Multiple Inheritance and Automated Delegation

Method lookup (binding)

```
// Square inherits from Rectangle
Square s = new Square();
s.set_width(12); // Meaning is obvious
Rectangle r = s; // substitutability!
r.set_width(12);
```

- Dynamic lookup: Square's method is called.
- Static lookup: Rectangle's method is called

- Java: Dynamic Lookup
- C++: `virtual` keyword
- (Liskov) Substitution Principle

Deferred implementations

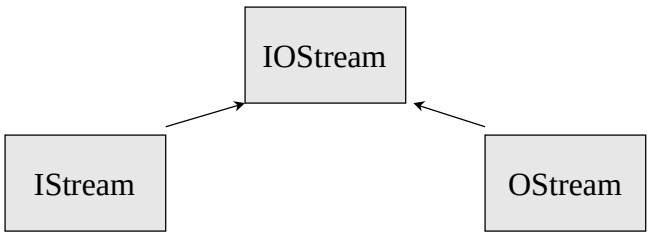
```
class Widget {
   // Bounding box information
   int height, width, xpos, ypos;
   void set_height(int height)
    …
   abstract void draw();
}
```

Then derive classes Menu, Window, Button, etc..
Derived class must define `draw` to be instantiable.

In C++:

```
virtual void draw() = 0;
```

Multiple Inheritance



Multiple Inheritance

```
class FileIStream:
   def __init__(self, filename):
      self.input_file = open(filename, “r”)
   def read(self, numbytes):
      return self.input_file.read(numbytes)
   def close(self): self.input_file.close()
class FileOStream:
   def __init__(self, filename):
      self.output_file = open(filename, “w”)
   def write(self, data):
      self.output_file.write(data)
   def close(self): self.output_file.close()
```

`self` (`this` in C++) is implicit in most languages.

Multiple Inheritance example

```
class FileIOStream(FileIStream, FileOStream):
   def __init__(self, filename):
      FileIStream.__init__(self, filename)
      FileOStream.__init__(self, filename)

class FileIOStream(FileIStream, FileOStream):
   def __init__(self, filename):
      self.file = open(filename, “rw”)
      self.input_file = self.file
      self.output_file = self.file
```

- Address base class, not the object for ambiguity resolution.
- Note the passing of `self`.

Terminology

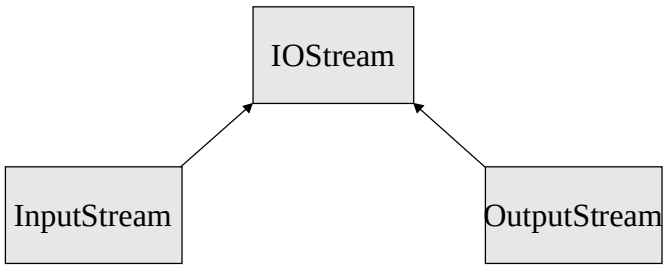
- **Subclassing** – Derivation of methods and variables.
- **Subtyping** – Derivation of types.
- **Specialization** – “Is a kind of” relationship.
- **Inheritance** – Subclassing + subtyping, intended for specialization.
- **Delegation** – Forwarding requests to an instantiated object.

Multiple Inheritance

- Q: If MI is so bad, why do people use it?
- A: It has its advantages
- The things MI does well, a replacement should try to do well.
- It should also avoid the shortcomings

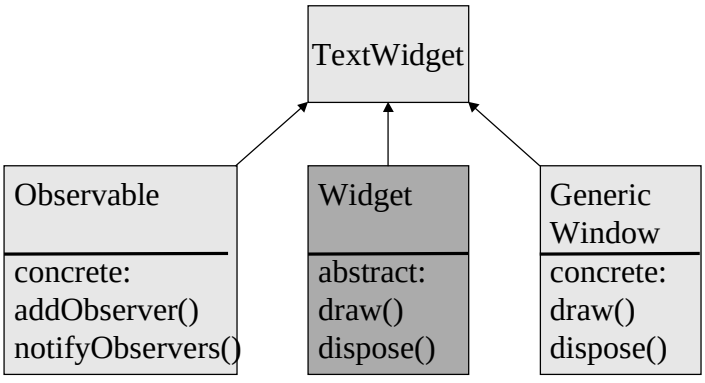
Pro 1: Multiple Specialization

- Two IS-A relationships



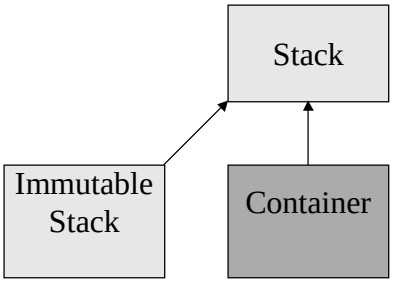
Pro 2: Mixin Inheritance

Attribute / functionality encapsulation

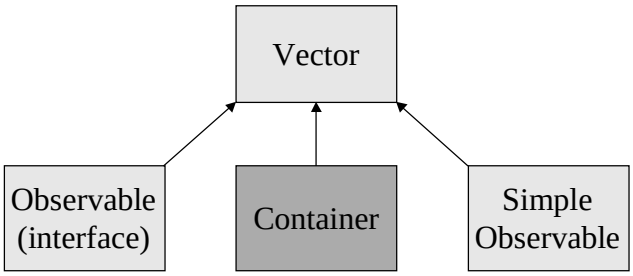


Pro 3: Multiple Subtyping

- Interface Segregation Principle: Wide interface? Provide narrow ones
- Not all clients need mutablity

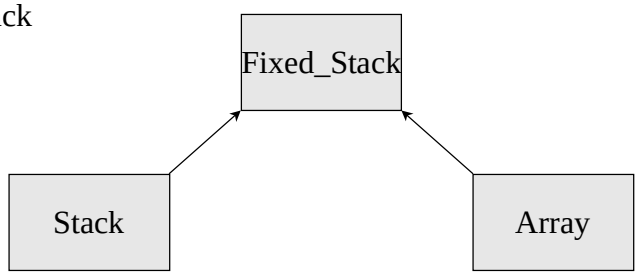


Pro 4: Pairing Interfaces and Implementations



Pro 5 / Con 1: Implementation Inheritance

Example: fixed stack



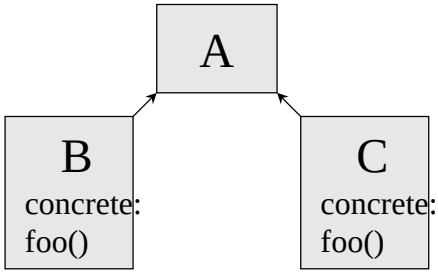
`Stack` deferred class: `empty, append, pop`
`Array` implementation: `empty, append, remove`
Copy `Array`'s implementation

Con 2: Misuse

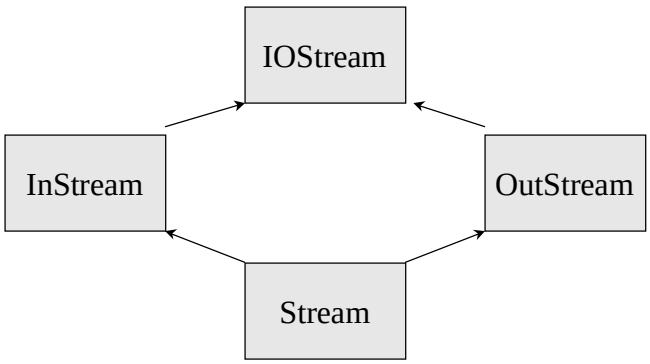
- Inheritance without specialization
- Implementation inheritance
- Facility inheritance

Con 3: Name conflicts

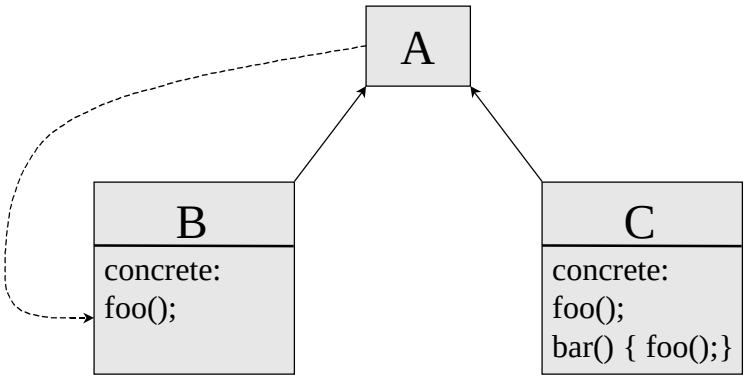
- Throw a compile error
- Require explicit addressing:
  `FileIStream.close()`
- Pick one
- Require renaming
  `rename FileOStream.close to unused_close`



Con 4: Repeated Inheritance



Con 5: Obscurity



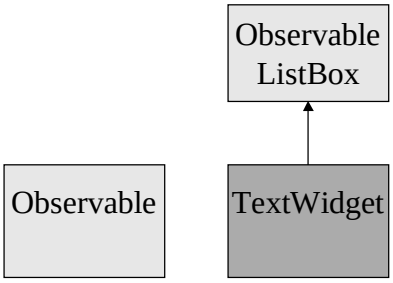
Interfaces

- Types without implementation

```
interface Cloneable(){
  void copy();
}
public class Vector implements Cloneable, Serializable {
  …
}
Incapapable of subclassing
```

Copying Schemes

- Copy code from one class into another



- Error prone, little reuse

Reference Passing

- Return a reference to an object
- E.g., ObservableTextWidget → getObservable()
- This solution isn't even subclassing!

Delegation

- Instantiate an object
- Forward methods to it

```
    boolean protect(Object x) throws InvalidObject {
      return myArmor.protect(x);
    }
  }
```

- Useful, but tedious and error prone

Automated Delegation

- Automatically generate delegation code.
- Syntax:

```
class C forwards T to tvar, X to xvar {
    U tvar;  Y xvar;
```

Where:

- U is a subtype of T (can be the same)
- T can be an interface or a class

Exclusion

- Declare interfaces to exclude from delegation:

```
class C extends B
 forwards T without S1, S2 to a
```

- Alleviate name conflicts
- Doesn't affect substitutability

Accessing the Delegating Class

- Forwarder keyword to access delegator
- Allow for type safety

```
class Delegate ... forwarder implements X {
```

- In Jamie, type safety doesn't always apply: Limitation of Java's type system

Analysis: Pros

- Addresses MI's drawbacks
  - name conflicts, repeated inheritance, misuse, obscurity
- Keeps the advantages
- Promotes black box reuse
- Good abstraction for non-specializing relationships
- Dynamic subclassing

Analysis: Cons

- Doesn't handle multiple specialization well
- Not as efficient as MI

Classless languages

- Objects only, no classes
- Failings of the class model:
  - All class instances have identical representations
  - Representation must include superclass repr.
  - Class hierarchy and instance hierarchy intertwined
- Delegation instead of inheritance
  - Subclasses and subtypes