Java講座

第3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

- クラス(複合データ)
- アクセス制御
  - private
  - public
  - (protected)
- static
- おまけ
- 参照と実体(発展)

クラス(複合データ)

クラスとは

クラスとは、1つの物(オブジェクト)を表す単位のようなもので、その物(オブジェクト)が持つ属性、ステータスなどを一つのまとまりとして定義できる。

例えば、
人というオブジェクトを考えた場合。
名前、体重、身長、などなど色々なステータスがある。

クラスを作成してみよう

- 試しにモンスターを表すMonsterクラスを作成してみよう。
- 今までと同様に新規クラスを作成。
(新規→クラス→名前付けて完了)
- クラス名の最初の文字は必ず大文字。
  - 逆に最初の文字が大文字であれば、それがクラスであるということ
    - System, JOptionPane, Math, String, Integer, Double, ...etc.
- mainメソッドは記述しなくても良い。

- これ以降のスライドでは、実際にプログラムを記述しながら進めていこう。

クラスのフィールド

```
public class Monster{
    String name;
    int hp;
    int atk;
}
```

Monsterクラスを定義した。
String型のname, int型のhp, int型のatkをステータスとして持っている。
このように、クラスに定義する変数をフィールドという。
フィールドにはクラスを型にも指定できる。

クラスのフィールドとメソッド

```
public class Monster{
        String name;
        int hp;
        int atk;

        void showStatus(){
                System.out.println("名前:" + name);
                System.out.println("HP:" + hp);
                System.out.println("ATK:" + atk);
        }
}
```

フィールド

メソッド

メンバー

- メソッドを記述すると、上記のようになる。
  ここで覚えてほしいのが、フィールド(変数)は引数として渡さなくてもクラス内では自由に使えるということ。
- クラスにおいて、フィールドとメソッドをまとめてメンバーという。

クラスを利用する

- 前頁ではクラスの定義を記述しただけ。
- 記述したクラスをプログラム中で使うには、

クラス名　変数名;
変数名 = new クラス名();

または、
クラス名　変数名 = new クラス名();
と記述すれば、オブジェクト(インスタンス)を生成できる。

オブジェクト(インスタンス)

- 生成したオブジェクトのメンバー(フィールド、メソッド)にアクセスするには「.」を用いる。
  例えば、
  Monster m = new Monster();
  m.name = “monster”;
  m.hp = 100;
  m.atk = 30;
  m.showStatus();
- 実際に記述して試してみよう。

メンバーの初期化

- 前頁の例において、
  m.showStatus();
  をフィールドに値を入れる前に記述したら、どうなる？

- コンソールには、
  名前:null
  HP:0
  ATK:0
  と出力される。
  名前がnullって。。。

コンストラクターとは

- 前頁のように意図しないことを避けるため、コンストラクターというものを記述する。

- コンストラクターとは、
クラス内に定義するvoid型のメソッドのようなもので、そのクラスのオブジェクトが生成されたとき(newされたとき)に一度だけ呼ばれる。メソッドと同様に引数を渡せる。

- コンストラクターでは、主に初期化処理などを行わせる。

コンストラクターの記述

```
public class Monster{
        String name;
        int hp;
        int atk;

        //コンストラクター
        Monster(String name, int hp, int atk){
                this.name = name;
                this.hp = hp;
                this.atk = atk;
        }
        void showStatus(){
                System.out.println("名前:" + name);
                System.out.println("HP:" + hp);
                System.out.println("ATK:" + atk);
        }
}
```

thisとは

- 前頁に記述した「this」とは、「オブジェクト自身(自分自身)」を表す語で、
「this.」とするとそのオブジェクト自身を表すことを強調、または区別し、そのメンバーにアクセスできる。

- 基本的にフィールド名と引数名が重複した場合などに使う。
したがって、重複させなければ使わなくても良い。

- 使ううちに覚えていこう。

コンストラクターのオーバーロード

- コンストラクターもメソッドと同様にオーバーロードできる。
  例えば、Monsterクラスにおいて、「名前」が引数に渡されなかった場合、nameフィールドを”Monster”とする、ということが出来る。

コンストラクターのオーバーロード

```
public class Monster{
        String name;
        int hp;
        int atk;

        Monster(int hp, int atk){
                name = “Monster”;
                this.hp = hp;
                this.atk = atk;
        }
        Monster(String name, int hp, int atk){
                this.name = name;
                this.hp = hp;
                this.atk = atk;
        }
        void showStatus(){
                 System.out.println(“名前:” + name);
                System.out.println(“HP:” + hp);
                System.out.println(“ATK:” + atk);
        }
}
```

コンストラクターを記述しないと？

- コンストラクターを定義しないと、
  空のコンストラクター
  クラス名(){
  　　//空
  }
  が定義されているのと同じである。

- コンストラクターを1つでも定義すると、その形式(引数の数、型など)でしかオブジェクト(インスタンス)を生成できなくなる。

アクセス制御

アクセス制御

- もし、あるクラスがパスワードのような外部には知られたくない、または変更されたくないフィールドを持っていたとしたら、どうだろう？

  これまでは外部のクラスからはオブジェクト(インスタンス)さえ生成してしまえば、アクセスし放題だった。不正な値でさえ入れられた。

- これらを制限するために、アクセスを制限するための修飾子がある。

private修飾子

- private修飾子を付けると、他のクラスからは一切アクセスを受け付けないメンバーとなる。
  private int hp;
  private void showStatus(){ ~ }
  など。
- つまり、Monsterクラス内で呼び出さない限り使われなくなる。

public修飾子

- public修飾子を付けると、privateとは逆にプロジェクト内の全てのクラスからアクセスが可能なメンバーとなる。
  public String name;
  など。

protected修飾子

- protected修飾子を付けると、異なるパッケージのクラスからはアクセス不可となる。
  ただし、スーパークラスのフィールドに対して、サブクラスからはパッケージが異なっていてもアクセスできるようになる。

- スーパークラス、サブクラスについては第4回でやる。

修飾子を付加しないと？

- 何も修飾子を付けないと、異なるパッケージのクラスからはアクセス不可となる。

Monsterクラス

```
public class Monster{
        private String name;
        private int hp;
        private int atk;

        public Monster(int hp, int atk){
            name = “Monster”;
            this.hp = hp;
            this.atk = atk;
        }
        public Monster(String name, int hp, int atk){
                this.name = name;
                this.hp = hp;
                this.atk = atk;
        }
        public void showStatus(){
                System.out.println(“名前:” + name);
                System.out.println(“HP:” + hp);
                System.out.println(“ATK:” + atk);
        }

}
```

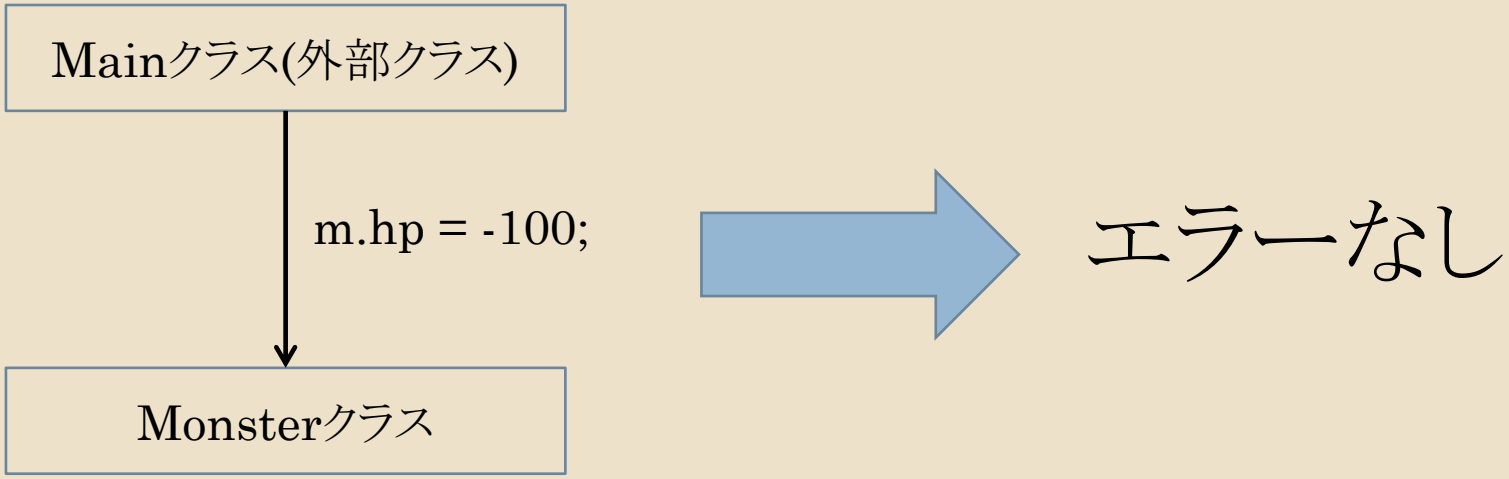
上記プログラムでは他のクラスからフィールドにアクセスできなくなり、値を見る場合は、showStatus()メソッドを呼び出すしかないので、安全なクラスと言える。

アクセス制御について

- 基本的にフィールドには全てprivateを、コンストラクターとメソッドにはpublicを付けると覚えておけばよい。
- 他のクラスからprivateなフィールドにアクセスさせる場合はsetterメソッド、getterメソッドというものを定義しておく必要がある。(後述)
- メソッドにprivateを付ける場合は他のクラスが呼び出す必要のないメソッドである場合など。

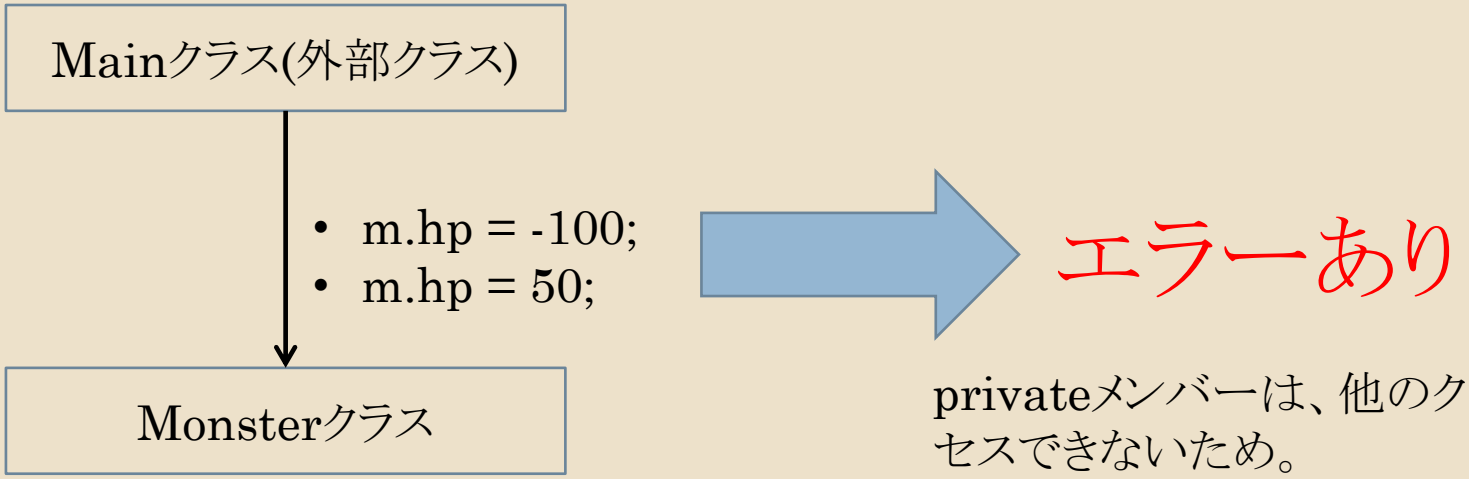
アクセス制御しないと．．．

```
Monster m = new Monster(/*何か適当な値*/);
…
```



アクセス制御すると．．．

```
Monster m = new Monster(/*何か適当な値*/);
…
```



privateメンバーは、他のクラスからアクセスできないため。

アクセス制御まとめ

- サンプルコードのjava_lec03.samples.p1.AccessTest.javaに対してテストを行う。
- テスト内容は「~.p1.A.java」と「~.p2.B.java」から「AccessTest.java」の各メンバーにアクセスできるかどうかである。
- 結果を以下の表にまとめる。

修飾子＼クラス	**p1.A.java**	**p2.B.java**
publicメンバー	○	○
修飾子なしのメンバー	○	×
protectedメンバー	○	×
privateメンバー	×	×

setter, getterのフォーマット

フィールドの最も基本的なsetterフォーマット

```
void setフィールド名(フィールドの型 フィールド名){
    this.フィールド名 = フィールド名;
}
```

フィールドの最も基本的なgetterフォーマット

```
フィールドの型 getフィールド名(){
    return フィールド名;
}
```

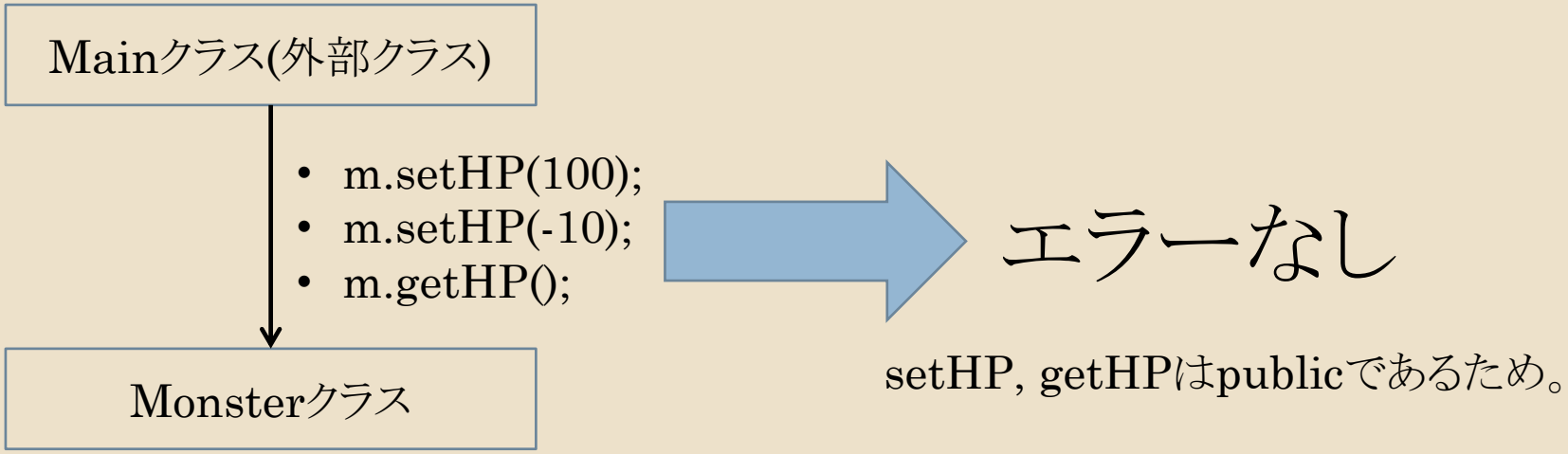
int hpのsetter, getter

```
public void setHP(int hp){
        if(0 <= hp)
                this.hp = hp;
        else
                this.hp = 0;
}
public int getHP(){
        return hp;
}
```

- 例として、Monsterクラスのフィールドhpのsetter, getterメソッドを定義した。
  特にsetterは不正な値が入れられないようになっている。

setter, getterを定義すると...

```
Monster m = new Monster(/*何か適当な値*/);
…
```



問題1

- 作成したMonsterクラスの全てのフィールド(変数)にsetter, getterメソッドを定義せよ。

オブジェクト生成しないメソッド

- Javaには簡単にダイアログ表示を行うJOptionPaneクラスというものがある。
  今まで何気なく利用していた
  JOptionPane.showInputDialog(“文字列”);
  などのメソッドが定義されている。

- しかし、これはJOptionPaneクラスのオブジェクト(インスタンス)を生成して、そのメソッドを呼んでいるわけではない。

staticについて

- 前頁のようにオブジェクト(インスタンス)を生成せずに、そのクラスのメソッド、変数を呼べたほうが便利な場合がある。

- このようにアクセスすることをstatic参照するといい、メソッド、変数の型の1つ前に「static」という語をつけることで行える。

  static……　静的

staticの使い方

public static フィールドの型　フィールド名;
public static メソッドの型　メソッド名;
private static フィールドの型　フィールド名;
private static メソッドの型　メソッド名;

- staticの付いた変数をクラス変数、メソッドをクラスメソッドという。
- 一方、staticの付いていない、すなわちオブジェクト(インスタンス)を生成してアクセスする変数をインスタンス変数、メソッドをインスタンスメソッドという。

static

- mainメソッドがpublic staticであるのは、呼ばれるタイミングが一番最初であるため、オブジェクトを生成できないから、と考えられる。
- 今までを振り返ってみよう。

```
public static void main(String[] args){
        new First().start();
}
```

上のコードは、
new First()でFirstクラスのオブジェクトを生成し、Firstクラスのインスタンスメソッドstartを呼び出していることになる。

これ以降、余裕のある人向け

参照と実体

参照と実体

```
Monster m1 = new Monster(“Monster1”, 100, 10);
Monster m2 = m1;
m1.setName(“Monster1111111”);//名前を変更
m2.showStatus();
```

- 上の例において、m2.showStatus()の出力結果を考えてみよう。

参照と実体

- 出力結果は、
  名前:Monster1111111
  HP:100
  ATK:10
  となっただろう。

- new クラス名();とすると、生成されたオブジェクトには、そのオブジェクトの「実体」を「参照」するためのアドレスのようなものが自動で割り当てられる。

- そのアドレスの場所にオブジェクトの「実体」が格納されている。

参照と実体

- 試しに先ほどの例で
  System.out.println(m1);
  System.out.println(m2);
  を記述して、その「参照」(アドレス)を出力してみよう。
- Monster m2 = m1;としているので同じ値が出力されるはずである。

- 「参照」が同じということは、同じ「実体」にアクセスしているということになるので、前頁の出力結果が得られた。

参照と実態のイメージ図

```
Object o1 = new Object();
```

Javaの仮想メモリ領域	
参照アドレス	実体
@0000000	オブジェクト
@0000001	
@0000002	

……

@00000FF	
@0000100	
@0000101	

……

参照と実態のイメージ図

```
Object o1 = new Object();
```

Javaの仮想メモリ領域	
参照アドレス	実体
@0000000	オブジェクト
@0000001	
@0000002	

……

@00000FF	
@0000100	
@0000101	

……

参照と実態のイメージ図

```
Object o1 = new Object();
Object o2 = new Object();
```

Javaの仮想メモリ領域	
参照アドレス	実体
@0000000	オブジェクト
@0000001	オブジェクト
@0000002	

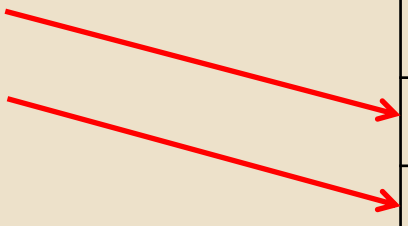
……

@00000FF	
@0000100	
@0000101	

……

参照と実態のイメージ図

```
Object o1 = new Object();
Object o2 = new Object();
```



Javaの仮想メモリ領域	
参照アドレス	実体
@0000000	オブジェクト
@0000001	オブジェクト
@0000002	

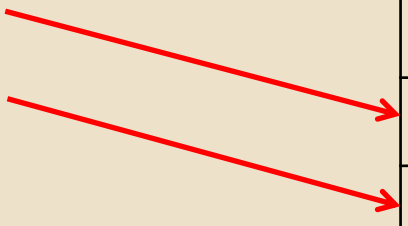
……

@00000FF	
@0000100	
@0000101	

……

参照と実態のイメージ図

```
Object o1 = new Object();
Object o2 = new Object();
Object o3 = o2;
```



Javaの仮想メモリ領域	
参照アドレス	実体
@0000000	オブジェクト
@0000001	オブジェクト
@0000002	

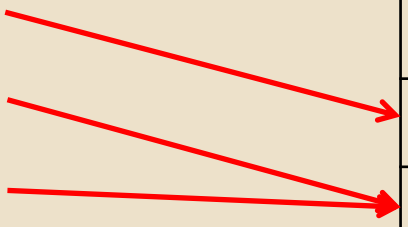
……

@00000FF	
@0000100	
@0000101	

……

参照と実態のイメージ図

```
Object o1 = new Object();
Object o2 = new Object();
Object o3 = o2;
```



Javaの仮想メモリ領域	
参照アドレス	実体
@0000000	オブジェクト
@0000001	オブジェクト
@0000002	

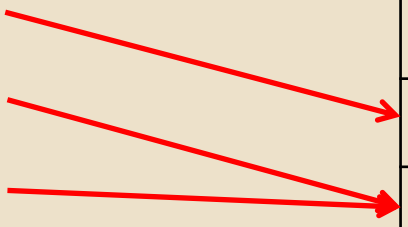
……

@00000FF	
@0000100	
@0000101	

……

参照と実態のイメージ図

```
Object o1 = new Object();
Object o2 = new Object();
Object o3 = o2;
o2 = o1;
```



Javaの仮想メモリ領域	
参照アドレス	実体
@0000000	オブジェクト
@0000001	オブジェクト
@0000002	

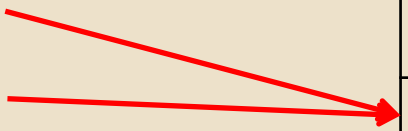
……

@00000FF	
@0000100	
@0000101	

……

参照と実態のイメージ図

```
Object o1 = new Object();
Object o2 = new Object();
Object o3 = o2;
o2 = o1;
```



Javaの仮想メモリ領域	
参照アドレス	実体
@0000000	オブジェクト
@0000001	オブジェクト
@0000002	

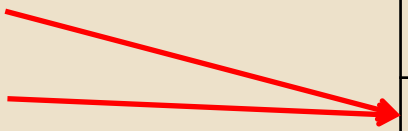
……

@00000FF	
@0000100	
@0000101	

……

参照と実態のイメージ図

```
Object o1 = new Object();
Object o2 = new Object();
Object o3 = o2;
o2 = o1;
o3 = null;
```



Javaの仮想メモリ領域	
参照アドレス	実体
@0000000	オブジェクト
@0000001	オブジェクト
@0000002	

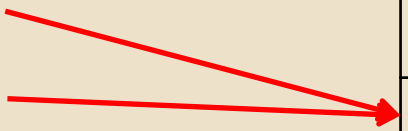
……

@00000FF	
@0000100	
@0000101	

……

参照と実態のイメージ図

```
Object o1 = new Object();
Object o2 = new Object();
Object o3 = o2;
o2 = o1;
o3 = null;
```



Javaの仮想メモリ領域	
参照アドレス	実体
@0000000	オブジェクト
@0000001	オブジェクト
@0000002	

……

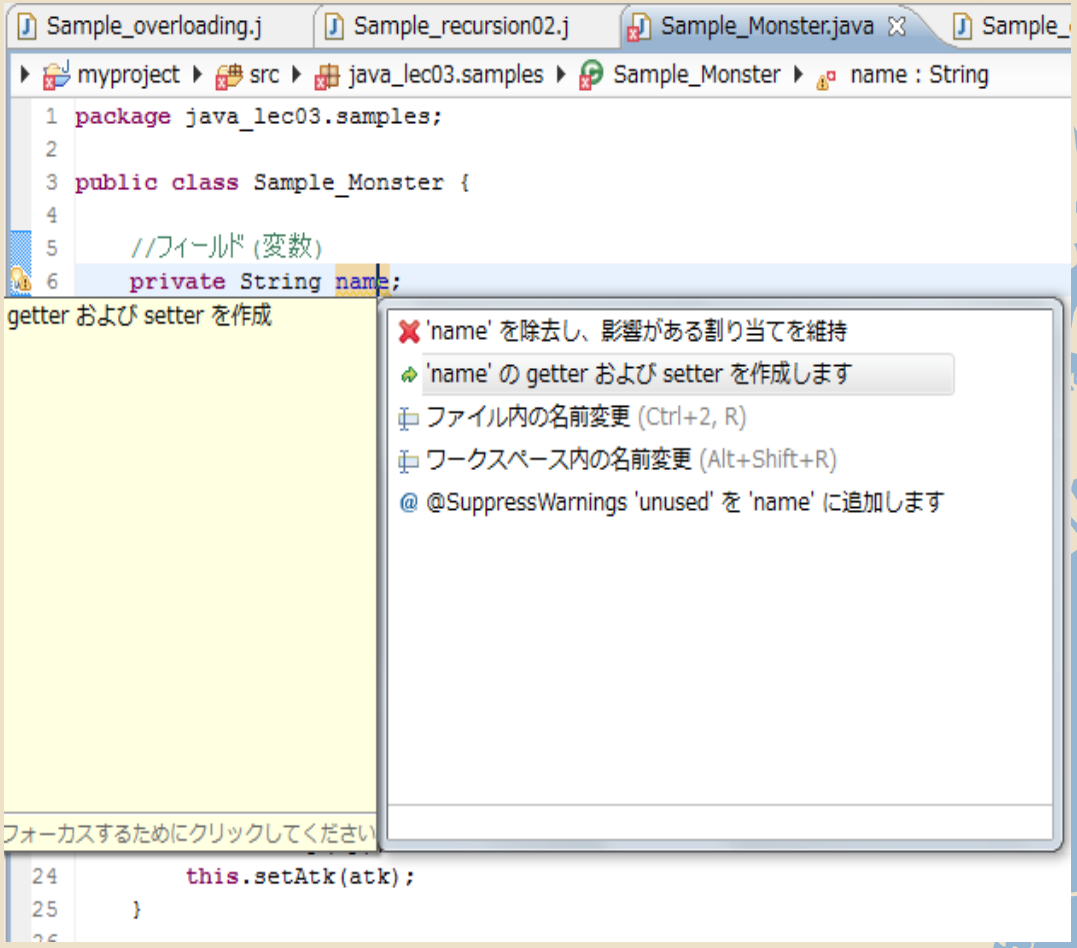
@00000FF	
@0000100	
@0000101	

……

Appendix

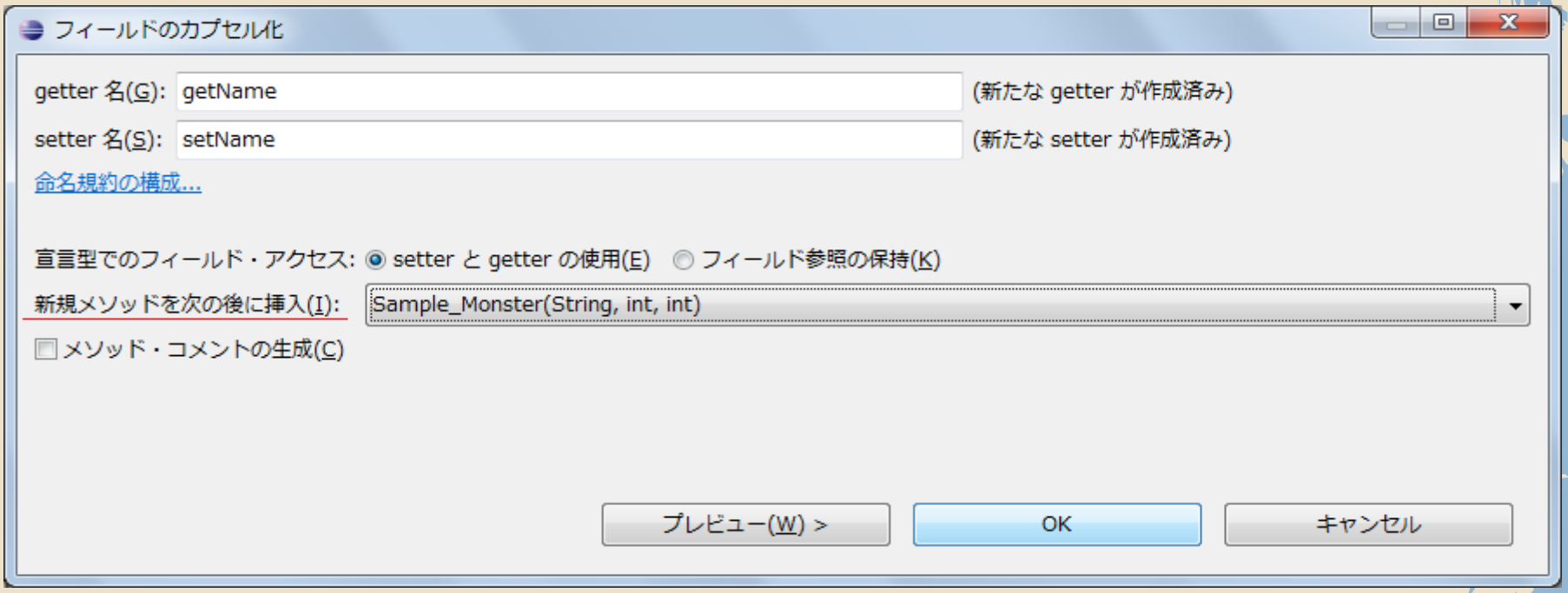
setter, getterを自動入力

- setter, getterメソッドを作成したいフィールドにカーソルを合わせ、[Ctrl+1]を押すと右図のようになる。



setter, getterを自動入力

- 前頁の図の「'name'のgetterおよびsetterを作成します」をクリックすると下図になるので、「OK」を押す。
  ウィンドウが出た場合、「継続」をクリック。



setter, getterを自動入力

- ‘name’フィールドのsetter, getterメソッドを自動で記述してくれる。
- 内容は先に述べたフォーマットの通りに記述されるので、条件分岐等をさせたい場合は、自分で追加しよう。
- 便利ですね。

コメントアウトについて

- Javaではコメントアウトが3種類ある。

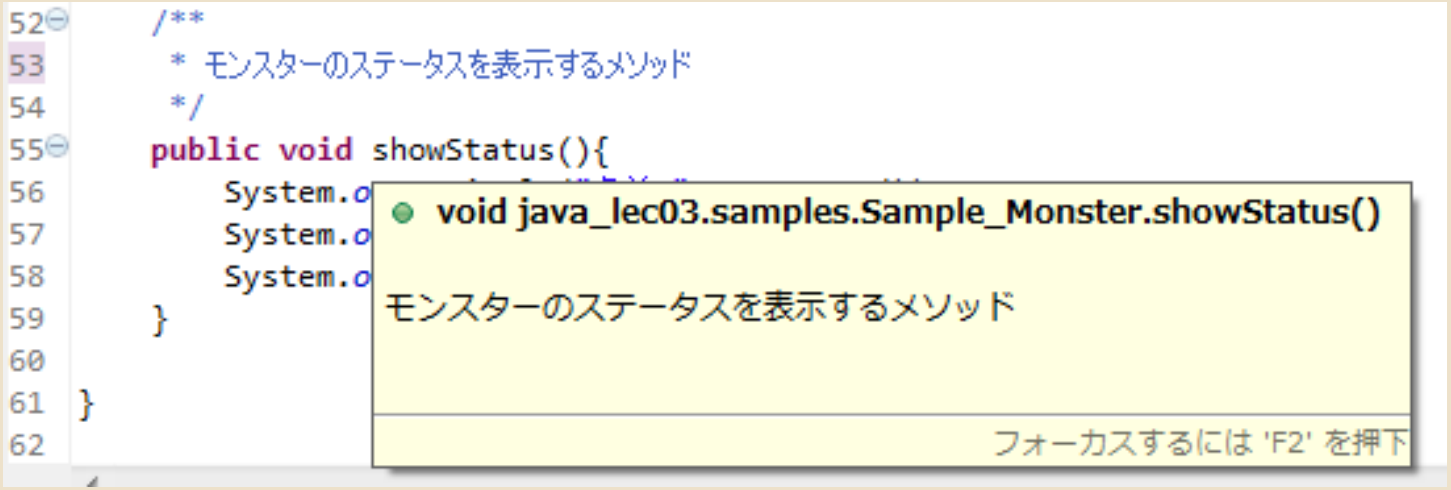
- //　　1行コメントアウト
- /* */　　複数行コメントアウト
- /** */　　Javadoc

Javadocについて

```
52      /**
53       * モンスターのステータスを表示するメソッド
54       */
55      public void showStatus(){
56          System.out.println("名前:" + getName());
57          System.out.println("HP:" + getHp());
58          System.out.println("ATK:" + getAtk());
59      }
```

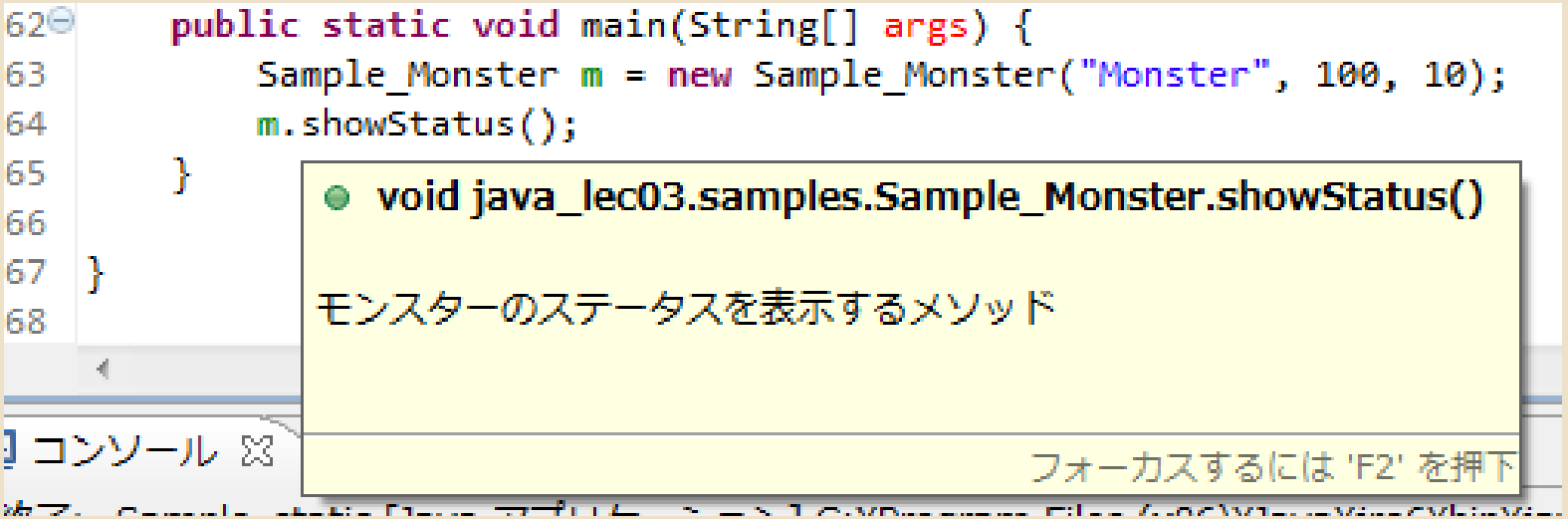
- 上図のように記述し、
  メソッド名(showStatus)にマウスオーバーすると…

Javadocを表示



- コメントアウトに記述した内容がポップアップ表示される！

Javadocを表示



- メソッドの呼び出し側のコード中で、
そのメソッド名にマウスオーバーしても良い。

- メソッド以外にも、変数、クラスにも利用できる。

Javadocとは

- ここで改めてJavadocとは、
/** */の間に記述する文章で、主にコメントアウトとして使う。
- わざわざ定義を見なくても、呼び出し側でコメントが確認できる。
- Javadocは、HTML形式で記述できるため、ハイパーリンクを貼ったり、文字をボールド体で強調したりすることも可能。
- 是非活用してみよう！

終わり