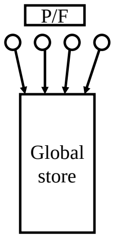
Object-Oriented Programming & Java vs C++

Imperative Programming

- Based on **statements (commands)** that update **variables** held in storage.
  - variable, statements (command), procedures
- Close to machine architecture
  - can be implemented very efficiently
- A long history

Fundamental Problems with IP

- Variables can be potentially accessed and updated by every part of the program!



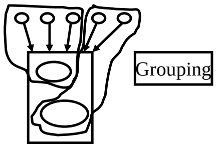
Idea!

- **Encapsulate**
  each global variable in a module with a group of operations that alone have direct access to the variable.
- Other modules can access the variable only indirectly by calling these operations.
  - Called **Objects**
  - Similar to Variables of **abstract data type (ADT)**

Object-Based Programming (OBP)

Object-Based Programming (OBP)

- Based on **Encapsulation**
  - Instead of variable, use an **encapsulated variable + operations** that have the exclusive right to access it.
  - **Object** = **An encapsulated variable + Ops**



Objects

- An encapsulation of a hidden variable (local data, state) and operations operating on that state.
- The data in the object may only be accessed/updated by the operations in the object.

Classes

- **A description of a set of similar objects.**
- A template for objects.
  - Descriptions of the actions an object can perform
  - Definition of the structure of an object's internal state.
  - **Grouping similar objects into a class**
- Each object is an **instance** of some class.

Messages

- A request sent from one object to another for the receiving object to produce some desired result.
  - Subprogram call
  - **A message = a selector that uniquely identifies the desired operation + a set of arguments**
- **Send requests (message) to object**, rather than calling subprograms.

Methods

- An operation that an object performs when it receives a message.
  - Subprograms

Computation with Objects

- Interacting objects:
  - Send requests (**message**) to object.
  - When an object receives a message (**receiver**), it determines whether it has an appropriate operation (**methods**).
  - The object reacts according to the definition of method.
- Message Passing

Binding Messages to Methods

- When an object receives a message, it determines whether it has an appropriate operation (methods).
  - **How to determine?**
    - Match messages with methods
  - **When to determine?**
    - At compile time (**Static binding**)
    - At run time (**dynamic binding**)

OBP with IPL?

- Questions: **OOP can be practiced in an Imperative Language?**
  - Yes, if the language supports the concept of encapsulation.
    - **Ada -> packages**
    - **Modula ->modules**

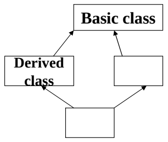
Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP)

- OBP + More:
  - **Subclassing and Inheritance**
  - **Dynamic Binding and Inclusion Polymorphism**

Subclassing

- An ability to organize object classes into a hierarchy of subclasses & superclasses and for operations of a given class to be applicable to objects of its subclasses.
  - **Grouping objects into class hierarchy**



Subclasses and Superclasses

- A class that inherits is a **derived class** or a **subclass**.
- The class from which another class inherits is a **parent class** or **superclass**.

Inheritance

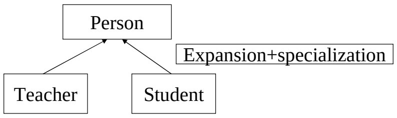
- Variables (state) and methods are inherited.
- Why?
  - **Software reuse!**

When Subclassing?

- **Expansion**
- **Specialization** (***overriding/overridden***)

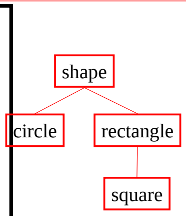
Expansion and Specialization

- Add new additional data & operation
  - **expansion**
- Redefine existing operations supported by the superclass
  - **specialization** (***overriding/overridden***)



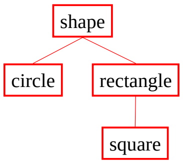
Example: Subclassing and Inheritance in Java

```
public class shape {
 public void draw(){ …}
 …
}
public class circle extends shape {
 public void draw() { …}
 …
}
public class rectangle extends shape {
 public void draw() { …}
 …
}
public class square extends rectangle
{
 public void draw() { …}
 …
}
```



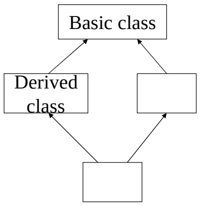
Example: Subclassing and Inheritance in C++

```
class shape {
 public: virtual void draw(){ …}
 …
}
class circle : public shape {
 public: virtual void draw() { …}
 …
}
class rectangle : public shape {
 public: virtual void draw() { …}
 …
}
class square : public rectangle {
 public: virtual void draw() { …}
 …
}
```



Single vs. Multiple Inheritance

- Single inheritance
  - Only one parent
  - Tree
- Multiple inheritance
  - More than one parent
  - Graph



Inheritance

- Any disadvantage of inheritance for reuse:
  - Creates interdependencies among classes that complicate maintenance.

Polymorphism

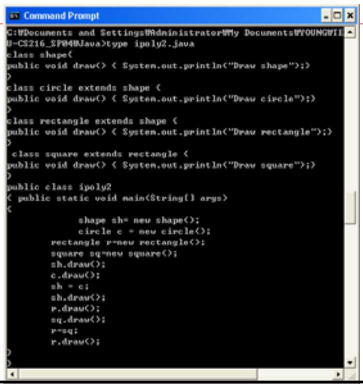
- A **polymorphic variable** of the type of the parent class is able to reference (or point to) objects of any of the subclasses of that class.
  - **Inclusion polymorphism**
- Why?
  - Software reuse!

Example: Polymorphism in Java

```
public class shape {
 public void draw(){ …}
 …
}
public class circle extends shape {
 public void draw() { …}
 …
}
public class rectangle extends shape
{
 public void draw() { …}
 …
}
public class square extends
rectangle {
 public void draw() { …}
 …
}
```

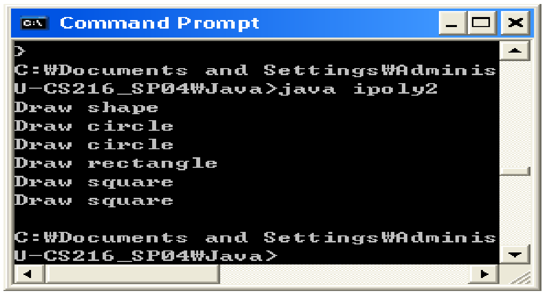
```
shape sh=new shape();
circle c=new circle();
rectangle r=new
rectangle();
square sq=new square();
        sh.draw();
        c.draw();
        sh = c;
        sh.draw();
        r.draw();
        sq.draw();
        r=sq;
        r.draw();
```

Example: Polymorphism in Java



```
C:\Documents and Settings\Administrator\My Documents\YOUNG\TI
U-CS216_SP04\Java>type ipoly2.java
class shape{
public void draw() { System.out.println("Draw shape");}
}
class circle extends shape {
public void draw() { System.out.println("Draw circle");}
}
class rectangle extends shape {
public void draw() { System.out.println("Draw rectangle");}
}
 class square extends rectangle {
public void draw() { System.out.println("Draw square");}
}
public class ipoly2
{ public static void main(String[] args)
{
             shape sh= new shape();
             circle c = new circle();
        rectangle r=new rectangle();
        square sq=new square();
        sh.draw();
        c.draw();
        sh = c;
        sh.draw();
        r.draw();
        sq.draw();
        r=sq;
        r.draw();
}
}
```

Example: Polymorphism in Java



```
>
C:\Documents and Settings\Adminis
U-CS216_SP04\Java>java ipoly2
Draw shape
Draw circle
Draw circle
Draw rectangle
Draw square
Draw square

C:\Documents and Settings\Adminis
U-CS216_SP04\Java>
```

Example: Polymorphism in Java

```
shape sh=new shape();
circle c=new circle();
rectangle r=new
rectangle();
square sq=new square();
        sh.draw();
        c.draw();
        sh = c;
        sh.draw();
        r.draw();
        sq.draw();
        r=sq;
        r.draw();
```

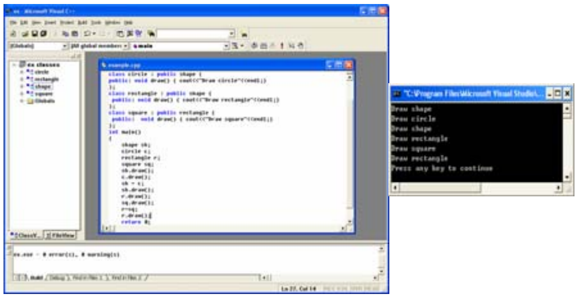
Draw Shape!
Draw Circle!
Draw **Circle**!
Draw Rectangle!
Draw Square!
Draw **Square**!

Example: Polymorphism in C++

```
class shape {
 public: virtual void draw(){…}
 …
}
class circle : public shape {
 public: virtual void draw() { …}
 …
}
class rectangle : public shape {
 public: virtual void draw() { …}
 …
}
class square : public rectangle {
 public: virtual void draw() { …}
 …
}
```

```
shape sh;
circle c;
rectangle r;
square sq;
        sh.draw();
        c.draw();
        sh = c;
        sh.draw();
        r.draw();
        sq.draw();
        r=sq;
        r.draw();
```

Example: Polymorphism in C++



ex - Microsoft Visual C++
File Edit View Insert Project Build Tools Window Help
(Globals)
(All global members)
main
ex classes
circle
rectangle
shape
square
Globals
ClassV...
FileView
example.cpp
"C:\Program Files\Microsoft Visual Studi...
Build / Debug / Find in Files 1 / Find in Files 2 /
Ln 27, Col 14

```
class circle : public shape {
public: void draw() { cout<<"Draw circle"<<endl;}
};
class rectangle : public shape {
 public: void draw() { cout<<"Draw rectangle"<<endl;}
};
class square : public rectangle {
 public:  void draw() { cout<<"Draw square"<<endl;}
};
int main()
{
    shape sh;
    circle c;
    rectangle r;
    square sq;
    sh.draw();
    c.draw();
    sh = c;
    sh.draw();
    r.draw();
    sq.draw();
    r=sq;
    r.draw();
    return 0;
```

```
Draw shape
Draw circle
Draw shape
Draw rectangle
Draw square
Draw rectangle
Press any key to continue
```

```
ex.exe - 0 error(s), 0 warning(s)
```

Example: Polymorphism in C++

```
shape sh;
circle c;
rectangle r;
square sq;
sh.draw();
c.draw();
sh = c;
sh.draw();
r.draw();
sq.draw();
r=sq;
r.draw();
```

Draw Shape!
Draw Circle!
Draw **Shape**!
Draw Rectangle!
Draw Square!
Draw **Rectangle**!

Example: Polymorphism in C++

```
class shape {
 public: virtual void draw(){…}
 …
}
class circle : public shape {
 public: virtual void draw() { …}
 …
}
class rectangle : public shape {
 public: virtual void draw() { …}
 …
}
class square : public rectangle {
 public: virtual void draw() { …}
 …
}
```

```
shape *sh;
circle *c;
rectangle *r;
square *sq;
sh = new shape;
Sh->draw();
c =  new circle;
C->draw();
sh = c;
sh->draw();
r =new rectangle;
r->draw();
sq = new square;
sq->draw();
r=sq;
r->draw();
```

Example: Polymorphism in C++

```
shape *sh;
circle *c;
rectangle *r;
square *sq;
sh = new shape;
sh->draw();
c =  new circle;
c->draw();
sh = c;
sh->draw();
r =new rectangle;
r->draw();
sq = new square;
sq->draw();
r=sq;
r->draw();
```

Draw Shape!
Draw Circle!
Draw **Circle**!
Draw Rectangle!
Draw Square!
Draw **Square**!

Example: Polymorphism in C++

```
shape sh;
circle c;
rectangle r;
square sq;
sh.draw();
c.draw();
sh = c;
sh.draw();
r.draw();
sq.draw();
r=sq;
r.draw();
```

Draw Shape!
Draw Circle!
Draw Shape!
Draw Rectangle!
Draw Square!
Draw Rectangle!

```
shape *sh;
circle *c;
rectangle *r;
square *sq;
sh = new shape;
sh->draw();
c =  new circle;
c->draw();
sh = c;
sh->draw();
r =new rectangle;
r->draw();
sq = new square;
sq->draw();
r=sq;
r->draw();
```

Draw Shape!
Draw Circle!
Draw Circle!
Draw Rectangle!
Draw Square!
Draw Square!

C++ vs Java

- A C++ variable can hold an object or a pointer to an object. There are two selectors:
  - **`a->x`** selects method or field x when a is a pointer to an object
  - **`a.x`** selects x when a is an object
- A Java variable cannot hold an object, only a reference to an object. Only one selector:
  - **`a.x`** selects x when a is a reference to an object

Method Binding

- Binding of messages to methods
  - Static binding
  - Dynamic binding

Dynamic Binding

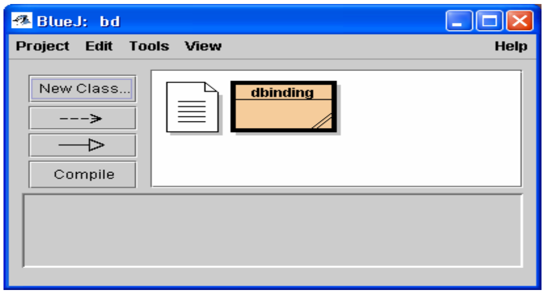
- When a class hierarchy includes classes that override methods and such methods are called through a polymorphic variable, the binding to the correct method must be dynamic.
- **Binding of messages to methods at run-time!**
- Why?
  - Allows software to be more easily extended during development and maintenance.

Example: Java

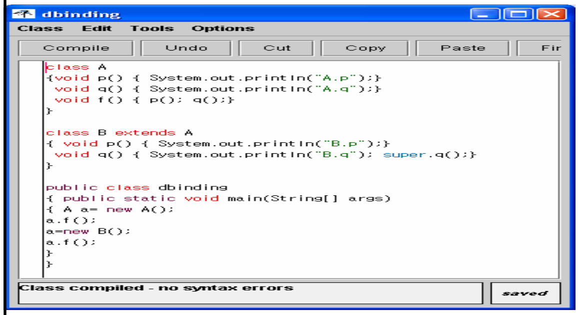
```
class A
{void p() { System.out.println("A.p");}
 void q() { System.out.println("A.q");}
 void f() { p(); q();}
}

class B extends A
{ void p() { System.out.println("B.p");}
 void q() { System.out.println("B.q"); super.q();}
}

public class dbinding
{ public static void main(String[] args)
{ A a= new A();
a.f();
a=new B();
a.f();
}
}
```

```
A::p
A::q
B::p
B::q
A::q
```

Example: Java



Example: Java

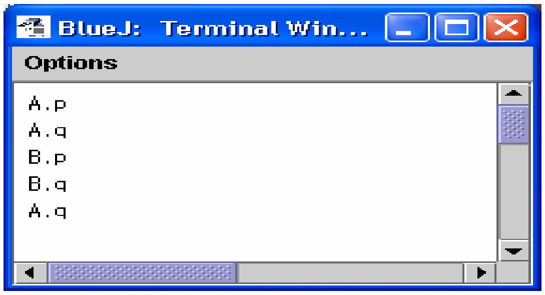


```
class A
{void p() { System.out.println("A.p");}
 void q() { System.out.println("A.q");}
 void f() { p(); q();}
}

class B extends A
{ void p() { System.out.println("B.p");}
 void q() { System.out.println("B.q"); super.q();}
}

public class dbinding
{ public static void main(String[] args)
{ A a= new A();
a.f();
a=new B();
a.f();
}
}
```

Class compiled - no syntax errors

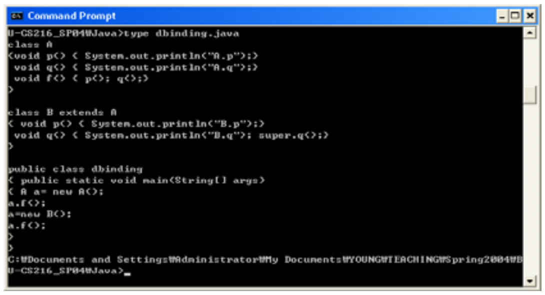
saved

Example: Java



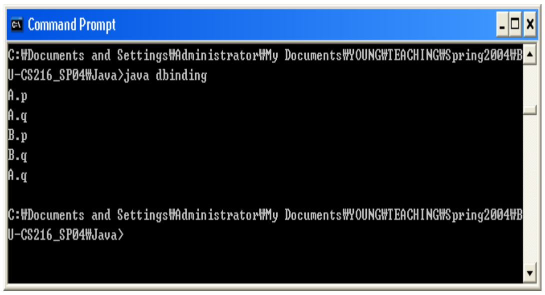
```
A.p
A.q
B.p
B.q
A.q
```

Example: Java



```
U-CS216_SP04₩Java>type dbinding.java
class A
{void p() { System.out.println("A.p");}
 void q() { System.out.println("A.q");}
 void f() { p(); q();}
}

class B extends A
{ void p() { System.out.println("B.p");}
 void q() { System.out.println("B.q"); super.q();}
}

public class dbinding
{ public static void main(String[] args)
{ A a= new A();
a.f();
a=new B();
a.f();
}
}
C:₩Documents and Settings₩Administrator₩My Documents₩YOUNG₩TEACHING₩Spring2004₩B
U-CS216_SP04₩Java>_
```

Example: Java



```
C:₩Documents and Settings₩Administrator₩My Documents₩YOUNG₩TEACHING₩Spring2004₩B
U-CS216_SP04₩Java>java dbinding
A.p
A.q
B.p
B.q
A.q

C:₩Documents and Settings₩Administrator₩My Documents₩YOUNG₩TEACHING₩Spring2004₩B
U-CS216_SP04₩Java>
```

Static Binding

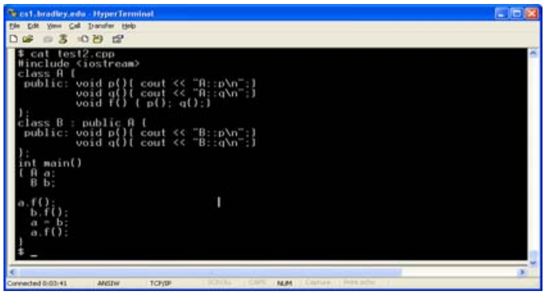
- **Binding of messages to methods at compile-time!**
- Why?
  - Efficient!

Example 1: C++

```
class A {
 public: void p(){ cout << "A::p\n";}
         void q(){ cout << "A::q\n";}
         void f() { p(); q();}
};
class B : public A {
 public: void p(){ cout << "B::p\n";}
         void q(){ cout << "B::q\n";}
};
int main()
{ A a;
  B b;
  a.f();
  b.f();
  a = b;
  a.f();
}
```

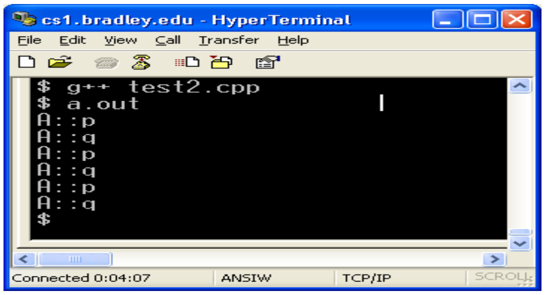
```
A::p
A::q
A::p
A::q
A::p
A::q
```

Example 1: C++


cs1.bradley.edu - HyperTerminal
File Edit View Call Transfer Help
Connected 0:03:41 ANSIW TCP/IP SCROLL CAPS NUM Capture Print echo

```
$ cat test2.cpp
#include <iostream>
class A {
 public: void p(){ cout << "A::p\n";}
         void q(){ cout << "A::q\n";}
         void f() { p(); q();}
};
class B : public A {
 public: void p(){ cout << "B::p\n";}
         void q(){ cout << "B::q\n";}
};
int main()
{ A a;
  B b;

a.f();
  b.f();
  a = b;
  a.f();
}
$ _
```

Example 1: C++


cs1.bradley.edu - HyperTerminal
File Edit View Call Transfer Help
Connected 0:04:07 ANSIW TCP/IP SCROLL

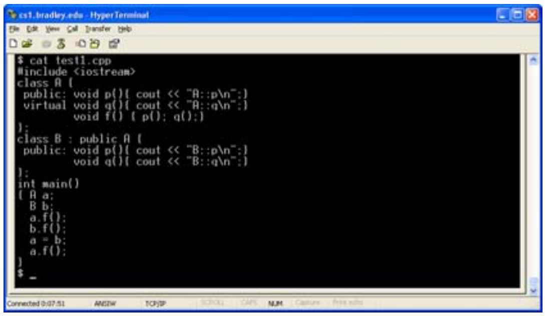
```
$ g++ test2.cpp
$ a.out
A::p
A::q
A::p
A::q
A::p
A::q
$
```

Example 2: C++ virtual for Dynamic Binding

```
class A {
 public: void p(){ cout << “A::p\n”;}
 virtual void q(){ cout << “A::q\n”;}
         void f() { p(); q();}
};
class B : public A {
 public: void p(){ cout << “B::p\n”;}
         void q(){ cout << “B::q\n”;}
};
int main()
{ A a;
  B b;
  a.f();
  b.f();
  a = b;
  a.f();
}
```

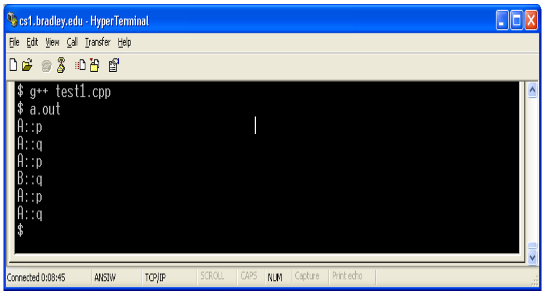
```
A::p
A::q
A::p
B::q
A::p
A::q
```

Example 2: C++


cs1.bradley.edu - HyperTerminal
File Edit View Call Transfer Help
Connected 0:07:51 ANSIW TCP/IP SCROLL CAPS NUM Capture Print echo

```
$ cat test1.cpp
#include <iostream>
class A {
 public: void p(){ cout << "A::p\n";}
 virtual void q(){ cout << "A::q\n";}
         void f() { p(); q();}
};
class B : public A {
 public: void p(){ cout << "B::p\n";}
         void q(){ cout << "B::q\n";}
};
int main()
{ A a;
  B b;
  a.f();
  b.f();
  a = b;
  a.f();
}
$ _
```

Example 2: C++


cs1.bradley.edu - HyperTerminal
File Edit View Call Transfer Help
Connected 0:08:45 ANSIW TCP/IP SCROLL CAPS NUM Capture Print echo

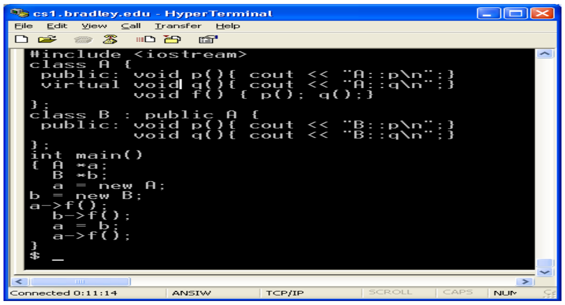
```
$ g++ test1.cpp
$ a.out
A::p
A::q
A::p
B::q
A::p
A::q
$
```

Example 3: C++

```
class A {
 public: void p(){ cout << “A::p\n”;}
 virtual void q(){ cout << “A::q\n”;}
         void f() { p(); q();}
};
class B : public A {
 public: void p(){ cout << “B::p\n”;}
         void q(){ cout << “B::q\n”;}
};
int main()
{ A *a;
  B *b;
  a = new A;
  b = new B;
  a->f();
  b->f();
  a = b;
  a->f();
}
```

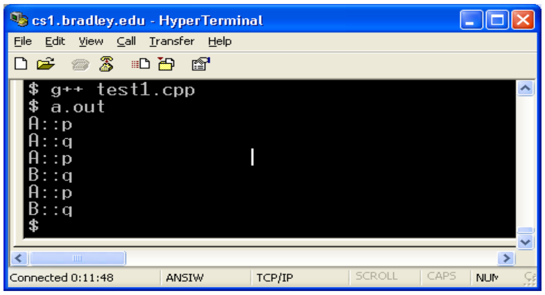
```
A::p
A::q
A::p
B::q
A::p
B::q
```

Example 3: C++


cs1.bradley.edu - HyperTerminal
File Edit View Call Transfer Help
Connected 0:11:14 ANSIW TCP/IP SCROLL CAPS NUM

```
#include <iostream>
class A {
 public: void p(){ cout << "A::p\n";}
 virtual void q(){ cout << "A::q\n";}
         void f() { p(); q();}
};
class B : public A {
 public: void p(){ cout << "B::p\n";}
         void q(){ cout << "B::q\n";}
};
int main()
{ A *a;
  B *b;
  a = new A;
b = new B;
a->f();
  b->f();
  a = b;
  a->f();
}
$ _
```

Example 3: C++


cs1.bradley.edu - HyperTerminal
File Edit View Call Transfer Help
Connected 0:11:48 ANSIW TCP/IP SCROLL CAPS NUM

```
$ g++ test1.cpp
$ a.out
A::p
A::q
A::p
B::q
A::p
B::q
$
```

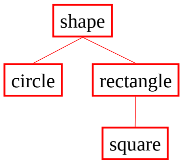
Example All: C++

```
class A {
public: void p(){ cout << "A::p";}
 virtual void q(){ cout << "A::q";}
         void f() { p(); q();}
};
class B : public A {
 public: void p(){ cout << "B::p";}
         void q(){ cout << "B::q";}
};
int main()
{ A a; A *aa=new A;
  B b; B *bb=new B;
  a = b;
  a.f();a.p();a.q();
  aa=bb;
  aa->f();aa->p();aa->q();
}
```

```
A::p
A::q
A::p
A::q
A::p
B::q
A::p
B::q
```

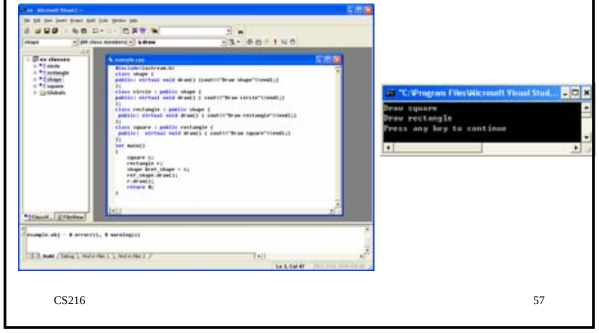
Example 4: C++

```
class shape {
 public: virtual void draw() { …};
 …
}
class circle : public shape {
 public: virtual void draw() { …}
 …
}
class rectangle : public shape {
 public: virtual void draw() { …}
 …
}
class square : public rectangle {
 public: virtual void draw() { …}
 …
}
```



```
square s;
rectangle r;
shape &ref_shape = s;
ref_shape.draw();
r.draw();
```

Example 4: C++



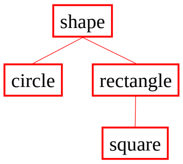
```
#include<iostream.h>
class shape {
public: virtual void draw() {cout<<"Draw shape"<<endl;}
};
class circle : public shape {
public: virtual void draw() { cout<<"Draw circle"<<endl;}
};
class rectangle : public shape {
 public: virtual void draw() { cout<<"Draw rectangle"<<endl;}
};
class square : public rectangle {
 public:  virtual void draw() { cout<<"Draw square"<<endl;}
};
int main()
{
    square s;
    rectangle r;
    shape &ref_shape = s;
    ref_shape.draw();
    r.draw();
    return 0;
}
```

```
example.obj - 0 error(s), 0 warning(s)
```

```
Draw square
Draw rectangle
Press any key to continue
```

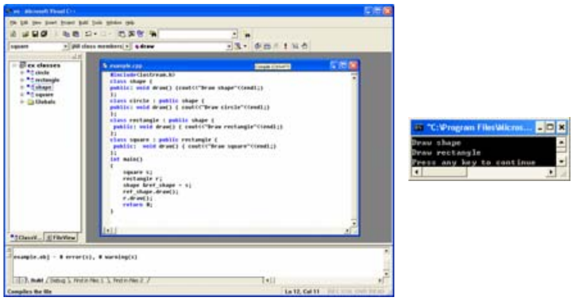
Example 5: C++

```
class shape {
 public: void draw() { …};
 …
}
class circle : public shape {
 public: void draw() { …}
 …
}
class rectangle : public shape {
 public: void draw() { …}
 …
}
class square : public rectangle {
 public: void draw() { …}
 …
}
```



```
square s;
rectangle r;
shape &ref_shape = s;
ref_shape.draw();
r.draw();
```

Example 5: C++



```
#include<iostream.h>
class shape {
public: void draw() {cout<<"Draw shape"<<endl;}
};
class circle : public shape {
public: void draw() { cout<<"Draw circle"<<endl;}
};
class rectangle : public shape {
 public: void draw() { cout<<"Draw rectangle"<<endl;}
};
class square : public rectangle {
 public:  void draw() { cout<<"Draw square"<<endl;}
};
int main()
{
    square s;
    rectangle r;
    shape &ref_shape = s;
    ref_shape.draw();
    r.draw();
    return 0;
}
```

```
example.obj - 0 error(s), 0 warning(s)
```

```
Draw shape
Draw rectangle
Press any key to continue
```

Design Issues for OOPLs

- OOPLs are programming languages that support OOP well.

The Exclusivity of Objects

- Everything is an object.
  - – Elegance and purity
  - – Slow operations on simple objects (e.g., float)

The Exclusivity of Objects

- Include an imperative-style typing system for primitives but make everything else objects.
  - – Fast operations on simple objects and a relatively small typing system
  - – Still some confusion because of the two type systems

Single and Multiple Inheritance

- **Multiple inheritance**
  - – Disadvantages:
    - Language and implementation complexity
    - A class may inherit from the same base class through more than one path.
    - Potential inefficiency - dynamic binding costs more with multiple inheritance (but not much)
  - – Advantage:
    - Sometimes it is extremely convenient and valuable.

Allocation and Deallocation of Objects

- From where are objects allocated?
  - – If they all live in the heap, references to them are uniform.
- Is deallocation explicit or implicit?

Dynamic and Static Binding

- Should all bindings of messages to methods be dynamic?
  - – If none are, you lose the advantages of dynamic binding.
  - – If all are, it is inefficient.

Java As OOPL

- Single inheritance
- Dynamic binding
- Inclusion polymorphism
- Implicit object deallocation (Garbage collection)

C++ As OOPL

- Multiple inheritance
- Static binding & Dynamic binding (virtual)
- Inclusion polymorphism
- Explicit object deallocation

Implementation of OOPLs

- An object of a class as a structure
- An object of a subclass as an extension of the object of a class
- A method as a function
- Dynamic binding using a **virtual method table** (VMT)