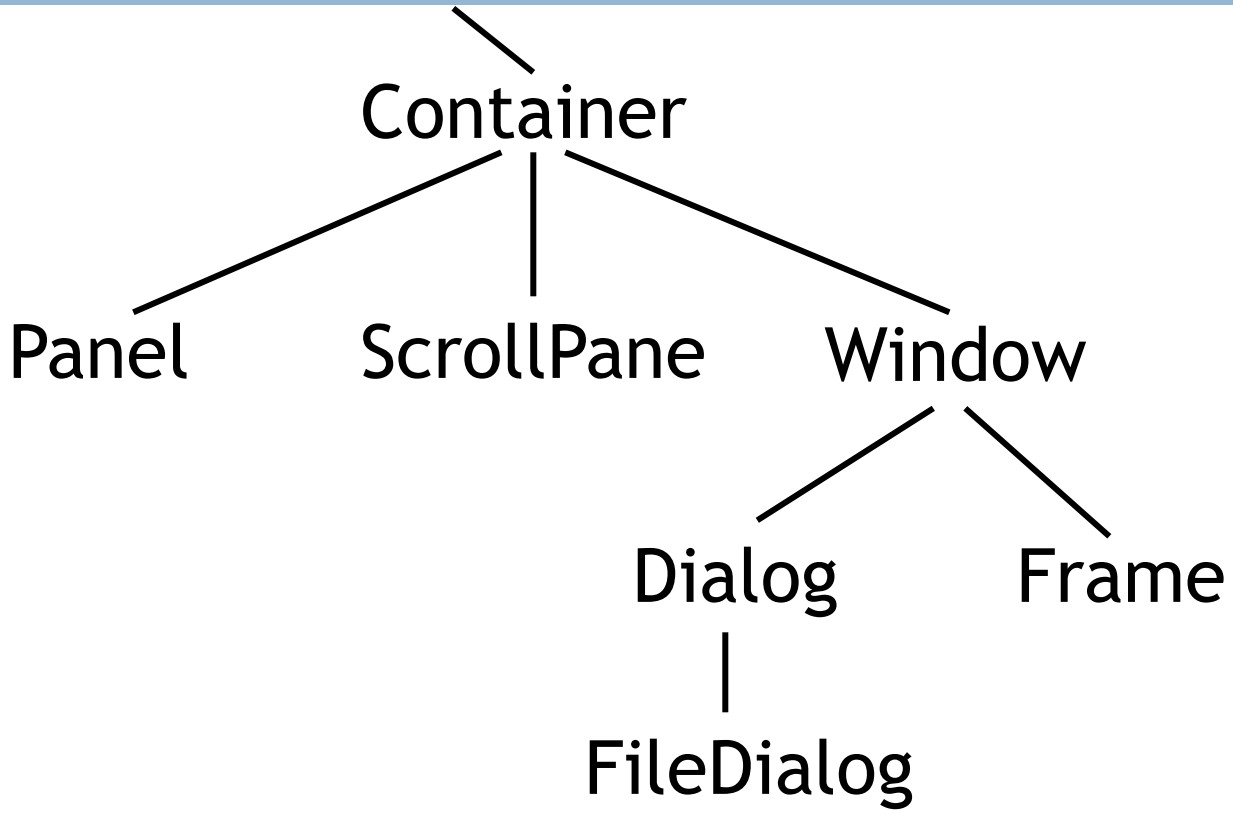
- Typically:

```
class Foo {
  private static Foo Instance;
  // no constructors
  public static getTheFoo() {
    if (Instance == null)
      Instance = new Foo();
    return Instance;
  }
}
```

- Why?
  - Situation calls for 1 single object
  - Object is more expressive than the class itself
    - e.g. can inherit, override methods, etc.
  - Zoom API components?

Code Reuse

Inheritance: example of a hierarchy



A FileDialog is a Dialog is a Window is a Container

➔ Conceptual Modeling leads to code reuse

C++Multiple Inheritance

- In C++ there may be more than one root
  - but not in Java, Python, C#!
- In C++ an object may have more than one parent (immediate superclass)
  - but not in Java, Python, C#!
- Java, Python, C# have a single, strict hierarchy

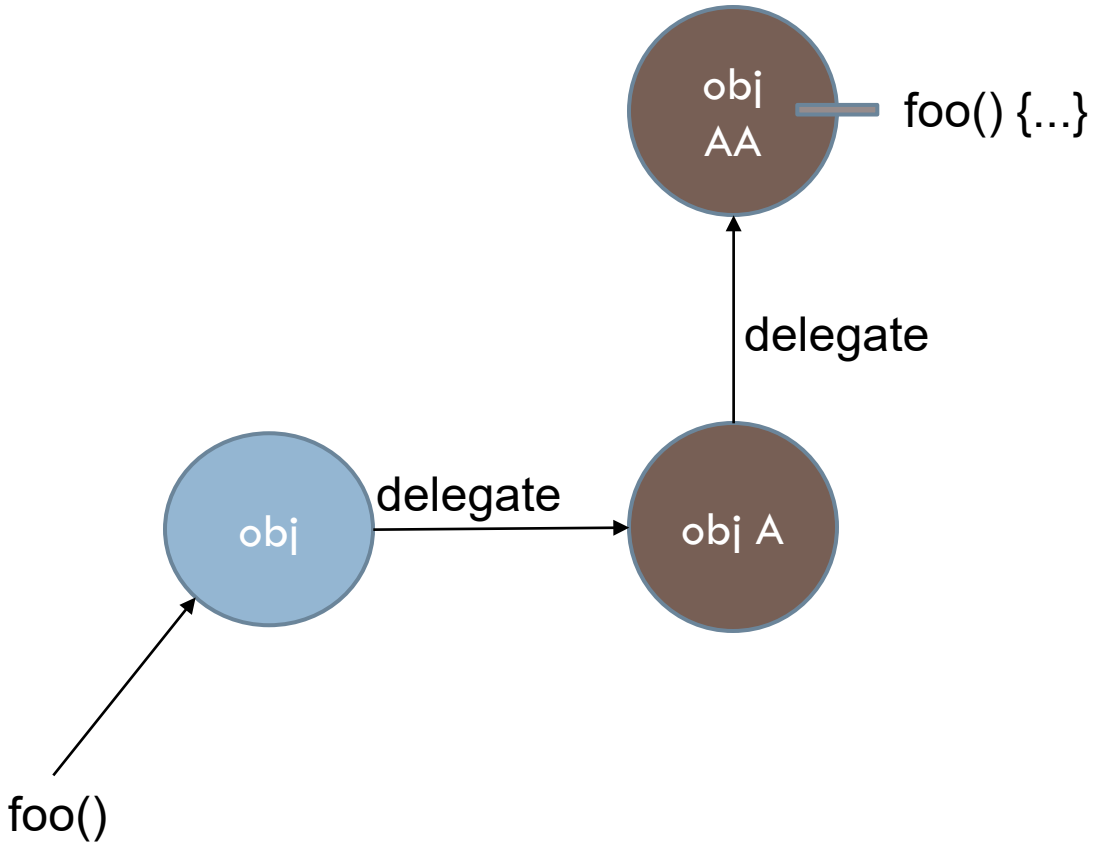
Mixins: mix-and-match without inheritance

```
class Vehicle(object):
    """A generic vehicle class."""

    def __init__(self, position):
        self.position = position

    def travel(self, destination):
        route = calculate_route(from=self.position, to=destination)
        self.move_along(route)

class Car(Vehicle):
    ...

class Boat(Vehicle):
    ...

class Plane(Vehicle):
    ...
```

Mixins: mix-and-match without inheritance

```
class RadioUserMixin(object):
    def __init__(self):
        self.radio = Radio()

    def play_song_on_station(self, station):
        self.radio.set_station(station)
        self.radio.play_song()

class Car(RadioUserMixin, Vehicle):
    ...

class Boat(RadioUserMixin, Vehicle):
    ...

class Plane(RadioUserMixin, Vehicle):
    ...
```

Delegation: runtime inheritance



E.G. Ruby

- Every object has a pointer to its class
- A class is represented by a “class object”
  - Every class object contains a hash table with method names and code
- Every class object has a pointer to its superclass
- Search for applicable methods starting in the object and moving up
  - If you hit the top without finding it, “message not understood”

Dynamic Dispatch

Types for O-O Languages

- Java, C++, and others are *strongly typed*
- Purpose of the type system: prevent certain kinds of runtime errors by compile-time checks (i.e., static analysis)

O-O Type Systems

- "Usual" guarantees
  - Program execution won't
    - Send a message that the receiver doesn't understand
    - Send a message with the wrong number of arguments
- "Usual" loophole
  - Type system doesn't try to guarantee that a reference is not null

Typing and Dynamic Dispatch

- The type system allows us to know in advance what methods exist in each class, and the potential type(s) of each object
  - Declared (static) type
  - Supertypes
  - Possible dynamic type(s) because of downcasts
- Use this to engineer fast dynamic type lookup

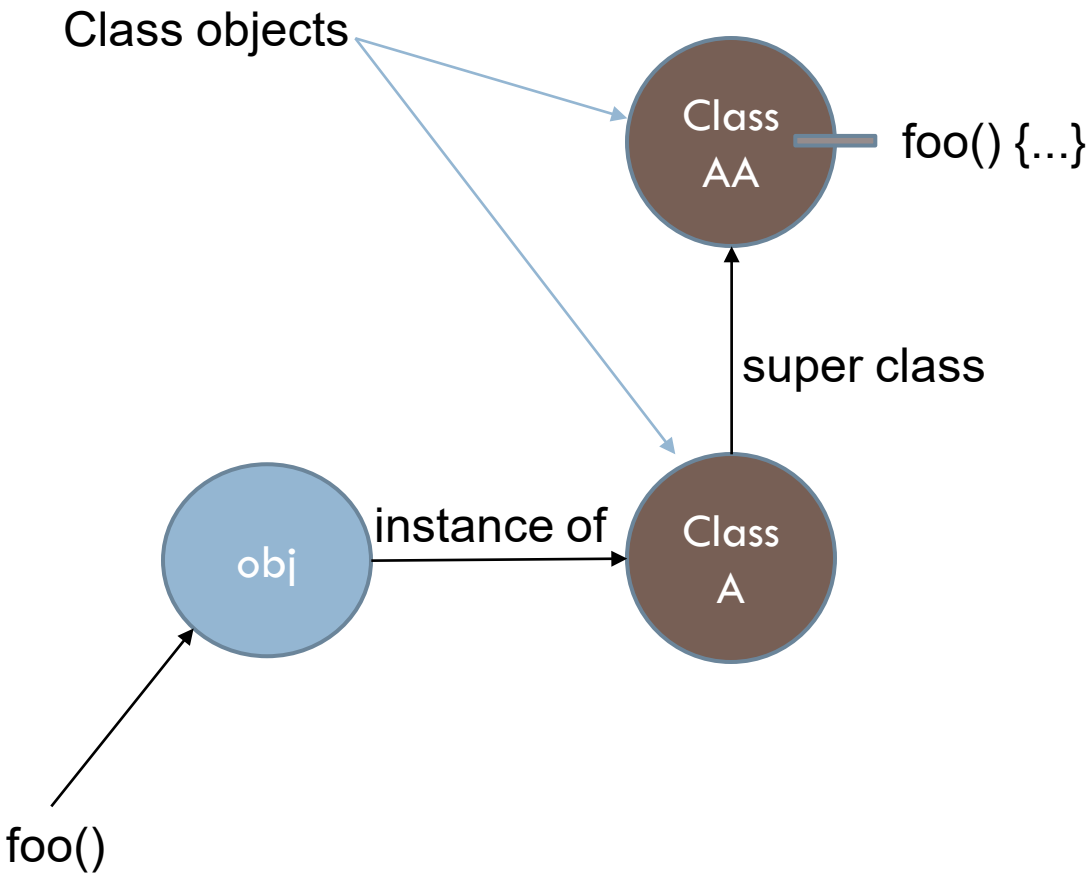
Object Layout

- Whenever we execute “new Thing(…)”
  - We know the class of Thing
  - We know what fields it contains (everything declared in Thing plus everything inherited)
- We can guarantee that the initial part of subclass objects matches the layout of ones in the superclass
  - So when we up- or down-cast, offsets of inherited fields don’t change

Per-Class Data Structures

- As in Ruby, an object contains a pointer to a per-class data structure
  - (But this need not be a first-class object in the language)
- Per-class data structure contains a table of pointers to appropriate methods
  - Often called “virtual function table” or vtable
  - Method calls are indirect through the object’s class’s vtable

Class as **repository of methods**



Vtables and Inheritance

- Key to making overriding work
  - Initial part of vtable for a subclass has the same layout as its superclass
    - So we can call a method indirectly through the vtable using a known offset fixed at compile-time *regardless of the actual dynamic type of the object*
  - Key point: offset of a method pointer is the same, but it can refer to a different method in the subclass, not the inherited one

Self Reference

Self reference: Java

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void print() {
      System.out.println(this.x + ":" + this.y);
    }
}
```

Self reference: Python

```
class Point():
    def __init__(self, x, y):
        self.x = x;
        self.y = y;

    def print(self):
      print(f'{self.x}:{self.y}')

p = Point(3, 4)
p.print()
```

- "self" is not a reserved keyword
- Python methods are class-level functions that take an object as first argument
- Object is passed implicitly, but received explicitly
- p.print() ➔ Point.print(p)

Self reference: JavaScript

```
var apple = new Apple('macintosh');
apple.color = "reddish";
apple.getInfo();
```

var context = { this : apple };
getInfo.call(context)