発展プログラミング演習II

6. メソッド

メソッド method

- クラスの機能、操作
- 定義方法

```
修飾子 返値の型 メソッド名（引数1の型 引数名1, ..., 引数nの型 引数名n） {
    ...
}
```

例

```
public void moveTo(int x, int y) {
    posX = x;
    poxY = y;
}
```

インスタンスメソッドへのアクセス

- オブジェクト外から

  オブジェクト式.メソッド名（引数）

- オブジェクト内から

  this.メソッド名（引数）

thisは状況に応じて省略可能
staticメソッドからは不可
必要に応じて返値を受け取る

メソッドの修飾子

- アクセスレベル
  public, protected, private
- static, final
- abstract, synchronized
- 他

- 複数の修飾子を付ける場合、推奨される順序は

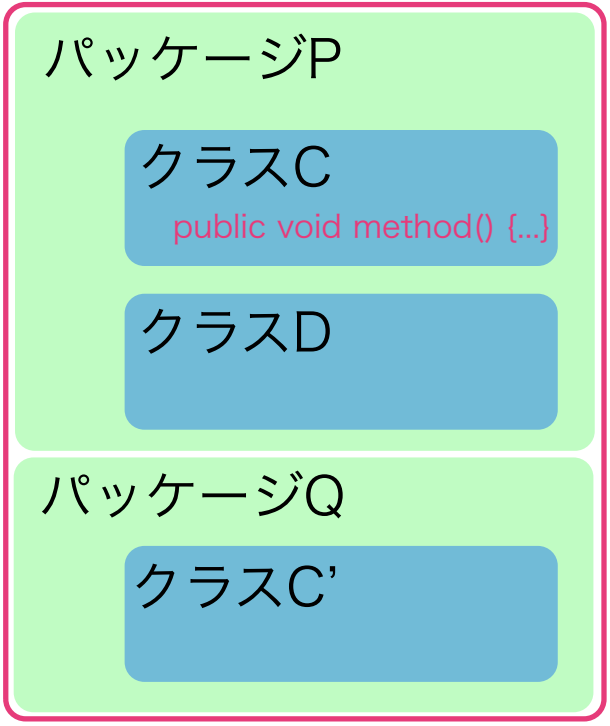
```
{public|protected|private} abstract static final
   transient volatile synchronized native strictfp interface
```

アクセス修飾子

- public　どこからでも呼び出し可能
- 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

- フィールドと同様

publicの呼び出し可能範囲

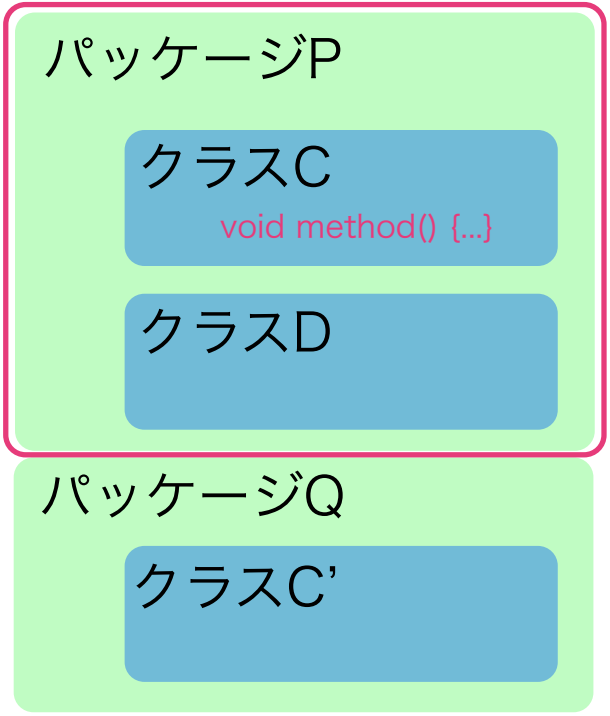


- クラス外からも自由に呼び出し可能

protectedの呼び出し可能範囲

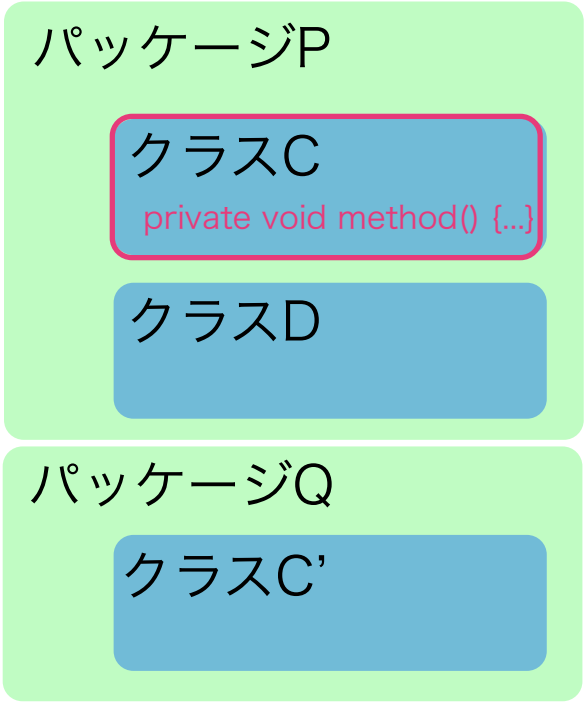
- 同じパッケージと、サブクラスから呼び出し可能
- 関係者限定
- 外部パッケージにサブクラスがある場合に使える

省略時のアクセス可能範囲



- 同じパッケージ内から呼び出し可能
- protectedと若干違う

privateの呼び出し可能範囲



- クラス内でのみ呼び出し可能
- クラス内でのみ使うメソッドに使う
  - 共通の処理の関数化
  - 再帰処理
  - プログラムの可読性の向上
  - など

カプセル化（再掲）

- 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
- メソッドの場合、外部から呼び出されて困るものは隠す

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6.1

- 車を表すクラスに対して以下のメソッドのそれぞれについて、privateかpublicかどちらがよいと思われるか？

```
void setAccel(int val) {.....}  // アクセル量を設定する
void calcSpeed() {.....}  // 現在のアクセル量から速度を計算する
double getSpeed() {.....}  // 現在の速度を返す
void burnFuel() {.....}  // 燃料を消費する
void setBreak(int val) {.....}  // ブレーキ量を設定する
void setSteer(int angle) {.....}  // ハンドルを切る
boolean checkCrash(Car anotherCar) {.....}  // 衝突判定する
```

static

- クラスメソッド（staticメソッド）を定義
- クラスインスタンス外に存在
  ＝特定のオブジェクトに結びつけられていない
  - staticメンバのみにアクセス可能
  - インスタンス変数を使うことはできない
  - インスタンスメソッドを呼び出せない
- 外部から呼び出し可能なstatic関数は
  C言語の汎用関数的な使い方ができる
  例：java.lang.Mathクラス

クラスメソッドへのアクセス

オブジェクト式.メソッド名（引数）

または

クラス名.メソッド名（引数）

クラス内の場合、クラス名は省略可能

必要に応じて返値を受け取る

mainメソッド

- mainメソッドのシグネチャ：

```
public static void main(String[] args)
```

- 形式的には他のstaticメソッドと同じ
- メインクラスとして指定された（javaコマンドの引数として与えられた）クラスの持つmain関数が起動時に自動的に呼ばれる
- 複数のクラスがmainメソッドを持ってもよい

final

- 上書きできないことを指定する修飾子
- メソッドの場合はオーバーライドを禁止する
  - オーバーライド：
    上位クラスのメソッドと同じシグネチャを持つメソッドを定義すると上位クラスのメソッドが遮蔽される →継承、多態性

その他のメソッドの修飾子*

- abstract　抽象メソッド →抽象クラス
  - 下位クラスで必ず実装しなければならない
- synchronized　マルチスレッド時の排他制御
- native　他言語で実装されるメソッド
- strictfp　浮動小数点演算を厳密に行う

引数

- メソッドの定義の引数の型と呼び出し時の引数の型は一致している必要がある
- 引数は常に値渡し
  - 但し、参照型の変数（オブジェクト、配列）は参照渡しと同様の振る舞いとなる

例題6.2

右のプログラムを実行したとき、run()の最後でiとjとkの保持する値はどのようになるか？
値を出力するコードを追加して実行し確認せよ

```
class MyInt {
   private int value;

   MyInt(int value)  {this.value = value;}
   void setValue(int val) {value = val;}
   int getValue()  {return value;}
}
```

```
class ArgTest {
   void run() {
      int i = 1;
      MyInt j = new MyInt(2);
      MyInt k = new MyInt(3);
      iMod(i);
      jMod(j);
      kMod(k);
   }

   void iMod(int val) {
      val = 4;
   }

   void jMod(MyInt val) {
      val.setValue(5);
   }

   void kMod(MyInt val) {
      val = new MyInt(6);
   }

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

返値とreturn文

- Java言語では必ず返値の型を指定する
  返値が無い場合はvoid型を指定
- メソッド内に返値の型を返すreturn文が必要
  - return文が無いとコンパイルエラーとなる
- 返値が無い場合はreturn文は省略可能
  - return; と書いてもよい

返値の利用(1/2)

- 返値は基本的に同じ型で受け取る
- 上位クラスの型で受け取ることは可能

```
Integer calc() {.....}
.....
Number number = calc();
```

- 型キャスト可能ならば、明示的に型キャストして受け取ることも可能（だが危険）

```
Number calc() {.....}  // Integerを返す
.....
Integer number = (Integer)calc();
```

返値の利用(2/2)

- 返値をそのまま引数として渡してもよい

```
method2(obj.method1());
```

- メソッドの呼び出しを連結してもよい
  （型に注意）

```
obj.method1().method2();
```

コンストラクタ

- クラスのオブジェクトを作成するメソッド
  - クラスと同じ名前
  - 返値を持たない（インスタンスが返値）
- 通常、オブジェクトの初期化処理を行う
- 定義されていない場合は空のデフォルトコンストラクタが自動的に提供される

オーバーロード overload

- メソッド名が同じでも引数（型、個数）が異なると別のメソッドとして扱われる
  - シグネチャは異なっていることになる
  - 返値だけ異なるのは不可
- いずれが呼ばれるかは引数の型に応じてコンパイル時に決定される

オーバーロードの例

```
// 合計を計算するメソッド
int sum(int x, int y)  {.....}
int sum(int x, int y, int z)  {.....}
int sum(int[] numbers)  {.....}
int sum(Integer[] numbers)  {.....}
int sum(Collection<Integer> collection)  {.....}
```

- オーバーロードが有効なケース
  - 多様な引数に対して同じ挙動をするメソッドを提供したい場合
  - 引数の省略（デフォルト値）を許す場合
- むやみに使うのは混乱のもと!!

例題6.3

- 次のプログラムを入力し、
  コンパイル時のエラーを確認の上、
  正しく動作するように修正せよ

```
class Exercise63 {
   void doubleAndPrint(int x) {
      System.out.println(x * 2);
   }

   void run() {
      int i = 3;
      doubleAndPrint(i + 2);
      doubleAndPrint(i + 2.0f);
      doubleAndPrint(i + 3.5);
   }

   public static void main(String[] args) {
      Exercise63 app = new Exercise63();
      app.run();
   }
}
```

int型の引数に対して
float型やdouble型の引数を挿入する

課題6.1

- ロボットのクラスについてコンストラクタと必要なアクセッサメソッドを実装せよ
  - 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

- アイテムのクラスについてコンストラクタと必要なアクセッサメソッドを実装せよ
  - 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。

ソースファイルは.zip圧縮して提出すること。

例題6.4

- 以下の4つのメソッドを持つクラスCalcを作成し、異なる引数でそれぞれを呼び出し、引数に応じて呼ばれるメソッドが異なることを確認せよ
  - メソッド内で、どのメソッドか分かるような出力をすればよい

```
// 合計を計算するメソッド
int sum(int x, int y)  {.....}
int sum(int x, int y, int z)  {.....}
int sum(int[] numbers)  {.....}
int sum(Integer[] numbers)  {.....}
```

コンストラクタのオーバーロード

- 基本的にはメソッドのオーバーロードと同じ

```
class Greeting {
  private String message;
  private static final String DEFAULT_MESSAGE = “Hello”;

  Greeting() {
    message = DEFAULT_MESSAGE;
  }

  Greeting(String message) {
    this.message = message;
  }
}
```

this()での呼び出し

- this()で別のコンストラクタを呼び出せる
- コンストラクタ中の最初の文でのみ可

```
class Greeting {
  private String message;
  private static final String DEFAULT_MESSAGE = “Hello”;

  Greeting() {
    this(DEFAULT_MESSAGE);
  }

  Greeting(String message) {
    this.message = message;
    .....
    .....
  }
}
```

コンストラクタの隠匿化*

- コンストラクタをprivateに設定すると
  クラス外から呼び出せなくなる!!
  - ＝クラスインスタンスを作れない
- 有効なケース
  - インスタンスを作られると困るクラス
  - インスタンスを一つに限定したいクラス

シングルトン Singleton*

- インスタンスが一つだけのクラス
- 二つ以上作られると困るクラスに適用する
  - 例：リソースを排他的に占有するクラス

```
class Singleton {
  private static final Singleton _instance = new Singleton();

  private Singleton() {}  // インスタンスの作成を禁止する

  Singleton instance()  {return _instance;}
}
```

※ガベージコレクションの対象にならないように、
他のクラスでインスタンスを保持しておくべき

例題6.5

- テストの点数を表すクラスExamScoreを次のように作成せよ（教科書 演習4.2）

・フィールド
  ・private int math;（数学の点数）
  ・private int phys;（物理の点数）
  ・private double ave;（2科目の平均点）
・メソッド
  ・public void setMath(int m);（数学の点数を設定する）
  ・public void setPhys(int p);（物理の点数を設定する）
  ・public int getMath();（数学の点数を得る）
  ・public int getPhys();（物理の点数を得る）
  ・public double getAve();（2科目の平均点を得る）
・コンストラクタ
  ・private ExamScore();（点数を全て0点とする）
  ・public ExamScore(int m, int p);（点数を設定する）
・クラスが持つべき機能
  ・点数は全て0点から100点の間とする。
  ・それ以外の点数を指定しようとした場合、「点数が範囲外です」と画面に出力し、そのフィールドは0とする。

例題6.6

- 10人の学級の成績を管理するExamClassを次のように作成せよ

・フィールド
  ・private[] ExamScore;（10人分の成績）
・メソッド
  ・public void setExamScore(int num, ExamScore score);（num番目の成績を登録する）
  ・public double getMathAve();（数学の学級平均点を得る）
  ・public double getPhysAve();（物理の学級平均点を得る）
  ・public double getAve();（2科目の学級平均点を得る）
・コンストラクタ
  ・public ExamClass();（学級を作成する）

・mainメソッドで以下の処理を行え
  ・3学級分のExamScoreクラスオブジェクトを作る
  ・それぞれのクラスに対して、10人分の成績を乱数で作成し、登録する
  ・各学級の数学、物理の平均点を表示
  ・全学級の両科目の平均点を表示

課題5+6(1/3)

残りはロボットクラスのメソッド4つのはずです

- 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