
MPC
Club
rogramming
uroran

# 今日やること

1.抽象クラス
2.インターフェース
3.ポリモーフィズム

# 今日やること

1.抽象クラス
2.インターフェース
3.ポリモーフィズム

# 抽象メソッド

定義がないメソッド

```
abstract void method(); //宣言だけ
```

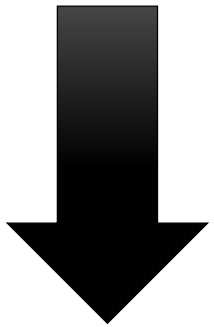
# 抽象クラス

抽象メソッドを1つ以上持つクラス

```
public abstract class SuperClass{
        abstract void method(); //抽象メソッド
}
```

# 抽象メソッドの定義はどこでする？？

抽象メソッドの定義はどこでする？？



抽象クラスを継承したサブクラスで
**オーバーライド**する

# 抽象クラス

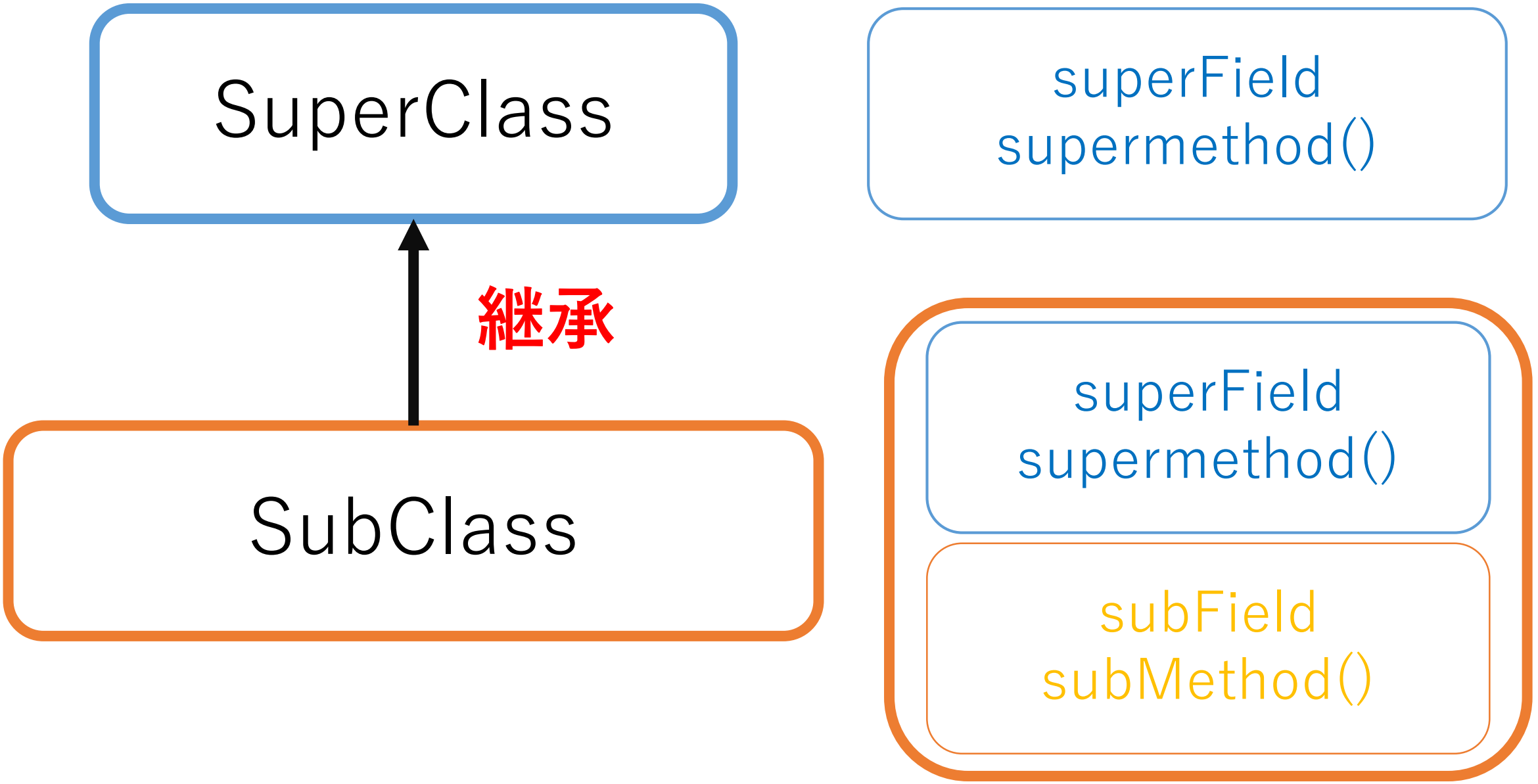
```
//抽象クラスを継承
class SubClass extends SuperClass{
        //オーバーライド
        public void method(){
                //処理を定義
        }
}
```

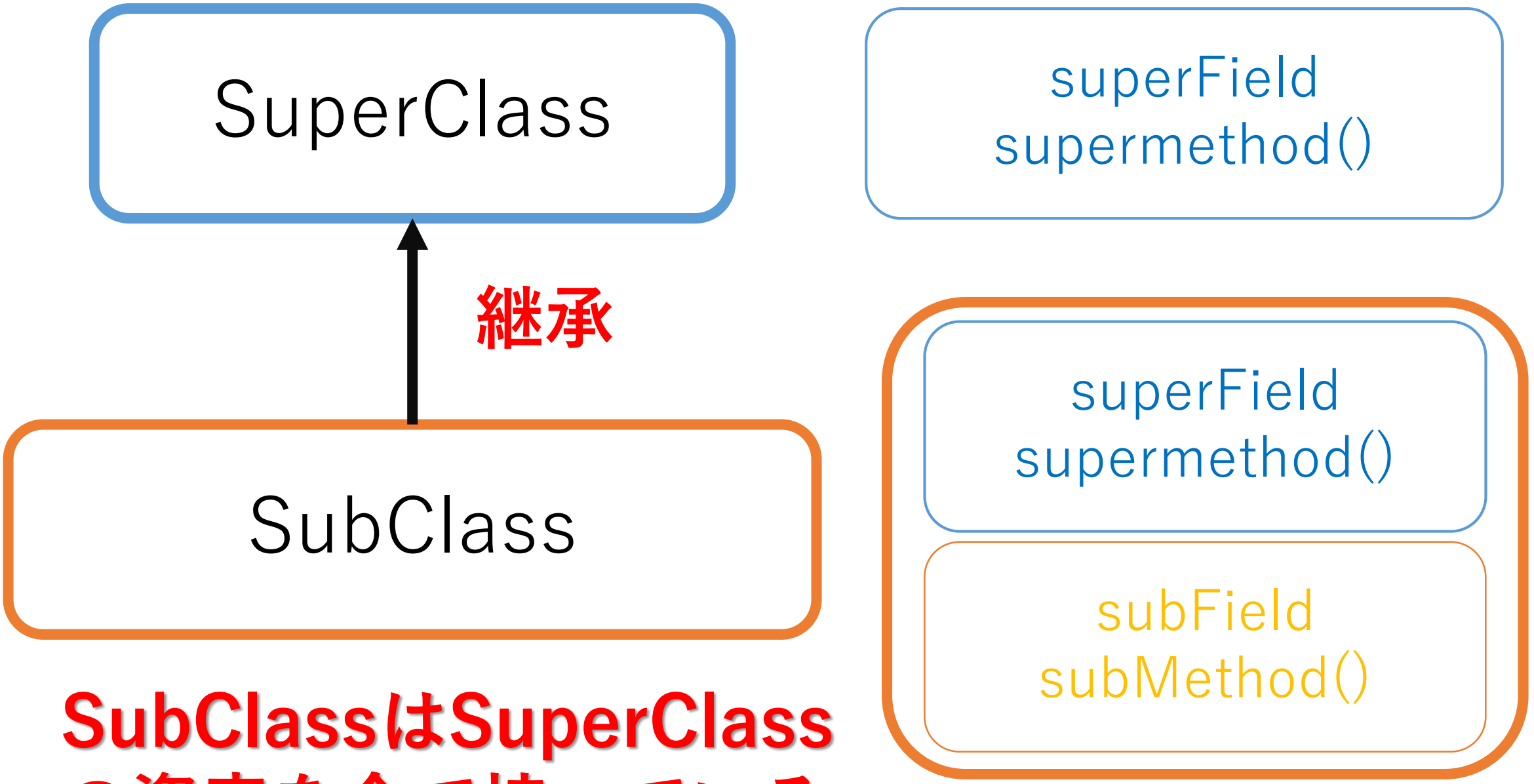
# 抽象クラス

抽象メソッドを定義していないクラスは

```
SuperClass s;    //変数の宣言OK

//s = new SuperClass();    インスタンスの生成NG

new SubClass();//抽象メソッドをオーバーライドすれば生成OK
```

# アップキャスト


SuperClass
継承
SubClass
superField
supermethod()
superField
supermethod()
subField
subMethod()




**SubClassはSuperClassの資産を全て持っている**

# アップキャスト

```
//親クラス型の変数に子クラスのインスタンスを代入
Super s = new Sub();          //←アップキャスト

s.superMehod();          //インスタンスのメソッドが呼ばれる

//親クラスの持っていないメソッドは呼び出せない
//s.subMethod();
```

# 今日やること

1.抽象クラス
2.インターフェース
3.ポリモーフィズム

# インターフェース

抽象メソッドのみを持つクラス

```
public interface Interface{
        int field = 100;//クラス定数(public static final)

        void method(); //public abstractになる
}
```

# インターフェース

インターフェースを継承　→　**実現**

```
//インターフェースを実現
public class Implementer implements Interface{
        //抽象メソッドをオーバーライド
        public void method();{
                処理を定義する
        }
}
```

# インターフェース

標準で用意されている様々なインターフェースがある

Clonable　　… ディープコピー
Throwable　　… 例外
Runnable　　… スレッド
List<>　　… リストや配列

# 今日やること

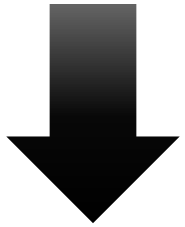
1.抽象クラス
2.インターフェース
3.ポリモーフィズム

# ポリモーフィズム

```
SuperClass s;           //スーパークラス型の変数 s

s = new SubA();         //サブクラスのインスタンス代入
s.method();             //SubAでオーバーライドしたmethod();

s = new SubB();         //サブクラスのインスタンス代入
s.method();             //SubBでオーバーライドしたmethod();

s = new SubC();         //サブクラスのインスタンス代入
s.method();             //SubCでオーバーライドしたmethod();
```

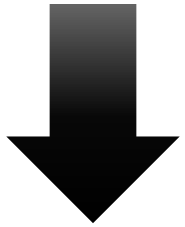
# メソッドの呼び出し方は同じ

メソッドの呼び出し方は同じ



**代入するインスタンスによって処理が異なる**

メソッドの呼び出し方は同じ



**代入するインスタンスによって処理が異なる**

ポリモーフィズム(多態性)

# ポリモーフィズム

new SubA();　　new SubB();　　new SubC();

```
void function(SuperClass s){
        s.method();
}
```

# ポリモーフィズム

```
new SubA();          new SubB();          new SubC();
```

```
void function(SuperClass s){
        s.method();
}
```

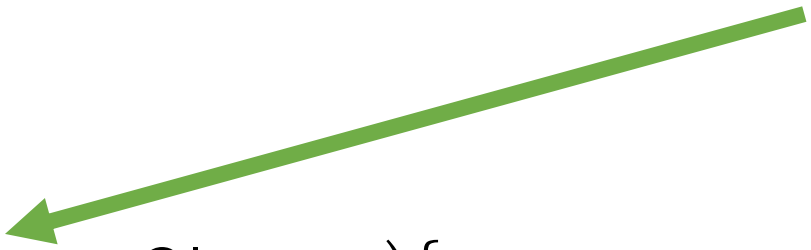
# ポリモーフィズム

new SubA();　　new SubB();　　new SubC();

```
void function(SuperClass s){
        s.method();
}
```

# ポリモーフィズム

new SubA();　　new SubB();　　new SubC();



```
void function(SuperClass s){
        s.method();
}
```

# 設計と実装の分離

function をつくるひと（設計者）は

SuperClassがmethod()を

持っていることを知っていれば良い

# 設計と実装の分離

function をつくるひと（設計者）は

SuperClassがmethod()を

持っていることを知っていれば良い

**サブクラスがどのような実装をしているかは**

**知らなくて良い**

# 設計と実装の分離

サブクラス SubA, SubB, SubCをつくるひと(実装者)は
SuperClassのmethod()をオーバーライドすれば良い

# 設計と実装の分離

サブクラス SubA, SubB, SubCをつくるひと(実装者)は
SuperClassのmethod()をオーバーライドすれば良い

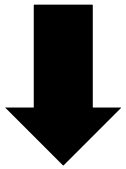
**function(SuperClass s)の設計を知らなくても良い**

# 設計と実装の分離

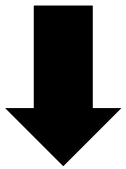
お互いに詳しいことは知らなくて良い

# 設計と実装の分離

お互いに詳しいことは知らなくて良い



**隠蔽**する



**カプセル化**

# カプセル化

カプセル化

知っておくべき事が少ない

カプセル化

知っておくべき事が少ない

**管理が楽になる**

# 演習問題