ECE 462
Object-Oriented Programming using C++ and Java

Inheritance and Polymorphism

A little terminology - methods and functions

- Methods are any function that is declared within a class and can access class information
- All functions in Java are methods. As we will see later this is not true for C++.

Derived classes and terminology

- When class *X* ISA *Y* we say *X is a derived class of Y*, and *Y is the base class of X*
- A class may have multiple derived classes:
  - Car: Sedan, Truck, Sport Utility Vehicle, Sport Car ...
  - Computer: Laptop, Desktop, Server
- A derived class may also have derived classes:
  - Vehicle: Car, Bike ... Car: Sedan, Truck ...

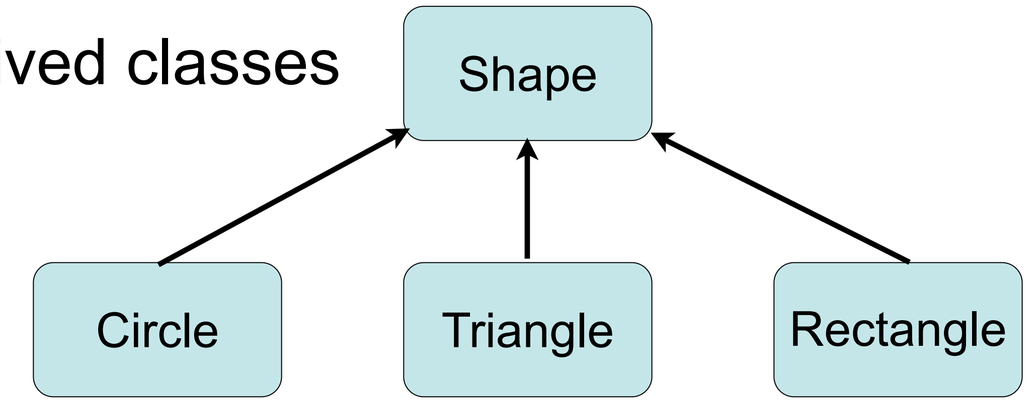
    Animal: Bird, Mammal ... Mammal: Dog, Cat ...
- Vehicle is the immediate base class of Car, and is a base class of Sedan
- Mammal is the immediate base class of Cat and Animal is a base class of Cat.
- We will use "base" and "derived" class. ***Do not*** use "super" and "sub" class.
  - A base class or a superclass is "smaller" (fewer attributes and behaviors) and seems like a subclass, but has more objects, which is like a super class
  - I, and many other people, find this hard to remember

Summary - Why Object-Oriented?

- Object-oriented programming (OOP) is a more natural way to describe the interactions between "things" (i.e. objects).
- OOP provides better code reuse:
  - commonalities among objects described by a class
  - commonalities among classes described by a base class (inheritance)
- Objects know what to do using their attributes and actions:
  - Each object responds differently to "What is your name?"
- OOP provides encapsulation
  - Objects hide data that are should not be visible to other objects or protect data from unintentional, inconsistent changes.
  - This allows changes to be made to the internals of objects and not affect code outside the object.

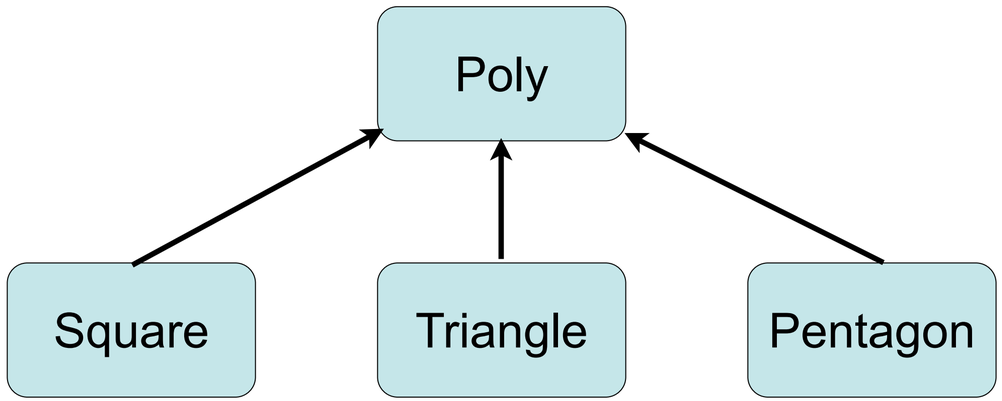
Interface ≠ Implementation

- If a behavior is common among classes, the behavior should be available in their base class.
- This behavior may need additional information from derived classes and must be handled in derived classes.
  - Shape: contains color, lineStyle ... attributes
  - Shape *supports* getArea behavior (using an approximating function)
  - getArea cannot be efficiently or precisely *implemented* by Shape since different shapes have different area formulas
  - Shape may want getArea to be handled by each derived classes
  - Shape can force getArea to be implemented in derived classes
  - Java and C++ provide a way to do this

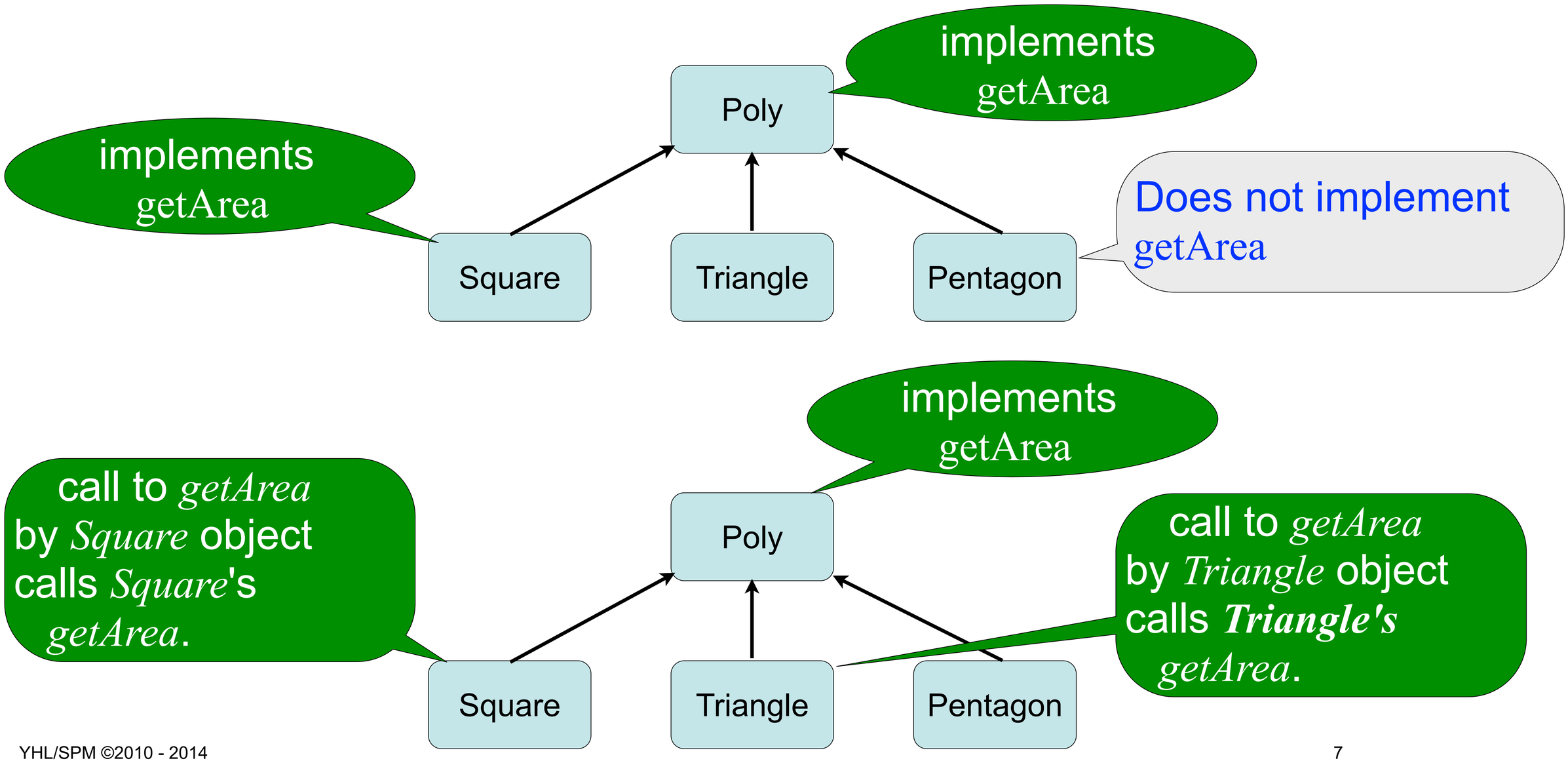


Override Behavior

- Poly can also support getArea.
- Derived classes (such as Triangle, Square, and Pentagon) **may** have better (faster) ways to getArea than Polygon.
- getArea is implemented in Poly and optionally in its derived classes.
- A Poly object calls getArea in Poly
- A Square object calls getArea in Square ***if*** getArea ***is*** implemented in Square.
- But, a Pentagon object calls getArea in Poly ***if*** getArea ***is not*** implemented in Pentagon.



Override Behavior



Overriding a function (method) M

Base	Derived	Object	Execute
Y	Y	Base	Base M
Y	Y	Derived	Derived M
Y	N	Base	Base M
Y	N	Derived	Base M
N	Y	Base	Error
N	Y	Derived	Derived M
N	N	Base	Error
N	N	Derived	Error

The behavior implemented in a sibling class (such as Square-Triangle) has no effect.

Y means M is implemented by the class.
N means M is not implemented by the class.

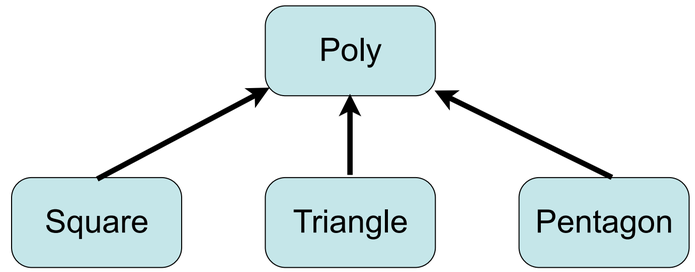
Class, Object and Polymorphism -- general OO

Poly p1; // Let *p1* be a ~~reference to~~ Poly object

p1.getArea( ); // call the implementation of *getArea* used by *Poly*

Square s2; // *s2* is a reference to a *Square* object

p1 = s2; // *p1* behaves likes a *Square* - ***polymorphism!***



a *Square* object acts like both a poly (kinds of variables it can be assigned to) and square object (methods and fields it has)

p1.getArea(); // implementation in *Square* (if available) is called

illegal attempt at polymorphism follows

s2 = p1; // error

// a *Poly* object IS NOT A *Square* object

Emphasis: This is general OO, not C++ or Java

p1 = s2 is not an example of type conversion!

Emphasis: This is general OO, not C++ or Java

Square s2;	// *s2* is a *Square* object
p1 = s2;	// *p1* behaves likes a *Square* - ***polymorphism!***
a *Square* object acts like both a *Poly* (kinds of variables it can be assigned to) and *Square* object (methods and fields it has)	

What is happening here is fundamentally different than:

```
int i = 0;
double f = 4.0
f = i;
```

In the C code to the left, the *int 0* is being converted to a *float 0* and assigned to the *float* variable *f*.

In the blue code above, an unconverted actual *Square* object is being assigned to a variable of type *Poly*.

Let's look at a Poly, etc., class in Java

```
// get access to math routines
import java.lang.Math;

public class Poly {
        private int n; // number of sides
        private double s; // length of side

    public Poly(int fn, double fs) {
            n = fn;
            s = fs;
    }

    public String toString( ) {
            return n+" "+s;
    }

    public double getLenSides( ) {
       return s;
    }

    public double getArea( ) {
       System.out.println(" poly area");
       return (s*s*n)/(4*Math.tan(Math.PI/n));
    }
}
```

Poly.java

```
// get access to math routines
import java.lang.Math;

public class Poly {
        private int n; // number of sides
        private double s; // length of side

        public Poly(int fn, double fs) {
                n = fn;
                s = fs;
        }

        public String toString( ) {
                return n+" "+s;
        }

        public double getLenSides( ) {
           return s;
        }

        public double getArea( ) {
           System.out.println(" poly area");
           return (s*s*n)/(4*Math.tan(Math.PI/n));
        }
}
```

Poly.java

```
// get access to math routines
import java.lang.Math;

public class Poly {
        private int n; // number of sides
        private double s; // length of side

        public Poly(int fn, double fs) {
                n = fn;
                s = fs;
        }

        public String toString( ) {
                return n+" "+s;
        }

        public double getLenSides( ) {
           return s;
        }

        public double getArea( ) {
           System.out.println(" poly area");
           return (s*s*n)/(4*Math.tan(Math.PI/n));
        }
}
```

```
private int n;
private double s;
Poly(int, double)
ToString( )
getArea( );
```

Poly.java

```
// get access to math routines
import java.lang.Math;

public class Poly {
        private int n; // number of sides
        private double s; // length of side

        public Poly(int fn, double fs) {
                n = fn;
                s = fs;
        }

        public String toString( ) {
                return n+" "+s;
        }

        public double getLenSides( ) {
           return s;
        }

        public double getArea( ) {
           System.out.println(" poly area");
           return (s*s*n)/(4*Math.tan(Math.PI/n));
                }
        }
```

Red declares variables to hold the state of a Poly object.
Green defines a *constructor* of a poly object. Used to define the initial state of the object when it is formed.
Blue are *methods* that defines the actions of a Poly object

Poly.java

```
// get access to math routines
import java.lang.Math;

public class Poly {
        private int n; // number of sides
        private double s; // length of side

        public Poly(int fn, double fs) {
                n = fn;
                s = fs;
        }

        public String toString( ) {
                return n+" "+s;
        }

        public double getLenSides( ) {
           return s;
        }

        public double getArea( ) {
           System.out.println(" poly area");
           return (s*s*n)/(4*Math.tan(Math.PI/n));
        }
}
```

Poly.java

```
// get access to math routines
import java.lang.Math;

public class Poly {
        private int n; // number of sides
        private double s; // length of side

        public Poly(int fn, double fs) {
                n = fn;
                s = fs;
        }

        public String toString( ) {
                return n+" "+s;
        }

        public double getLenSides( ) {
           return s;
        }

        public double getArea( ) {
           System.out.println(" poly area");
           return (s*s*n)/(4*Math.tan(Math.PI/n));
        }
}
```

Poly.java

```
// get access to math routines
import java.lang.Math;

public class Poly {
        private int n; // number of sides
        private double s; // length of side

        public Poly(int fn, double fs) {
                n = fn;
                s = fs;
        }

        public String toString( ) {
                return n+" "+s;
        }

        public double getLenSides( ) {
           return s;
        }

        public double getArea( ) {
           System.out.println(" poly area");
           return (s*s*n)/(4*Math.tan(Math.PI/n));
        }
}
```

Square.java

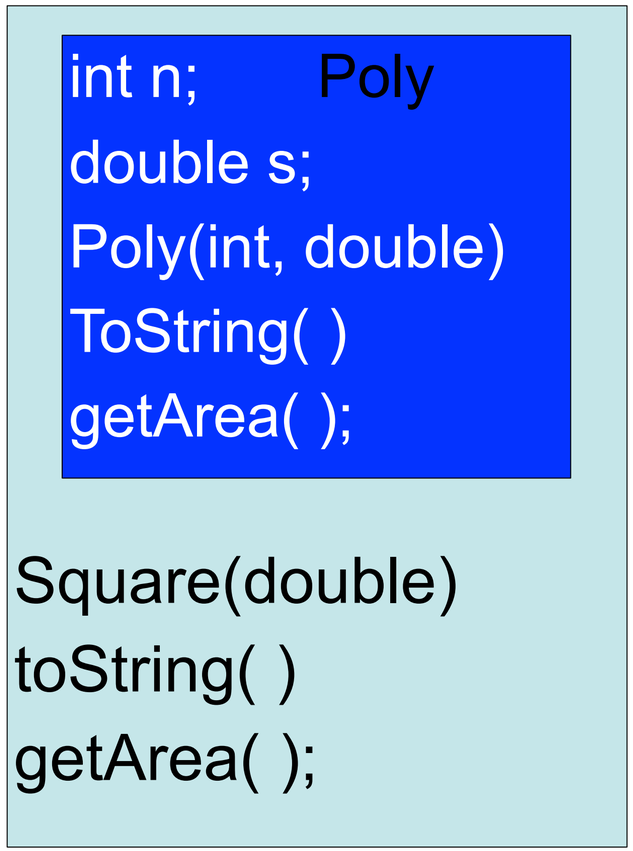
```
public class Square extends Poly {

        public Square(double fs) {
                super(4,fs);
        }

        public String toString( ) {
                return "4 "+getLenSides( );
        }

        public double getArea( ) {
           System.out.println(" square area");
           return getLenSides( )*getLenSides( );
        }
}
```

Square.java

```
public class Square extends Poly {

        public Square(double fs) {
                super(4,fs);
        }

        public String toString( ) {
                return "4 "+getLenSides( );
        }

        public double getArea( ) {
           System.out.println(" square area");
           return getLenSides( )*getLenSides( );
        }
}
```

Square.java

```
public class Square extends Poly {

        public Square(double fs) {
              super(4,fs);
        }

        public String toString( ) {
              return "4 "+getLenSides( );
        }

        public double getArea( ) {
           System.out.println(" square area");
           return getLenSides( )*getLenSides( );
        }
}
```



Square.java

```
public class Square extends Poly {

        public Square(double fs) {
                super(4,fs); // what if no base class constructor called here?
                             // when is base class constructor called?
        }

        public String toString( ) {
                return "4 "+getLenSides( );
        }

        public double getArea( ) {
           System.out.println(" square area");
           return getLenSides( )*getLenSides( );
        }
}
```

Square.java

```
public class Square extends Poly {

        public Square(double fs) {
                super(4,fs);
        }

        public String toString( ) {
                return "4 "+getLenSides( );
        }

        public double getArea( ) {
           System.out.println(" square area");
           return getLenSides( )*getLenSides( );
        }
}
```

Square.java

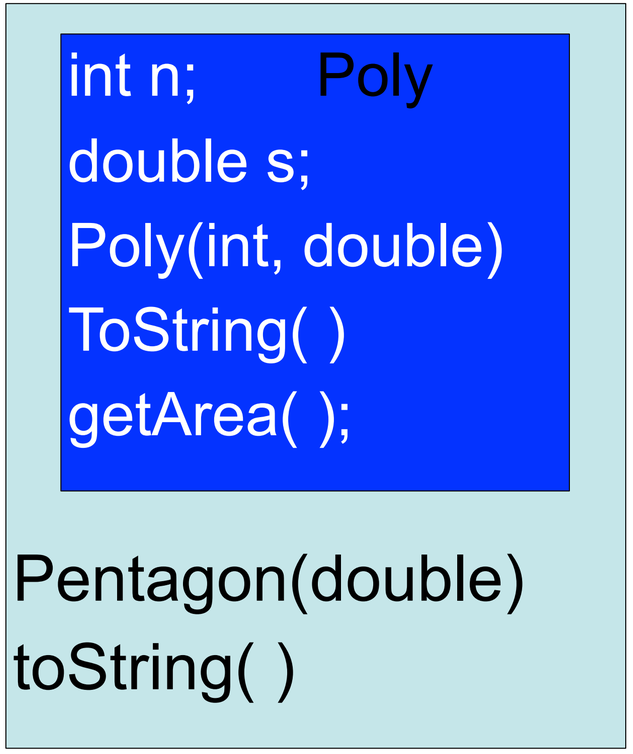
```
public class Square extends Poly {

        public Square(double fs) {
                super(4,fs);
        }

        public String toString( ) {
                return "4 "+getLenSides( );
        }

        public double getArea( ) {
           System.out.println(" square area");
           return getLenSides( )*getLenSides( );
        }
}
```

Pentagon.java

```
public class Pentagon extends Poly {

        public Pentagon(double fs) {
                super(5, fs);
        }

                public String toString( ) {
                        return "Pentagon with side of length"+getLenSides( );
                }
        }
```

Pentagon.java

```
public class Pentagon extends Poly {

        public Pentagon(double fs) {
                super(5,fs);
        }

        public String toString( ) {
                return "Pentagon with side of length"+getLenSides( );
        }
}
```



Pentagon.java

```
public class Pentagon extends Poly {

        public Pentagon(double fs) {
                super(5,fs);
        }

                public String toString( ) {
                        return "Pentagon with side of length"+getLenSides( );
                }
        }
```

Test.java -- contains the *main* function

```
public class Test {

  public static void main(String[] args) {

      Poly p1 = new Poly(6, 2.0);
      Square s1 = new Square(2.0);
      Pentagon pe1 = new Pentagon(2.0);

      System.out.println("Poly p1 is "+p1+", area is "+p1.getArea( ));
      System.out.println(" ");

      System.out.println("Square s1 is "+s1+", area is "+s1.getArea( ));
      System.out.println(" ");

      System.out.println("Pentagon pe1 is "+pe1+", area is "+pe1.getArea( ));
      System.out.println(" ");

      p1 = s1;
      System.out.println("Poly p1 is "+p1+", area is "+p1.getArea( ));
    }
}
```

Test.java -- contains the *main* function

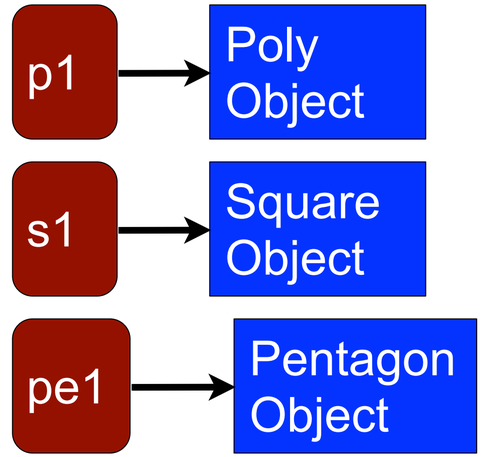
```
public class Test {

  public static void main(String[] args) {

      Poly p1 = new Poly(6, 2.0);
      Square s1 = new Square(2.0);
      Pentagon pe1 = new Pentagon(2.0);

      System.out.println("Poly p1 is "+p1+", area is "+p1.getArea( ));
      System.out.println(" ");

      System.out.println("Square s1 is "+s1+", area is "+s1.getArea( ));
      System.out.println(" ");

      System.out.println("Pentagon pe1 is "+pe1+", area is "+pe1.getArea( ));
      System.out.println(" ");

      p1 = s1;
      System.out.println("Poly p1 is "+p1+", area is "+p1.getArea( ));
    }
}
```

Test.java -- contains the *main* function

```
public class Test {

  public static void main(String[] args) {

      Poly p1 = new Poly(6, 2.0);
      Square s1 = new Square(2.0);
      Pentagon pe1 = new Pentagon(2.0);

      System.out.println("Poly p1 is "+p1+", area is "+p1.getArea( ));
      System.out.println(" ");

      System.out.println("Square s1 is "+s1+", area is "+s1.getArea( ));
      System.out.println(" ");

      System.out.println("Pentagon pe1 is "+pe1+", area is "+pe1.getArea( ));
      System.out.println(" ");

      p1 = s1;
      System.out.println("Poly p1 is "+p1+", area is "+p1.getArea( ));
    }
}
```

Test.java -- contains the *main* function

```
public class Test {

  public static void main(String[] args) {

      Poly p1 = new Poly(6, 2.0);
      Square s1 = new Square(2.0);
      Pentagon pe1 = new Pentagon(2.0);

      System.out.println("Poly p1 is "+p1+", area is "+p1.getArea( ));
      System.out.println(" ");

      System.out.println("Square s1 is "+s1+", area is "+s1.getArea( ));
      System.out.println(" ");

      System.out.println("Pentagon pe1 is "+pe1+", area is "+pe1.getArea( ));
      System.out.println(" ");

      p1 = s1;
      System.out.println("Poly p1 is "+p1+", area is "+p1.getArea( ));
    }
}
```



Test.java

```
public class Test {

  public static void main(String[] args) {
. . .

      System.out.println("Poly p1 is "+p1+", area is "+p1.getArea( ));
      System.out.println(" ");
```

poly area
Poly p1 is 6 2.0, area is 10.392304845413264

```
      System.out.println("Square s1 is "+s1+", area is "+s1.getArea( ));
      System.out.println(" ");
```

square area
Square s1 is 4 2.0, area is 4.0

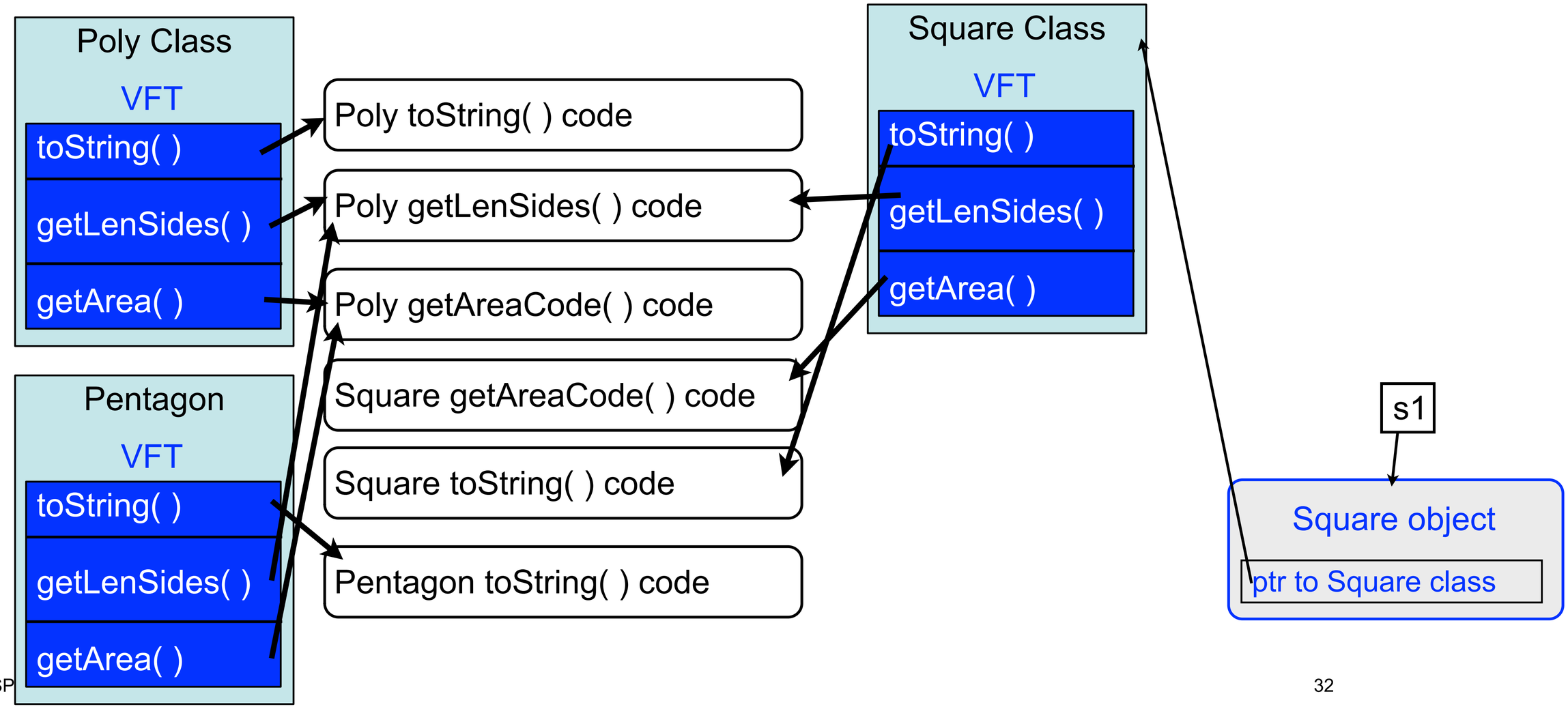
```
      System.out.println("Pentagon pe1 is "+pe1+", area is "+pe1.getArea( ));
      System.out.println(" ");
```

poly area
Pentagon pe1 is Pentagon with side of length 2.0, area is 6.881909602355868

```
      p1 = s1;
      System.out.println("Poly p1 is "+p1+", area is "+p1.getArea( ));
```

???????? area
Poly p1 is 4 2.0, area is 4.0

How is it known which *getArea* to call?
Virtual Function Tables



Another Polymorphism example

```
import java.io.*;

public class Foo {

   private final String fooString;

   public Foo( ) {fooString = null;}
   public Foo(String ln) {fooString = ln;}
   public void print( ) {System.out.println("Foo: "+fooString);}
}
```

```
import java.io.*;

public class DFoo extends Foo {

   private final String dfooString;

   public DFoo(String ln) {dfooString = ln;}
   public void print( ) {System.out.println("DFoo: "+dfooString);}
}
```

```
import java.io.*;

class Test {

   public static void main(String args[ ]) {
      Foo f = new Foo("a new foo");
      f.print( );

      DFoo d = new DFoo("a new dfoo");
      d.print( );

      ((Foo) d).print( );

      f = d;
      f.print( );
   }
}
```

From java/baseDerived/

```
import java.io.*;

class Test {

   public static void main(String args[]) {
      Foo f = new Foo("Foo object");
      f.print( );

      DFoo d = new DFoo("DFoo object");
      d.print( );

      ((Foo) d).print( );

      f = d;
      f.print( );
   }
}
```

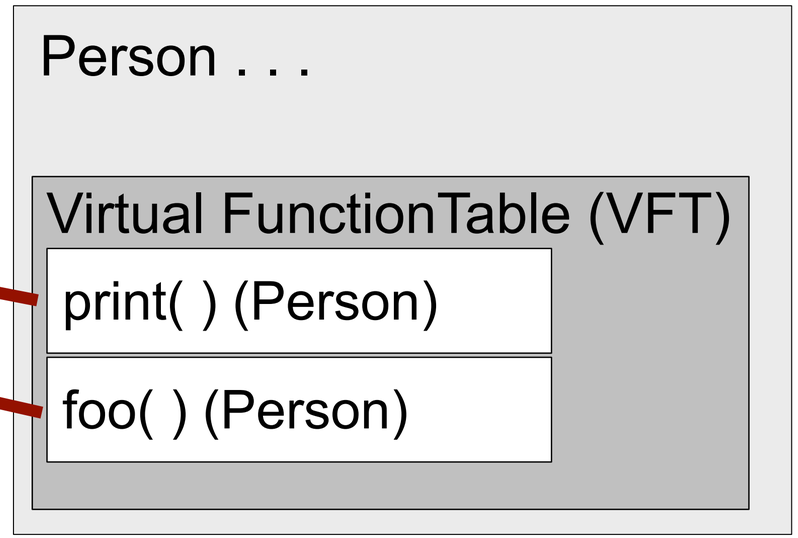
```
smidkiff% javac Test.java
smidkiff% java Test
Foo: Foo object
DFoo: DFoo object
DFoo: DFoo object
DFoo: DFoo object
[ece-76-55:code/java/baseDerived] smidkiff%
```

The class of the object on which the method is invoked is the class whose methods are called

Another Virtual Function Table (VFT) Example

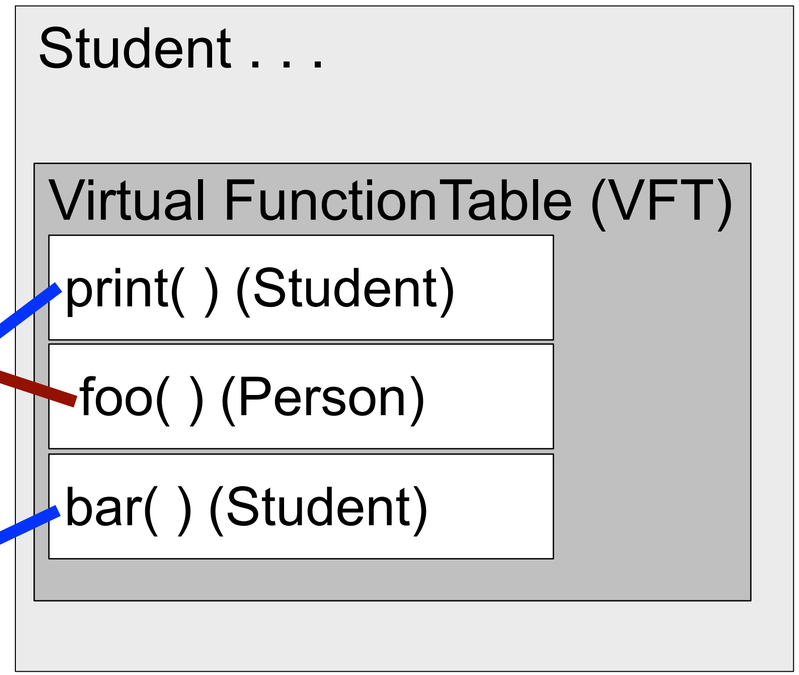
```
public class Person {
        final String p_lastName;
        final String p_firstName;

        public Person(String ln, String fn) {. . .}
        public void print( ) {. . .}
        public void foo( ) {. . .}
}
public class Student extends Person {
        String s_school;
        String s_major;

        public Student(. . .) {. . .}
        public void print( ) {. . .}
        public void bar( ) {. . .}
}
```

Person class VFT



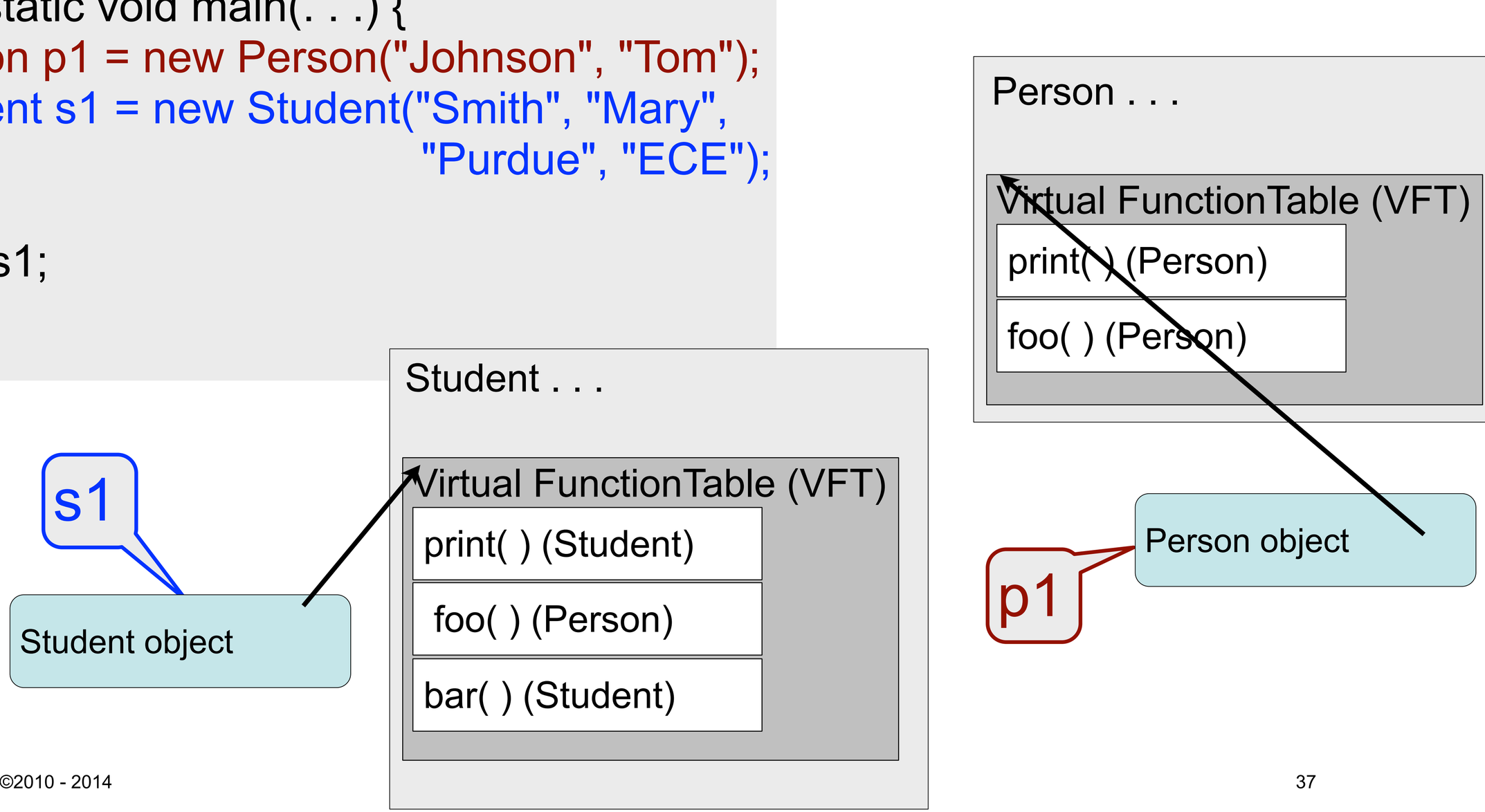
The Student class VFT

```
public class Person {
        final String p_lastName;
        final String p_firstName;

        public Person(String ln, String fn) {. . .}
        public void print( ) {. . .}
        public void foo( ) {. . .}
}
public class Student extends Person {
        String s_school;
        String s_major;

        public Student(. . .) {. . .}
        public void print( ) {. . .}
        public void bar( ) {. . .}
}
```



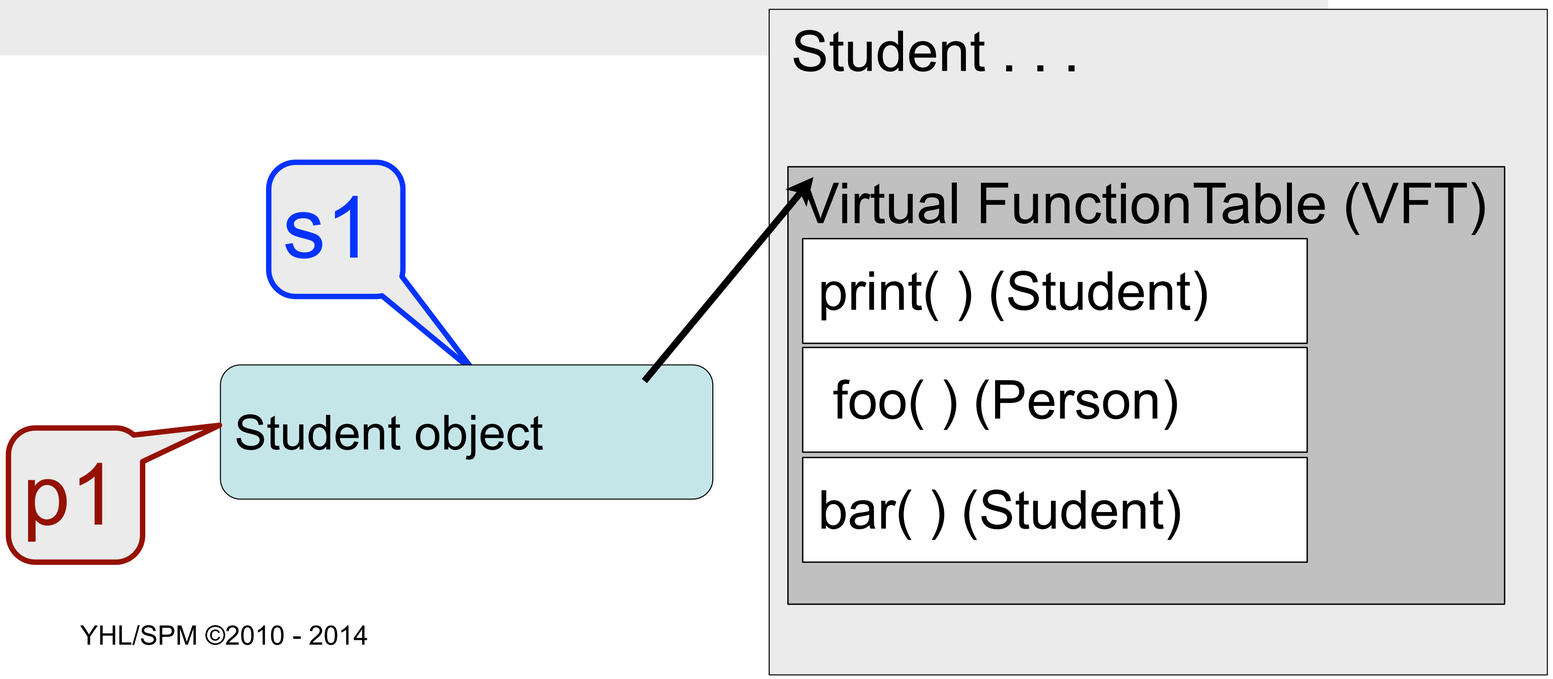
How the VFT enables polymorphic behavior

```
public static void main(. . .) {
   Person p1 = new Person("Johnson", "Tom");
   Student s1 = new Student("Smith", "Mary",
                                    "Purdue", "ECE");
   . . .
   p1 = s1;
   . . .
}
```



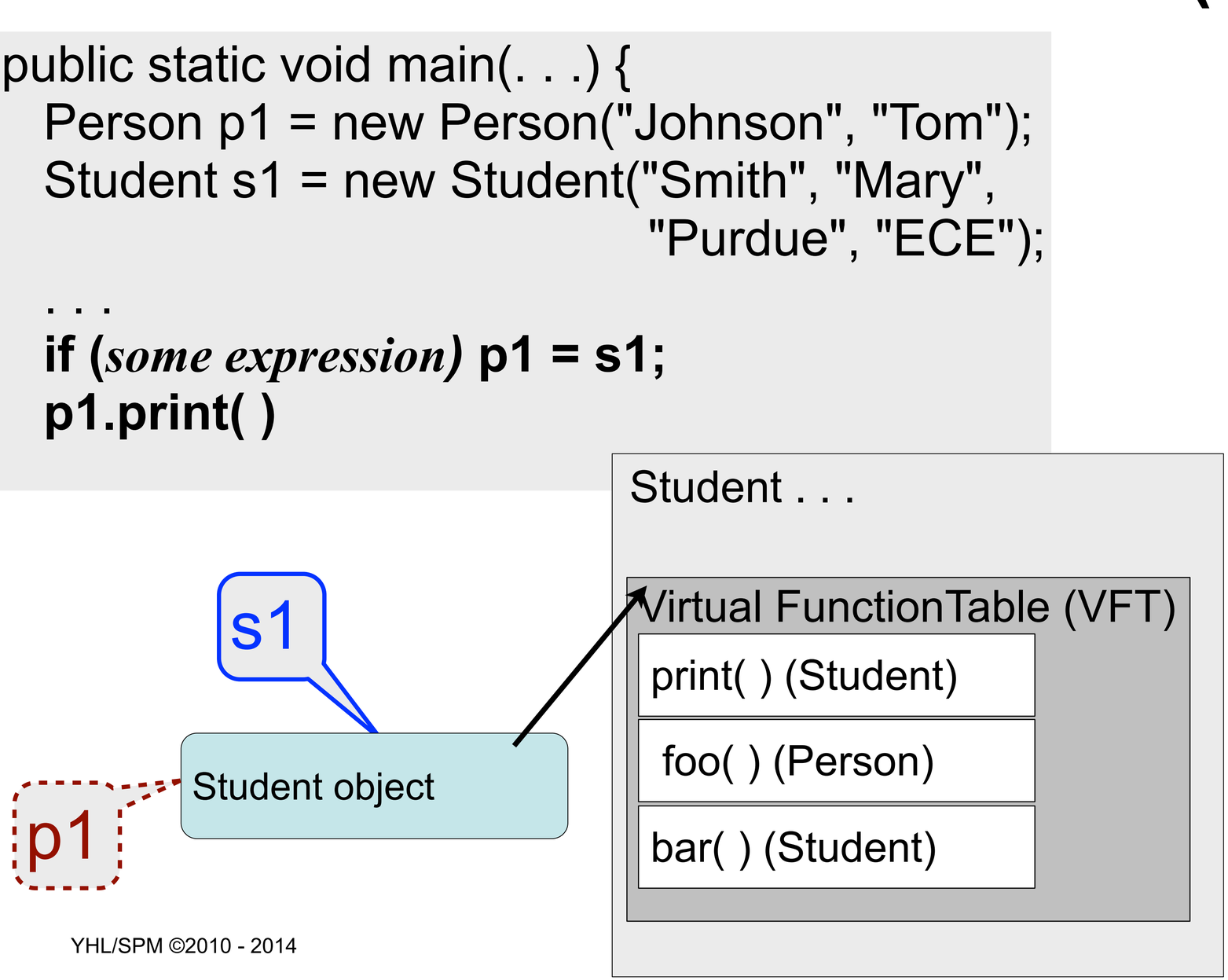
This will show polymorphic behavior

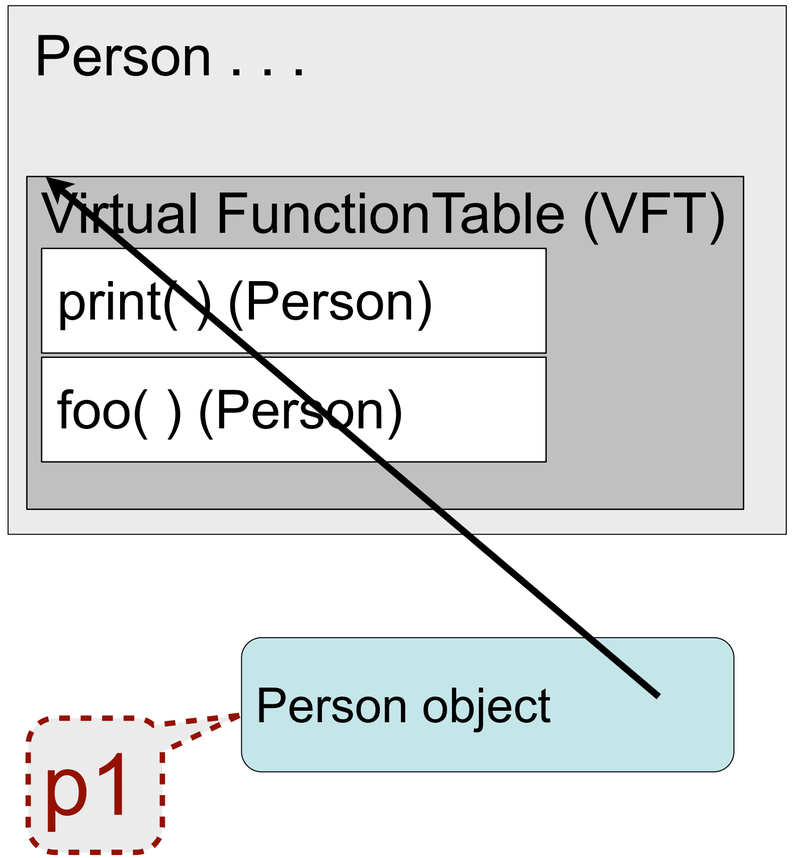
```
public static void main(. . .) {
   Person p1 = new Person("Johnson", "Tom");
   Student s1 = new Student("Smith", "Mary",
                                    "Purdue", "ECE");
   . . .
   p1 = s1;
   p1.print( )
```



What about this case? (new slide)

```
public static void main(. . .) {
   Person p1 = new Person("Johnson", "Tom");
   Student s1 = new Student("Smith", "Mary",
                                    "Purdue", "ECE");
   . . .
   if (some expression) p1 = s1;
   p1.print( )
```





Another example of how polymorphism is implemented - example code

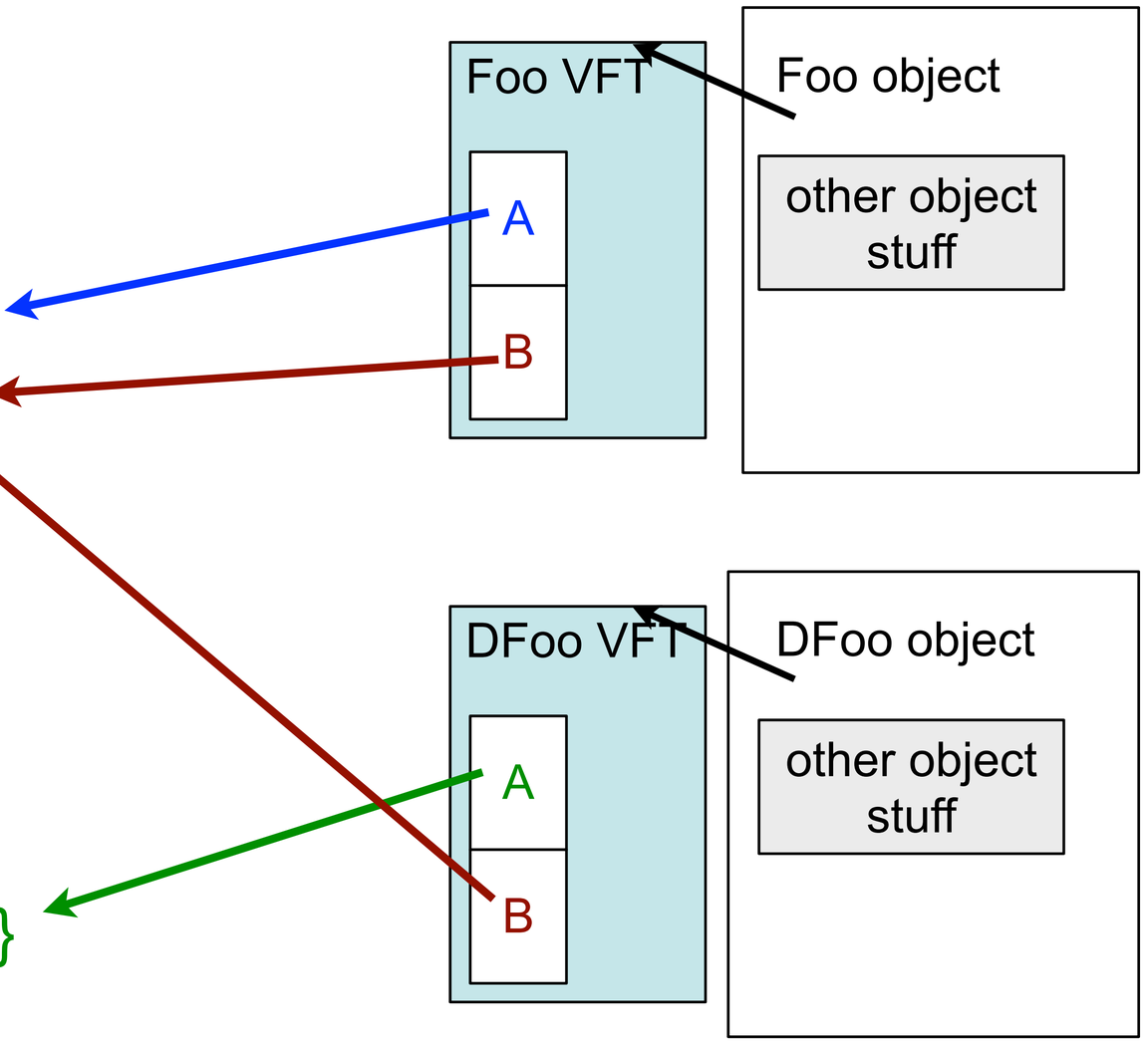
```
import java.io.*;

public class Foo {

   private final String fooString;

   public Foo( ) {fooString = null;}
   public Foo(String ln) {fooString = ln;}
   public void A( ) {System.out.println("fA");}
   public void B( ) {System.out.println("fB");
}
```

```
import java.io.*;

public class DFoo extends Foo {

   private final String dfooString;

   public DFoo(String ln) {dfooString = ln;}
   public void A( ) {System.out.println("dA");}
}
```

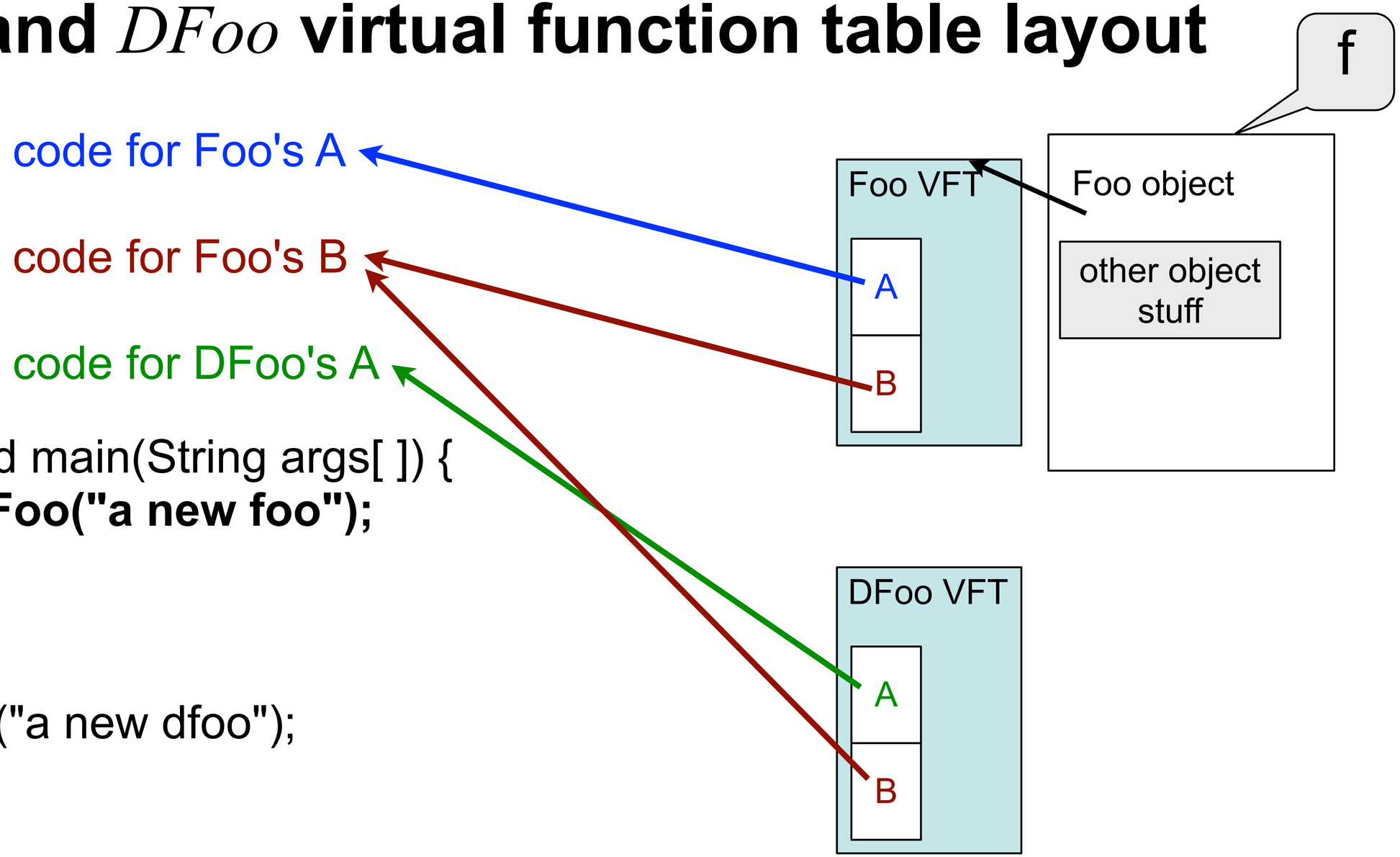
```
import java.io.*;

class Test {

   public static void main(String args[ ]) {
      Foo f = new Foo("a new foo");
      f.A( );
      f.B( );

      f = new DFoo("a new dfoo");
      f.A( );
      f.B( );
   }
}
```

Foo and *DFoo* virtual function table layout

```
public class Foo {
   private final String fooString;

   public Foo( ) {fooString = null;}
   public Foo(String In) {fooString = In;}
   public void A( ) {System.out.println("fA");}
   public void B( ) {System.out.println("fB");
}
 public class DFoo extends Foo {

    private final String dfooString;

    public DFoo(String In) {dfooString = In;}
    public void A( ) {System.out.println("dA");}
 }
```

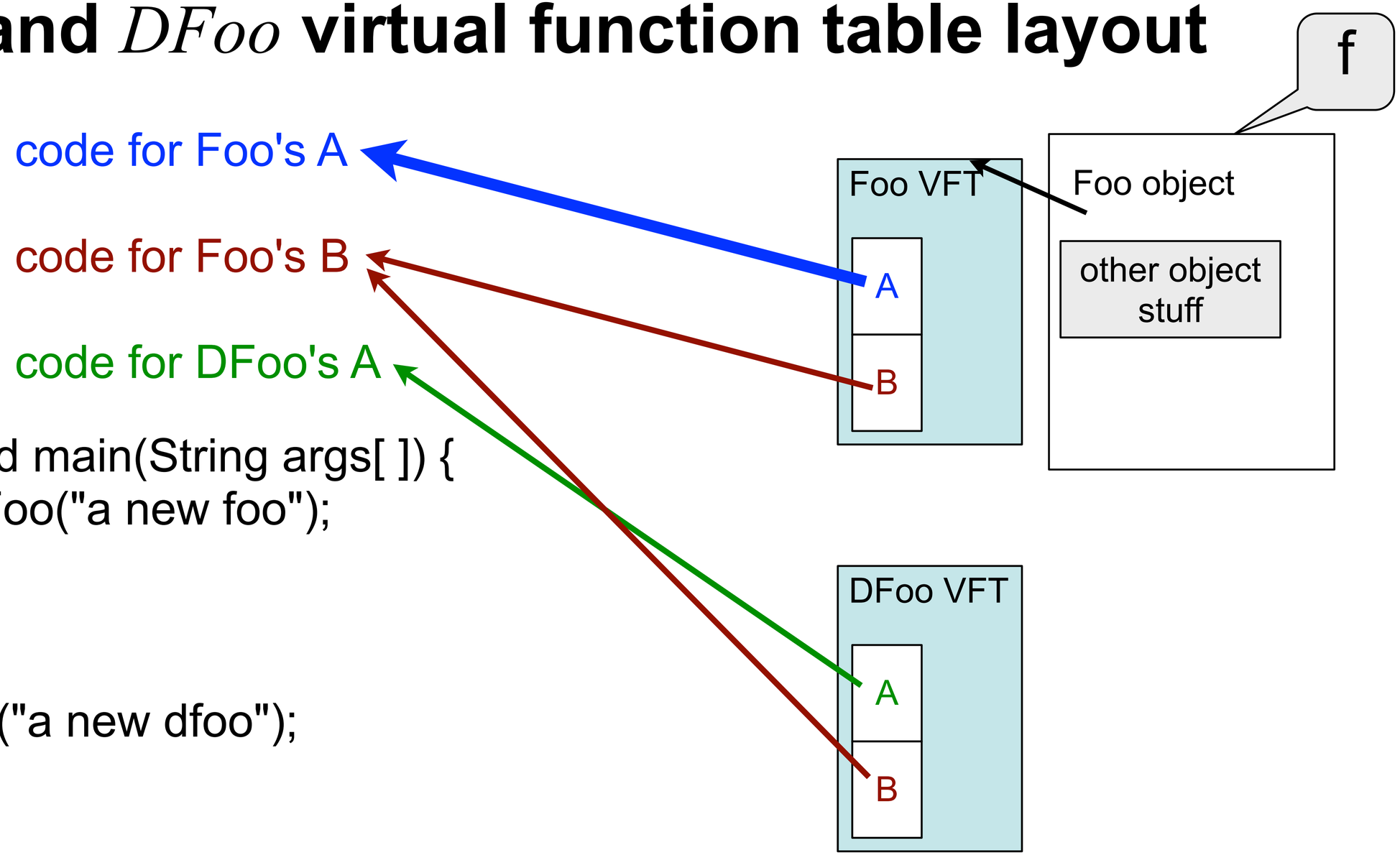


Foo and *DFoo* virtual function table layout



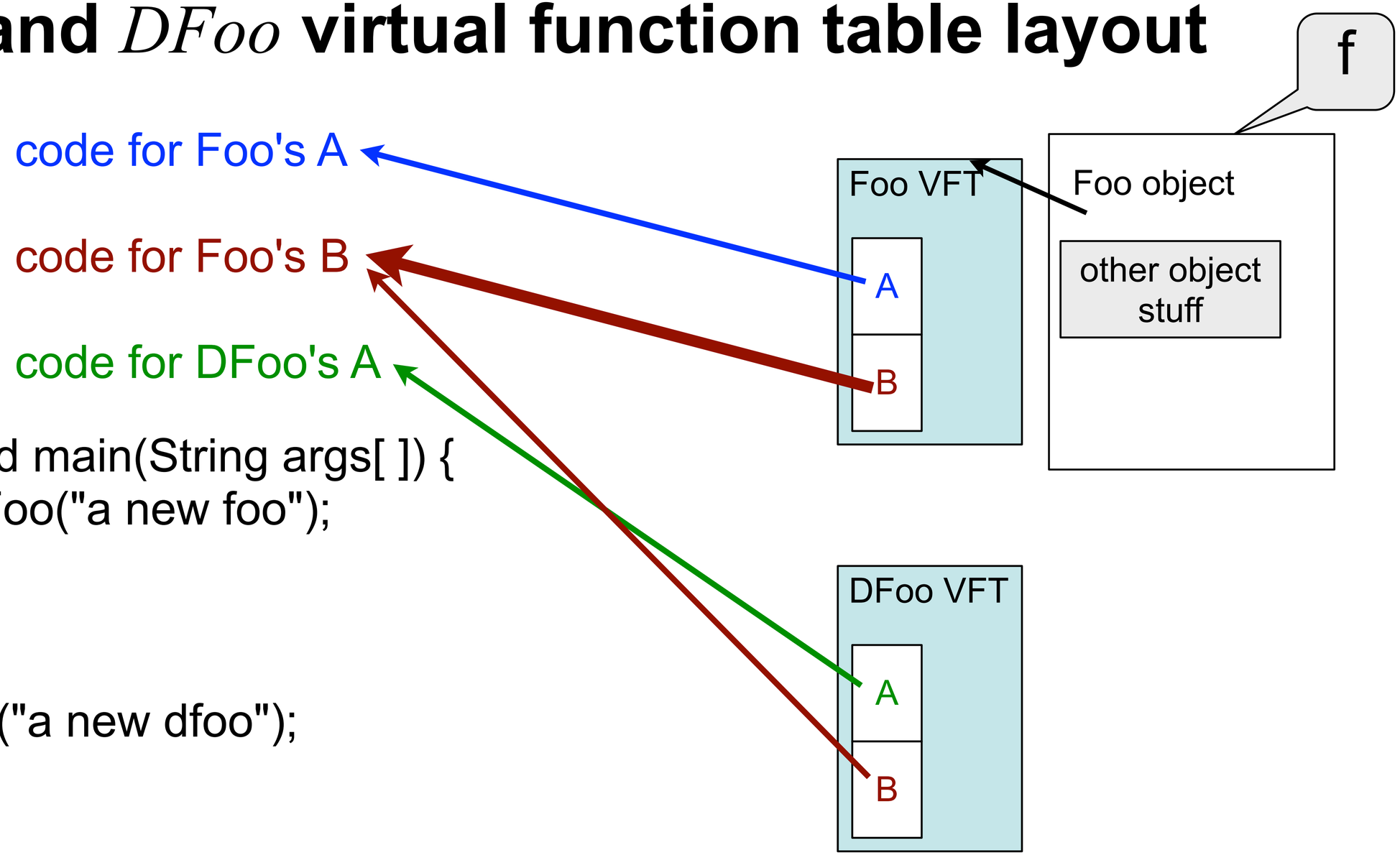

```
class Test {
   public static void main(String args[ ]) {
      Foo f = new Foo("a new foo");
      f.A( );
      f.B( );

      f = new DFoo("a new dfoo");
      f.A( );
      f.B( );
   }
}
```

Foo and *DFoo* virtual function table layout




```
class Test {
   public static void main(String args[ ]) {
      Foo f = new Foo("a new foo");
      f.A( );
      f.B( );

      f = new DFoo("a new dfoo");
      f.A( );
      f.B( );
   }
}
```

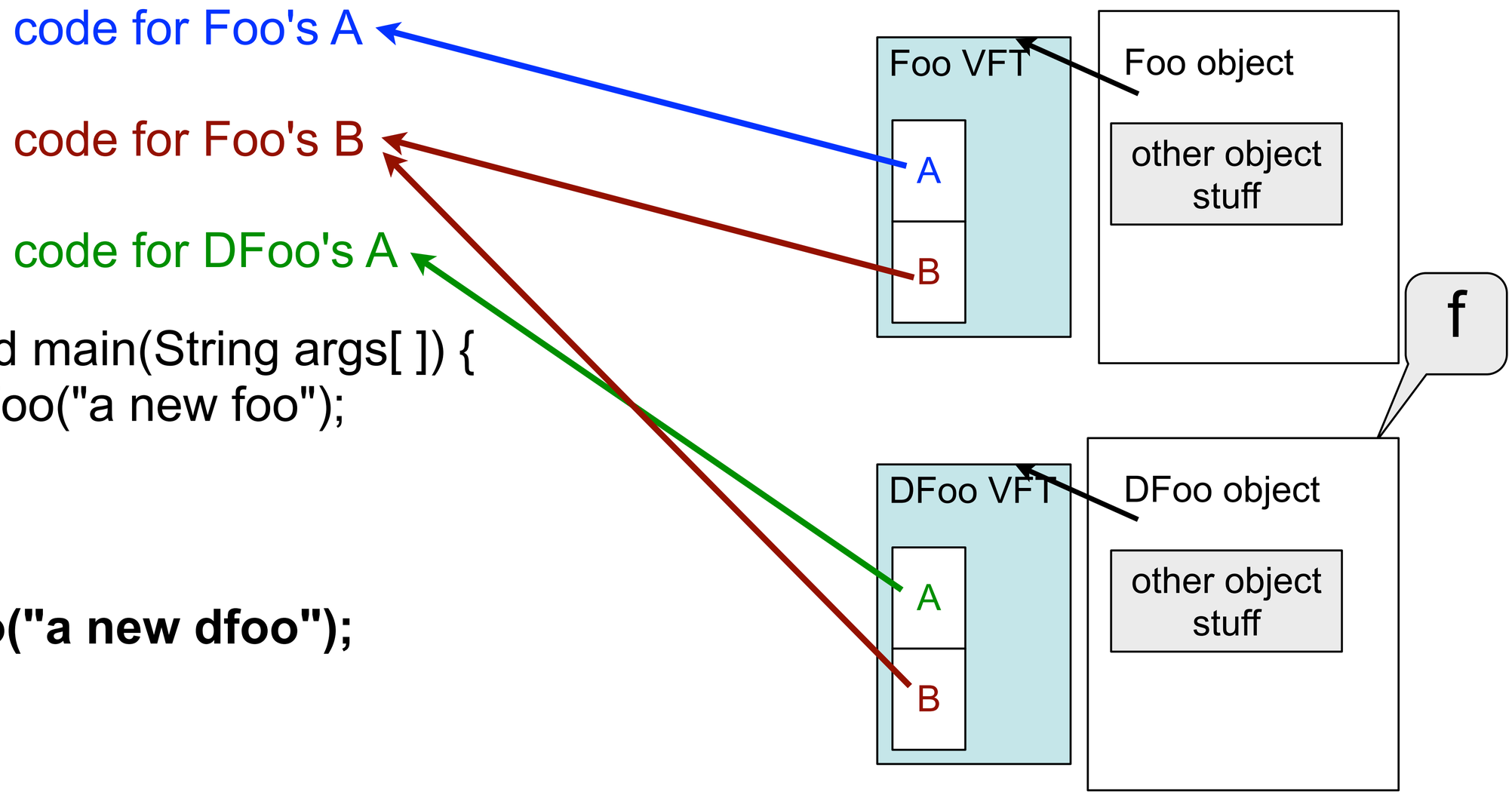
Foo and *DFoo* virtual function table layout




```
class Test {
   public static void main(String args[ ]) {
      Foo f = new Foo("a new foo");
      f.A( );
      f.B( );

      f = new DFoo("a new dfoo");
      f.A( );
      f.B( );
   }
}
```

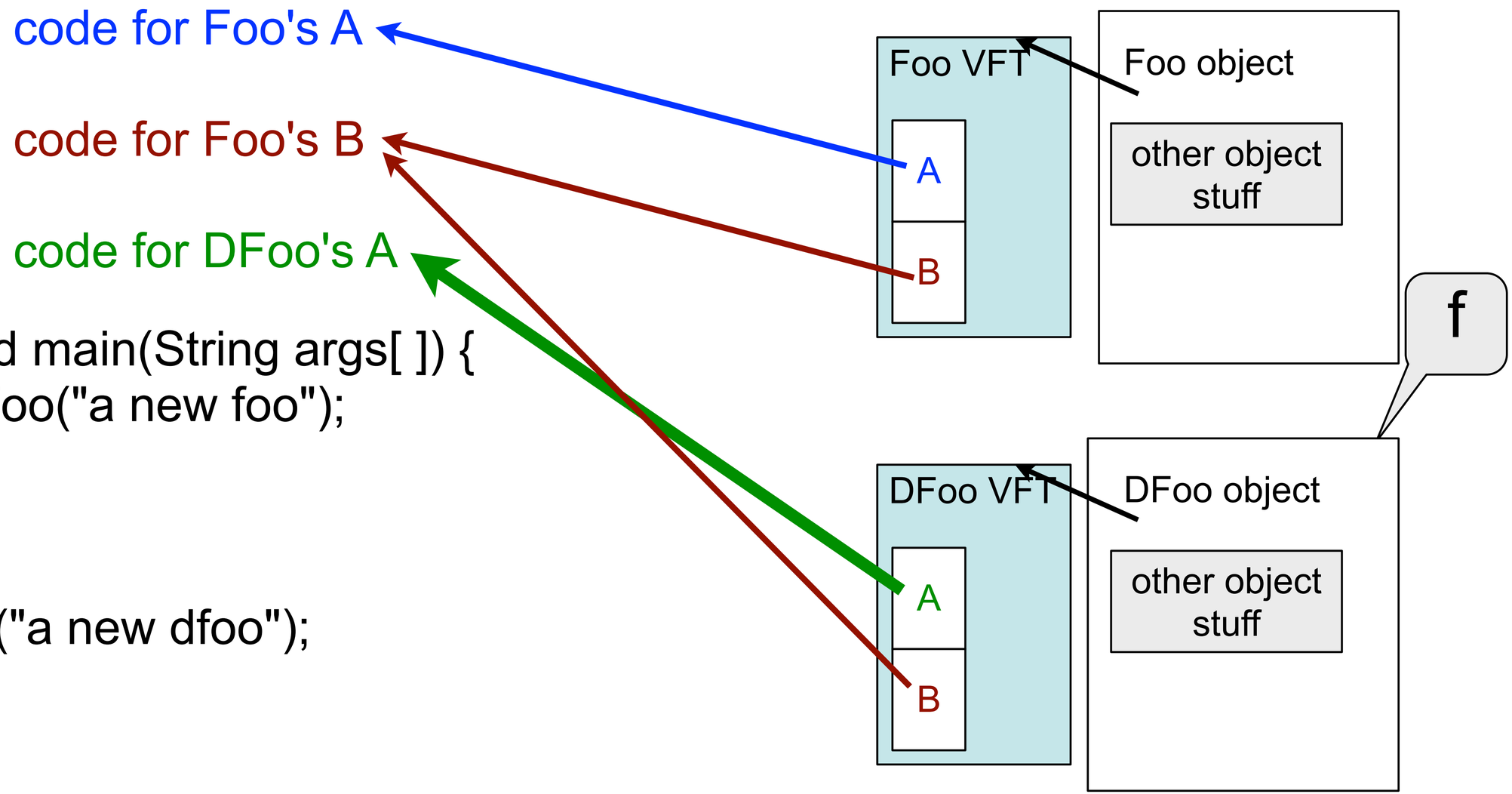
Foo and *DFoo* virtual function table layout

```
class Test {
   public static void main(String args[ ]) {
      Foo f = new Foo("a new foo");
      f.A( );
      f.B( );

      f = new DFoo("a new dfoo");
      f.A( );
      f.B( );
   }
}
```



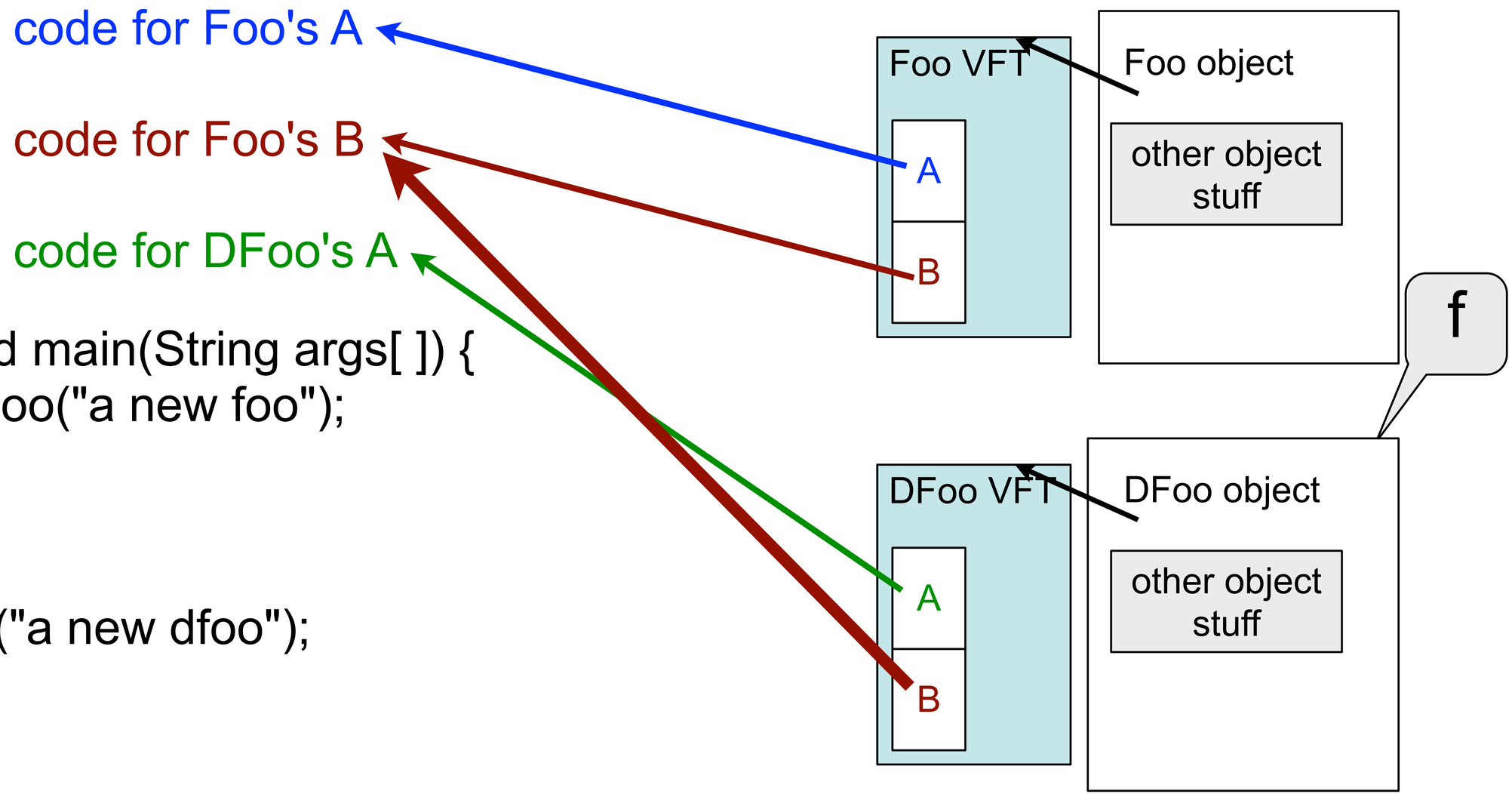
Foo and *DFoo* virtual function table layout

```
class Test {
   public static void main(String args[ ]) {
      Foo f = new Foo("a new foo");
      f.A( );
      f.B( );

      f = new DFoo("a new dfoo");
      f.A( );
      f.B( );
   }
}
```



Foo and *DFoo* virtual function table layout

```
class Test {
   public static void main(String args[ ]) {
      Foo f = new Foo("a new foo");
      f.A( );
      f.B( );

      f = new DFoo("a new dfoo");
      f.A( );
      f.B( );
   }
}
```



Forcing base methods to be invoked in Java

```
public class Foo {
   private final String fooString;

   public Foo( ) {fooString = null;}
   public Foo(String In) {fooString = In;}
   public void print( ) {
      System.out.println("Foo: "+fooString);
   }
}
```

```
public class DFoo extends Foo {
   private final String dfooString;

   public DFoo(String In) {dfooString = In;}
   public void print( ) {
      System.out.print("DFoo, printing super: ");
      super.print( ); // invokes print in base
                      // (super) class
      System.out.println("DFoo: "+dfooString);
   }
}
```

From java/SuperInvoke/

An example of when “overriding” does not work

```
class Base {

   public Base( ) { };
   public void print( ) {
        System.out.println("in Base print");}
}


class Derived extends Base {
   Derived( ) { }
   public void print( ) {
      System.out.println("Derived");
   }
   public void print2( ) {
      System.out.println("Derived 2");
   }
}
```

```
class Test {
   public static void main(String args[]) {
      Derived d1 = new Derived( );
      Base b2 = d1;
      d1.print2( );
      b2.print2( );  //
      ((Derived) b2).print2( );
   }
}
```

Not really overriding since print2 is not defined in the Base class.

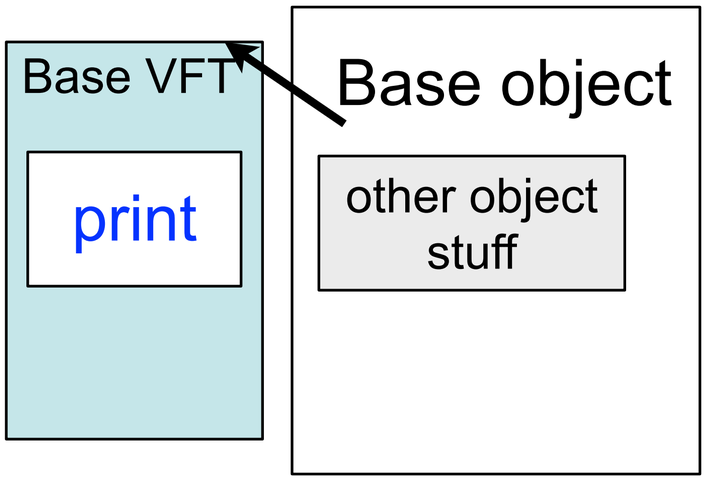
```
TestA.java:7: cannot find symbol
symbol  : method print2()
location: class Base
      b2.print2( );  //
        ^
1 error
```

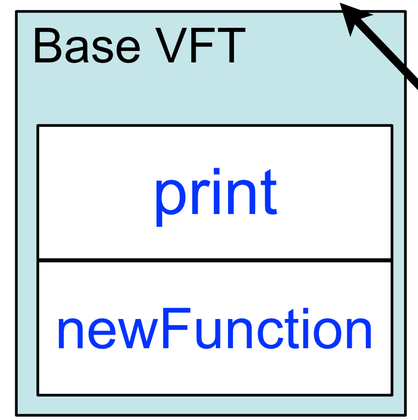
Let another class extend Base

```
class Base {

   public Base( ) { };
   public void print( ) {
        System.out.println("Base print");}
}


 class Derived extends Base {
    Derived( ) { }
   public void print( ) {
      System.out.println("Derived");
   }
   public void print2( ) {
      System.out.println("Der print2");
   }
}
```

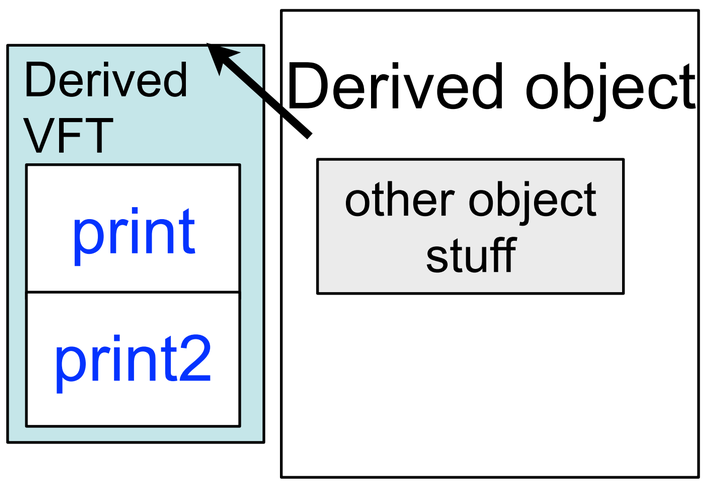
```
class Derived2 extends Base {
   public Derived2( ) { }
   public void print( ) {
      System.out.println("Derived");
   }
   public void newFunction( ) {
      System.out.println("new function");
   }
}
```

Why is this wrong, operationally?






```
From main
Derived d1 = new Derived( );
Base b2 = d1;
d1.print2( );
b2.print2( );  //
((Derived) b2).print2( );
```



Note that one derived object has a print2, the other does not and the base object only has a print.

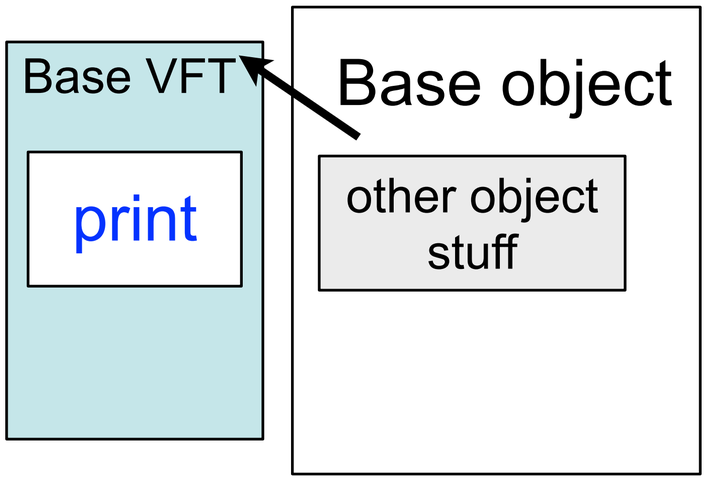
When doing a call b2.print2( ) should the function at position 1 be called, or an error? **Java says an error** -- does not check the type of object at runtime and see if the call is ok and where to call from. Java does do a check on Base b2 = ...

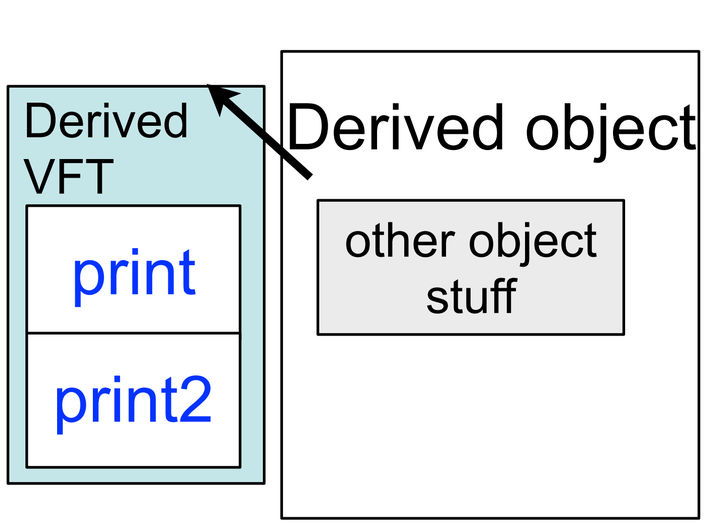
Let yet another class extend Base

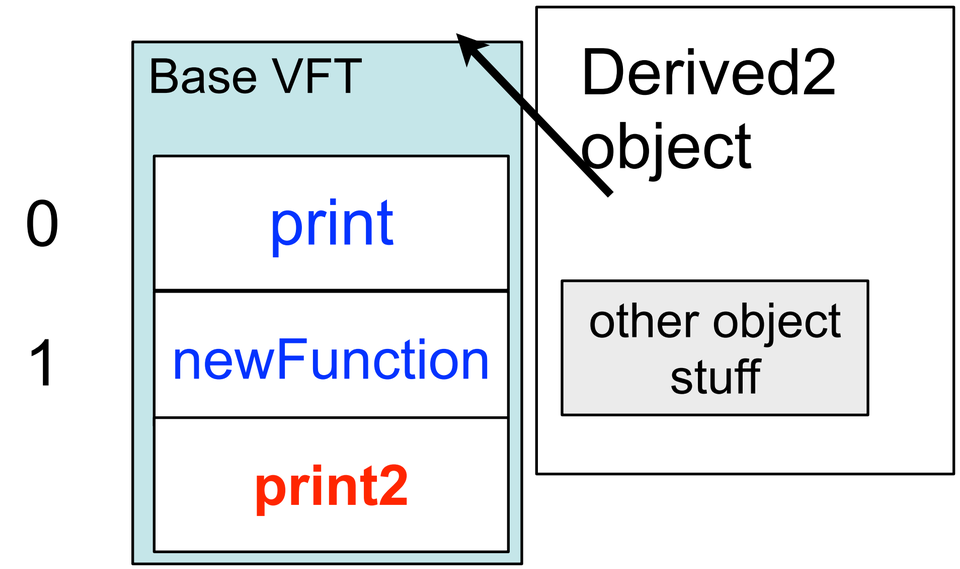
```
class Base {

   public Base( ) { };
   public void print( ) {
        System.out.println("Base print");}
}


 class Derived extends Base {
    Derived( ) { }
   public void print( ) {
      System.out.println("Derived");
   }
   public void print2( ) {
      System.out.println("Der print2");
   }
}
```

```
class Derived2 extends Base {
   public Derived2( ) { }
   public void print( ) {
      System.out.println("Derived");
   }
   public void newFunction( ) {
      System.out.println("new function");
   }
   public void print2( ) {
      System.out.println("D2 print2");
   }
}
```

Why is this the case, operationally?







Note that print2 is in two different slots in the Derived and Derived2 VFTs

When doing a call b2.print2( ) should the function at position 1 or 2 be called? Or an error, in case the object is a Base object? Java chooses an error.

Summary of problems with this

- In Java the compiler may not know if a Derived, Derived2 or Base class is the object the call to print2 is being made on
  – Again, cannot tell what slot to call print2 from, or if a slot exists
- Java looks at the type C of the reference or pointer to some object obj
  – Find all methods defined in the class C and its base classes
  – If the method called is not found, it is an error

A Java Gotcha

- *private* functions are not overridden - the *base print( )* will be called when using a reference to a base object (a *Foo* in this example).

```
public class Base {

   public Base( ) { }
   private void print( ) {
      System.out.println("Base print");
   }

   public void callPrint(Base b) {
      b.print( );
   }
}
```

```
public class Derived extends Base {

   public Derived( ) { }
   public void print( ) {
      // super.print( ); // invokes print in base
      System.out.print("Derived Print");
   }
}
public class Test {

   public static void main(String[ ] s) {
      Base b = new Base( );
      Derived  der = new Derived( );
      b.callPrint(der);
      der.callPrint(der);
   }
}
```

A Java Gotcha

- *private* functions are not overridden - the *base print( )* will be called when using a reference to a base object (a *Foo* in this example).

```
public class Base {

  public Base( ) { }
  private void print( ) {
    System.out.println("Base print");
  }

  public void callPrint(Base b) {
    b.print( );
  }
}
```

```
public class Derived extends Base {

  public Derived( ) { }
  public void print( ) {
    // super.print( ); // invokes print in base
    System.out.print("Derived Print");
  }
}
public class Test {

  public static void main(String[ ] s) {
    Base b = new Base( );
    Derived  der = new Derived( );
    b.callPrint(der);
    der.callPrint(der);
  }
}
```

```
Base print
Base print
```

Downcasts or specializing casts

- Most~~, if not all,~~ casts we have seen have been from a *derived* to a *base* object
  - These are called *upcasts* or *generalizing* casts
- In the TestA example, and the example on the right, we have a cast from a Base reference to a Derived reference
  - This is a specializing cast or down cast

```
class Test {
   public static void main(String args[]) {
      Derived d1 = new Derived( );

      Base b2 = d1;
      . . .
      ((Derived) b2).print2( );
   }
}
```

- Unlike upcasts or generalizing casts downcasts can lead to errors when what is being referred to by the Base type is not the type of the cast or something derived from that type.

Downcasts or specializing casts

- Most if not all casts we have seen have been from a *derived* to a *base* object
  - These are called *upcasts* or *generalizing* casts
- In the TestA example, and the example on the right, we have a cast from a Base reference to a Derived reference
  - This is a specializing cast or down cast

```
class Test {
   public static void main(String args[]) {
      Derived d1 = new Derived( );

      Base b2 = d1;
      . . .
      ((Derived) b2).print2( );
   }
}
```

- In the case above *b2* refers to a base object which has no print2( ) defined in its VFT, thus no print2 exists to be called.

Example of bad *implicit* down casting

```
public class Base {
   public Base( ) { }
   public void print( ) {System.out.println("Base");}

}

public class Derived extends Base {
   public Derived( ) { }
   public void print( ) {System.out.println("Derived");}

}

public class Main {
   public static void main(String[] args) {
      Base b = new Base( );
      Derived d = new Derived( );
      b = d; // OK, Derived ISA Base
      d = b; // ILLEGAL! Base ISA not a Derived
    }
}
```

- Even though the Java compiler, in this case, could know
  - The object referenced by **d** is a Derived object
  - The **d** reference can legally point to a Derived object
- This is still illegal because for assignment **l = r**, it must be true that **r ISA l. This is a Java rule that you must follow**

Assume previous implicit down cast were allowed

```
public class Base {
   public Base( ) { }
   public void print( ) {System.out.println("Base");}
   public int zero( ) {return 0;}

}

public class Derived extends Base {
   public Derived( ) { }
   public void print( ) {System.out.println("Derived");}

}

public class Main {
   public static void main(String[] args) {
      Base b = new Base( );
      Derived d = new Derived( );
      b = d; // Derived ISA Base
      d = b; // ILLEGAL! Base ISA not a Base
    }
}
```

What should happen here?

- A smart compiler would figure out that at the red statement **b** references a Derived object and program would be legal.
- A dumb compiler would not know what **b** pointed to in the red statement and program would be illegal.
- Legality of the program would depend on the compiler.
- Kills portability and generally a bad thing to do.

Assume previous cast were allowed
What should happen here?

```
public class Base {
   public Base( ) { }
   public void print( ) {System.out.println("Base");}
}

public class Derived extends Base {
   public Derived( ) { }
   public void print( ) {System.out.println("Derived");}

}

public class Main {
   public static void main(String[] args) {
      Base b = new Base( );
      Derived d = new Derived( );
      b = d; // Derived ISA Base
      d = (Derived) b; // possible runtime test
    }
}
```

- This is legal but may require a runtime test.
- A smart compiler would figure out that at the red statement **b** references a Derived object and not do a runtime test
- A dumb compiler would not know what **b** pointed to in the red statement and do a runtime test.
- The cast indicates the programmer might have a clue and thus Java does a runtime test, if necessary, to ensure legality of the down cast.

Assume previous cast were allowed

```
public class Base {
   public Base( ) { }
   public void print( ) {System.out.println("Base");}
}

public class Derived extends Base {
   public Derived( ) { }
   public void print( ) {System.out.println("Derived");}

}

public class Main {
   public static void main(String[] args) {
      Base b = new Base( );
      Derived d = new Derived( );
     if (foo( ) == 0) b = d;
      d = (Derived) b; // possible runtime test
    }
}
```

What should happen here?

- This may or may not be legal, depending on the result of the *if* statement
- Doing a runtime test, as before, makes it all work because an error will be called if it is illegal and the program will run if it is legal.
- Unless you *know*, as a programmer, the downcast is legal, you should not do this
  - It is a rich source of errors that will only be caught at runtime
  - Embarrassing when it brings down Amazon or during a demo.

How to execute and run a Java program from a terminal window

```
smidkiffs-MacBook-Air:L1PolyOverride smidkiff$ ls
Pentagon.javaSquare.java   TestAlt.java spoor
Poly.java     Test.java     Triangle.java
smidkiffs-MacBook-Air:L1PolyOverride smidkiff$ javac Test.java
smidkiffs-MacBook-Air:L1PolyOverride smidkiff$ ls
Pentagon.class      Poly.class    Square.class Test.class    TestAlt.java spoor
Pentagon.javaPoly.java     Square.java   Test.java     Triangle.java
smidkiffs-MacBook-Air:L1PolyOverride smidkiff$ java Test
```

output from the run

```
smidkiffs-MacBook-Air:L1PolyOverride smidkiff$
```

How not to compile a Java program from a terminal window

```
smidkiffs-MacBook-Air:L1PolyOverride smidkiff$ java Test.java
Exception in thread "main" java.lang.NoClassDefFoundError: Test/java
Caused by: java.lang.ClassNotFoundException: Test.java
        at java.net.URLClassLoader$1.run(URLClassLoader.java:202)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:190)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:301)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
smidkiffs-MacBook-Air:L1PolyOverride smidkiff$
```