発展プログラミング演習II

5. フィールド

オブジェクト指向プログラミング

- オブジェクト ＝ 物 ＝ 物質＋機能
  - 例：机　　天板、脚
    物を上に置く、動かす、倒す、……
- オブジェクト間の相互作用で全体の動作を記述
  - 例：ボールが壁に当たる→ボールが跳ね返る
    オブジェクト指向の場合　どう跳ね返るかはボールと壁が知っている
    手続き型の場合　神様（全体を記述するプログラム）が決める
- オブジェクト ＝ データ＋手続き

フィールド
＋メソッド

  - cf. 手続き型プログラミングでは　データ構造＋アルゴリズム
  - クラス：オブジェクトの定義＝データ構造ごとに固有の処理を一体化したもの

オブジェクト指向のメリット

- プログラムの機能ごとにクラスに切り分ける

- ソフトウェアの再利用がやりやすい
- プログラムの保守性がよい
- 大規模なプログラムの作成に適している

クラス（復習）

- クラス＝オブジェクト*の設計図　　*インスタンスとも呼ぶ
- Java言語ではすべてのプログラムは
  クラスで記述されなければならない
- プログラムはクラスを記述するが、
  そのプログラム自体が実行されるのではない。
  オブジェクト化されて初めて動作する。
  （staticと付いたものを除く）

小さなプログラム例

```
class HelloJavaObject {
  void sayHello() {
    System.out.println(“Hello Java!”);
  }

  public static void main(String[] args) {
    HelloJavaObject app = new HelloJavaObject();
    app.sayHello();
  }
}
```

- java HelloJavaObject として実行
  →まずmainが呼ばれる
- HelloJavaObjectオブジェクトを作って
  （コンストラクタを呼ぶ）変数appに格納する
- appのsayHello()メソッドが呼ばれる

mainメソッドは特別

```
class HelloJavaObject {
  void sayHello() {
    System.out.println(“Hello Java!”);
  }
}
```

```
class HelloJava {
  public static void main(String[] args) {
    HelloJavaObject app = new HelloJavaObject();
    app.sayHello();
  }
}
```

- public static void main(String[])
- コマンド　java クラス名
  で指定したクラス名のmainが最初に実行される
- 複数のクラスがmainを持っていても構わない

クラスメンバ

- クラスを構成する要素（メンバ member）
  - フィールド(field)　モノの属性
  - メソッド(method)　モノの機能
- 例：車というクラスがあったとして、
  - 属性：車種名、色、位置、向き、速度、etc.
  - 機能：走る、止まる、曲がる、etc.

C言語との比較

- フィールド（メンバ変数）
  - 構造体の変数に似ている
- メソッド（メンバ関数）
  - 関数に似ている
- （かなり乱暴に言えば）クラス＝変数＋関数

フィールド field

- クラス内で定義（宣言）される変数
  - クラス内で自由にアクセス（読み書き）可能
  - クラス外からのアクセスは設定次第
  - オブジェクトごとに変数の実体が作られる
- メソッド内で定義されるローカル変数と違う
  - ローカル変数は定義されたスコープ内でのみ有効

フィールドの定義方法

- 定義方法：通常の変数と同じ
- 初期値を与えなければ自動的に初期値が設定される（型に応じて、0、false、null）

修飾子　型名　フィールド名;

例

```
public int hogehoge = 5;
```

```
private Double doubleObject;
```

例題5.1

```
class HelloJavaObject {
  void sayHello() {
    System.out.println(“Hello Java!”);
  }
}
```

```
class HelloJava {
  public static void main(String[] args) {
    HelloJavaObject app = new HelloJavaObject();
    int num = Integer.parseInt(args[0]);
    for (int i = 0; i < num; i++) {
      app.sayHello();
    }
  }
}
```

コマンドラインで与えた引数の回数だけ”Hello Java! ?回目”（?は1から順に増える）と表示するように修正せよ

引数？フィールド？

- 複数のHelloJavaObjectを扱うとき、
  呼び出し側が回数を数える変数を扱うのが面倒
- どこからでもseyHello()を呼べるようにしたい

- → HelloJavaObjectインスタンス自体が呼ばれた回数を覚えていればよい

修飾子 modifier

- クラス、変数、メソッドのアクセス可能な範囲や振る舞いを指定するキーワード
  - public, protected, private
  - static, final
  - 他、transit, volatile など

static

- クラスのフィールドには2種類ある：
- インスタンス変数
  - 実体はクラスインスタンスが個別に持つ
  - 変数名はインスタンス間で同じ
- クラス変数（static変数）
  - 実体はクラスが持つ（1個だけ!!）
  - インスタンス間で共有される

教科書3.5節参照

例題5.2

- 次のプログラムを実行した結果を予想せよ

```
class Exercise52 {
  static int classVar = 0;
  int instanceVar = 0;

  public static void main(String[] args) {
    Exercise52 obj1 = new Exercise52();
    Exercise52 obj2 = new Exercise52();

    obj1.classVar = 1;
    obj1.instanceVar = 1;

    System.out.println("obj2.classVar = " + obj2.classVar);
    System.out.println("obj2.instanceVar = " + obj2.instanceVar);
  }
}
```

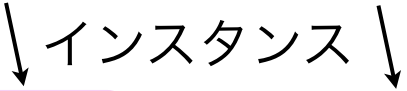
クラス変数とインスタンス変数

```
class Exercise52 {
  static int classVar = 0;
  int instanceVar = 0;

  .....
}
```

クラスの定義

クラス変数は
クラス自体が持つ
（インスタンス間
で共有される）




```
obj1
   instanceVar
```

```
obj2
   instanceVar
```

インスタンス変数は各インスタンス自身が持つ

クラス変数とインスタンス変数の使い分け

- 基本的にはインスタンス変数
- クラス変数にするべき場合
  - オブジェクト間で共有される場合

    例 `static int numMember; // 教科書例3.7`
  - 定数（finalと併せて宣言する）

    例 `static final int SIZE = 800;`
  - staticメソッドから使いたい場合（後述）

例題5.3

```
class HelloJavaObject {
  int count;
  void sayHello() {
    count++;
    System.out.println(“Hello Java! ” + count + “回目”);
  }
}
```

```
class HelloJava {
  public static void main(String[] args) {
    HelloJavaObject app1 = new HelloJavaObject();
    HelloJavaObject app2 = new HelloJavaObject();
    int num = Integer.parseInt(args[0]);
    for (int i = 0; i < num; i++) {
      app1.sayHello();
      app2.sayHello();
    }
  }
}
```

- int count; にstaticを付けたらどうなるか？

フィールドとローカル変数

- フィールド（メンバ変数）はクラス内のメソッドすべてからアクセス可能
  - クラス外については後述
- メソッド内で宣言されたローカル変数はスコープ内（＝ 宣言の含まれている{}内 ）のみで有効

例題5.4

- 次のプログラムを実行して動作を確認せよ

```
class Exercise54 {
   int x = 0;
   int z = 0;

   void method1() {
      int y = 1;
      x = 2;
   }

   void method2() {
      int y = 2;
      int z = 3;
      System.out.println("x = " + x + ", y = " + y + ", z = " + z);
   }

   public static void main(String[] args) {
      Exercise54 obj = new Exercise54();
      obj.method1();
      obj.method2();
   }
}
```

遮蔽

- ローカル変数とインスタンス変数が同じ名前を持つ場合、ローカル変数が優先される
  - 例題5.4では int method2() 内の z は、クラス Exercise54 のインスタンス変数 z を遮蔽する

this

- オブジェクト自分自身を指す特別な変数
- 宣言しなくても使用可能
- 通常は省略している
- 遮蔽された変数を明示的に指定するのに使える

- 例題5.4の13行目を次のように修正して動作を確認せよ

```
System.out.println("x = " + x + ", y = " + y + ", z = " + this.z);
```

final

- 上書きできないことを指定する修飾子
- 変数の場合は値を変更できなくする
  - 定数の宣言に使う
    （通常staticと同時に使う）
  - 定数であることを明確にするため、
    慣例として変数名は大文字のみにする
- クラス、メソッドにも使う

課題5+6(1/3)

最終的にこの課題を提出してもらいますが、途中経過の課題を順番にこなすとよいでしょう。

- とあるゲームプログラムにおいて使用する、ロボットとパワーアップアイテムのクラスを実装したい
  - ロボットには攻撃値と装甲値と耐久値が設定される
    - 攻撃値は攻撃したときに敵に与えるダメージである
    - 装甲値は攻撃されたときにダメージを受ける確率である
    - 耐久値はダメージを受けると減り、0になるとロボットは壊れてしまう
  - 攻撃値、装甲値、耐久値はロボットオブジェクト作成時に外部から設定される
  - ロボットは修理することで耐久値を最大値まで回復できる
  - ロボットには8つまでのパワーアップアイテム（以下、単にアイテム）を装備できる
    - ロボットにアイテムを装備すると、アイテムごとに設定された値だけロボットの攻撃力がアップする
    - 装備したアイテムを使うとロボットの攻撃値、装甲値、耐久値の最大値が変化し、そのアイテムは消滅する（修正値は装着時の攻撃力アップ値とは異なっていてもよい）
  - ロボットはオーバーヒート状態になることがある（オーバーヒート状態の効果は考慮しなくてよい）
    - 使うとオーバーヒートをおこすアイテムがある
  - ロボット、アイテムともにそれぞれのオブジェクトは名前を持つ（名前はオブジェクト作成時に外部から指定される）
  - アイテムごとに設定される値もオブジェクト作成時に外部から設定される

続く

課題5+6(2/3)

- 次の仕様の2つのクラス（ロボットとアイテム）を実装せよ
  - ロボットを表すクラスRobo
    - 最低限、次のパラメータを表すフィールドを持つ：
      - 名前　・攻撃値　・装甲値　・耐久値の最大値　・現在の耐久値
      - オーバーヒート状態の有無　・アイテムの枠（8つ分）
    - 最低限、次のメソッドを持つ：
      - コンストラクタ、および、必要なアクセッサメソッド。
        コンストラクタは名前、攻撃値、装甲値、耐久値の最大値を受け取る。
      - アイテムを指定された番号（0〜7）の枠に装着するメソッド。装着するアイテムオブジェクトと枠の番号を引数で受け取る。既に指定された番号の枠にアイテムが装着されている場合は新たなアイテムと入れ替えて、外した方のアイテムオブジェクトを返値で返す（無い場合はnullを返す）。アイテムを装着すると攻撃値が変化することに注意。
      - 指定された番号の枠に装着されているアイテムを使用するメソッド。返値は無し。アイテムを使用すると、アイテムに設定されているだけロボットの能力値や状態が変化する。また、使用したアイテムは無くなってしまう。指定された番号の枠にアイテムが装着されていなければ何も起こらない。
      - 攻撃値を引数に受け取りロボットの被攻撃判定を行うメソッド。乱数を一つ生成し装甲値以下であれば受けた攻撃値だけ耐久値が減る。ただし、1/50の確率で装甲値に関係なく攻撃値の2倍だけ耐久値が減る（クリティカルヒット）。 耐久値は0より小さくならない。判定後の耐久値を返値で返す。
      - 回復するメソッド。引数で渡された値だけ現在の耐久値が回復する（最大値を超えて回復しない）。回復後の耐久値を返値で返す。

続く

課題5+6(3/3)

- アイテムを表すクラスItem
  - 最低限、次のパラメータを表すフィールドを持つ：
    - 名前
    - 装着時の攻撃値修正値
    - 使用時の攻撃値修正値
    - 使用時の装甲値修正値
    - 使用時の最大耐久値修正値
    - 使用時にオーバーヒートになるか否か
  - 最低限、次のメソッドを持つ：
    - コンストラクタおよび必要なアクセッサ。コンストラクタは各フィールドの初期値を受け取る。各パラメータの値は初期値から変更されることはない。
- その他の要求：
  - アイテムの枠の数は将来変更するかもしれないので定数化しておくこと
  - クラス名には学生証番号を最後につけておくこと。例： Robo123456
- ヒントと注意：
  - 0.0 以上、1.0 より小さい正の乱数を得るには Math.random()を使う（double型の値が返る）
  - この2つのクラスは、全体のプログラムの一部であることに注意（これらだけで実行はできない）
  - コンパイルが通ることをチェックして提出すること。
    通らない場合はコメントに何がうまくいかなかったのか記入しておくこと。
  - mainメソッドは不要である（テスト用に作成してもよい）。

moodleで提出。ソースファイルは.zip圧縮して提出すること。締切11/19 16:45

課題5.1

・ロボットのクラスRoboについて、以下のフィールドを実装せよ
  ・名前
  ・攻撃値
  ・装甲値
  ・耐久値の最大値
  ・現在の耐久値
  ・オーバーヒート状態の有無

  （アイテム枠は課題5.2で実装）

アクセス修飾子

- クラス外からのアクセス（値の読み書き）の制限を指定する（教科書4.1, 5.5節参照）
  - public　どこからでもアクセス可能
  - protected　同じパッケージと、サブクラスからのみ
  - 省略時　同じパッケージ内のみ
  - private　同じクラス内のみ

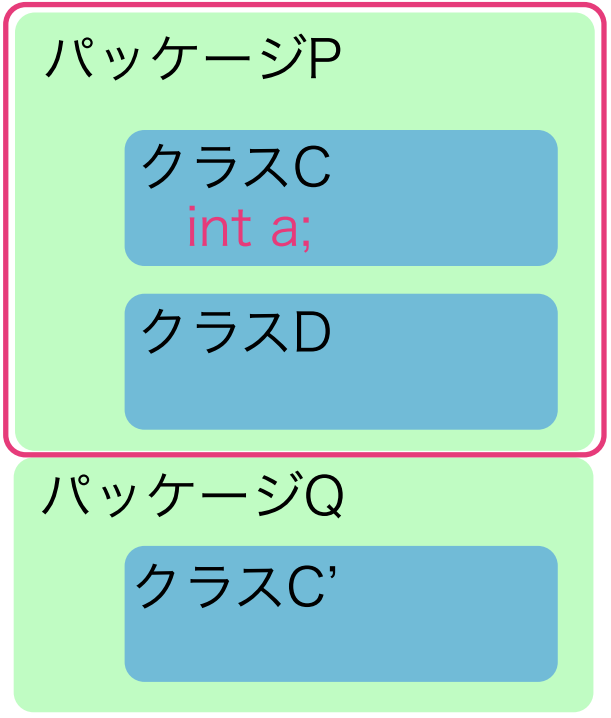
publicのアクセス可能範囲

- クラス外からも自由にアクセス可能
- 便利なように見えるが極力使わないこと（カプセル化に反する）

protectedのアクセス可能範囲

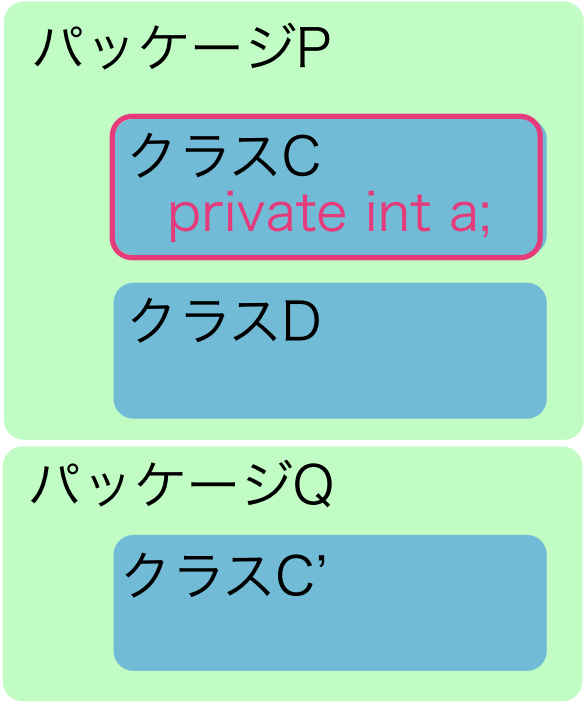
- 同じパッケージと、サブクラスからアクセス可能
- 平たく言えば関係者限定
- 外部パッケージにサブクラスがある場合に使える

省略時のアクセス可能範囲



- 同じパッケージ内からアクセス可能
- protectedと若干違う

privateのアクセス可能範囲



- クラス内でのみアクセス可能

インスタンス変数へのアクセス

- オブジェクト外から

  オブジェクト式.フィールド名

- オブジェクト内から

  `this.`フィールド名

thisは状況に応じて省略可能

staticメソッドからは不可

クラス変数へのアクセス

オブジェクト式.フィールド名

または

クラス名.フィールド名

クラス内の場合、クラス名は省略可能

フィールドのアクセス制限の基本

- 原則、できるだけ制限を強くする
  private > 省略 > protected > public
- 注意!! Java言語にはグローバル変数は無い
  （クラス外で変数を定義できない）
- 変数の場合、publicを使う理由は汎用の定数や仕方がない場合のグローバル変数の代わりのみ

例

```
public static final double PI;
```
（Mathクラス）

```
public static final PrintStream out;
```
（Systemクラス）

カプセル化

- クラス（＝ソフトウェアの部品）を「ブラックボックス化」する
  - 外部から見えなくてもよいものは見せない
- 変数の場合、外部からアクセス可能にすると、想定外の書き換えが行われるおそれがある
- フィールドは極力privateとし、必要に応じてアクセッサメソッドを用意する

アクセッサメソッド accessor

```
class accesserExample {
 private int hogehoge;

 int getHogehoge() {
  return hogehoge;
 }

 void setHogehoge(int hogehoge) {
  this.hogehoge = hogehoge;
 }
}
```

例題5.5

- 以下のフィールドを持つクラスCarを設計せよ
  - ドライバー名
  - 位置（2次元座標）
  - 速度（スカラー値）
  - 向き（北を0度とした角度）
- ドライバー名を引数で受け取って初期化する
  コンストラクタを作れ
- 各フィールドはprivateとしゲッターメソッドを用意せよ
- 変数名は任意。mainメソッドは無くてもよい（実行テスト用に作ってもよい）

例題5.6

- 以下のフィールドを持つクラスTireを設計せよ
  - 摩耗度
    （0〜1の実数値で表現、1:新品、0:要交換）
  - 空気圧（実数値、単位 kg/cm^3）
- 各フィールドはprivateとしゲッターメソッドを用意せよ
- オブジェクト作成時には、摩耗度と空気圧を初期値に設定すること
  - 初期値は摩耗度1.0、空気圧2.5kg/cm^3

例題5.7

- Carクラスにタイヤ4つ分のフィールドを追加せよ（配列でよい）
- Carクラスのコンストラクタ中に、
  タイヤを初期化する処理（新品のタイヤを装着する処理）を追加せよ

課題5.2

- アイテムのクラスItemについて、以下のフィールドを実装せよ
  - 名前
  - 装着時の攻撃値修正値
  - 使用時の攻撃値修正値
  - 使用時の装甲値修正値
  - 使用時の最大耐久値修正値
  - 使用時にオーバーヒートになるか否か

- ロボットのクラスRoboに以下のフィールドを追加せよ
  - アイテムの枠（8つ分）
    - アイテムの枠の数は将来変更するかもしれないので定数化しておくこと

ソースファイルは.zip圧縮して提出すること。