第５週　抽象クラス・インターフェース

抽象クラス

Java ではメソッドの処理を定義せずに宣言だけすることができます.これを抽象メソッドと言います.そして,抽象メソッドを１つ以上持つクラスを抽象メソッドと言います．抽象クラスはインスタンスを生成する事ができません．抽象クラスを使用するためには抽象クラスを継承したサブクラス(子クラス)を作成して抽象メソッドをオーバーライドする必要があります．抽象クラスと抽象メソッドにはキーワード「abstract」を付けます．

```
//抽象クラス
public class abstract SuperClass{
                    ⋮
        public abstract void superMethod(); //抽象メソッド
                    ⋮
}
```

それでは,抽象クラスを実際に使ってみましょう．以下のサンプルソースを実行してみます．

```
//抽象クラス
public abstract class SuperClass {


        //抽象メソッド
        public abstract void superMethod();   //抽象クラスに定義はない
```

```
public class SubClass extends SuperClass{


        public void superMethod(){
                System.out.println("superMethod in SubClass");
        }
        public void subMethod(){
                System.out.println("subMethod in SubClass");
        }
}
```

```
public class Sample1 {

	public static void main(String[] args) {
		// TODO 自動生成されたメソッド・スタブ
		//SuperClass s = new SuperClasss(); //抽象クラスはインスタンスを生成できない

		SuperClass s = new SubClass(); //アップキャスト

		s.superMethod();   //サブクラスのメソッドが呼び出される

		//s.subMethod();   //サブクラスで定義したメソッドは呼び出せない
	}
}
```

サブクラスはスーパークラスを拡張したものなのでスーパークラス型の変数に代入する事ができます．このとき,スーパークラス型への型変換が行われるのでキャスト演算子を使ってキャストする必要はありません．これを**アップキャスト**と言います.サンプルソースのようにサブクラスのインスタンスをスーパークラス型の変数に代入した場合はスーパークラスで宣言したメソッドのみを使用する事ができ,サブクラスしか持っていないメソッドを使用することはできません.

インターフェース

メソッドがすべて抽象メソッドであるクラスをインターフェースと呼びます．インターフェースを作成するためにはクラス名の前にキーワード「**interface**」を付けます.

```
//インターフェース
class interface Interface{
		⋮
}
```

インターフェースのフィールドは暗黙的にクラス定数(public static final), メソッドは暗黙的に抽象メソッド(public abstract)になります.

インターフェースを継承してメソッドをオーバーライドすることを実現といいます．インターフェースを継承するためには名前のあとにキーワード「**implements**」を付けます.

Java ではクラスの多重継承が禁止されていますがインターフェースはいくつでも実現することができます.

実際にインターフェースを作成してみましょう.以下のサンプルソースを入力し, 実行してください.

```java
public interface Interface {
	int field =100;			//public static final int field = 100; となる

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

```java
public class Implementer implements Interface{
	//オーバーライド
	public void method(){
		System.out.println("method in Implementer");
	}
}
```

```java
public class Sample2 {

	public static void main(String[] args) {
		// TODO 自動生成されたメソッド・スタブ
		Interface inter = new Implementer(); //アップキャスト

		inter.method(); //Inplementerのメソッドが呼ばれる
	}
}
```

ポリモーフィズム(多態性)

スーパークラス型の変数にサブクラスのインスタンスを代入するとオーバーライドされているメソッドはサブクラスによって振る舞いが異なります．このように同じメソッドの呼び出しで振る舞いを変えることを**ポリモーフィズム**(多態性)といいます．実際にサンプルソースで見てみましょう．

```
//図形クラス(抽象クラス)
public abstract class Shape {

	//描画
	public abstract void draw();		//抽象メソッド

}
```

```
//直線クラス
public class Line extends Shape {

	private int length;			//長さ

	public Line(int length){
		this.length = length;
	}

	@Override		//←　書かなくても良いがあったほうがいい
	public void draw() {
		System.out.println("直線------------------------------");
		for(int i = 0; i < length; i++){
			System.out.print("■");
		}
		System.out.println();
	}

}
```

```java
//四角形クラス
public class Rectangle extends Shape {
	private int width; //幅
	private int height;		//高さ

	public Rectangle(int width, int height){
		this.width = width;
		this.height = height;
	}

	@Override
	public void draw() {
		System.out.println("四角形------------------------------");
		for(int i = 0; i < width; i++){
			for(int j = 0; j < height; j++){
				System.out.print("■");
			}
			System.out.println();
		}
	}
}
```

```java
//三角形クラス
public class Triangle extends Shape {
	private int height;

	public Triangle(int height){
		this.height = height;
	}
	@Override
	public void draw() {
		System.out.println("三角形----------------------------");
		for(int i = 0; i < height; i++){
			for(int j = 0; j < height - i -1 ; j++){
				System.out.print("  ");
			}
			for(int j = 0; j < 2*i + 1; j++){
				System.out.print("■");
			}
			System.out.println();
		}
	}
}
```

```java
public class Sample3 {
	public static void main(String[] args) {
		// TODO 自動生成されたメソッド・スタブ
		Shape s; //図形
		s = new Line(5);
		s.draw();

		s = new Rectangle(5,9);
		s.draw();

		s = new Triangle(8);
		s.draw();
	}
}
```

オブジェクト指向

これまでオブジェクト指向について学んできました．継承はスーパークラスのメソッドやフィールドを引き継いで機能を拡張するのでクラスを再利用する事ができます.

抽象メソッドをオーバーライドしていないサブクラスはインスタンスを生成することができません．つまり,抽象クラスはサブクラスに抽象メソッドのオーバーライドを強制します．これによってコンパイルをした際にオーバーライドをしてないサブクラスがあった場合にミスを検出する事ができます.

ポリモーフィズムを利用することで,設計と実装を分離する事ができます．つまり,お互いが詳細を知らなくても開発する事ができます．例えば,ソフトウェアの設計をする人はサブクラスの詳細を知らなくてもスーパークラスを使って設計する事ができます．また、サブクラスの機能を実装する人はソフトウェア全体の設計を知らなくてもサブクラスで要求された使用を満たすようにスーパークラスの抽象メソッドをオーバーライドして処理を実装する事ができます.

演習問題

1　Java には標準で使える Runnable インターフェースが用意されています.

```
public interface Runnable{
        void run();
}
```

1.1　つぎの ThreadMain.java, Lecture.java, CLecture.java を写してください

```
public class ThreadMain {

        public static void main(String[] args) {
                System.out.println("start Main");

                new Thread(new CLecture()).start();

                //new Thread(new JavaLecture()).start();

                System.out.println("finish Main");
        }
}
```

```
public class Lecture {
        String name;
        Lecture(String name){
                this.name = name;
        }
        public void start(){
                System.out.println( name + " start");
        }
        public void finish(){
                System.out.println(name + " finish");
        }
}
```

```
public class CLecture extends Lecture implements Runnable {
	CLecture(){
		super("C講座");
	}
	@Override
	public void run() {
		// TODO 自動生成されたメソッド・スタブ
		start();
		for(int i = 0; i < 50; i++){
			System.out.println("C講座");
		}
		finish();
	}
}
```

1.2　次の要件を満たす JavaLecture.java を実装してください

- JavaLecture クラスは Lecture クラスを継承する
- JavaLecture クラスは Runnable インターフェースを実現する
- JavaLecture クラスは run()をオーバーライドする
- JavaLecture クラスは run()の最初に start()を呼び出す
- JavaLecture クラスは次に以下の処理を行う

```
for(int i = 0; i < 50; i++){
	System.out.println("Java講座");
}
```

- JavaLecture クラスは run()の最後に finish()を呼び出す

1.3　ThreadMain.java のコメントをはずして実行してください

2　以下の順番でクラスを作成し，動作を確認してください

2.1　次の Character.java, Hero.java, GameMain.java を写してください

```
public abstract class Character {
	protected String name;	//名前
	protected int hp; //体力
	protected int ap; //攻撃力

	Character(String name, int hp, int ap){
		this.name = name;
		this.hp = hp;
		this.ap = ap;
	}

	//targetに攻撃する
	public void attack(Character target){
		System.out.println(name + "が" + target.getName() + "に攻撃!!");
		target.damage(this);	//引数として自分自身を渡す
	}

	//ステータスを表示
	public void putStatus(){
		System.out.println(name + ":(" + hp + "," + ap + ")");
	}

	//抽象メソッド
	abstract void damage(Character attacker);

	// getter
	String getName(){return name;}
	int getHp(){return hp;}
	int getAp(){return ap;}
}
```

```java
public class Hero extends Character{

	Hero(String name, int hp, int ap){
		super(name, hp, ap);
	}

	@Override
	public void damage(Character attacker){
		System.out.println(name + "は" + attacker.getAp() + "のダメージを受けた!!");
		hp -= attacker.getAp();	//attackerの攻撃力だけhpから引く
	}
}
```

```java
public class GameMain {
	public static void main(String[] args) {

		Hero hero = new Hero("勇者", 500, 50);
		Enemy maou = new Enemy("魔王", 1000, 80, 20);

		//ステータスを表示
		hero.putStatus();
		maou.putStatus();

		//攻撃
		hero.attack(maou);
		maou.attack(hero);

		//ステータスを表示
		hero.putStatus();
		maou.putStatus();
	}
}
```

2.2　次の要件を満たす Enemy クラスを実装してください

- Enemy クラスはフィールドとして int dp （防御力）をもつ
- Enemy クラスは damage メソッドをオーバーライドする

> damage メソッドの処理：
> attacker の攻撃力から防御力を引いた分のダメージを受ける
> 計算した結果が 0 以下ならばダメージを受けない

- Enemy クラスは putStatus メソッドで dp の値も表示する

2.3　最後に GameMain.java を実行してください