ACCESS CONTROL, CODE REUSE, INHERITANCE, POLYMORPHISM

17TH LECTURE

엄현상(Eom, Hyeonsang)
School of Computer Science and Engineering
Seoul National University

Outline

- **Access Control**
  - Access Control
  - Class Access
- **Code Reuse**
  - Reusing Class
- **Inheritance**
  - Composition vs Inheritance
  - Composition syntax
  - Inheritance syntax
  - final
- **Polymorphism**
  - Interface & Implementation
  - Dynamic Binding
  - Abstract classes
  - Constructors
  - Pure Inheritance vs Extension

Java Access Control

- Introduction
- Function Templates
- Overloading Function Templates
- Class Templates
- Nontype Parameters and Default Types for Class Templates

Java Access Control

public	Interface Access
private	Only Accessible Within the class
protected	“Sort of private” deals with inheritance

“Friendly”

- Default access : no keyword
- **Public**
  - Other members of the same package
- **Private**
  - Anyone outside the package
  - Easy interaction for related classes (that you place in the same package)
  - Also referred to as “package access”

Protected

- **protected**
  - Inheritors (and the package) can access protected members

  BUT
  - They are then vulnerable to changes in the base-class implementation
  - Users of classes not in the class hierarchy are prevented from accessing protected members

public: Interface Access

```
package c05.dessert;

public class Cookie {
  public Cookie() {
    System.out.println("Cookie
constructor");
  }
  void bite()
{ System.out.println("bite"); }
} ///:~
```

```
//: c05:Dinner.java
// Uses the library.
import c05.dessert.*;

public class Dinner {
  public Dinner() {
    System.out.println("Dinner
constructor");
  }
  public static void main(String[]
args) {
    Cookie x = new Cookie();
    //! x.bite(); // Can't access
  }
} ///:~
```

private: Can't Touch That!

```
//: c05:IceCream.java
// Demonstrates "private" keyword.

class Sundae {
  private Sundae() {}
  static Sundae makeASundae() {
    return new Sundae();
  }
}

public class IceCream {
  public static void main(String[] args) {
    //! Sundae x = new Sundae();
    Sundae x = Sundae.makeASundae();
  }
} ///:~
```

Class Access

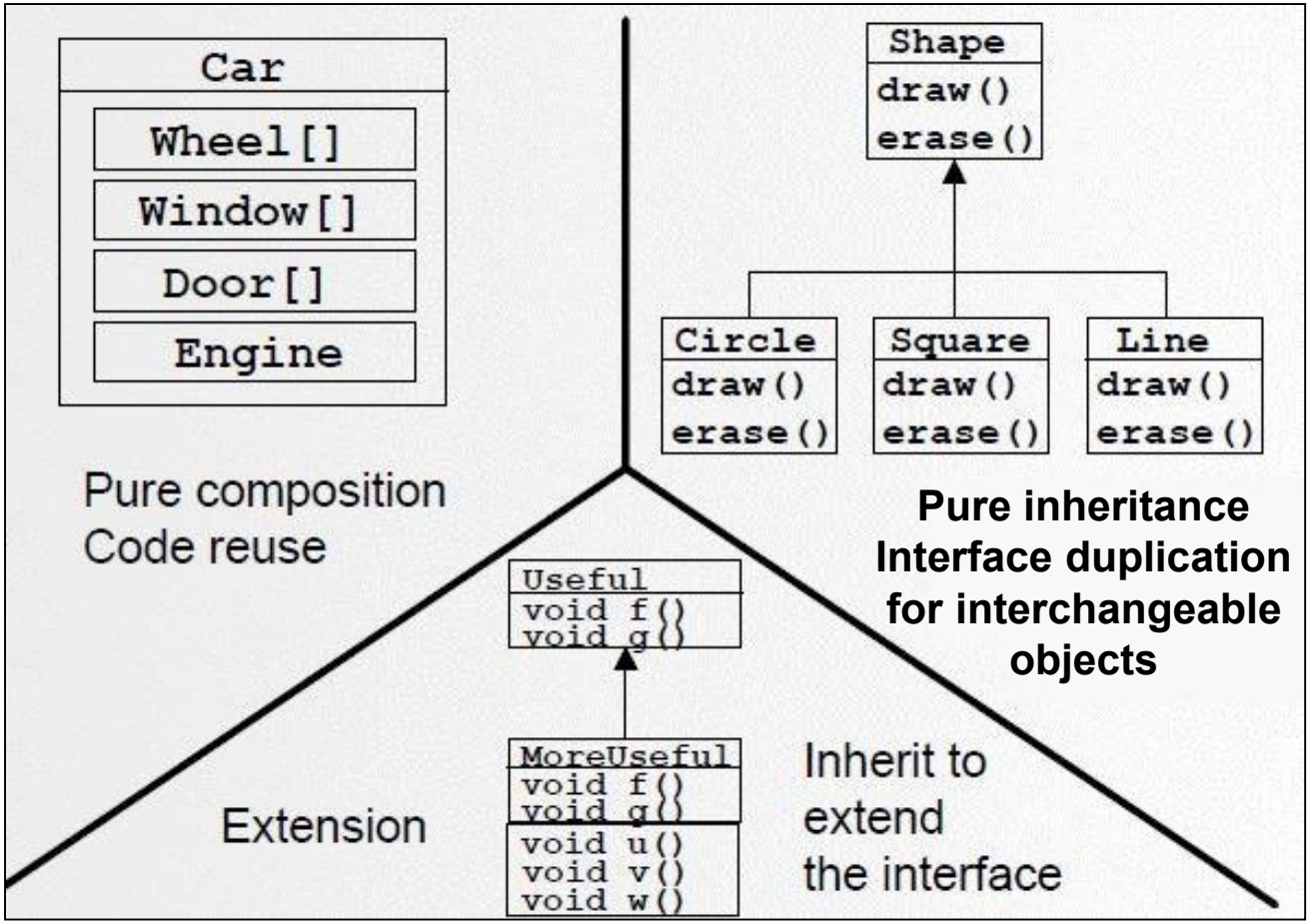
- Classes as a whole can be **public** or "friendly"
- Only one **public** class per file, usable outside the package
- All other classes "friendly," only usable within the package

Reusing Classes

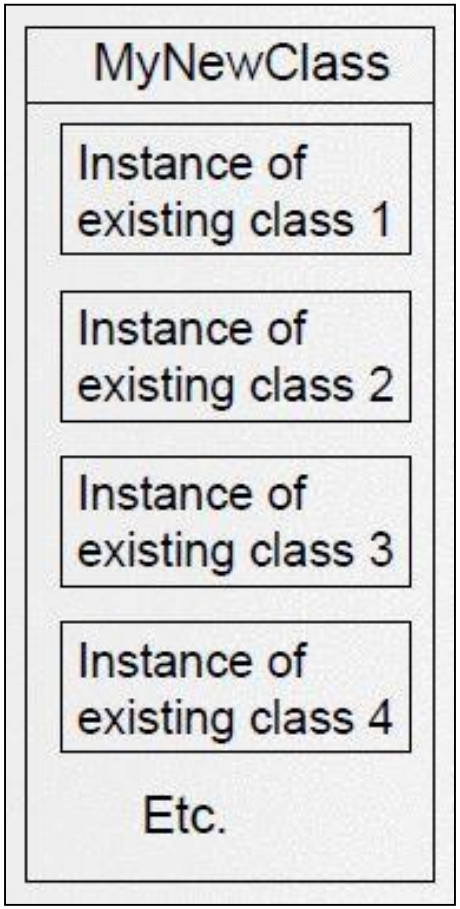
- When you need a class, you can:
  1) Get the perfect one off the shelf (one extreme)
  2) Write it completely from scratch (the other extreme)
  3) Reuse an existing class with composition
  4) Reuse an existing class or class framework with inheritance

현재 statement 하에서는 즉, 현재 패키지 내에서는
디렉토리를 다 쓰지 않아도 괜찮다. 클래스 파일 실행 가능하다.

Composition vs. Inheritance



Composition Syntax




```
class MyNewClass {
        Foo x = new Foo();
        Bar y = new Bar();
        Baz z = new Baz();
        // ...
}
```

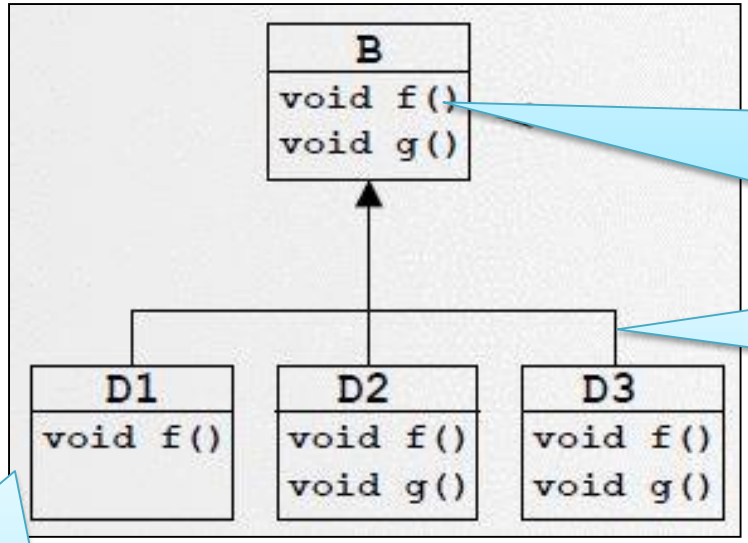
- Can also initialize in the constructor
- Flexibility: Can change objects at run time!
- "Has-A" relationship

Composition Syntax

```
class Soap {
  private String s;
  Soap() {
    System.out.println("Soap()");
    s = new String("Constructed");
  }
  public String toString() { return s; }
}

public class Bath {
  private String
    // Initializing at point of definition:
    s1 = new String("Happy"),
    s2 = "Happy",
    s3, s4;
  Soap castille;
  int i;
  float toy;
  Bath() {
    System.out.println("Inside Bath()");
    s3 = new String("Joy");
    i = 47;
    toy = 3.14f;
    castille = new Soap();
  }
  void print() {
    // Delayed initialization:
    if(s4 == null)
      s4 = new String("Joy");
    System.out.println("s1 = " + s1);
    System.out.println("s2 = " + s2);
    System.out.println("s3 = " + s3);
    System.out.println("s4 = " + s4);
    System.out.println("i = " + i);
    System.out.println("toy = " + toy);
    System.out.println("castille = " + castille);
  }
  public static void main(String[] args) {
    Bath b = new Bath();
    b.print();
  }
} ///:~
```

Inheritance Syntax

```
class B {
        public void f(){
        /* … */ }
        public void g(){
        /* … */ }
}
```



Base interface automatically duplicated in derived classes

Base data members also duplicated

If a derived member is not redefined, base definition is used

```
class D1 extends B {
public void f() {
        /* … */
        }
}
```

Inheritance Syntax

```
class Cleanser {
  private String s = new
String("Cleanser");
  public void append(String a) { s +=
a; }
  public void dilute() { append("
dilute()"); }
  public void apply() { append("
apply()"); }
  public void scrub() { append("
scrub()"); }
  public void print()
{ System.out.println(s); }
  public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    x.print();
  }
}
```

```
public class Detergent extends Cleanser {
  // Change a method:
  public void scrub() {
    append(" Detergent.scrub()");
    super.scrub(); // Call base-class version
  }
  // Add methods to the interface:
  public void foam() { append(" foam()"); }
  // Test the new class:
  public static void main(String[] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    x.print();
    System.out.println("Testing base class:");
    Cleanser.main(args);
  }
} ///:~
```

Initializing the Base class

- Java automatically calls default constructors

```
class Art {
  Art() {
    System.out.println("Art constructor");
  }
}

class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing constructor");
  }
}
public class Cartoon extends Drawing {
  Cartoon() {
    System.out.println("Cartoon constructor");
  }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
  }
} ///:~
```

>>

Art constructor
Drawing constructor
Cartoon constructor

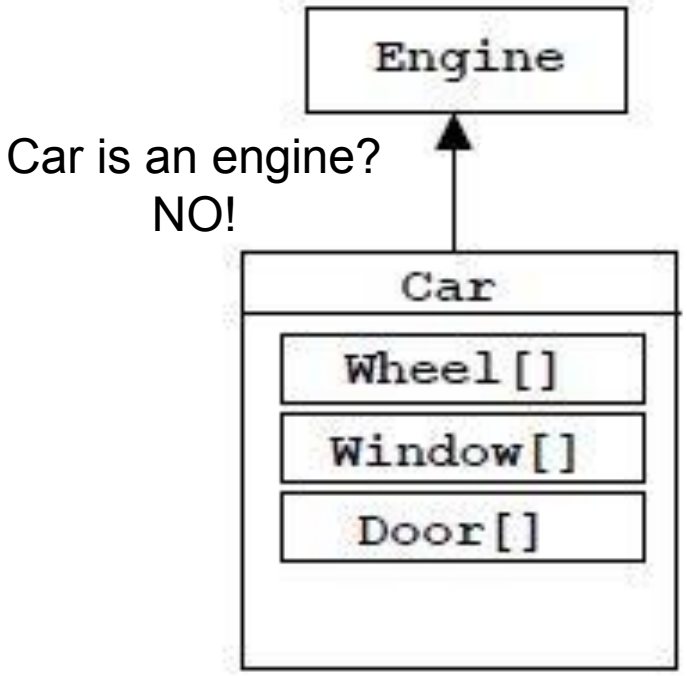
Constructors with Arguments

```
class Game {
  Game(int i) {
    System.out.println("Game constructor");
  }
}

class BoardGame extends Game {
  BoardGame(int i) {
    super(i);
    System.out.println("BoardGame constructor");
  }
}

public class Chess extends BoardGame {
  Chess() {
    super(11);
    System.out.println("Chess constructor");
  }
  public static void main(String[] args) {
    Chess x = new Chess();
  }
} ///:~
```

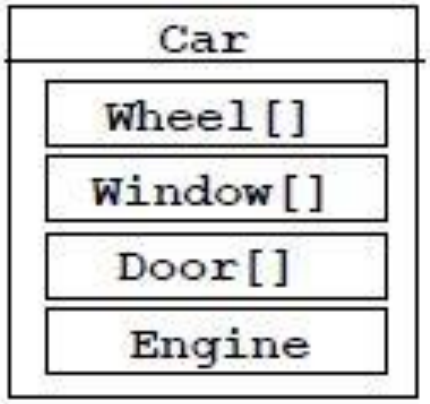
- Base constructor call must happen first
- Use **super** keyword

차라리 파라매터 없는 constructor거나 둘다 없으면 괜ㅊ낳지만
하나라도 있으면 문제?
여기서는 super를 불러줘야한다.

Choosing Composition vs. Inheritance



WRONG



RIGHT

- Inheritance is determined at compile time; member binding can be delayed until run time
- Member rebinding is possible
- In general, prefer composition to inheritance as a first choice

Initialization with Inheritance

```
class Insect {
  int i = 9;
  int j;
  Insect() {
    prt("i = " + i + ", j = " + j);
    j = 39;
  }
  static int x1 =
    prt("static Insect.x1 initialized");
  static int prt(String s) {
    System.out.println(s);
    return 47;
  }
}
public class Beetle extends Insect {
  int k = prt("Beetle.k initialized");
  Beetle() {
    prt("k = " + k);
    prt("j = " + j);
  }
  static int x2 =
    prt("static Beetle.x2 initialized");
  public static void main(String[] args) {
    prt("Beetle constructor");
    Beetle b = new Beetle();
  }
} ///:~
```

>>

static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 47
j = 39

main이 시작하기도 전에 static initial?
base -> sub 나중

non static base 먼저

The final Keyword

- Slightly different meanings depending on context
- "This cannot be changed"
- A bit confusing: two reasons for using it
  - Design
  - Efficiency
- **final** fields
- **final** methods
- **final** class

각각 무엇을 할 수 없는지...

final Fields

1) Compile-time constant

Must be given a value at point of definition

final static int NINE = 9;

May be "folded" into a calculation by the compiler

2) Run-time constant

Cannot be changed from initialization value

final int RNUM = (int)(Math.random()*20);

- **final static**: only one instance per class, initialized at the time the class is loaded, cannot be changed.
- **final** references: cannot be re-bound to other objects

final arguments & final Classes

사용은 일반적으로 굳이...? 어차피 영향 안주는디!
걍 이렇게도 있답 reference rebind 할 수 없고,
primitive는 값 변경 불가능 //////

- **Final arguments**
  - **void f(final int i) { // ...**
    - Primitives can't be changed inside method
  - **void g(final Bob b) { // ...**
    - References can't be rebound inside method
    - Generally, neither one is used
- **Final Class** overriding 고려할 것도 없이 implicitly final 이다. 필요에 따라 선언 가능 혹은 안할수도!
  - Cannot inherit from **final** class
  - All methods are implicitly **final** == > overide 안한다는 말과 같음
  - Fields may be **final** or not, as you choose

field는 변경 가능?

final Methods

1) Put a "lock" on a method to prevent any inheriting class from overriding it (design)

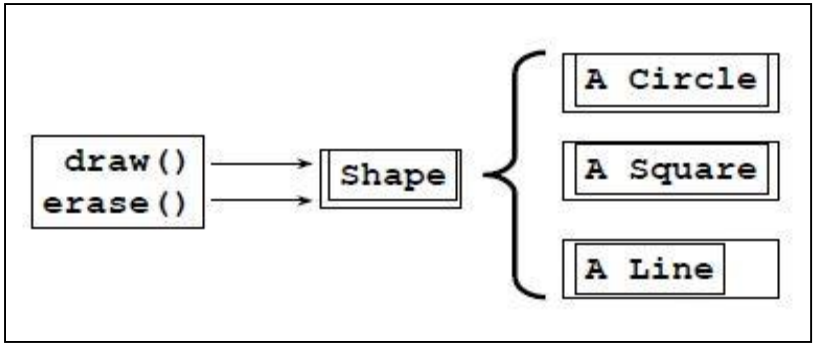
2) Efficiency (try to avoid the temptation…)

- Compiler has permission to "inline" a final method
  - Replace method call with code
  - Eliminate method-call overhead
- Programmers are characteristically bad about guessing where performance problems are
- Limits use of class (example: can't override Vector)

•Private methods are implicitly final

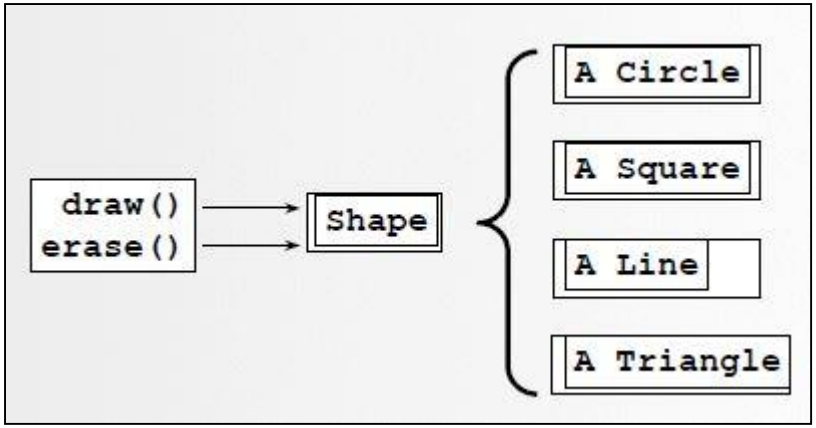
overiding 될 일 없겠쥬?
** final 자체 시험 내기 좋음

Polymorphism



dynamic binding 이다
class 위계에서 대체, 추가 다 가능하다.

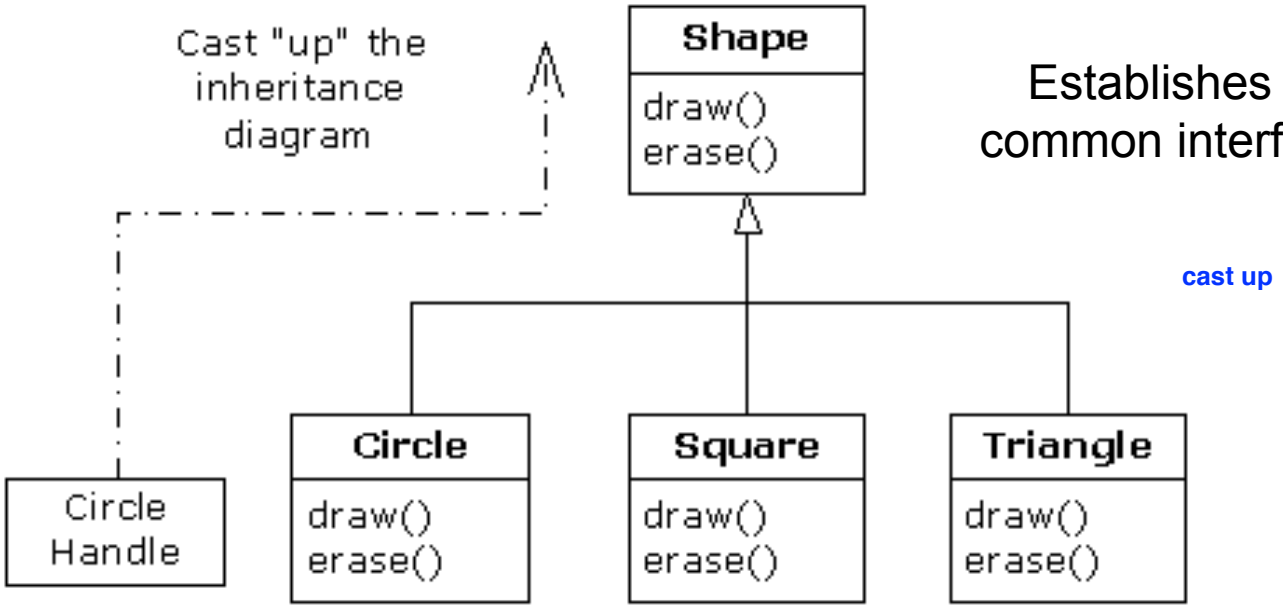
"Substitutability"



"Extensibility"

Interface & Implementation

Shape s = new Circle();



Implementations of the common interface

Why Upcast?

새로운 sub class 생길 수 있는데, 해당 위계 upcast 사용하는데, 나중에 sub 추가 되더라도 , super objext 처럼 해 놓으면 읽기 쉽고, 일관성이 잇으며, 좋다. 하나의 공통 인터페이스니까

슈퍼 인터처럼 해놓으면 쓰기에도 읽기에서 쉽고, 좋다

- All true OOP programs have some upcasting somewhere
- Upcasting allows us to isolate type specific details from the bulk of your code, i.e. decoupling
- Code is simpler to write and read
- Most important
  - Changes in type do not propagate changes in code

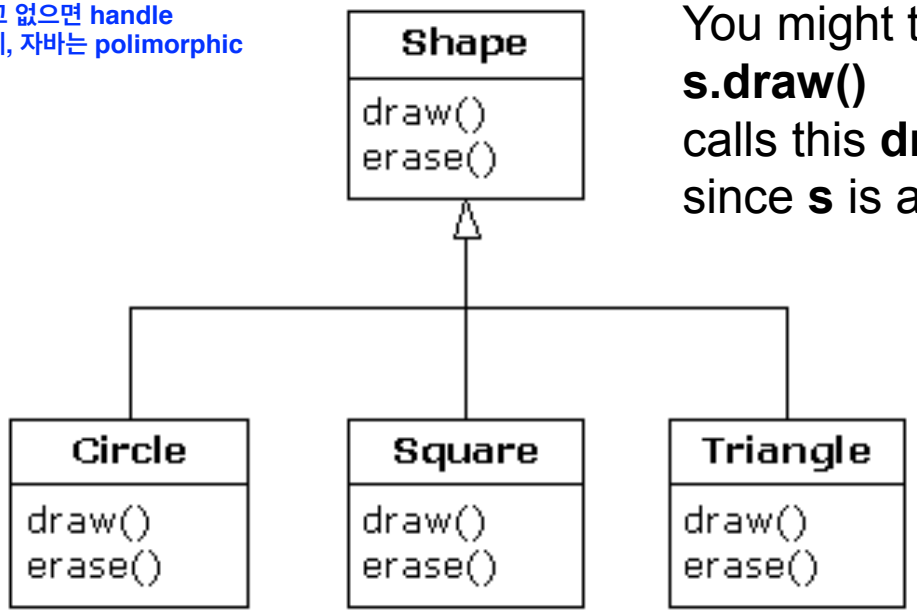
이렇게 하면, sub class 타입의 변화가 안나타난다. ?? 슈퍼 어쩌고로 하게되면

실제로는 가르키는 오브젝트이지만, 마치 invoke를 하면, invoke 될거라 착각 놉
실제적으로는

A problem

Shape s = new Circle();

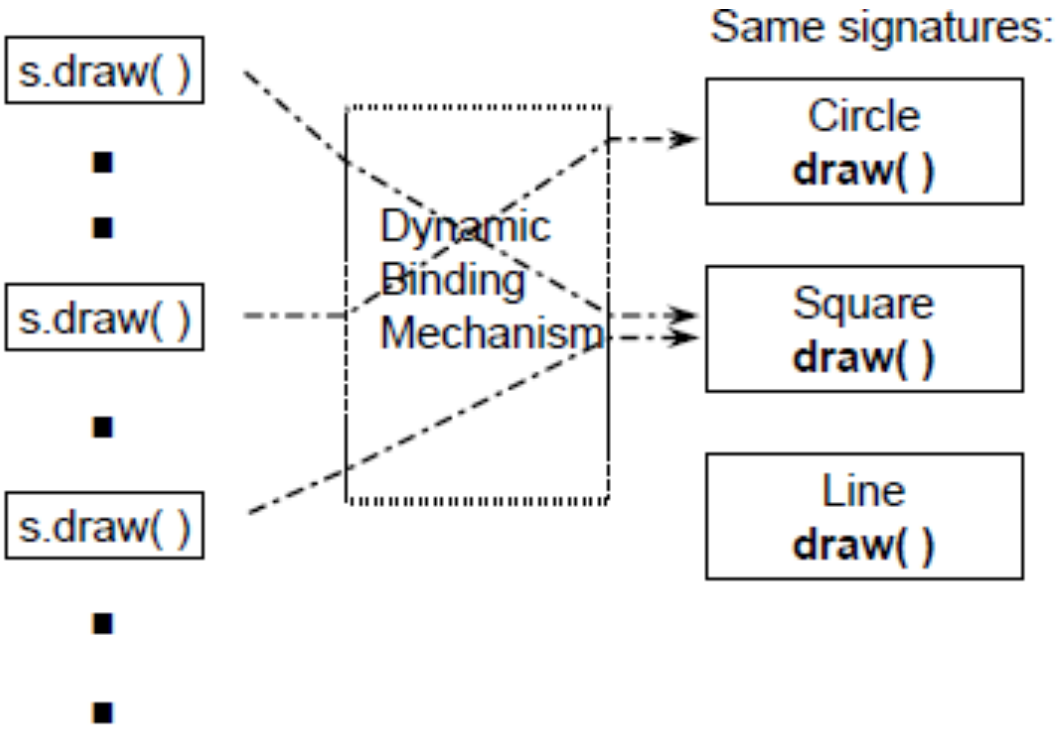
c++은 virtual 보고 없으면 handle type을 보고 하는데, 자바는 polimorphic 하게…



You might think
s.draw()
calls this **draw()**
since **s** is a **Shape**

We want it to call this **draw()** since **s** actually points to a **Circle**

Dynamic Binding in Java



Dynamic Binding in Java

```
class Shape {
  void draw() {}
  void erase() {}
}

class Circle extends Shape {
  void draw() {
    System.out.println("Circle.draw()"); }
  }
  void erase() {
    System.out.println("Circle.erase()");
  }
}

class Square extends Shape {
  void draw() {
    System.out.println("Square.draw()");
  }
  void erase() {
    System.out.println("Square.erase()");
  }
}
```

```
class Triangle extends Shape {
  void draw() {
    System.out.println("Triangle.draw()");
  }
  void erase() {
    System.out.println("Triangle.erase()");
  }

public class Shapes {
  public static Shape randShape() {
    switch((int)(Math.random() * 3)) {
      default:
      case 0: return new Circle();
      case 1: return new Square();
      case 2: return new Triangle();
    }
  }
```

return을 하면서 upcast가 발생한다.

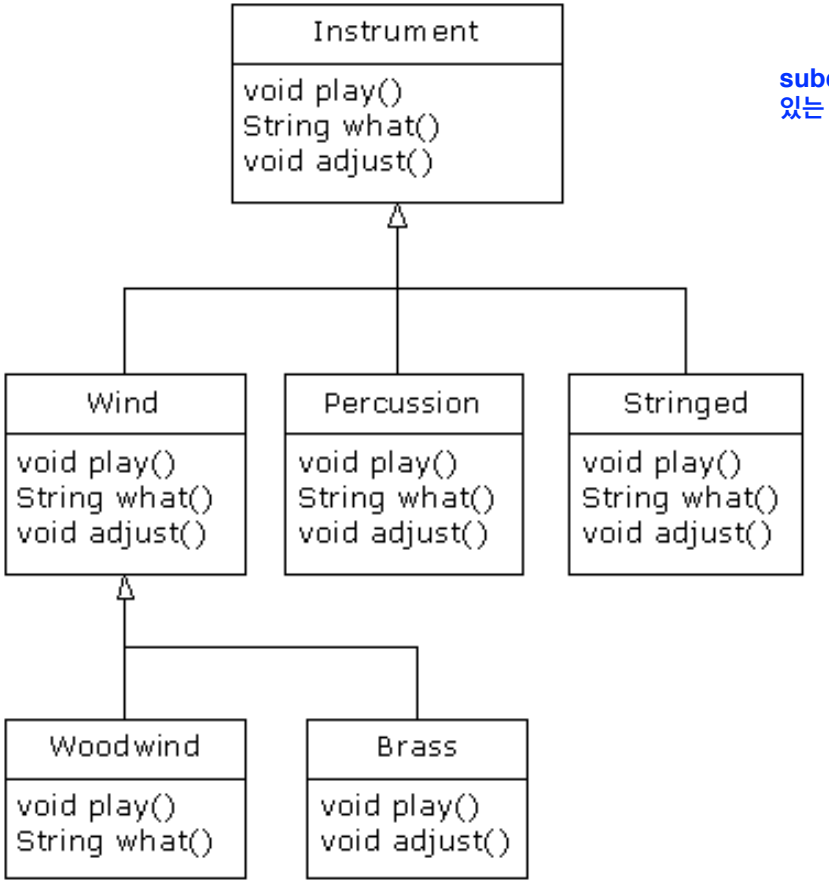
해당 클래스 실행시 main 실행 되겠쥬~

```
public static void main(String[] args) {
    Shape[] s = new Shape[9];
    // Fill up the array with shapes:
    for(int i = 0; i < s.length; i++)
      s[i] = randShape();
    // Make polymorphic method calls:
    for(int i = 0; i < s.length; i++)
      s[i].draw();
  }
} ///:~
```

이 배열 요소들은 shape이 아니라
각각의 오브젝트가 가르키는 class의
reference(?)가 가르키는 것이다
random number seed에 의해서 진행이라서
우리가 프로그램 짜서 돌려도 같은 결과일듯

```
>>
Circle.draw()
Triangle.draw()
Circle.draw()
Circle.draw()
Circle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Square.draw()
```

Extensibility

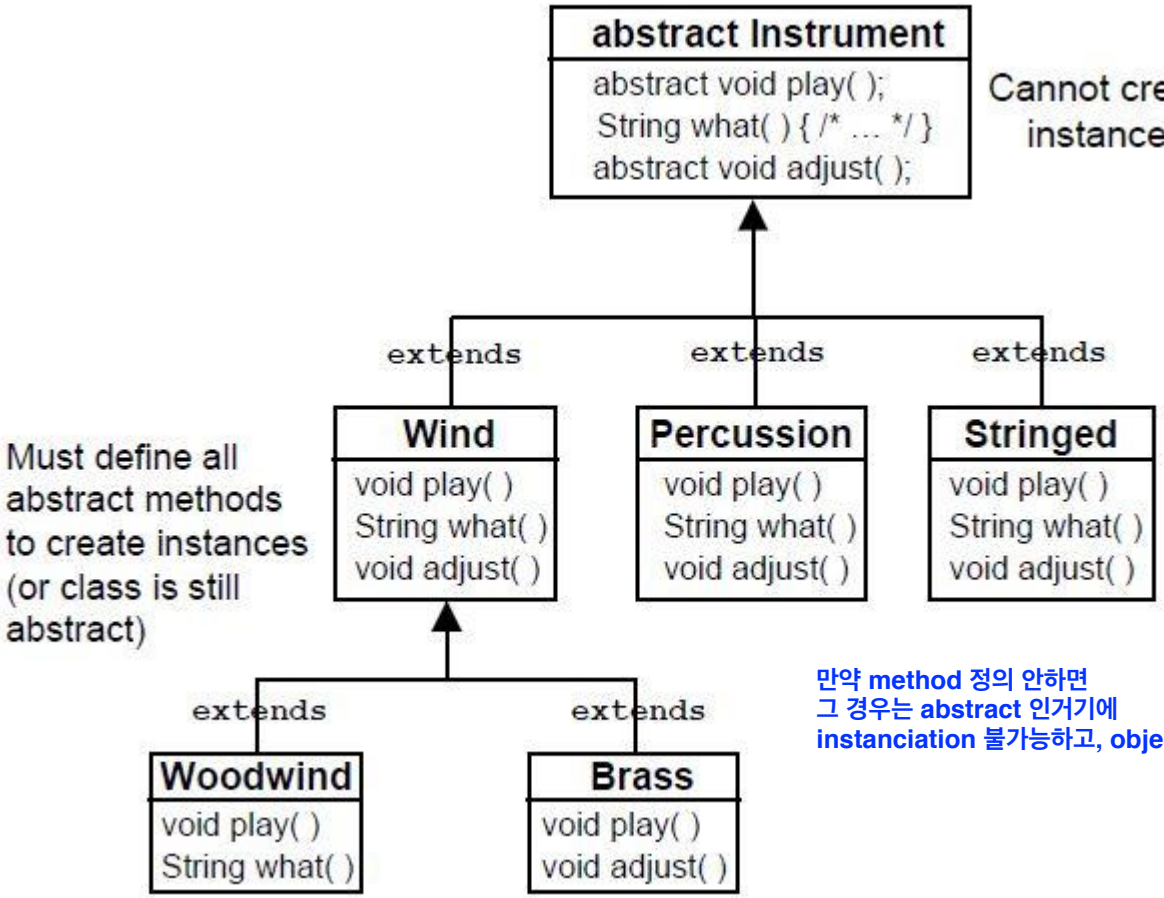


subclass에 subclass가 있는 경우이다 ㅎㅎ

```
import java.util.*;

class Instrument {
  public void play() {
    System.out.println("Instrument.play()");
  }
  public String what() {
    return "Instrument";
  }
  public void adjust() {}
}

class Wind extends Instrument {
  public void play() {
    System.out.println("Wind.play()");
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Percussion extends Instrument {
  public void play() {
    System.out.println("Percussion.play()");
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed extends Instrument {
  public void play() {
    System.out.println("Stringed.play()");
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play() {
    System.out.println("Brass.play()");
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
}

class Woodwind extends Wind {
  public void play() {
    System.out.println("Woodwind.play()");
  }
  public String what() { return "Woodwind"; }
}

class Woodwind extends Wind {
  public void play() {
    System.out.println("Woodwind.play()");
  }
  public String what() { return "Woodwind"; }
}

public class Music3 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument i) {
    // ...
    i.play();
  }
  static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    tuneAll(orchestra);
  }
} ///:~
```

```
>>
Wind.play()
Percussion.play()
Stringed.play()
Brass.play()
Woodwind.play()
```

Abstract classes & abstract methods



pure만 있음 놉 concret으로 재정의 필요 -> c++

자바는 abstract 클래스에서 method들은 pure virtual function 같이 method body가 없어서 inherit define 필요 따라서 후에 instanciation가능. Object 만들 수 있답

만약 method 정의 안하면
그 경우는 abstract 인거기에
instanciation 불가능하고, object만들 수 없다.

1

An abstract Instrument

- Some methods and data may be defined

```
abstract class Instrument {
        int i; // storage allocated for each
        public abstract void play();
        public String what() {
                return "Instrument";
        }
        public abstract void adjust();
}
```

- Rest of the code is the same…
- This maps with C++, may have been an early design

Constructors & Polymorphism

Order of constructor calls

```
class Meal {
  Meal() { System.out.println("Meal()"); }
}

class Bread {
  Bread() { System.out.println("Bread()"); }
}

class Cheese {
  Cheese() { System.out.println("Cheese()"); }
}

class Lettuce {
  Lettuce() { System.out.println("Lettuce()"); }
}
```

상위 클래스 에서도 이니셜 컨스트럭터 등 있을 때,
initialization -> constructor 순이다? 이것도
밑의 경우랑 같다??

```
class Lunch extends Meal {
  Lunch() { System.out.println("Lunch()");}
}

class PortableLunch extends Lunch {
  PortableLunch() {
    System.out.println("PortableLunch()");
  }
}
```

실행 call 순서랑 invocation 순서랑 다르다.
initialization 전에 super class 먼저 만들어지고,
본인 constructor는 그 후? 등등. 순서 잘 알아야한다.
super -> object들 -> 본인 초기화 ??

```
public class Sandwich extends PortableLunch {
  Bread b = new Bread();
  Cheese c = new Cheese();
  Lettuce l = new Lettuce();
  Sandwich() {
    System.out.println("Sandwich()");
  }
  public static void main(String[] args
    new Sandwich();
  }
} ///:~
```

```
> >
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()
```

Order of Initialization

아 이거 초기화 잘해야, 한다는 의미인듯
내 과제에서처럼!!! 초기화 먼저하고 변수사용

상속 -. constructor overwritten된 거 사용한다.
만약에 먼저 overwrittend 된거 invoke 후 다른거 initialize 한다고 생각해 보아라. overwriiten 된거, initialization 될 거를 가지고 사용해야 한다고 할 때 문제가 생긴다. ** 중요 **

1) Base-class constructor is called

   This step is repeated recursively such that the very root of the hierarchy is constructed first, followed by the next derived class, etc., until the most derived class is reached

2) Member initializers are called in the order of declaration

3) Body of the derived-class constructor is called

is a relationship vs

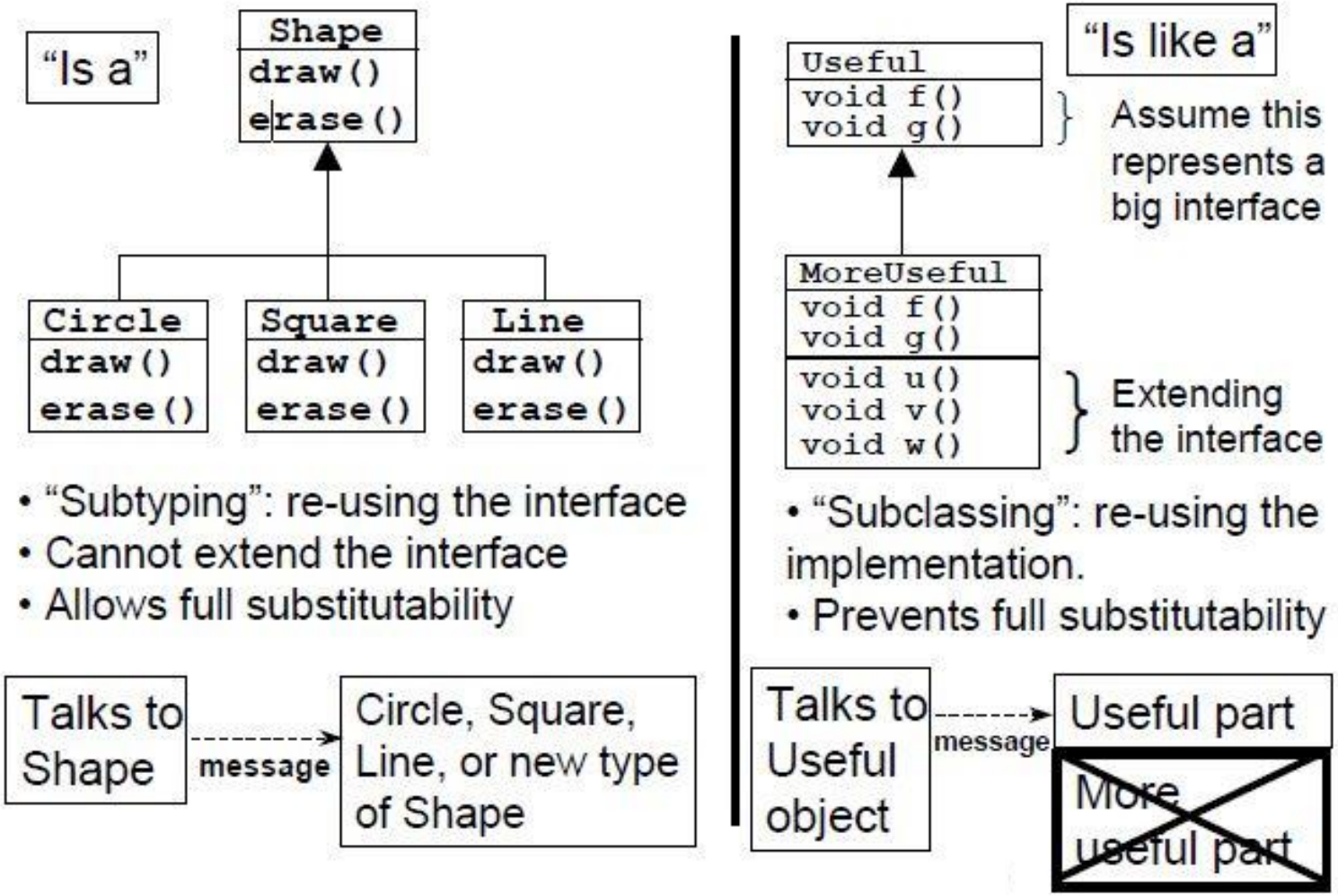
Polymorphism & Constructors

- Inside constructor
  - Overridden method *is* used!
- This can produce incorrect results
  - Overridden method assumes all parts of derived class have been initialized
  - But you're still in the base part of the constructor
  - Derived constructor hasn't been called yet!

```
class Base {
    int  x=10;

    Base() {
      show();
    }

    void show() {
        System.out.print ("Base Show " +x + "  ");
    }
}

class Child extends Base {
    int x=20;

    Child() {
        show();
    }

    void show() {
        System.out.print("Child Show " + x +"  ") ;
    }

    public static void main( String s[ ] ) {
        Base obj = new Child();
    }
}
```

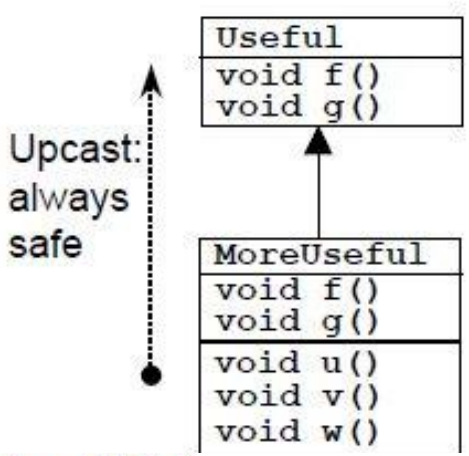
이 경우에, Base() constructor에서 overriden 된 Child class의 show() method를 call하게 되는데, 이때 아직 child class가 초기화되지 않았기 때문에 문제가 발생한다. 출력값은 “Child Show 0”

Pure Inheritance vs. Extension



Downcasting & Run-Time Type Identification (RTTI)

- Normally try to do everything with upcasting
- If you extend the class, you must downcast to access extended methods
- Java casts are always checked at run-time (safe)



x[1].g() -? 무엇?
subclass의 g()
reference가 가르키는 class의 것이다.
example 물어볼것 ㅎㅅㅎ

base derived
t실행결과 물어볼 수 있답

```
class Useful {
  public void f() {}
  public void g() {}
}

class MoreUseful extends Useful {
  public void f() {}
  public void g() {}
  public void u() {}
  public void v() {}
  public void w() {}
}
```

Extends the interface

```
public class RTTI {
  public static void main(String args[]) {
    Useful x[] = {
      new Useful(),
      new MoreUseful()
    };
    x[0].f();
    x[1].g();
    // Compile-time: method not found in Useful:
    //! x[1].u();
    ((MoreUseful)x[1]).u(); // Downcast/RTTI
    ((MoreUseful)x[0]).u(); // Exception thrown
  }
}
```

redifine이 되어 있으면, 불리고 아니면, useful게 불린다는 것~~~

실제적으로는 useful object가르키고, 정의가 안되어 있으니, 당근 exception

down cast 괜찮지만, // 뒤의 경우는 정의 x

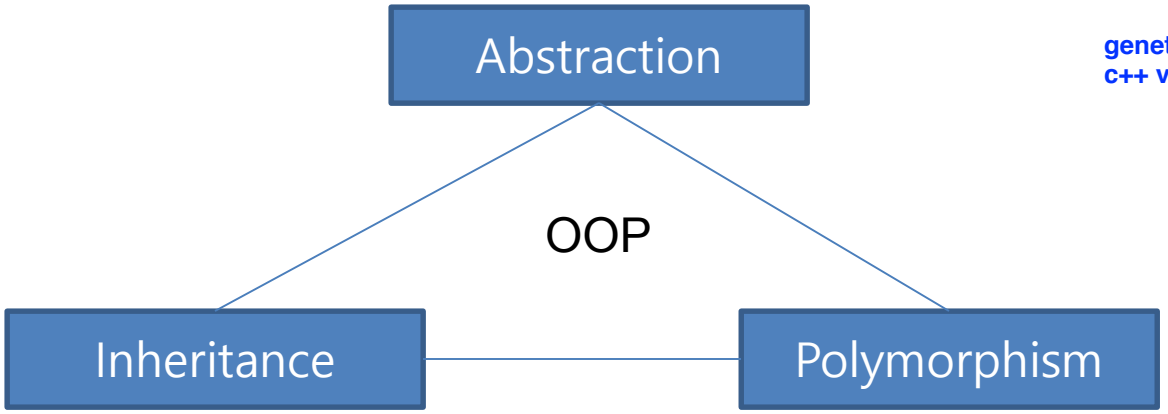
Summary

- Access control
  - What users can & can't use (shows the area of interest)
  - Separating interface & implementation
  - Allowing the class creator to change the implementation later without disturbing client code
  - An important design & implementation flexibility
- Design guideline
  - Always make elements "as **private** as possible"

Summary Cont'd

- Easy to think that OOP is *only* about inheritance
- Often easier and more flexible to start with composition code reuse 가능하게 함.
  - Remember to say "has-a" and "is-a" is s 는 inheritance 사용해야한다.
- Use inheritance when it's clear that a new type is a kind of a base type

Summary Cont'd



genetic stack 만들기
c++ vs java

- Not just creating types, but proper behavior in all situations
- Allows decoupling of code from specific type it's acting on: easy reading, writing & extensibility

c & c++ pointer 다시 복습!

self reference -> linked list & Node 만들기 부분

원하는 클래스 오브젝트를 스택에 push, pop 할 수 있는지 c++ java 둘다 genetic stack module을 사용할 수 있는지/ void*

Reference / stack

Self reference structure

```
Struct node {

int fata;

Struct node *next;

// pointer로 가르킨다.
}
```

Array 통해서 연결

c에서는 memory allocation을 이용해서 사용한다.

Stack queue module을 만들 수 있음.

Top of stack 은 가자아 마지막에 있는 것을 가르키도록 한다.

Stack module
c++ // java
code 구현해보기!

상세 spec
java method proto type
attibute type 에 해당?
c++ 의 경우는,,,?

Data Structure (linked list & Node)

module modular design
stack 모듈과 queue 모듈을 사용할 때,

c++

```
typedef void* SDT;
class Node {
    public :
     SDT data;
    Node *next;
};
```

interface가 같은 한 변화 없다? implementation
이 source 파일이 되는 것이다.
user defined header file) stack.h
interface 가 변하지 않는 한 module 변화 없다??
관련 있는 것들로, operation들 등, stackd의 경우
array 혹은 linked 이거 각각 operaion를
interface에 정의되어 있으니, 이거 high cohercion

java

```
class Node {
    Object data;
    Node next;
}
```

집합들이다. 결국, 관련된ㄱ ㅓㅅ들만 모아서 interface로
작업을 하면, 다른 모듈에 영향 안미치고, 개개인별 작업 가능?

static file scope ->
다른 곳에서 access 못한댱

abstract object는 하나의 스택?
지금 ppt 는 여러 개의 스택?
pStack stack =. new ~~
이런 식으로 object 만들어서 필요한 스택 사용

high cohersion
law…. 의 원칙에 의해 만든다(못들음 ㅠ—ㅠ)

stack은 interface 있는게 합리적

levelup abstraction 의 경우에는 괄호 안 내용에 의해 CU에 의해 정해짐
context 잘 알아야 한다.(common understanding)에 의해

텍스트

c++

```
int i = 3;
char c = ‘c’;
float f = 3.14;

Stack s;
s.push(&i);
s.push(&c);
s.push(&f);

cout << *(float *) s.pop() << endl;
//              char
//              int

/* 어떤 primitive type의 주소값을 넣든,
어떤 것이 들어가 있는지 알아서 cast 후 꺼내야한다.
구조를 알고, 정보 access 할 수 있어야 한답.
*/
```

Java

```
int i = 3;
char c = ‘c’;
float f = 3.14;

Stack s = new Stack();
s.push(new Integer(i));
s.push(new character(c));


System.out.println(“ “ + s.pop());
// “ “ 를 통해서 string concatenation 통해서 바로 string 값 출력 가능하다.

// String valaueOf(s.pop());
// valueOf method -> integer는 가능
// 다른 type은 확인해볼 것!

/* test program 작성해서 genetic stack module 작성해보기
*/
```

*** 힙 공간에 메모리 할당 후 실행인데,
이때 full의 경우에는 null을 return한다.
-> stack이 꽉찬 경우이다. 이땐 exception handling끝내거나 혹은 공간 space 더 늘린다.

old top - stack object 가르킬 수 있는 것
저장하고, (메모리 해재해주어야 하므로)
i 의 경우는 SDT type 이겠쥬

텍스트

C++

```
void Stack::push(SDT i) {

Node * new node = new Node();
……….
new_node -> data = i;
new_node -> next = top;
top = new_node;
}

SDT Stack :: pop() {
i = top -> data;
top = top -> next;
return i ;

}
```

java

```
public void push(Object i) {
Node new_node = new Node();
new_node.data = i;
new_node.next = top;
top = new_node;
}

public Object pop() {
i = top data;
top = top.next;
return i;
}
```