OOP with Java

Yuanbin Wu
cs@ecnu

OOP with Java

- 通知
  - Project 4: 5月6日晚9点
  - Project 5: 5月17日晚9点

- 复习
  - Protected
    - 可以被子类 / 同一包中的类访问，不能被其他类访问
    - 弱化的 private
    - 同时赋予 package access

```
class MyType {
    public int i;
    public double d;
    public char c;
    protected void set(double x) { d = x;}
    protected void set(int y) {i = y;}
    public double get() { return d; }
}
```

```
public class MySubType extends MyType{
    public void set(double x){ i = (int)x; }
    public void set(char z) {c = z; }
    public static void main(String [ ]args){
        MySubType ms = new MySubType();
        ms.set(1.0);
        System.out.println(ms.get());
        System.out.println(ms.i);
        System.out.println(ms.d);
    }
}
```

- 复习
  - Upcasting
    - 继承
    - 子类具有父类的所有方法和数据
    - Sub-class is a type of base class
  - 类型转换：父类的引用可以指向子类对象

```
class Instrument {
    public void play() {}
    static void tune(Instrument i) {
        // ...
        i.play();
    }
}
public class Wind extends Instrument {
    public static void main(String[] args) {
    Wind flute = new Wind();
    Instrument.tune(flute);
    }
}
```

- 复习
  - final 关键字
  - final 数据
    - static final int j = 1;
    - final int[ ] a = new int [10];
    - Blank final，构造函数中初始化
  - final 参数
  - final 方法：不能重写
  - final 类：不能继承
  - immutable

多态

- Upcasting 与多态
- 动态绑定
- Downcasting

Upcasting

- 类型
  - 基本类型 (byte, short, char, int, long, float, double)
  - 类 (class, array)
- 类型检查
  - 基本类型的转换关系
  - class A 的引用只能指向 class A 的对象 (即，类型需要一致)

```
class A{ … }
class B{ … }
A a = new A();
B b = new B();

// A a = new B(); compile error
```

Upcasting

- Upcasting
  - 同一基类的不同子类可以被视为同一类型 (基类)
  - 放宽类型一致性

```
class A{ … }
class B{ … }
A a = new A();
B b = new B();

// A a = new B(); compile error
```

```
class A{ … }
class B extends A{ … }
A a = new A();
B b = new B();

A a = new B(); // upcasting
```

Upcasting

- Upcasting 的优点
    - 简化接口

```
class Instrument {
    public void play(int note) {
        System.out.println(“Instrument.play()” + n);
    }
}
```

```
public class Wind extends Instrument {
    public void play(int note) {
        System.out.println(“Wind.play()” + n);
    }
}
```

```
public class Stringed extends Instrument {
    public void play(int note) {
        System.out.println(“Stringed.play()” + n);
    }
}
```

```
public class Brass extends Instrument {
    public void play(int note) {
        System.out.println(“Brass.play()” + n);
    }
}
```

```
public class Music {
    public static void tune(Wind i) {
        i.play();
    }
    public static void tune(Stringed i) {
        i.play();
    }
    public static void tune(Brass i) {
        i.play();
    }
    public static void main(String []args){
       Wind flute = new Wind();
             Stringed violin = new Stringed();
       Brass frenchHorn = new Brass();
             tune(flute);
             tune(violin);
             tune(frenchHorn);
    }
}
```

Without upcasting

```
class Instrument {
    public void play(int note) {
        System.out.println(“Instrument.play()” + n);
    }
}
```

```
public class Wind extends Instrument {
    public void play(int note) {
        System.out.println(“Wind.play()” + n);
    }
}
```

```
public class Stringed extends Instrument {
    public void play(int note) {
        System.out.println(“Stringed.play()” + n);
    }
}
```

```
public class Brass extends Instrument {
    public void play(int note) {
        System.out.println(“Brass.play()” + n);
    }
}
```

```
public class Music {
    public static void tune(Instrument i) {
        i.play();
    }
    public static void main(String []args){
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Brass frenchHorn = new Brass();
        tune(flute);
        tune(violin);
        tune(frenchHorn);
    }
}
```

With upcasting

1. 接口变简洁
2. play() 方法能正确的调用对应的重写 (override) 后的子类方法

多态 (Polymorphism)

参数 Instrument i 可以代表不同的子类，并能正确调用它们的方法 (即，有多种表现形态)

多态

by Sinipull for codecall.net

```
class Super {
    public void f() {
        System.out.println(“In Super”);
    }
}
public class Base1 extends Super {
    public void f() {
        System.out.println(“In Base1”);
    }
}

public class Base2 extends Super {
    public void f() {
        System.out.println(“In Base2”);
    }
}
public class Tester {
    public static void main(String []args){
      Super s = new Base1();
      s.f();
      s = new Base2();
      s.f();
    }
}
```

upcasting

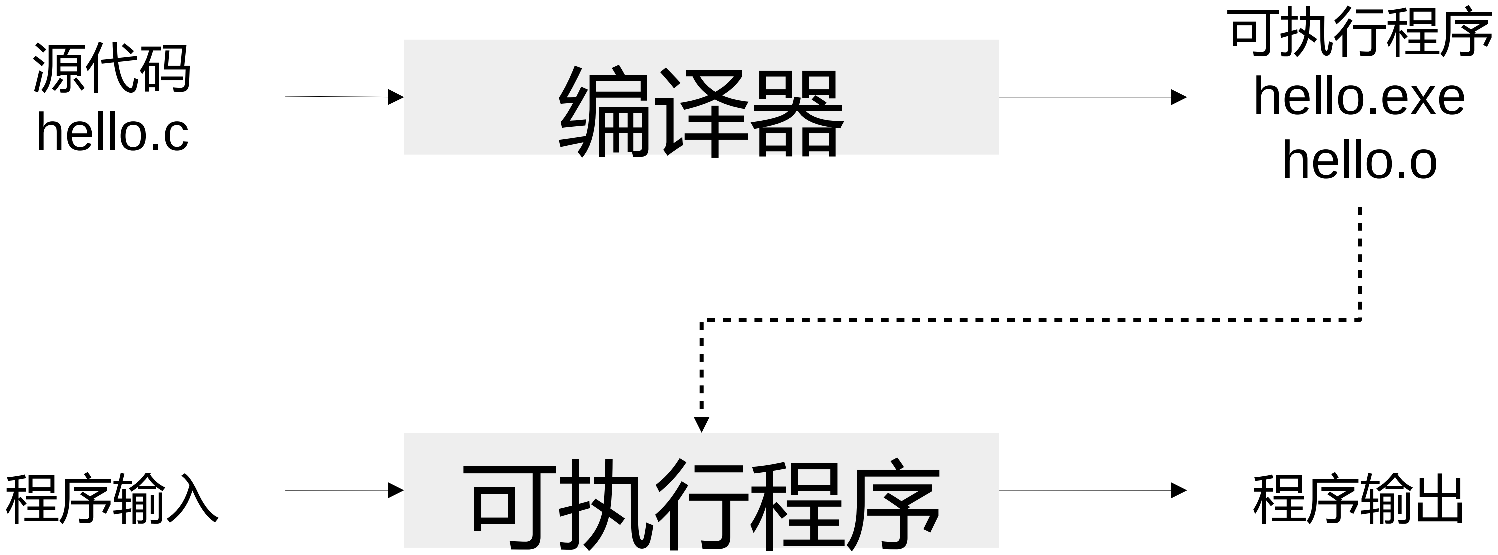
- 问题

```
public class Music {
    public static void tune(Instrument i) {
        i.play();
    }
    public static void main(String []args){
       Wind flute = new Wind();
             Stringed violin = new Stringed();
       Brass frenchHorn = new Brass();
             tune(flute);
             tune(violin);
             tune(frenchHorn);
    }
}
```

tune() 方法是如何知道调用哪一个子类的 play()?
多态是如何实现的？

动态绑定

- C 语言
  - 编译



源代码
hello.c

编译器

可执行程序
hello.exe
hello.o

程序输入

可执行程序

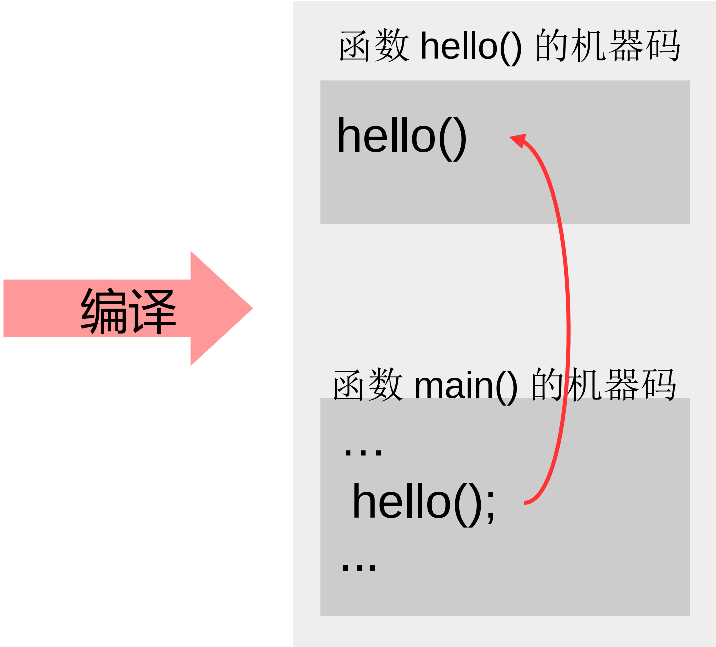
程序输出

- C 语言
  - 可执行文件

静态绑定 (static binding) :
函数的位置在编译时确定

```
#include <stdio.h>

void hello(){
    ...
}

int main(){
    …
    hello();
}
```

源代码
hello.c



编译

函数 hello() 的机器码

```
hello()
```

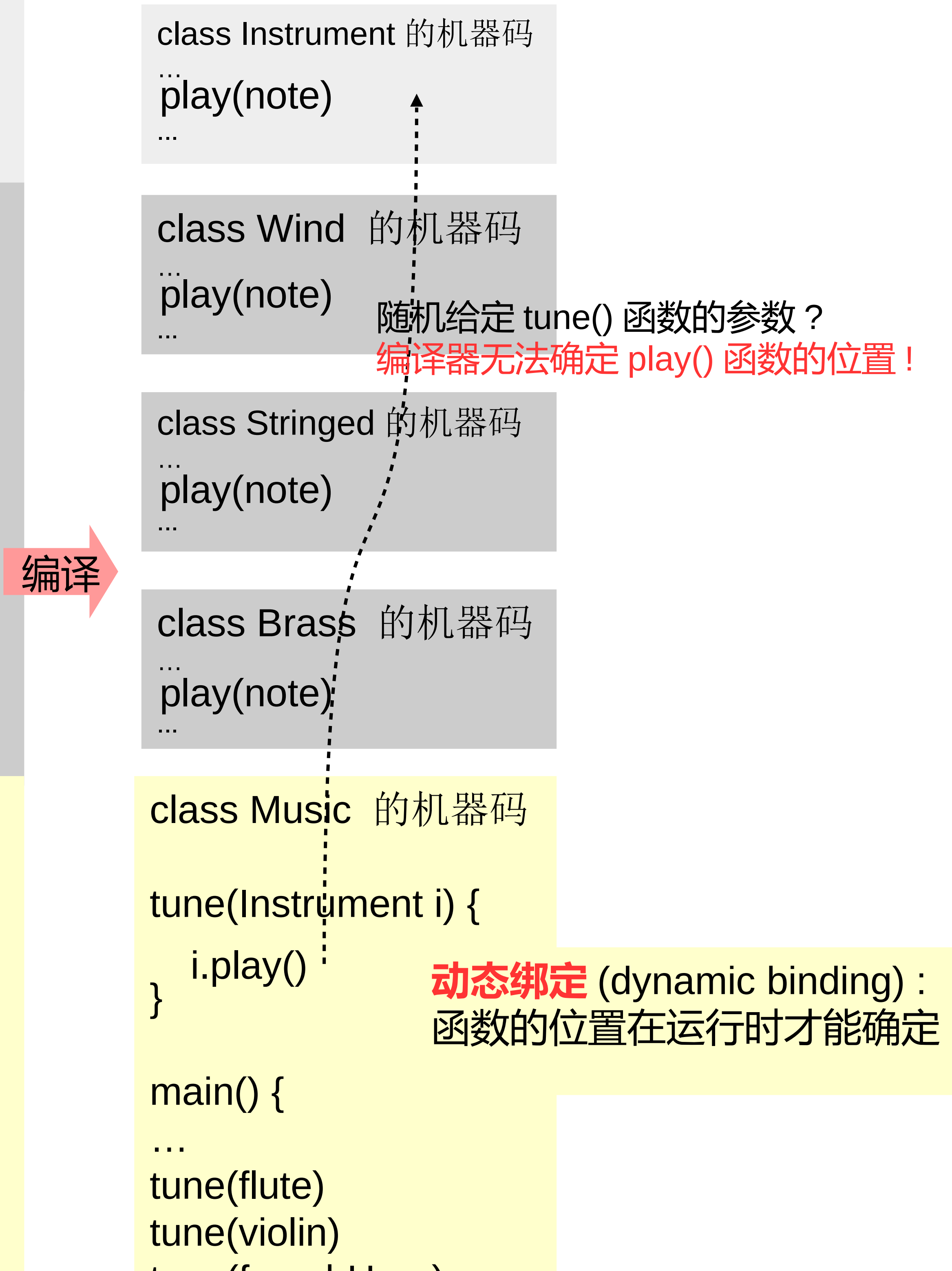
函数 main() 的机器码

```
…
 hello();
...
```

可执行程序
hello.exe
hello.o

编译后 , main() 函数能够确定的知道 hello() 函数的位置

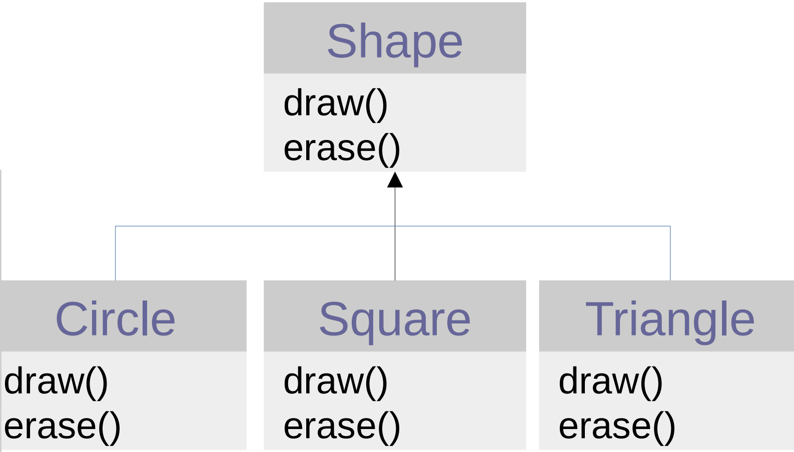
```
class Instrument {
    public void play(int note) {
        System.out.println(“Instrument.play()” + n);
    }
}
public class Wind extends Instrument {
    public void play(int note) {
        System.out.println(“Wind.play()” + n);
    }
}
public class Stringed extends Instrument {
    public void play(int note) {
        System.out.println(“Stringed.play()” + n);
    }
}
public class Brass extends Instrument {
    public void play(int note) {
        System.out.println(“Brass.play()” + n);
    }
}
public class Music {
    public static void tune(Instrument i) {
        i.play();
    }
    public static void main(String []args){
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Brass frenchHorn = new Brass();
        tune(flute);
        tune(violin);
        tune(frenchHorn);
    }
```



随机给定 tune() 函数的参数？
编译器无法确定 play() 函数的位置！

动态绑定 (dynamic binding)：
函数的位置在运行时才能确定

```
public class Shape {
    public void draw() { }
    public void erase() { }
}
public class  Circle extends Shape {
    public void draw() {System.out.println(“circle draw”);}
    public void erase() { System.out.println(“circle erase”);}
}
public class  Square extends Shape {
    public void draw() {System.out.println(“square draw”);}
    public void erase() { System.out.println(“square erase”);}
}
public class Triangle extends Shape {
    public void draw() {System.out.println(“triangle draw”);}
    public void erase() { System.out.println(“triangle erase”);}
}
```



```
public class RandomShapeGenerator {
    public Shape next() {
        double r = Math.random();
        if (r < 0.3)
            return new Circle();
        else if (r >= 0.6)
            return new Tirangle();
        else
            return new Square();
    }
}
```

upcasting

```
public class Shapes {
    private RandomShapeGenerator gen = new
RandomShapeGenerator();
    public static void main(String []args) {
        Shape[]s = new Shape[9];
        for (int i = 0; i < s.length; ++i)
            s[i] = gen.next();
        for (Shape shp:s)
            s.draw();
    }
}
```

Dynamic Binding

动态绑定

- 静态绑定
  - 函数的调用在编译后便确定
  - 也称 early binding
  - 优点：快速，易于 debug
  - 缺点：接口繁琐
- 动态绑定
  - 函数的调用在运行时才能确定
  - 也称 late binding
  - 优点：接口简洁
  - 缺点：函数调用需要额外开销，给 debug 带来困难

动态绑定

- “upcasting+ 多态”带来的扩展性

```
class Instrument {
    public void play(int note) {System.out.println(“Instrument.play()” + n);}
    public void adjust() {System.out.println(“Instrument.adjust”)}
}
```

```
public class Wind extends Instrument {
    public void play(int note) {System.out.println(“Wind.play()” + n);}
    public void adjust() {System.out.println(“Wind.adjust”)}
}
```

```
public class Stringed extends Instrument {
    public void play(int note) {System.out.println(“Stringed.play()” + n);}
    public void adjust() {System.out.println(“Stringed.adjust”)}
}
```

```
public class Brass extends Instrument {
    public void play(int note) {System.out.println(“Brass.play()” + n);}
    public void adjust() {System.out.println(“Brass.adjust”)}
}
```

```
public class Music {
    public static void tune(Instrument i) {
        i.play();
    }
    public static void main(String []args){
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Brass frenchHorn = new Brass();
        tune(flute);
        tune(violin);
        tune(frenchHorn);
    }
}
```

无需改变！

1. 增加新的接口，并不影响原有的只依赖于旧接口的代码
2. 原因：tune 的实现只与父类的相关

动态绑定

- 动态绑定
  - Java 中的所有方法都采用动态绑定，除了
    - final
    - static
  - 原因？

动态绑定

```
public class Super {
    public int field = 0;
    public int getField() {return field;}
}
```

```
public class Sub extends Super {
    public int field = 1;
    public int getField() {return field;}
    public int getSuperField() {return super.field;}
}
```

```
public class FieldAccess {

    public static void main(String []args){
        Super sup = new Sub();
        System.out.println(sup.field);
        System.out.println(sup.getField());

        Sub sub = new Sub();
        System.out.println(sub.field);
        System.out.println(sub.getField());

System.out.println(sub.getSuperField());
    }
}
```

数据成员不使用动态绑定

动态绑定

- 构造函数
  - 初始化顺序
    - 分配内存空间，默认初始化 (设置为 0)
    - 初始化父类 (递归 !)
    - 静态成员初始化 (首次创建该类对象)
    - 数据成员初始化 (按照定义顺序)
    - 调用构造函数

动态绑定

- 构造函数初始化顺序

```
public class Super {
    int  sup_field = 1;
    public Super(){
        ...
    }
}
```

```
public class Sub extends Super {
    public int sub_field = 1;
    public Sub(int f) {

        sub_field = f;
    }
}
```

1. 初始化父类
2. 初始化子类的数据

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

```
public class Sandwich extends PortableLunch {
    private Bread b = new Bread();
    private Cheese c = new Cheese();
    private Lettuce l = new Lettuce();
    public Sandwich() {
         System.out.println("Sandwich()");
    }

    public static void main(String[] args) {
      new Sandwich();
    }
}
```

Output:

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

动态绑定

- 构造函数中使用重写函数 → BUG!

```
public class Super {
    public Super() {
        System.out.println(“Before Super draw”);
        draw();
        System.out.println(“After Super draw”);
    }
    public void draw() {
       System.out.println(“draw”);
    }
```

```
public class Sub extends Super {
    public int field = 1;
    public Sub(int f) {
        field = f;
        System.out.println(“Sub” + field);
    }
    public void draw() {
        System.out.println(“draw” + field);
    }
}
```

```
public class Test {

    public static void main(String []args){
        Sub sub = new Sub(5);
    }
}
```

输出?

1. 子类的方法：在子类对象创建之后才有意义
2. 构造函数中，避免使用将被重写的函数

动态绑定

- 多态适用于协变返回值
  - 协变的返回值
    - 被重写的函数返回值可以是原函数的子类

```
class Grain {
    public String toString() { return “Grain”; }
}
class Wheat extends Grain{
    public String toString() { return “Wheat”; }
}
class Mill {
    Grain process() { return new Grain(); }
}
class WheatMill extend Mill{
    Wheat process() { return new Wheat(); }
}
```

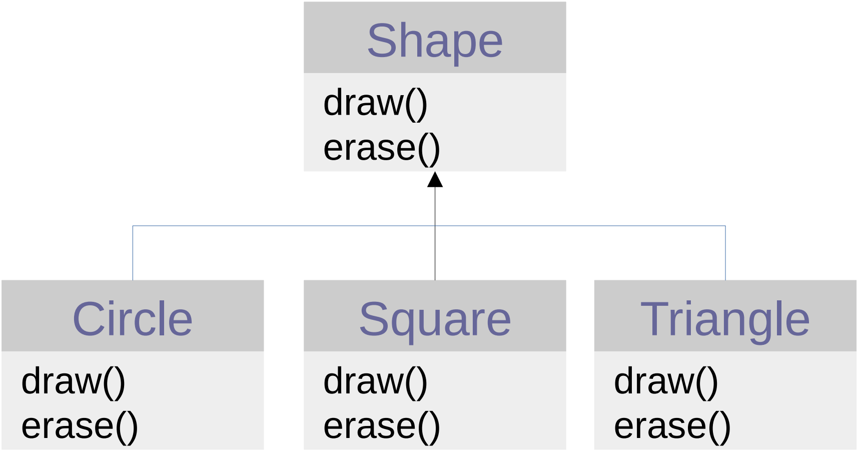
```
public class CovariantReturn {
    public static void main(String []args){
        Mill m = new Mill();
        Grain g = m.process();
        System.out.println(g);

        m = new WheatMill();
        g = m.process();
        System.out.println(g);
    }
}
```

动态绑定

- 总结
  - 静态绑定：函数在编译时确定
  - 动态绑定：函数在运行时才能确定
  - 除了 final, static 外所有函数都为动态绑定
  - 在构造函数中减少使用可能会被重写的函数

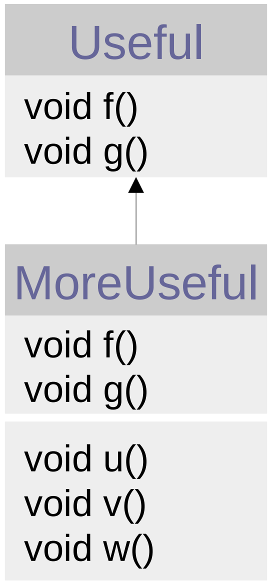
Downcasting

- Is-a 关系
  - 父类与子类的接口完全相同



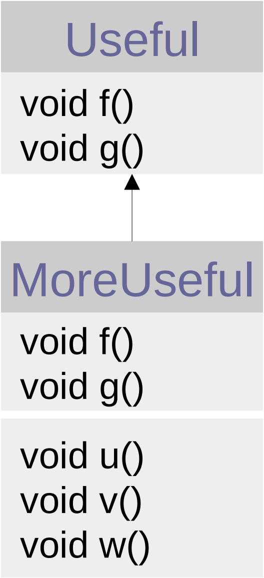
Downcasting

- Is-like-a 关系
  - 子类添加了新的方法
  - Upcasting:
    - 父类引用指向子类的对象
    - 安全的
  - Downcasting
    - 子类引用指向父类的对象
    - 不安全
    - 但当一个父类引用指向子类时，可以将该引用强制转换为子类引用



Downcasting

```
public class Downcasting {
    public static void main(String []args){
        Userful x = new Userful();
        Userful y = new MoreUseful();
        x.f();
        y.f()
        // y.u(); compile error, u() not in Useful
        ((MoreUseful)x).u(); // run time error
        ((MoreUseful)y).u(); // downcasting
    }
}
```


Useful
void f()
void g()
MoreUseful
void f()
void g()
void u()
void v()
void w()