Java講座

第3回－前半

情報科学部コンピュータ科学科
2年　根岸　拓郎

今回の内容

- クラスの拡張
  - スーパークラス(親クラス)
  - サブクラス(子クラス)
  - オーバーライド
- final修飾子
- 抽象クラス
- インターフェイス
- おまけ

まず始める前に

- 前回作成した「java_lec03.samples」パッケージの中に、先ほどダウンロードした「Sample_Dragon.java」と「Sample_overriding.java」を移しておいてください。

クラスの拡張

前回作成したプログラムから

- 前回、MovingBallクラスを作成しました。

```
public class MovingBall extends Applet implements Runnable{
```

- ここの赤枠で囲った部分について解説したいと思います。

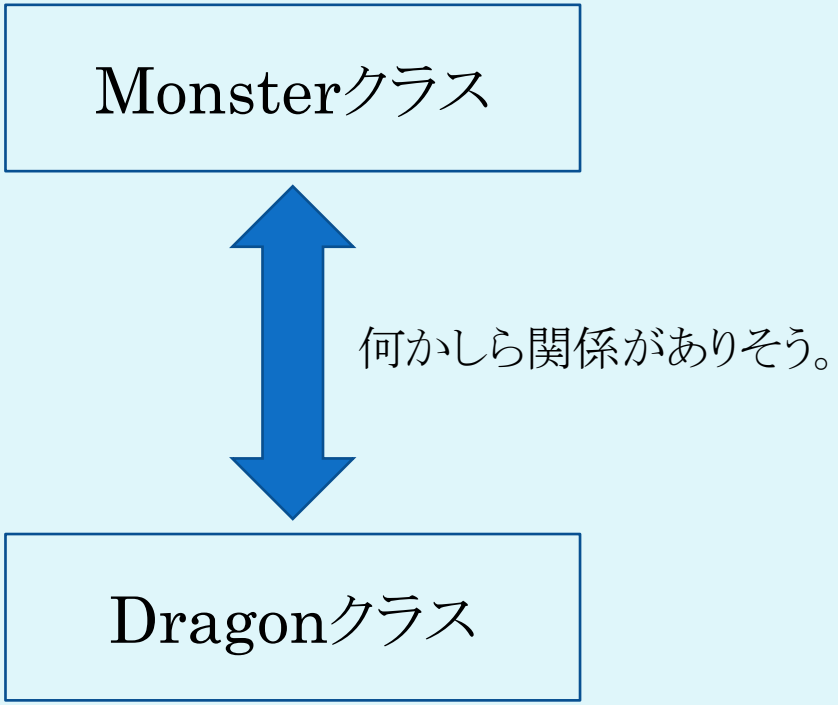
- まずは「extends⇒クラスの拡張」から

クラスの拡張って？

- 前回の例で、Monsterクラス(モンスターを表すクラス)を定義した。
  ここで、「ドラゴン」というモンスターを表すクラスを作成したいとする。
  ドラゴンもモンスターの一種なので、Monsterクラスが利用したいところである。

- Javaでは、既に作成したクラス(Monsterクラス)を基にして、新しいクラス(Dragonクラス)を作成できるようになっている。
- これをクラスの継承、またはクラスを拡張すると言う。

クラス同士の関係



クラスの拡張

- 新しいクラスは、継承したクラスのアクセス可能なメンバー(変数、メソッド)を「受け継ぐ」仕組みになっている。
- このとき、基になるクラス(この場合はMonsterクラス)をスーパークラス、または親クラスという。
- スーパークラスの性質や機能(メンバー)を継承するクラス(この場合はDragonクラス)をサブクラス、または子クラスという。

クラスの拡張(イメージ図)

スーパークラス(親クラス)

- Dragonクラスのスーパークラスである Monsterクラスは変更なし。
- スーパークラスにするための記述などはない。 サブクラス側が指定するだけである。

サブクラス(子クラス)

サブクラスの宣言

```
public class サブクラス名 extends スーパークラス名{
    サブクラスに追加するメンバー

    サブクラスのコンストラクター(引数リスト){

    }
}
```

Dragonクラス(定義)

```
public class Dragon extends Monster{

    public Dragon(int hp, int atk){
        super(hp, atk);
    }
    public Dragon(String name, int hp, int atk){
        super(name, hp, atk);
    }
}
```

```
Monster(int hp, int atk){
  this.hp = hp;
  this.atk = atk;
}
```
を、起動しているのと同じ

super()は次項で説明する。
サンプルコード:
java_lec03.samples.Sample_Dragon.java

サブクラスについて

- 前頁のsuperとは「そのオブジェクト自身(this)」のスーパークラスを表す語で、super(引数リスト)でコンストラクターを呼び出したり、「super.メンバー」でメンバーにアクセスすることが出来る。

- ただし、super()のコンストラクターの呼び出しは、サブクラスのコンストラクター内の最初の処理である必要がある。

this()とsuper()

- コンストラクター内では、this()、super()でそれぞれオブジェクト自身のコンストラクター、スーパークラスのコンストラクターを呼び出すことが出来る。

- this()
  そのクラスの別のコンストラクターを呼び出す
- super()
  そのクラスのスーパークラスのコンストラクターを呼び出す

- どちらもコンストラクター内の最初の処理でなければならないので注意。

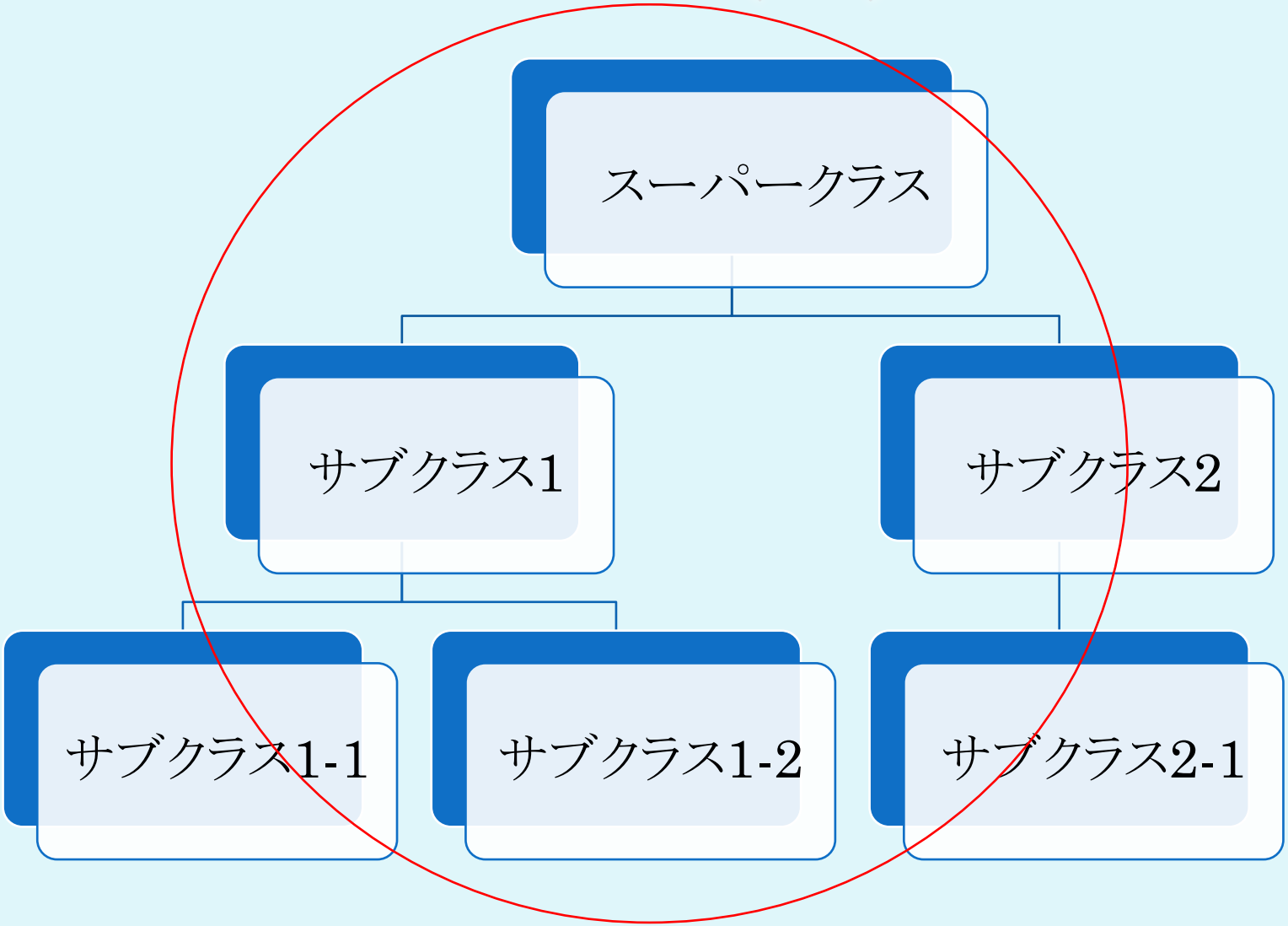
クラスの拡張

- Javaでは、1つのスーパークラスを拡張して複数のサブクラスを宣言することも出来る。

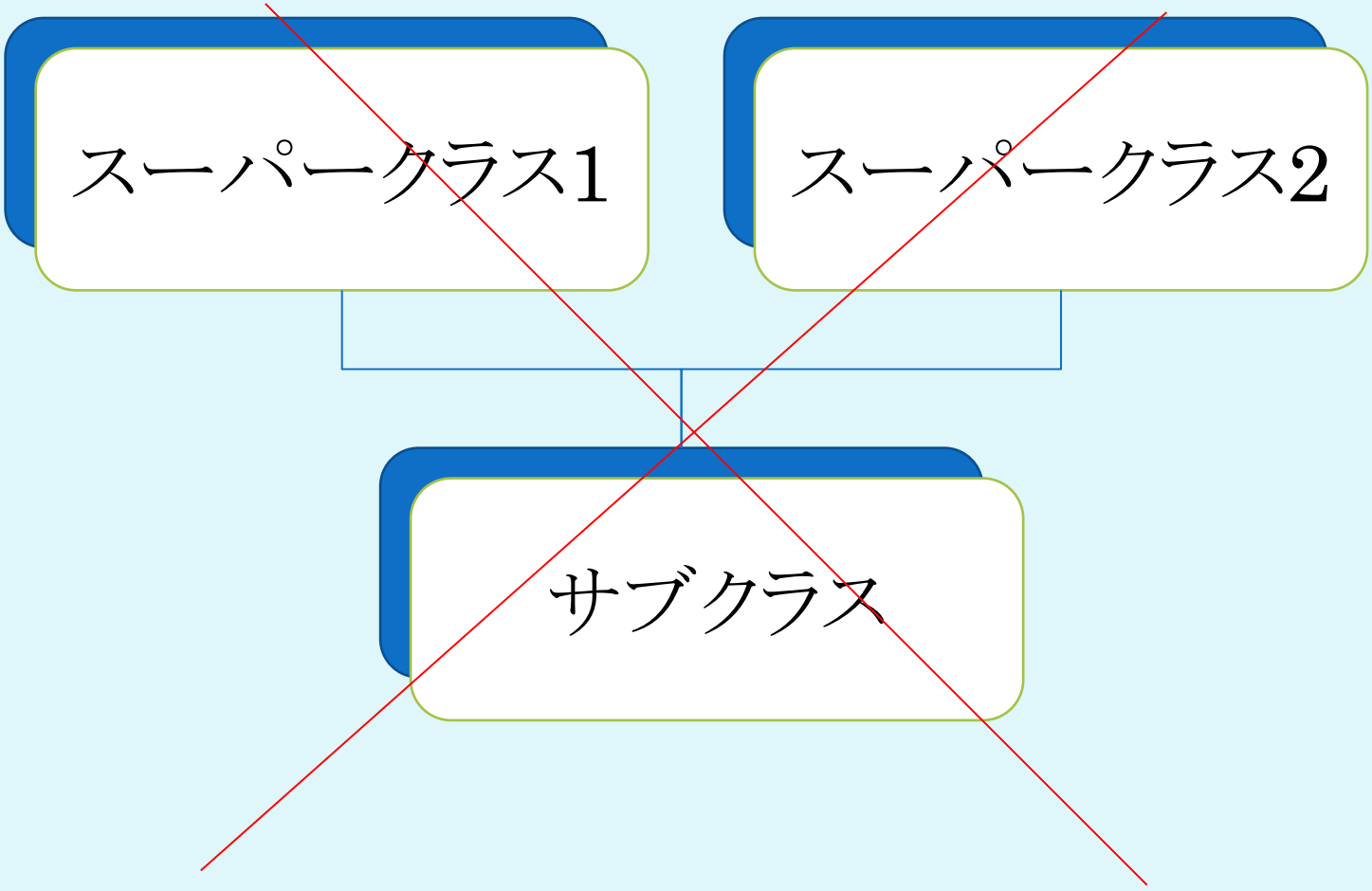
- また、そのサブクラスをさらに拡張して、さらに新しいサブクラスを作成することも出来る。
  最初のサブクラスは次に拡張したサブクラスから見れば、スーパークラスとなる。

- ただし、Javaでは1つのサブクラスで複数のスーパークラスを継承すること(多重継承)は出来ない。

クラス図(正)



クラス図(誤)

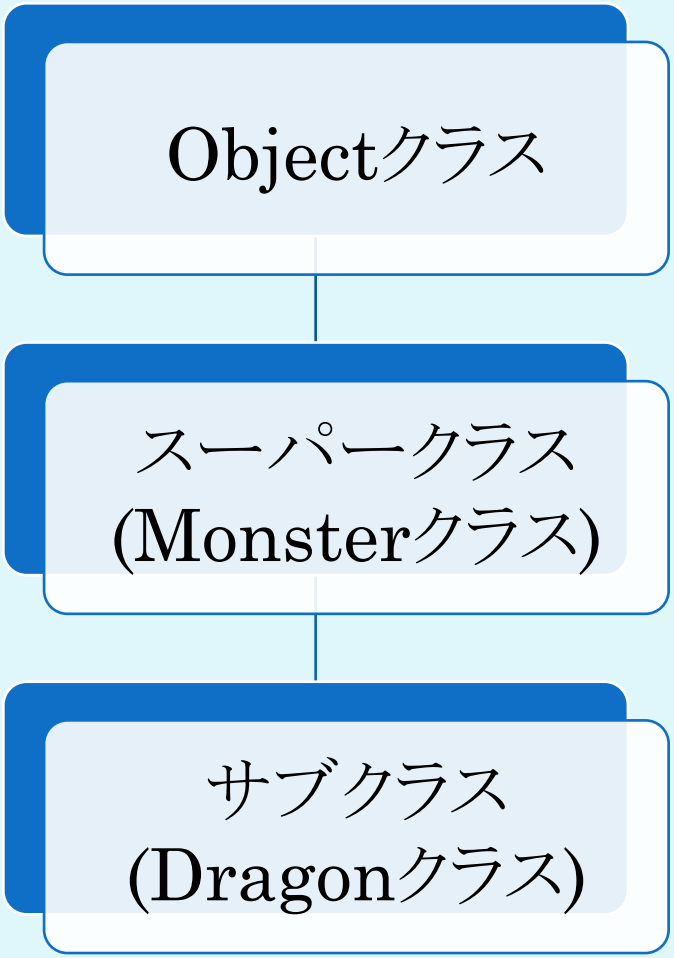


スーパークラスを指定しないと？

- 今まで、スーパークラスを指定しないクラスを宣言することがあった。

  Javaでは、クラスがスーパークラスを指定しない場合、
  そのクラスはObjectクラスというクラスをスーパークラスに持つ(継承する)
  という決まりになっている。

Objectクラスとのクラス図



オーバーライド

- サブクラスで新しくメソッドを記述する時に、スーパークラスとまったく同じメソッド(名前、引数の数・型、戻り値の型)を定義し、「上書き」することができる。
- 試しにDragonクラスに新しくshowStatus()メソッドを定義してみよう。

オーバーライド

DragonクラスのshowStatus()メソッド

```
public void showStatus(){
        System.out.println(“モンスターの種類:ドラゴン”);
        System.out.println(“名前:” + name);
        System.out.println(“HP:” + hp);
        System.out.println(“ATK:” + atk);
}
```

- MonsterクラスのshowStatus()メソッドに一行追加した。
- それ以外の処理は変わらない。

オーバーライド

```
Dragon d = new Dragon(“ドラゴン”, 1000, 100);
d.showStatus();
```

または、

```
Monster m = new Dragon(“ドラゴン”, 1000, 100);
m.showStatus();
```

- 上に例を二つ示したが、どちらでも記述できる。
  二つ目の記述例は継承関係にあるクラス同士でのみ可能である。
- どちらのオブジェクトもDragon型で生成しているので、自動的にDragonクラス(サブクラス)のshowStatus()メソッドが呼ばれる。

オーバーライド

- このようにサブクラス側でスーパークラスのメソッドを定義すること、すなわちサブクラス側でスーパークラスのメソッドを「上書き」することをオーバーライドといい、クラスの拡張には必須の機能である。

- overriding　……　上書き

オーバーライド

```
public void showStatus(){
        System.out.println("モンスターの種類:ドラゴン");
        //スーパークラスのshowStatus()を呼び出している
        super.showStatus();
}
```

- 上書きといっても、スーパークラスのshowStatus()メソッドが消えてしまったわけではないので、上の例のように「super.メソッド名」を使って呼び出すことができる。
- サンプルコード:
  java_lec03.samples.Sample_overriding.java

final修飾子

final修飾子

public final 戻り値の型 メソッド名(引数リスト){　}
private final 戻り値の型 メソッド名(引数リスト){　}

- メソッドの中には決してサブクラスによってオーバーライドされたくないメソッドがあるかもしれない。

  そのような場合には、メソッドの先頭にfinalをつけると、オーバーライドされないようにすることが出来る。

final修飾子

```
public final class クラス名{   }
```

- オーバーライドだけでなくサブクラス自体を拡張してほしくないクラスを設計する場合、クラスの先頭にfinalをつけておくことでサブクラスを拡張できなくなる。

final修飾子

public final フィールドの型 フィールド名 = 初期化値;
public static final フィールドの型 フィールド名 = 初期化値;

- 次にフィールド(変数)にfinalを付けた場合を考える。
- finalをつけたフィールド(変数)は値を変更することが出来なくなり、この決まった値を表すフィールドを定数という。

final修飾子

- 前頁の「public static final」が付いたフィールド(変数)について、クラス変数であるので、
  クラス名.フィールド名
  という記述によって決まった値を表すことができ、
  Javaではこのような「クラスの定数」の名前を全て大文字で書き、単語の区切りは「_」で記述する。

finalまとめ

- フィールドにfinalを付けると、初期値から値を変更できなくなる。
- メソッドにfinalを付けると、サブクラスでオーバーライドできなくなる。
- クラスにfinalを付けると、クラスを拡張できなくなる。

これ以降は難しいので、理解しにくい人はこんなものもあるのか、程度に考えておくと良いでしょう。

しかし、Javaのとても重要な部分でもあるので、いずれ自分で参考書などでチャレンジしてみてください。

抽象クラス

抽象クラスの記述

```
public abstract class AbstractMonster{
        private String name;
        他のフィールドは略;

        public showStatus(){
                これまでと同じ処理;
        }
        //抽象メソッド
        public abstract void move();
}
```

- 上の例は、これまで作成してきたMonsterクラスと同じメンバーを持つ抽象クラスAbstractMonsterクラスである。
- クラスの先頭部分にabstractという修飾子が付いていると、そのクラスは抽象クラスとなる。
- 抽象クラスは、「オブジェクトが生成できない」という特徴がある。
- 抽象クラスは、abstractの付いた
  処理内容が定義されていないメソッド(抽象メソッド)を定義できる。

抽象クラスのフォーマット

抽象クラスの宣言

```
public abstract class クラス名{
    フィールドの宣言;
    abstract 戻り値の型 メソッド名(引数リスト);
    ……
}
```

- メンバー(変数、メソッド)の宣言は通常と同じ
- 抽象メソッドは持っていなくても良い
- 抽象クラスのオブジェクト(インスタンス)は生成できない

抽象クラスを利用する場合

- 抽象クラスを利用するには、そのクラスを拡張しなければならない。
  また、抽象クラスから継承した抽象メソッドの内容をサブクラスできちんと定義してオーバーライドする(サブクラスも抽象クラスの場合はしなくても良い)
  という作業をしなければならない。

- 抽象クラスを使うことで、サブクラスごとにメソッドの内容が違うので、同じカテゴリで色々な種類のクラスを扱いやすくなる。

  例えば、モンスターというカテゴリでは、色々な種類のモンスターが存在するが、それぞれ移動範囲や攻撃方法、特性などが異なるが、それぞれに対して必ずある名前の抽象メソッドが定義されているというようにすれば、管理しやすくなる。

抽象クラスと同様、難易度高め。

インターフェイス

インターフェイスのフォーマット

```
public interface インターフェイス名{
    //フィールドは必ず初期化する
    型名   フィールド名 = 値;

    //メソッドの処理は定義しない
    戻り値の型 メソッド名();
}
```

- インターフェイスも抽象クラスと同様に「オブジェクトは生成できない。」
- フィールドは(public) static final、
  メソッドには(public) abstractしか定義できない。
  すなわち、インターフェイスのフィールドは定数、メソッドは抽象メソッドとなっている。

インターフェイスの利用

- インターフェイスはクラスと組み合わせて使う。

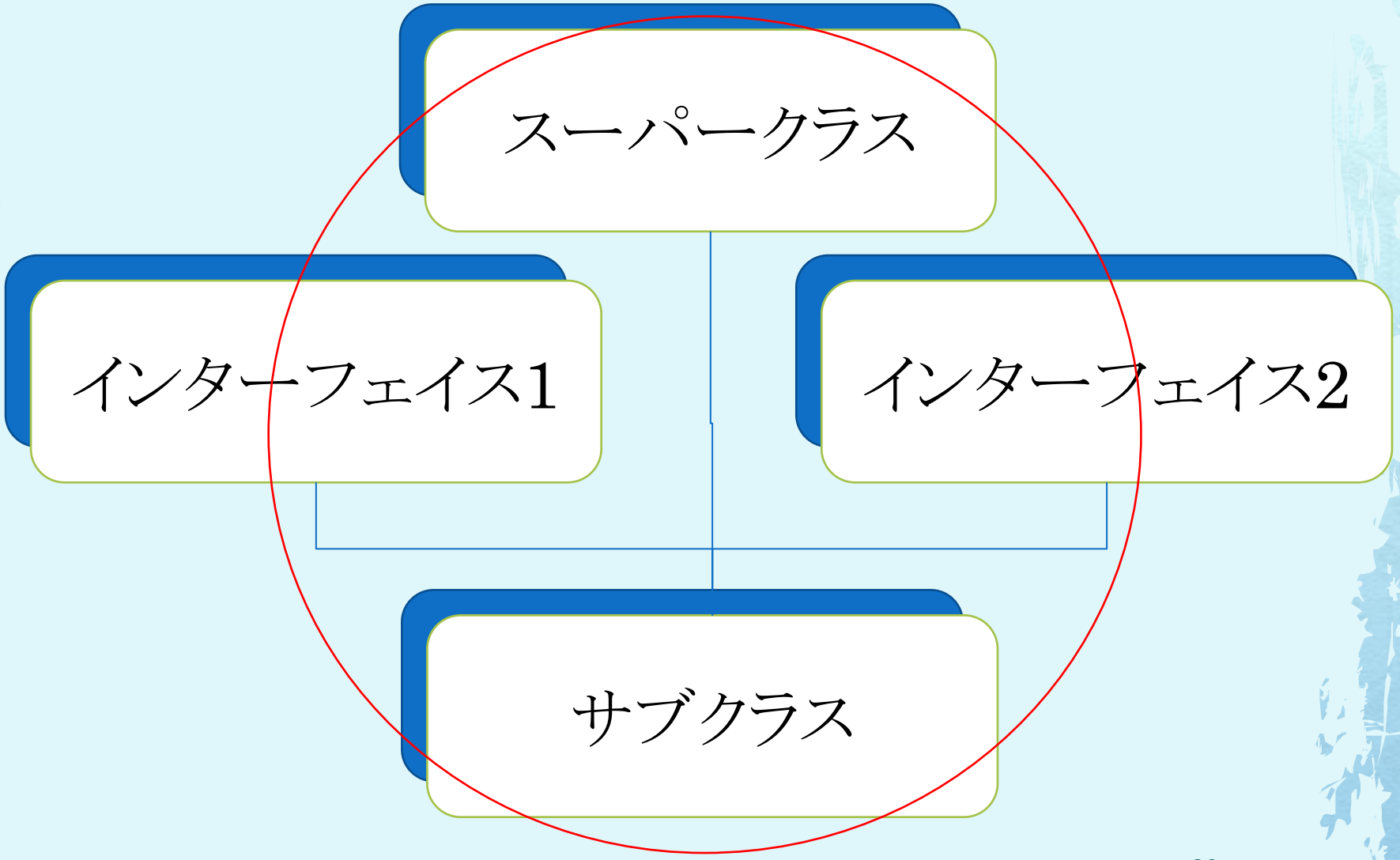
  これを、インターフェイスを実装する(implementation)という。

インターフェイスの実装

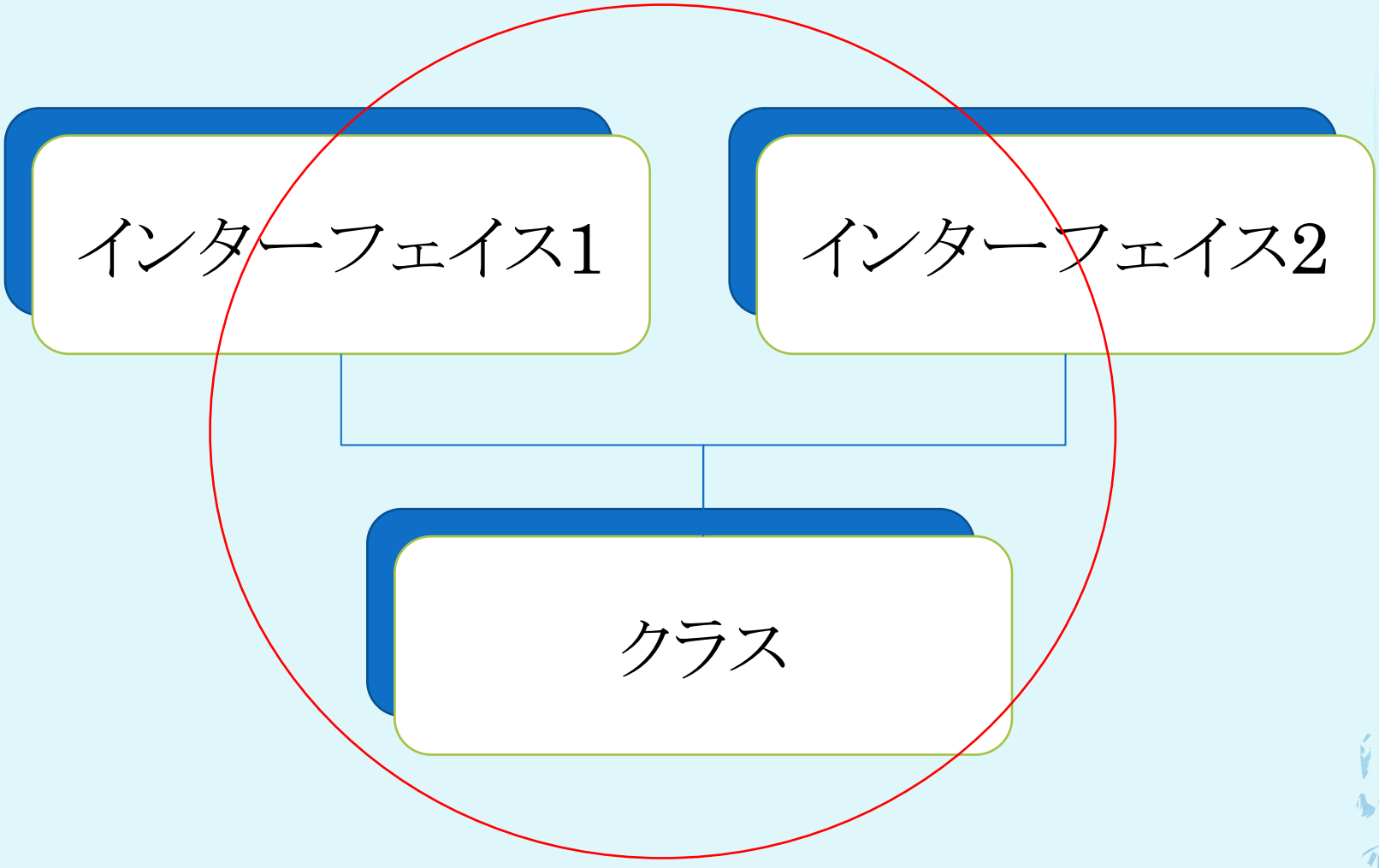
```
public class クラス名(またはインターフェイス名) implements
    インターフェイス名1, インターフェイス名2…{

    //実装したインターフェイスの持つ抽象メソッドを
    //全て定義しなければならない
}
```

- インターフェイスはカンマで区切れば、いくつでも実装(implements)できる。

インターフェイス(図)



インターフェイス(図)



なぜインターフェイスか？

- なぜ、インターフェイスを使うのか？
  前頁より、implementsの場合はいくつでもインターフェイスを実装できる。
  しかし、extendsの場合は一つのクラスの性質、機能しか継承できない。

  すなわち、インターフェイスを使うと多重継承の一部を実現することが出来る。

インターフェイスの拡張

```
public interface サブインターフェイス名 extends
        スーパーインターフェイス名1,
        スーパーインターフェイス名2…{

        …
}
```

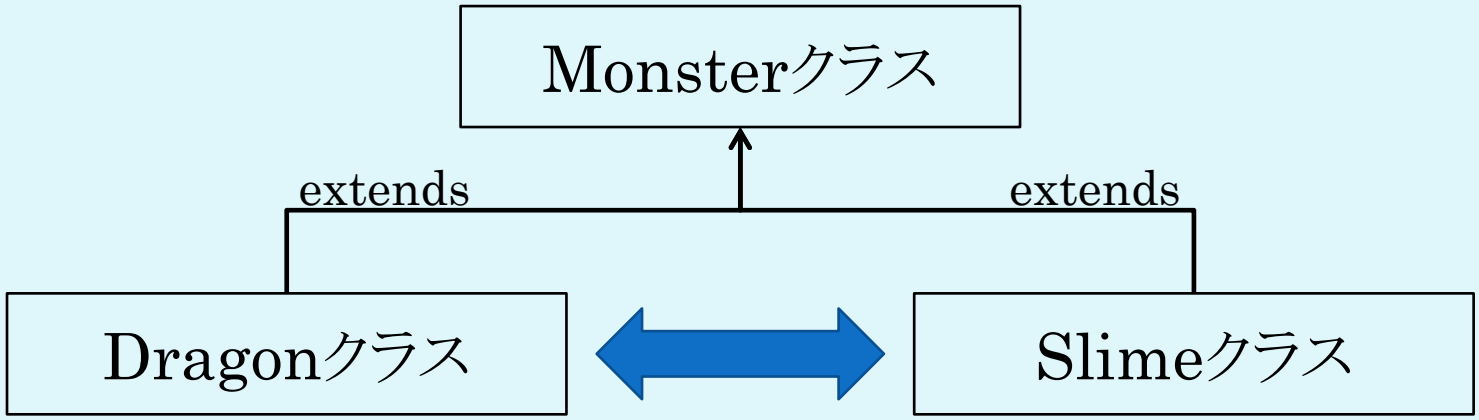
- インターフェイスもクラスと同様に拡張することが出来る。
- クラスと同じく拡張されるほうをスーパーインターフェイス、拡張したほうをサブインターフェイスという。
- 拡張にはextendsを使い、いくつでもスーパーインターフェイスに指定できる。
  (なぜextendsであるか考えてみよう。)

型の同一視

- Javaでは、同じクラス(インターフェイス)を継承(実装)している場合、
そのクラス(インターフェイス)を型として、オブジェクトを同一視することが出来る。

- つまり、
void attackTo(Monster m){ ~ }
というメソッドがあるとすれば、
Dragon d = new Dragon();
attackTo(d);
などということができる。これは、他のサブクラスを定義しても同様である。

同一視のイメージ



Monsterという型においては、
同一視できる。

- Monster m = new Slime(); → ○
- Monster m = new Dragon(); → ○
- Dragon d = new Monster(); → ×
- Slime s = new Monster(); → ×
- Dragon d = new Slime(); → ×

抽象クラスとインターフェイスの利点

- 抽象クラスとインターフェイスは、抽象メソッドを宣言することが出来る。
- この抽象メソッドこそ利点そのものである。

しかし、現時点では話がややこしいので割愛する。
(気になる人は「Javaのデザインパターン」を学ぶと良いだろう。)

既存のインターフェイス

- これ以降のページではJavaにもともと用意されている便利なインターフェイスを紹介していく。

- 慣れないうちはインターフェイスは自分で作らず、あるものを使えば十分である。

Runnableインターフェイス

抽象メソッド(自分で定義しなければならないメソッド)

```
public void run();
```

- 簡単に言うと、並列処理(スレッド)のためのインターフェイスだが、このインターフェイスを実装したからといって並列処理(スレッド)が出来るわけではない。
- Threadオブジェクトを生成し、ターゲットに指定する必要がある。
  Thread t = new Thread(Runnableを実装したオブジェクト);
- run()メソッドに並列に行ってほしい処理を記述する。

Listener系インターフェイス

- イベントと呼ばれるユーザー側が起こしたアクションに反応して、
それに応じたメソッドを自動的に呼び出してくれるインターフェイスで、リスナーと呼ばれる。

- 利用する場合、まずリスナーを登録しなければならない。

- 紹介するもの以外にもたくさんある。

ActionListenerインターフェイス

抽象メソッド(自分で定義しなければならないメソッド)

```
public void actionPerformed(ActionEvent e);
```

- ボタンが押されたとき、テキストフィールドでEnterが押されたときに何かしたい、という場合に実装するインターフェイス。
- 上記のアクションが起こると、自動的にactionPerformed()メソッドが呼ばれ、変数eにそのアクションの情報が格納されている。

KeyListenerインターフェイス

抽象メソッド

```
public void keyTyped(KeyEvent e);
public void keyPressed(KeyEvent e);
public void keyReleased(KeyEvent e);
```

- キーが押されたとき、keyTyped()が呼ばれる。
- Enter、Shiftなどのキーが押されたとき、keyPressed()が呼ばれる。
- 押されていたキーが離されたとき、keyReleased()が呼ばれる。
- 変数eにどのキーが押されたかの情報が格納されている。

MouseListenerインターフェイス

抽象メソッド

```
public void mouseClicked(MouseEvent e);
public void mouseEntered(MouseEvent e);
public void mouseExited(MouseEvent e);
public void mousePressed(MouseEvent e);
public void mouseReleased(MouseEvent e);
```

- マウスがウィンドウに入った時、mouseEntered()が呼ばれ、出た時、mouseExited()が呼ばれる。
- マウスが左クリックされた時、mouseClicked()が、押し続けられた時、mousePressed()が、押されていたのが離された時、mouseReleased()がそれぞれ呼ばれる。

MouseWheelListenerインターフェイス

抽象メソッド

```
public void mouseWheelMoved(MouseEvent e);
```

- マウスのホイールが回された時、mouseWheelMoved()が呼ばれる。
- 変数eに左右どちらの方向に回されたかの情報が格納されている。

MouseMotionListenerインターフェイス

抽象メソッド

```
public void mouseDragged(MouseEvent e);
public void mouseMoved(MouseEvent e);
```

- マウスがドラッグしていて、マウスが動いた時にmouseDragged()が呼ばれる。
- マウスがウィンドウ内で動いた時にmouseMoved()が呼ばれる。

Appendix

Objectクラスについて

- Javaでは、スーパークラスを指定しなかったクラスはObjectクラスを継承することになっている。
- すなわちObjectクラスとは、全てのクラスの最上位に位置するようなスーパークラス。
- したがって、全てのクラスはObjectクラスのメソッドを継承している。

Objectクラスの主なメソッド

いくつかあるが、最も重要な2つのみ紹介する。

- String toString()
  - 生成したオブジェクトを文字列として処理しようとした時に自動的に呼ばれるメソッド。
    このメソッドが返す値がそのオブジェクトの文字列表現となる。
- boolean equals(Object)
  - メソッドを呼ぶオブジェクトと引数のオブジェクトが等しいかどうかを判定するメソッド。
    戻り値がtrueの場合は、同じオブジェクトとなる。(オブジェクト同士は比較演算子==では正しく判定できない)

Stringクラスについて

- 今までString型を利用してきたが、あまり深くは説明してこなかった。
- まず、Stringはクラスである。(一文字目が大文字であることからも明らか。)
  - つまり、Objectクラスを継承している。

- 「String str = “Hello”;」は
  「String str = new String(“Hello”);」と同じ意味のコードである。
  - つまり、文字列はオブジェクトである。
  - したがって、比較演算子==では正しく判定できないことがあるので、継承したequalsメソッドを使用する。
  - if(“Hello”.equals(“Hello world!”){～} とか。

スーパーなObjectクラス

- Javaでは、同じスーパークラスを持つサブクラスを同じスーパークラスの型として同一視することが出来る。
- つまり、
  Object[] array = new Object[3];
  array[0] = new Object();
  array[1] = new String();
  array[2] = new Monster(～);

- メソッドでよく、
  「メソッド名(Object arg)」という様な定義がある(equalsメソッドなど)。
  このとき引数argは、どんなクラスのオブジェクトでも引数として受け取ることが出来る。

- 継承を使えば、クラスが同一視でき、プログラムの拡張性が増す。

終わり