Java講座

第4回

情報科学部コンピュータ科学科
2年　竹中　優

今回の内容

- final修飾子
- 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
- 抽象クラス
- インターフェイス

クラスの拡張

クラスの拡張

- 前回、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、多くの共通点がある。

- 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。

- このように、新しいクラスを作成することをクラスを拡張する(extends)という。

クラスの拡張

- 新しいクラスは、既存のクラスのメンバー(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。

スーパークラス

- 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

```
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);
    }
}
```

Dragonクラス

- Dragonクラスのスーパークラスであるクラスが違うパッケージにある場合はimportしなければならない。
  インポート文：import パッケージ名.クラス名;

- 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()もsuper()も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先頭で記述し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)は出来ません。

クラスの拡張

- 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、クラスを作成する時に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
  という決まりになっている。

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・引数の数・型を持つメソッドを定義することができる。
- 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

```
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型であるので、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、上の例のように呼び出すことができる。

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」の名前を大文字で書き、単語の区切りは「_」で記述することが多い。

finalのまとめ

- フィールドにfinalをつけると、値を変更できなくなる。
- メソッドにfinalをつけると、サブクラスでオーバーライドできなくなる。
- クラスにfinalをつけると、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、程度に考えておいて、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

```
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、処理内容が定義されていないメソッド(抽象メソッド)を持っていて、それらにもabstractを付ける。

抽象クラス

```
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);
    ……
}
```

抽象クラス

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

- 抽象クラスを利用するには、サブクラスを拡張しければならない。
  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
  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抽象メソッドが定義されているので、扱いやすくなる。

インターフェイス

インターフェイスの宣言

```
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 = 式;

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

- フィールドとメソッドを持つことができるが、コンストラクターは持たない。
- 通常インターフェイスには何も修飾子を付けない。何も付けなくてもフィールドにはpublic static final、メソッドにはabstractという修飾子を付けているのと同じになる。
つまり、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宣言

- インターフェイスも抽象クラスと同じようにオブジェクトは生成できない。

- インターフェイスはクラスと組み合わせて使うことになっていて、インターフェイスをクラスと組み合わせることを、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 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

- 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…{

        …
}
```

- インターフェイスの拡張にはextendsを使い、いくつでも拡張できる。
- 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。

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、または便利なインターフェイスを紹介しておきます。

- 正直、自分はインターフェイスを作ったことがないので、
  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だ、と思っています。

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。

Runnableインターフェイス

抽象メソッド

```
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。
- run()に並列に行ってほしい処理を記述する。

終わり