Overloading, simple constructors, static methods

Function/method overloading

- *Overloading* allows us to reuse method names, i.e., implement the same operator for different types of data
- C++ allows overloading of ***infix, unary and other operators,*** Java only allows overloading of methods
- While overloaded methods have the same name, their *signatures must differ*
  - Not enough for return values to differ, arguments themselves must differ

- Overloading **IS NOT** overriding.
  - ***Overriding*** is defining a method in a derived class with the same name and signature as a method in a base class
  - ***Overloading*** is using the same method name but different signatures within a class or in a base class, in Java.

Overloading in Java -- examples

```
import java.io.*;

class User {
   private String name;
   private int age;

   public User(String str, int yy)  {name = str; age = yy; }

   public void print( ) {
      System.out.println("name: " + name + " age: " + age);
   }

   public String foo(String s, int i) {
      System.out.println("User::foo was called with a string and an int");
      return s+i;
   }

   public String foo(String s1, String s2) {
      System.out.println("User::foo was called with a string and a string");
      return s1.concat(s2);
   }

   public String foo(int i, int j) {
      System.out.println("User::foo was called with an int and an int");
      return ""+i+""+i;
   }
```

Overloading in Java -- examples

```
class User {
    public String foo(String s, int i) {
      System.out.println("User::foo was called with a string and an int");
      return s+i;
    }

    public String foo(String s1, String s2) {
      System.out.println("User::foo was called with a string and a string");
      return s1.concat(s2);
    }

    public String foo(int i, int j) {
      System.out.println("User::foo was called with an int and an int");
      return ""+i+""+i;
    }
}
```

Example of calling overloaded methods

```
import java.io.*;

class Test2 {

   public static void main(String args[]) {
      User u = new User("Car", 54);
      String s1 = u.foo("hello ", 0);
      String s2 = u.foo(-1, 0);
      String s3 = u.foo("super", " man");
      System.out.println("s1: "+s1+"; s2: "+s2+"; s3: "+s3);
   }
}
```

```
[ece-76-55:codew2b/java/overLoad] smidkiff% java Test
User::foo was called with a string and an int
User::foo was called with an int and an int
User::foo was called with a string and a string
s1 hello 0, s2: -1-1, s3: super man
[ece-76-55:codew2b/java/overLoad] smidk
```

What if no function matches exactly?

```
Class C {
   . . .
   public void foo(int i) { . . . }
   public void foo(long l) { . . . }
   . . .
}

public static void main(String args[]) {
   short s = -4;
   C c = new C( );
   c.foo(s);
}
```

- Intuitively we would like to match the "closest" function, i.e., the function whose parameters are closest to the passed arguments
- Need to decide what is closest
- From the Java 5 spec: "A *widening* primitive conversion does not lose information about the overall magnitude of a numeric value." and "A *widening reference* conversion exists from any reference type S to any reference type T, provided S is a subtype (§4.10) of T.", i.e., from a derived to a base class.

What are the widening primitive conversions?

byte to short, int, long, float, or double

short to int, long, float, or double

char to int, long, float, or double

int to long, float, or double

long to float or double

float to double

This gives us a clue as to what parameters may be closest to the calling argument types

Widening operations

	byte	char	short	int	long	float	double
byte		No	Yes	Yes	Yes	Yes	Yes
char	No		No	Yes	Yes	Yes	Yes
short	No	No		Yes	Yes	Yes	Yes
int	No	No	No		Yes	Yes	Yes
long	No	No	No	No		Yes	Yes
float	No	No	No	No	No		Yes

converting `short` to `char` and `char` to `short`

- Both `char` and `short` are 16 bits long. Why are they not a widening conversion?
- `char` to `short`
  - `char` is unsigned
  - Moving a `char` to an integral primitive of length *n* is done by moving the low-order *n* bits of `char` to the integral primitive.
  - `short` is 16 bits long, so all 16 bits of a `char` are moved to a `short`. A `short` is also signed.
  - If the bit in `char` that corresponds to the sign bit in a `short` is set, then an unsigned positive `char` value becomes a negative `short` value. E.g., consider moving 1111 1111 1111 1110 to a short.
  - `short` to `char`
  - `short` is signed.
  - Moving the value 1111 1111 1111 1100 in a `short` (a negative number) to a `char` will give a large positive number, since the sign bit is not longer a sign bit, but just another value bit.

What if no function matches exactly?

- Form the set *M* of all visible methods *m* that match the call exactly or with only widening conversions.
- Examining the set *M*
  - If there is an exact match in *M*, use it
  - For each m_n and m_w, if the parameters of m_n can match m_w with widening conversions, discard m_w. If after doing this only one method remains, use it
- Otherwise, declare an error
- blue text above is basically looking for the method whose parameters are closest to the argument types of the call.

Example 1

```
public class C {

   public C( ) {System.out.println("construct a C object");}

   public void foo(long l, double d) {System.out.println("foo(l,d)");}
   public void foo(int i, float f) {System.out.println("foo(i,f)");}
   public void foo(byte b, float f) {System.out.println("foo(b,f)");}
   public void foo(short s, float f) {System.out.println("foo(s,f)");}
   public void foo(byte b, short s) {System.out.println("foo(b,s)");}
   public void foo(int i, short s) {System.out.println("foo(i,s)");}

   public static void main(String args[ ]) {
      int i = 0;
      short s = 0;
      byte b = 1;
      float f = (float) 1.0;
      double d = 1.0;
      C c = new C( );

      c.foo(i, d);
      c.foo(s, d);
      c.foo(b, i);
      c.foo((int) b, i);
   }
}
```

Example 1(a)

```
public class C {

  public C( ) {System.out.println("construct a C object");}

  public void foo(long l, double d) {System.out.println("foo(l,d)");}
  public void foo(int i, float f) {System.out.println("foo(i,f)");}
  public void foo(byte b, float f) {System.out.println("foo(b,f)");}
  public void foo(short s, float f) {System.out.println("foo(s,f)");}
  public void foo(byte b, short s) {System.out.println("foo(b,s)");}
  public void foo(int i, short s) {System.out.println("foo(i,s)");}

  public static void main(String args[ ]) {
    int i = 0;
    short s = 0;
    byte b = 1;
    float f = (float) 1.0;
    double d = 1.0;
    C c = new C( );

    c.foo(i, d);
    c.foo(s, d);
    c.foo(b, i);
    c.foo((int) b, i);
  }
}
```

Only `foo`(long l, double d) is left and `c.foo(i, d)` can match the parameters with widening operations.

`foo`(long l, double d) is called.

Example 1(b)

```
public class C {

  public C( ) {System.out.println("construct a C object");}

  public void foo(long l, double d) {System.out.println("foo(l,d)");}
  public void foo(int i, float f) {System.out.println("foo(i,f)");}
  public void foo(byte b, float f) {System.out.println("foo(b,f)");}
  public void foo(short s, float f) {System.out.println("foo(s,f)");}
  public void foo(byte b, short s) {System.out.println("foo(b,s)");}
  public void foo(int i, short s) {System.out.println("foo(i,s)");}

  public static void main(String args[ ]) {
    int i = 0;
    short s = 0;
    byte b = 1;
    float f = (float) 1.0;
    double d = 1.0;
    C c = new C( );

    c.foo(i, d);
    c.foo(s, d);
    c.foo(b, i);
    c.foo((int) b, i);
  }
}
```

Only `foo`(long l, double d) is left and `c.foo(s, d)` can match the parameters with widening operations.

`foo`(long l, double d) is called.

Example 1(c)

```
public class C {

  public C( ) {System.out.println("construct a C object");}

  public void foo(long l, double d) {System.out.println("foo(l,d)");}
  public void foo(int i, float f) {System.out.println("foo(i,f)");}
  public void foo(byte b, float f) {System.out.println("foo(b,f)");}
  public void foo(short s, float f) {System.out.println("foo(s,f)");}
  public void foo(byte b, short s) {System.out.println("foo(b,s)");}
  public void foo(int i, short s) {System.out.println("foo(i,s)");}

  public static void main(String args[ ]) {
    int i = 0;
    short s = 0;
    byte b = 1;
    float f = (float) 1.0;
    double d = 1.0;
    C c = new C( );

    c.foo(i, d);
    c.foo(s, d);
    c.foo(b, i);
    c.foo((int) b, i);
  }
}
```

foo(int i, short s) and foo(byte b, short s) cannot match (byte, int) by widening.

$M = \{$foo(long l, double d),
foo(int i, float f),
foo(byte b, float f),
foo(short s, float f)$\}$

Example 1(c)

```
public class C {

  public C( ) {System.out.println("construct a C object");}

  public void foo(long l, double d) {System.out.println("foo(l,d)");}
  public void foo(int i, float f) {System.out.println("foo(i,f)");}
  public void foo(byte b, float f) {System.out.println("foo(b,f)");}
  public void foo(short s, float f) {System.out.println("foo(s,f)");}
  public void foo(byte b, short s) {System.out.println("foo(b,s)");}
  public void foo(int i, short s) {System.out.println("foo(i,s)");}

  public static void main(String args[ ]) {
    int i = 0;
    short s = 0;
    byte b = 1;
    float f = (float) 1.0;
    double d = 1.0;
    C c = new C( );

    c.foo(i, d);
    c.foo(s, d);
    c.foo(b, i);
    c.foo((int) b, i);
  }
}
```

$$M = \{\texttt{foo(long l, double d)},\ \texttt{foo(int i, float f)},\ \texttt{foo(byte b, float f)},\ \texttt{foo(short s, float f)}\}$$

foo(int i, float f) (m_n) can match foo(long l, double d) (m_w) by widening. Remove m_w.

Example 1(c)

```
public class C {

  public C( ) {System.out.println("construct a C object");}

  public void foo(long l, double d) {System.out.println("foo(l,d)");}
  public void foo(int i, float f) {System.out.println("foo(i,f)");}
  public void foo(byte b, float f) {System.out.println("foo(b,f)");}
  public void foo(short s, float f) {System.out.println("foo(s,f)");}
  public void foo(byte b, short s) {System.out.println("foo(b,s)");}
  public void foo(int i, short s) {System.out.println("foo(i,s)");}

  public static void main(String args[ ]) {
    int i = 0;
    short s = 0;
    byte b = 1;
    float f = (float) 1.0;
    double d = 1.0;
    C c = new C( );

    c.foo(i, d);
    c.foo(s, d);
    c.foo(b, i);
    c.foo((int) b, i);
  }
}
```

$$M = \{\texttt{foo(int i, float f)},\ \texttt{foo(byte b, float f)},\ \texttt{foo(short s, float f)}\}$$

foo(byte b, float f) (m_n) can match foo(int i, float f) (m_w) by widening. Remove m_w.

Example 1(c)

```
public class C {

  public C( ) {System.out.println("construct a C object");}

  public void foo(long l, double d) {System.out.println("foo(l,d)");}
  public void foo(int i, float f) {System.out.println("foo(i,f)");}
  public void foo(byte b, float f) {System.out.println("foo(b,f)");}
  public void foo(short s, float f) {System.out.println("foo(s,f)");}
  public void foo(byte b, short s) {System.out.println("foo(b,s)");}
  public void foo(int i, short s) {System.out.println("foo(i,s)");}

  public static void main(String args[ ]) {
    int i = 0;
    short s = 0;
    byte b = 1;
    float f = (float) 1.0;
    double d = 1.0;
    C c = new C( );

    c.foo(i, d);
    c.foo(s, d);
    c.foo(b, i);
    c.foo((int) b, i);
  }
}
```

$M = \{$`foo(byte b, float f),`
`foo(short s, float f)`$\}$

foo(byte b, float f) (m_n) can match foo(short s, float f) (m_w) by widening.

Remove m_w.

$M = \{$`foo(byte b, float f)`$\}$

Call `foo(byte b, float f)`

Example 1(d)

```
public class C {

  public C( ) {System.out.println("construct a C object");}

  public void foo(long l, double d) {System.out.println("foo(l,d)");}
  public void foo(int i, float f) {System.out.println("foo(i,f)");}
  public void foo(byte b, float f) {System.out.println("foo(b,f)");}
  public void foo(short s, float f) {System.out.println("foo(s,f)");}
  public void foo(byte b, short s) {System.out.println("foo(b,s)");}
  public void foo(int i, short s) {System.out.println("foo(i,s)");}

  public static void main(String args[ ]) {
    int i = 0;
    short s = 0;
    byte b = 1;
    float f = (float) 1.0;
    double d = 1.0;
    C c = new C( );

    c.foo(i, d);
    c.foo(s, d);
    c.foo(b, i);
    c.foo((int) b, i);
  }
}
```

$M = \{$`foo(long l, double d),`
`foo(int i, float f)`$\}$

`foo(int i, float f)` (m_n) can match
`foo(long l, double d)` (m_w) by widening.
Remove m_w.

$M = \{$`foo(int i, float f)`$\}$
Call `foo(int i, float f)`

Example 2

visible methods:

```
public void foo(int i, double d) {...}
public void foo(char c, double d) {...}
public void foo(short i, double d) {...}
```

The call to be matched:

```
int j; double z; float f; short s;
foo(s, f);
```

- *M* is all blue functions
  - *short* matches *short* exactly, widens to *int*
  - ***float* widens to *double***
  - Nothing matches exactly
- let m_n = `foo(short i, double d)`
  m_w = `foo(int i, double d)`
  - short widens to int
  - double matches double
  - get rid of m_w = `foo(int i, double d)`

Example 2

- ***One function left -- use it.***

visible methods:

public void foo(int i, double d) {...}
public void foo(char c, double d) {...}
public void foo(short i, double d) {...}

The call to be matched:

int j; double z; float f; short s;
foo(s, f);

Example 3

visible methods:

public void foo(int i; float f) {...}
public void foo(char c; double d) {...}

The call to be matched:

char k; float x;
foo(k, x);

Intuitively, one visible method is closer match on one argument, another is closer on another argument. Neither is overall closer.

- *M* is all blue definitions
  - nothing matches directly
- let m_n = `foo(int i; float f)`
  m_w = `foo(char c; double d)`
  - *int* cannot widen to *char*
  - cannot remove `foo(char c; double d)`
- let m_n = `foo(char c; double d)`
  m_w = `foo(int i; float f)`
  - *char* widens to *int*
  - *double* does not match or widen to *float*
  - cannot get rid of foo(short s; float f)
- *Multiple elements left in M,* ***error!***

Simple constructors and using constructors and *final* to control inheritance

Constructors

- Constructors are the functions that assign values to an object, i.e. an instance of the class
- Storage is allocated on the heap for Java objects by *new*
- Constructors initialize the storage for an object's storage.
- Initialization can happen by default, and thus constructors perform both system and programmer specified actions
- Let's look at some Java examples

Sample base class constructor

```
import java.io.*;

class User {
   private String name;
   private int age;

   public User(String str, int yy)  {name = str; age = yy; }

   public void print( ) {
      System.out.println("name: " + name + " age: " + age);
   }
}
```

In codew2b/java/constructor

Sample derived class constructor

```
class StudentUser extends User {
   private String schoolEnrolled;

   public StudentUser(String nam, int y, String sch) {
      super(nam,y); // call base class constructor
      schoolEnrolled = sch;
   }

   public void print( ) {
      super.print( );
      System.out.println("School: " + schoolEnrolled);
   }
}
```

Creating a derived class object

```
import java.io.*;

class Test {

   public static void main(String args[]) {
      StudentUser student = new StudentUser("Ralph", 54, "Bug Tech");
      student.print( );
   }
}
```

The StudentUser constructor calls the User constructor (passing "Ralph" and 54) before initializing the rest of the StudentUser object.

Java zero-arg constructors

- Java is well-defined
  - Java is required to always have a defined state for objects
  - Java was a clean slate design - no backward compatibility
- Java calls a ***zero-arg constructor*** if no explicit call provided. You may not need to define this constructor.
- Like C++, an error results if *no* zero arg constructor supplied and non-zero arg constructors *are* supplied
  - This is done to catch typos typos and unexpanded constructor skeletons supplied by IDEs
  - If Java supplies a zero-arg constructor, *uninitialized* fields initialized according to the table on the right.
  - This table also applies to default variable initializations

Type	Initial Value
boolean	FALSE
char	\u0000
all integer types	0
float	0.0f
double	0
object reference	null

Java default zero arg constructor example

A default for name and age are defined. *C++ does not allow this to be done in class declarations.*

This code can be found in java/zeroarg

```
import java.io.*;

class User {
   private String name = "Default Name";
   private int age = -1;

//   public User(String str, int yy)  {
//      name = str; age = yy;
//   }

   public void print( ) {
      System.out.println("name: " + name + " age: " + age);
   }
}
```

Java zero-arg example

```
class StudentUser extends User {
   private String schoolEnrolled;

   public StudentUser(String nam, int y, String sch) {
      // super(nam,y);
      schoolEnrolled = sch;
   }

   public void print( ) {
      super.print( );
      System.out.println("School: " + schoolEnrolled);
   }
}
```

```
javac Test.java
[ece-76-55:codew2b/java/zeroInit] smidkiff% java Test
name: Default Name age: -1
School: Bug Tech
```

Another zero arg example

```
import java.io.*;

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);
   }
}
```

```
import java.io.*;

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);
   }
}
```

The DFoo constructor does not specify a base class constructor to be called. Therefore the default zero arg constructor for Foo is called. This constructor sets fooString = null.

From java/SuperInvoke/

Zero Arg example

```
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( );
   }
```

```
Foo: Foo object
DFoo, printing super: Foo: null
DFoo: DFoo object
DFoo, printing super: Foo: null
DFoo: DFoo object
DFoo, printing super: Foo: null
DFoo: DFoo object
```

Note call to Base class print. But why does it print *Foo: null* for DFoo objects?

From java/SuperInvoke/

The answer to why *Foo* is null lies in how the constructors are written

```
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.print("DFoo, printing super: ");
      super.print(); // invokes print in base (super) class
      System.out.println("DFoo: "+dfooString);
   }
}
```

Is Called implicitly

Would like this to be called

The DFoo constructor does not specify a base class constructor to be called. Therefore the default zero arg constructor for Foo is called. This constructor sets fooString = null.

From java/SuperInvoke/

This fixes the problem and calls the right constructor

```
public class DFoo extends Foo {

   private final String dfooString;

   public DFoo(String ln) {super(ln); dfooString = ln;}
   public void print( ) {
      super.print( );
   }
}
```

```
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( );
   }
}
```

```
Foo: Foo object
DFoo, printing super: Foo: DFoo object
DFoo: DFoo object
DFoo, printing super: Foo: DFoo object
DFoo: DFoo object
DFoo, printing super: Foo: DFoo object
DFoo: DFoo object
```

From java/SuperConstInvoke/

This fixes the problem and calls the right constructor

```
public class DFoo extends Foo {

   private final String dfooString;

   public DFoo(String ln) {super(ln); dfooString = ln;}
   public void print( ) {
      super.print( );
   }
}
```

Note that the call is within the constructor body.

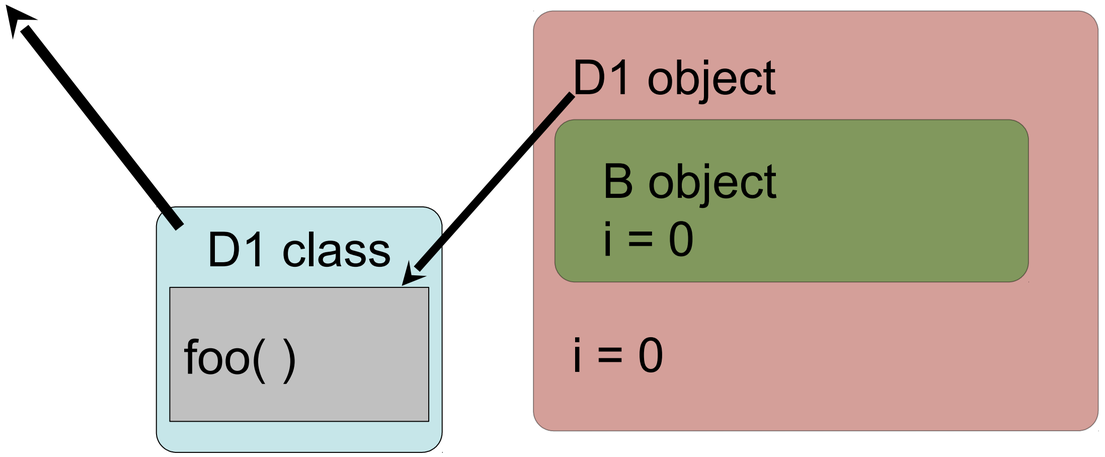
```
class Test {

   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( );
   }
}
```

From java/SuperConstInvoke/

An example of polymorphic calls in a constructor

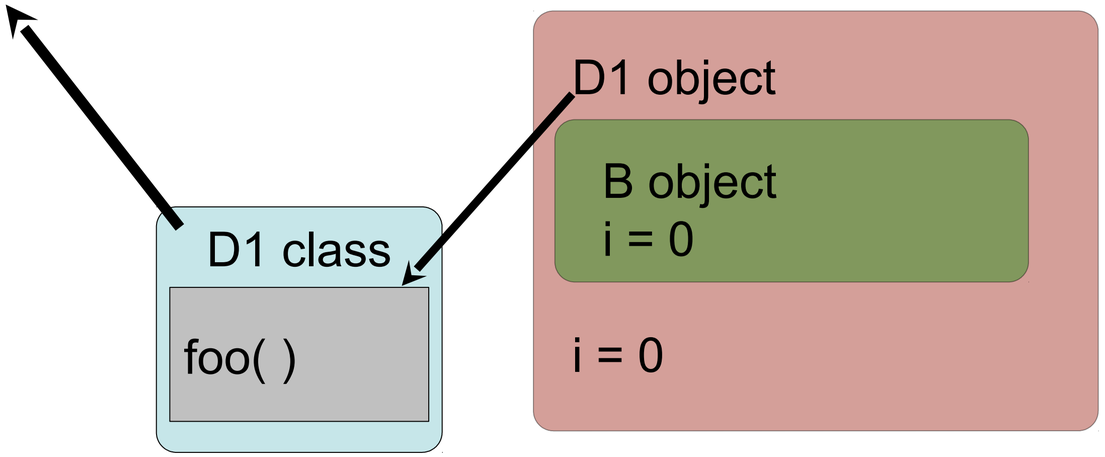
```
public class B {
   private int i;
   public B( ) {i=1;  foo( );}
   public void foo( ) {System.out.println("i: "+i);}
}

public class D1 extends B {
   private int i;
   public D1( ) {i=2;}
   public void foo( ) {System.out.println("i: "+i);}
}

public class Test {
   public static void main(String args[]) {
      D1 d1 = new D1( );
   }
}
```

This is dangerous for reasons we will see.



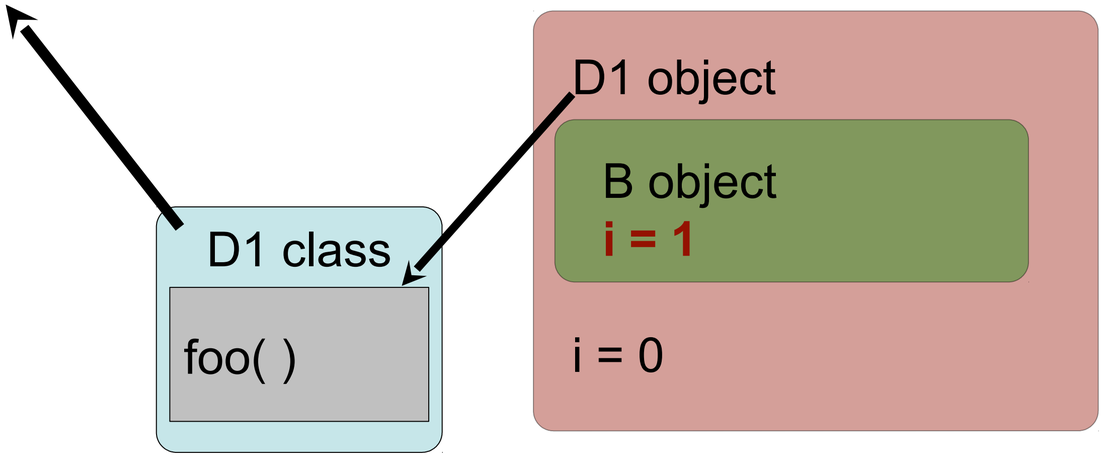
An example of polymorphic calls in a constructor

```
public class B {
   private int i;
   public B( ) {i=1;  foo( );}
   public void foo( ) {System.out.println("i: "+i);}
}

public class D1 extends B {
   private int i;
   public D1( ) {super( ); i=2;}
   public void foo( ) {System.out.println("i: "+i);}
}

public class Test {
   public static void main(String args[]) {
      D1 d1 = new D1( );
   }
}
```



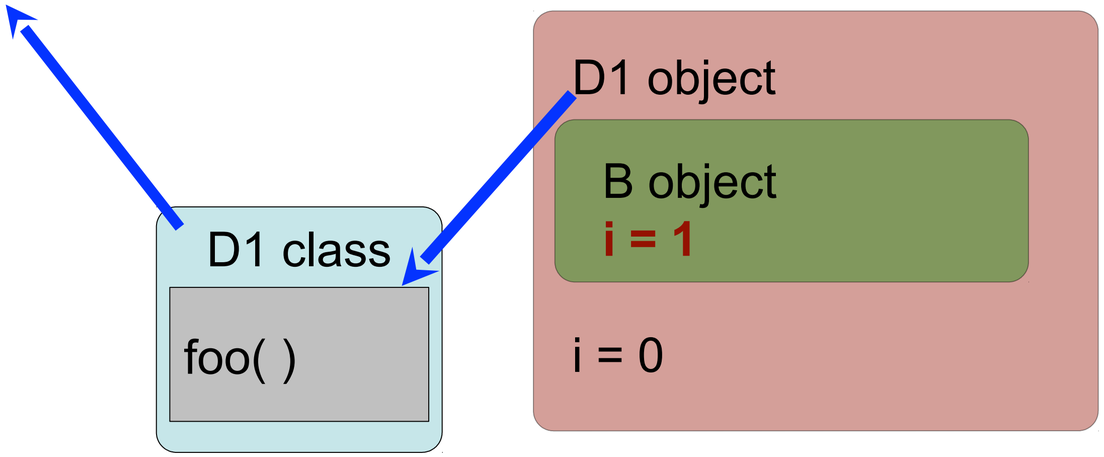
An example of polymorphic calls in a constructor

```
public class B {
   private int i;
   public B( ) {i=1;  foo( );}
   public void foo( ) {System.out.println("i: "+i);}
}

public class D1 extends B {
   private int i;
   public D1( ) {super( ); i=2;}
   public void foo( ) {System.out.println("i: "+i);}
}

public class Test {
   public static void main(String args[]) {
      D1 d1 = new D1( );
   }
}
```



An example of polymorphic calls in a constructor

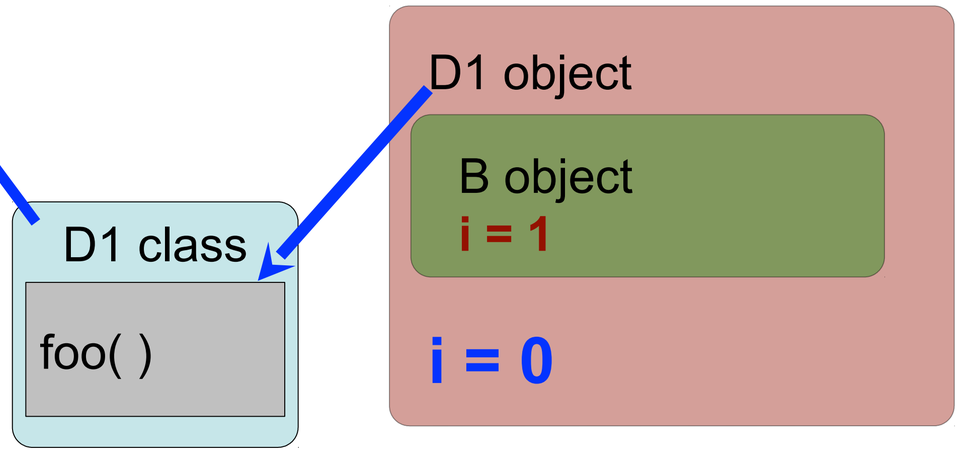
```
public class B {
   private int i;
   public B( ) {i=1;  foo( );}
   public void foo( ) {System.out.println("i: "+i);}
}

public class D1 extends B {
   private int i;
   public D1( ) {super( ); i=2;}
   public void foo( ) {System.out.println("i: "+i);}
}

public class Test {
   public static void main(String args[]) {
      D1 d1 = new D1( );
   }
}
```



An example of polymorphic calls in a constructor

```
public class B {
   private int i;
   public B( ) {i=1;  foo( );}
   public void foo( ) {System.out.println("i: "+i);}
}

public class D1 extends B {
   private int i;
   public D1( ) {super( ); i=2;}
   public void foo( ) {System.out.println("i: "+i);}
}

public class Test {
   public static void main(String args[]) {
      D1 d1 = new D1( );
   }
}
```

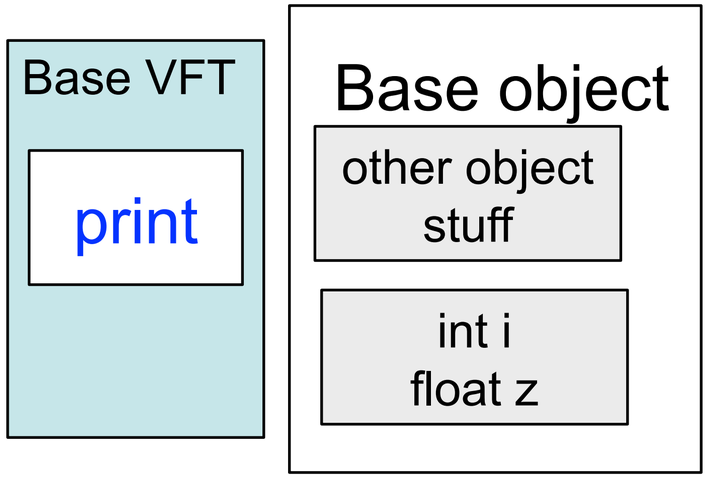
i: 0 *is printed*



Static members and methods

The *this* reference

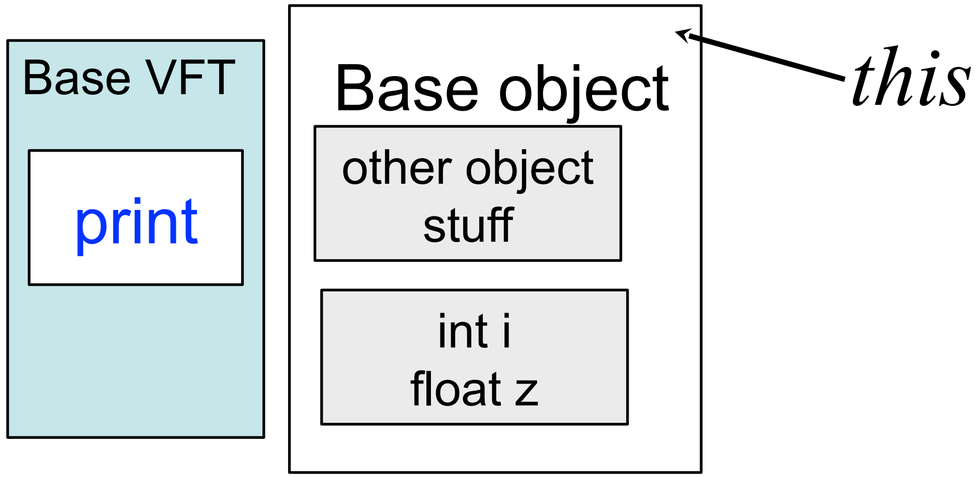
```
class Base {

   int i;
   float z;

   public Base( ) { };
   public void print( ) {
        System.out.println(i+” “+z);}
}
```



- When a Base object is allocated, it is created on the heap
- When the *print* method is called, it needs to access the variables *i* and *z* associated with the object
- This is done by having a hidden parameter, *this*, passed into the method that points to the object so that the variables on the heap can be accessed.

The *this* reference

```
class Base {

   int i;
   float z;

   public Base( ) { };
   public void print( ) {
        System.out.println(i+" "+z);}
}
```



- User code *r.print( );* generates a call that looks like *r.print(this);*
- Since the reference *r* either contains the address of the object on which print is called, or allows the address to be found (*r* is, after all, a handle that allows the object to be accessed), coming up with a value of *this* is easy.

Static members

- Static members are variables that are associated with the class
- There is only one copy of them for the entire class. In the example, $B.x$ accesses class B's x variable.

```
public class B {
   public static double x;
   . . .
}
```

```
B b = new B( );
B.x = 2.0;
b.x = 1.0;
```

Static methods

- Static methods are associated with the class and *not* an object
- Static methods do not have access to a *this* pointer since they are not associated with any object
- Since static methods do not have access to a *this* pointer, they cannot access object members (since they are associated with no object, it is not clear whose object's members they would access.

```
public class B {
   public static double x;
   public float y;
   static void foo( ) {
      // y = 1.0; would be an error!
      x = 1.0;
   . . .
}
```

```
B.foo( );
B b = new B( );
b.foo( );
```

Both call the same static *foo( )* in class *B*. *b.foo( );* gives a warning in some examples. Use *B.foo( )* if possible.

Controlling inheritance in Java

- Java provides the final keyword
- When applied to a method, final keeps the method from being overridden, since to override the method it needs to be redefined, and final means that the given definition is the final definition.
- final public void print( ) { System.out.println(“this is a final, un-extendable method”);}

Attempting to override a final method

```
import java.io.*;

class User {
   private String name;
   private int age;

   public User(String str, int yy)  {name = str; age = yy; }

   final public void print( ) {
      System.out.println("name: " + name + " age: " + age);
   }
}
```

```
class StudentUser extends User {
   private String schoolEnrolled;

   public StudentUser(String nam, int y, String sch) {
      super(nam,y);
      schoolEnrolled = sch;
   }

   public void print( ) {
      super.print( );
      System.out.println("School: " + schoolEnrolled);
   }
}
```

```
[ece-76-55:codew2b/java/finalField] smidkiff% javac Test.java
./StudentUser.java:9: print() in StudentUser cannot override print() in User; overridden method is final
   public void print( ) {
               ^
1 error
```

Classes can be final

public final class finalClass {

- This means that no one can derive from finalClass. Attempts to do so will give a compile time error.
- This is sometimes done for security reasons when you want the behavior of a method or all methods in the class to be unchangeable
  - Object has several final methods that return the runtime class of what is referenced, are used for synchronizing across threads, etc. Note that the Object class itself is not final, however.
- Can allow faster function calling and dispatch since a finalClass reference will always point to a finalClass, and not a derived method. No need to go through the VFT.

parameters can be final

```
void doSomething( final String arg ) {  // Mark argument as 'final'.
  String x = arg;
  x = "elephant";
  arg = "giraffe";  // Compiler error: The passed argument variable arg cannot
be re-assigned to another object.
}
```

The above case doesn't really change anything outside of the function call, but this is bad style and *final* allows the compiler to catch that.

```
void doSomething( final MyClass arg ) {  // Mark argument as 'final'.

   arg =  new MyClass();  // Compiler error: The passed argument variable arg
                          // cannot be re-assigned to another object.

   arg.setX(20); // allowed
  // We can re-assign properties of argument which is marked as final
 }
```

One source of Java OO impurity - not everything is an object

- Java has objects that are instantiations of classes, e.g. *Foo, DFoo*, . . .
- Java has primitives, e.g. *float, int, double*
- This causes problems when you try and write a function that takes either primitives or objects as arguments
  - The operations you can perform on each are often different (e.g. cannot invoke a method on an *int* or a *double*)
  - Have to be very careful when writing code like this
  - Can sometimes hide with other functions (We will do this as a mini programming assignment soon) but it requires effort on the part of the programmer

Abstract Classes

Abstract classes

- Abstract classes are classes for which objects *cannot* be constructed
- They can be derived from, however
- Abstract classes are a general OO concept, not a Java specific thing

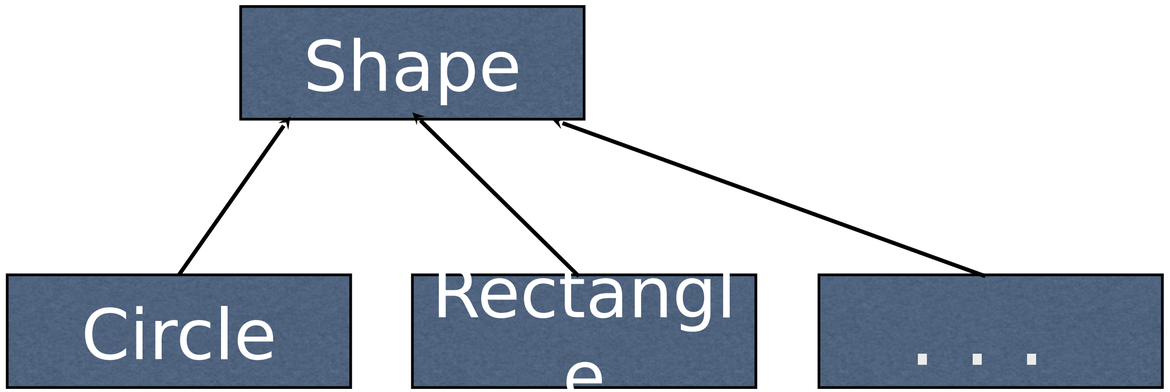
Good for 3 things

1. Can lend organization to a class hierarchy,
2. Provides a common base class
3. Can represent a specialized behavior that when mixed with other *interfaces* gives a desired behavior

Can help build up an implementation

Let’s look at a concrete example to make these concepts clearer. In particular, let’s look at a shape class such as might be used in a drawing program

A shape class



- It makes sense to construct a *Circle*, a *Rectangle*, etc., but not necessarily a shape
- It is useful to be able to refer to all shapes with a common class

A Shape abstract class in Java

```
abstract class Shape {
public:
   abstract public double area( );
   abstract public double circumference;
   public double notAbstract( ) {
      System.out.println("Abstract classes can contain non-abstract methods!");
   }
   . . .
}
```

A Shape interface in Java

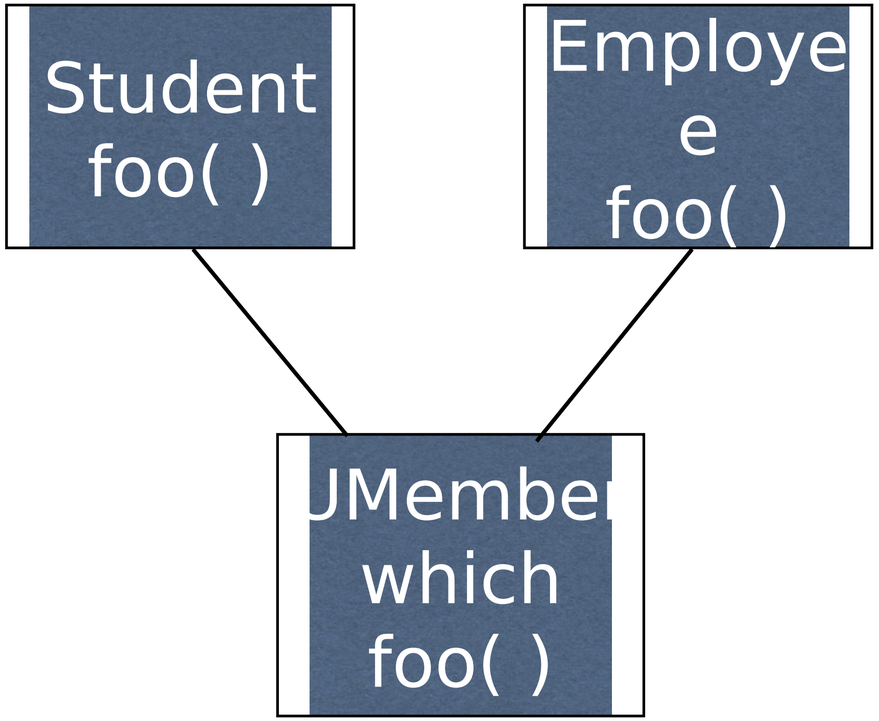
```
interface Shape {
public:
   double pi = 3.14;
   abstract public double area( );
   public double circumference; // note abstract is
optional
   // . . .
};
```

A Java *interface* - interfaces can only contain abstract **methods** and (non-abstract) constants.

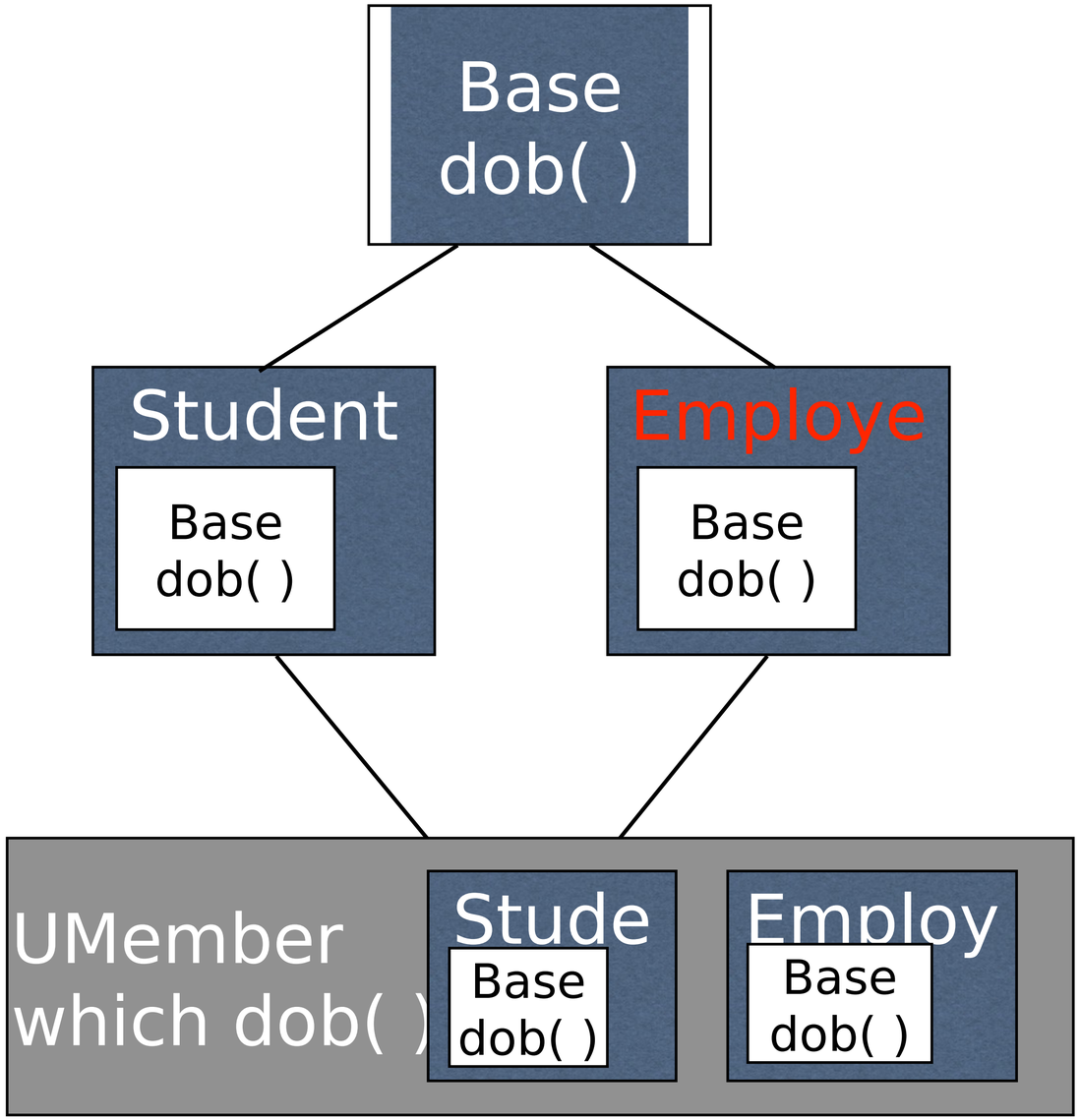
Why two ways with Java?

- C++ allows multiple inheritance, Java does not
- Thus, in C++ if a derived class *UMember* needs to be both a *Student* and an *Employee*, the *UMember* class can inherit from both *Student* and *Employee.*
  - Things get ugly if both *Student* and *Employee* implement (directly or via inheritance) some method or field, say DOB?
  - Which should be called in *UMember* when DOB not overridden? Implementation dependent, and "modern" compilers flagging it as an error. ***Don't do this!***

Multiple inheritance problem



Another problem



Interface or abstract?

- Use an abstract class A if the inheriting (derived) class D *ISA* A
- Use an interface *I* if the inheriting class D *has the capabilities of* an *I*
- When to use one or the other is not completely clear, however.
  - If you can define many, but not all, of the behaviors in the base class you might want to use an abstract class
  - If you want most or all methods overridden to have a specialized version in the derived class, you may want to use an interface
    - If most everything is overridden, there is not a strong *ISA* relationship

Java prohibits multiple inheritance

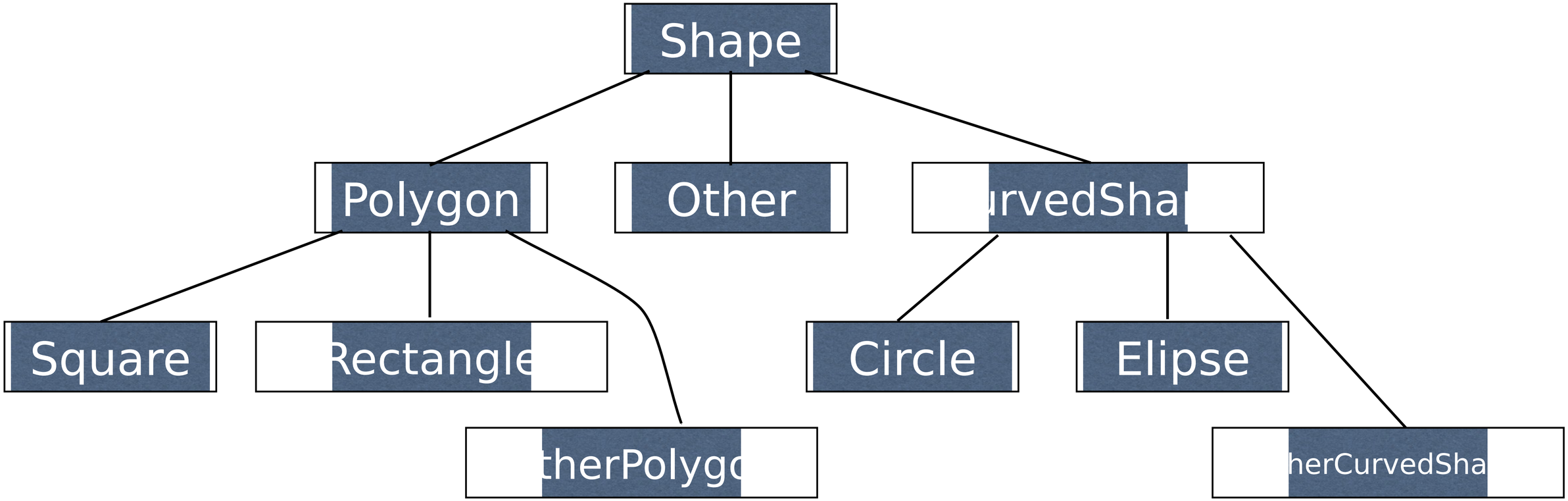
```
interface Shape {
   public double area ( );
   public double circumference( );
}

class Rectangle implements Shape
{
   // ...
   // implement code for area and
   // circumference
   // ...
}
```

- Java uses interfaces to force implementation of multiple base class properties
- But the Java class hierarchy designer still needs a way to force a sub-class to implement functionality for two "base classes"

Interfaces can extend other interfaces

- Only do this when you want to add to the functionality of the interface
- You cannot implement methods in the derived interface for the base methods of the interface
  - There was/is an Eclipse bug that will claim the base class interface method's implementation is in the derived interface that *extends* it.

A longer Java example



A longer Java abstract class example

```
abstract class Shape {
   abstract protected double area( );
   abstract protected double circumference( );
}

abstract class Polygon extends Shape {
     protected int numVertices;
     protected boolean starShape;
}

abstract class curvedShape extends Shape {
   protected void polygonalApprox( );
}
```

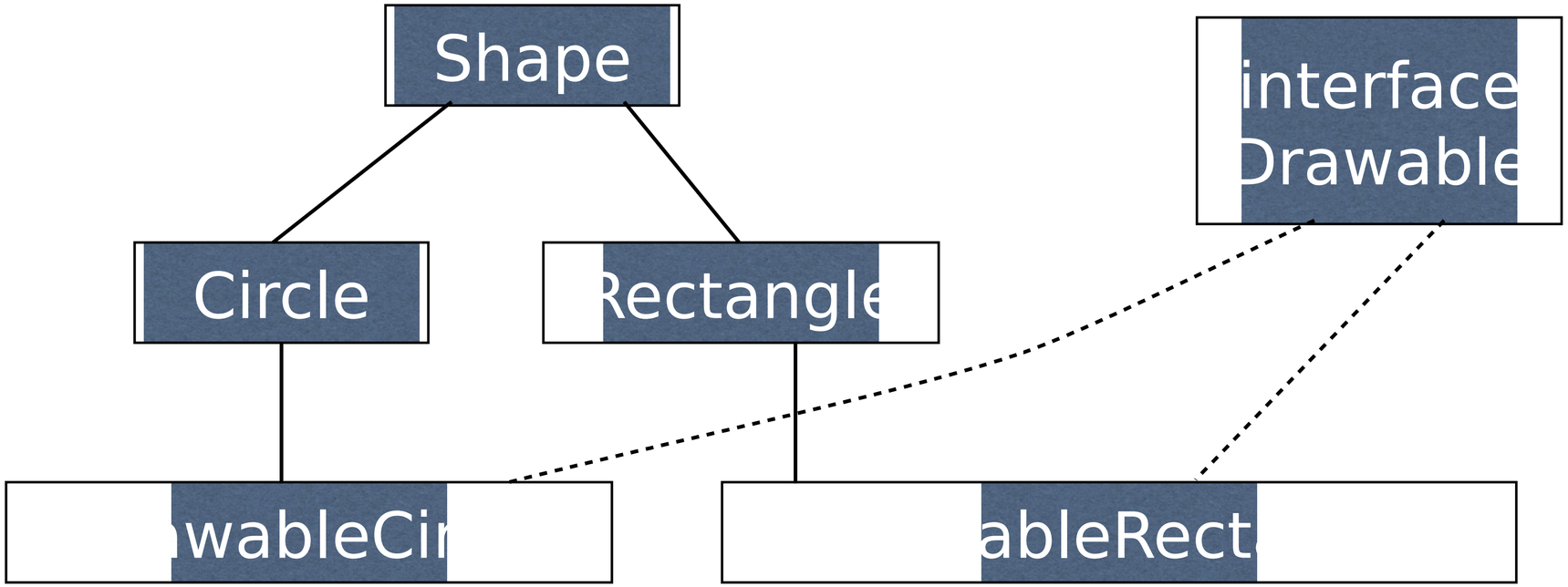
```
class Circle extends CurvedShape {
   protected double r;
   protected static double PI = 3.14159;

   public Circle( ) {r = 1.0;}
   public Circle (double r) {this.r = r;}
   public double area( ) {return PI*r*r;}
   public double circumference( ) {return 2*PI*r;}
   public double getRadius( ) {return r;}
   public void polygonalApprox( ) {
      System.out.println("polygonalApprox code goes here");
   }
}
```

```
class Rectangle extends Polygon {
   double w, h;
   public Rectangle( ) {
      w = 0.0; h = 0.0; numVertices = 0; starShaped = true;}
   }
   public Rectangle (double w, double h) {
      this.w = w;
      this.h = h;
      numVertices = 4;
      starShaped = true;
   }
   public double area( ) {return w*h;}
   public double circumference( ) {return 2*(w+h);}
   public double getWidth( ) {return w;}
   public double getHeight( ) {return h;}
}
```

```
class Test {
   public static void main(String[ ] args) {
      Shape [ ] shapes = new Shape[3];
      shapes[0] = new Circle(2.9);
      shapes[1] = new Rectangle(1.0, 3.0);
      shapes[2] = new Rectangle(4.0, 2.0);

      double totArea = 0;
      for (int i = 0; i < shapes.length; i++)
         totArea += shapes[i].area( );
      System.out.println("Total area = " + totArea);
}
```

Interface example in Java

- Consider a *Drawable* class that implements methods for drawing
  - *setColor*
  - *setPosition*
  - *draw*
- In C++ we could implement a *DrawRectangle* class by inheriting from both *Rectangle* and *Drawable*
- Java only allows single inheritance
- Interfaces allows some of the flexibility of multiple inheritance without the problems with same signature methods in both inherited classes


Shape
interface
Drawable
Circle
Rectangle
wableCir
ableRect

```
erface Drawable {
ublic void setColor(Color r);
ublic void setPosition(double x, double y);
ublic void draw (DrawWindow w);


ss DrawableRectangle extends Rectangle implements Drawable {
rivate Color c;
rivate double x, y;
ublic DrawableRectangle(double w, double h) {super(w, h); }
/ here are the implementations of the methods
/ inherited from the interface Drawable:
ublic void setColor(Color c) {this.c = c;}
ublic void setPosition(double x, double y) {
  this.x = x; this.y = y;

ublic void draw(DrawWindow dw) {
  dw.drawRect(x, y, w, h, c);
```

Interfaces

Interfaces and attributes tradeoffs

- In the previous example, could conceivably have an attribute (field in the class) that was a reference to a *drawable* object that handled the draw functions
  - Advantage: the drawable object might actually be able to implement many of the methods of the interface, saving extending classes from having to implement them
  - Disadvantage: if we need to treat Circle, Rectangle, etc., as a drawable objects, cannot do it as Circle, etc., already extend shape

```
Drawable d[ ] = new Drawable[4]; . . . initialize d

for (i=0; i < 4; i++} {d[i].setPosition = f(i);}
```

Multiple Interfaces/Constants

- Multiple interfaces can be *implemented*
- All abstract methods (i.e. all methods) declared in the interface must be implemented (defined)
  - That each method is defined in the implementing class removes any ambiguity as to which interface's method is called - *there is one implemented method that covers all interfaces!*
- Constants can be declared in interfaces and the constants become visible to the implementing class
  - The possibility of ambiguity exists with constants

Can variables be declared in interfaces?

- Yes, but . . .
- They must be a public static final variable

```
public interface foo {
   int i = 0;
}
```

- The declaration above is the same as *public static int i = 0;* even if it is not explicitly declared as such.

What if a static final is declared twice and used?

```
interface Bar {
   static final int c = 10;
}

interface Foo {
   static final double c = 5.0;
}
```

```
class Test implements Foo, Bar {
  void main(String[ ] args) {
     lc = c;
  }
  int lc;
}
```

```
smidkiff% javac Test.java
Test.java:3: reference to c is ambiguous, both variable c
in Foo and variable c in Bar match
     lc = c;
          ^
1 error
```

What if a static final is declared twice and not used?

```
interface Bar {
   static final int c = 10;
}

interface Foo {
   static final double c = 5.0;
}
```

```
class Test implements Foo, Bar {
  void main(String[ ] args) {
     lc = 10;
  }
  int lc;
}
```

```
smidkiff% !javac
javac Test.java
smidkiff%
```

No use of c
No error

Java 8 changes

- What if I define an interface with abstract method f( ), and lots of people implement it, and then
  - I later add the abstract method f2( )?
  - Everyones implementation will break because they don’t implement f2. This seems wrong since they don’t use f2
- Java 8 allows *default* implementations:

```
default public void f2( ) {
    System.out.println(“Default”0;
 }
```

- Cannot implement multiple interfaces with default implementations for multiple functions with the same signatures.

Packages and access modifiers

- Often we have multiple files that work together to provide the same functionality
- For example, we could have a graphics package with Draggable interface, a shape abstract class, and Circle, Rectangle, Line and Point classes.
- They are more closely related than, e.g., a set of classes and interfaces that implements Math functions (sin, logarithmic functions, coordinate system transforms) and a set of classes and interfaces that implement home address validation or SSN validation.
- Packages give us a way to group closely related interfaces and classes together

Some classes in a package

```
//in the Draggable.java file
package graphics;
public interface Draggable {
    ...
}
```

```
//in the Graphic.java file
package graphics;
public abstract class Graphic {
    ...
}
```

```
//in the Circle.java file
package graphics;
public class Circle extends Graphic
    implements Draggable {
    . . .
}
```

```
//in the Rectangle.java file
package graphics;
public class Rectangle extends Graphic
    implements Draggable {
    . . .
}
```

```
//in the Point.java file
package graphics;
public class Point extends Graphic
    implements Draggable {
    . . .
}
```

```
//in the Line.java file
package graphics;
public class Line extends Graphic
    implements Draggable {
    . . .
}
```

Benefits of packages

Being in the same package indicates that the classes are related

Documents where, in this case, *graphics* related code can be found

The package creates a *namespace* and you can have a graphics.Rectangle and a Rectangle class in another (perhaps unnamed) package

Can allow extra access to variables to members of the package while being protected from access outside of the package

Access Modifiers

Modifier	Class	Package	Derived or subclass	World
Public	Can access	Can access	Can access	Can access
Protected	Can access	Can access	Can access	No
None (package)	Can access	Can access	No, unless in the same package.	No
Private	Can access	No	No	No