Javaプログラミング実習

20. オブジェクト指向プログラミング基礎

株式会社ジードライブ

今回学ぶこと

- クラスの定義と利用方法
- クラスの関連用語と概念
- UMLのクラス図

クラスの定義（作成）

- クラス名とファイル名は一致させる
  - ファイル名はパスカルケース（先頭文字と単語の区切りを大文字）にする
  - ファイルの拡張子は .java とする
- クラス内にはフィールド(変数)とメソッドを定義する

書式

```
public class クラス名 {
   フィールド1
   フィールド2
   …
   メソッド1
   メソッド2
   …
}
```

クラス定義の例

例：Itemクラス の定義（ファイル名は Item.java）

```
public class Item {
  public String name;
  public int price;

  public void showInfo() {
    System.out.println("商品名：" + name);
    System.out.println("値段：" + price + "円");
  }
}
```

フィールド(変数)

メソッド

フィールド

- クラス内に定義される変数を**フィールド**と呼ぶ
  - フィールドはメソッドの外で定義される
- **static**(静的)が付いているフィールドは、**静的フィールド**と呼ばれる (クラスフィールド/クラス変数とも呼ばれる)
- **static**が付いていないフィールドは、**インスタンスフィールド**と呼ばれる

例

```
public class Item {
  public static int itemCount; // 静的フィールド
  public String name; // インスタンスフィールド
  ...
```

メソッド

- **static**(静的)が付いているメソッドは、**静的メソッド**と呼ばれる (クラスメソッドとも呼ばれる)
- **static**が付いていないメソッドは、**インスタンスメソッド**と呼ばれる

例

```
public class Item {
  ...
  public static void showCount() {
    System.out.println("商品数：" + count);
  }
  public void showPrice() {
    System.out.println("値段：" + price + "円");
  }
  ...
```

静的メソッド

インスタンスメソッド

クラスのメンバ

- 静的フィールド
- 静的メソッド
- インスタンスフィールド
- インスタンスメソッド

まとめて「メンバ」と呼ぶ

  - ドキュメントの中では、クラス名とメンバを#記号で繋げて表記することがある

例：Itemクラスの`showInfo()`メソッド

⇒ `Item#showInfo()`

インスタンスの生成と利用

- クラスからインスタンス(オブジェクト)を生成するには、new演算子を使う
  - 基本的にインスタンスは変数に代入して利用する
    ⇒ 代入先の変数の型はクラス型になる
    (クラス名が型名になる)

書式

```
new クラス名()
```

例：Itemクラスの利用 ⇒ Item型の変数にインスタンスを格納する

```
Item item1 = new Item();
Item item2 = new Item();
```

Item型の変数の宣言

インスタンスの生成と利用

- Itemクラスの使用例

```
public class MyApp {
  public static void main(String[] args) {
    // インスタンスの生成
    Item item1 = new Item();

    // フィールド(変数)の利用
    item1.name = "りんご";
    item1.price = 100;

    // メソッドの利用
    item1.showPrice();
  }
}
```

フィールドやメソッドにアクセスする際はドットを使う

練習

- 練習20-1

コンストラクタ

- インスタンス生成直後に自動的に呼び出されるメソッド
  - インスタンスの初期化処理などを記述する
  - コンストラクタの名前はクラス名と同じ
  - 戻り値は定義できない

書式

```
public クラス名(型 引数名, ...) {
  // コンストラクタ内で行う処理
  ...
}
```

コンストラクタ

- 例：Itemクラス のコンストラクタ

```
public class Item {
  ...
  public Item(String name, int price) {
    this.name = name;
    this.price = price;
  }
}
```

this

- this はnewで生成されたインスタンス自身を表す
  - フィールドやメソッドとドットでつなげる
  - 以下の例ではthisが必要だが、基本的には省略可能
  - 省略可能であっても、明示的に記す場合もある

例

```
public class Item {
  public String name;
  ...
  public Item(String name, int price) {
    this.name = name;
    ...
```

コンストラクタの引数

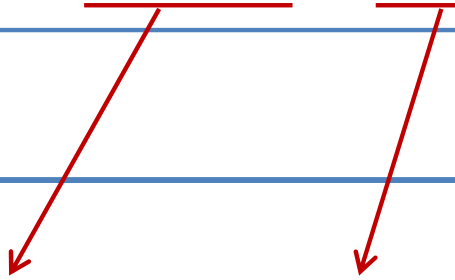
- コンストラクタの引数へ渡す値はインスタンス生成時に指定する

インスタンスの生成

```
Item item1 = new Item("りんご", 100);
```

コンストラクタの定義

```
...
public Item(String name, int price) {
  this.name = name;
  this.price = price;
}
```



コンストラクタの引数

- 引数を持つコンストラクタの使用例

```
public class MyApp {
  public static void main(String[] args) {
    // Itemオブジェクトの作成
    Item item1 = new Item("りんご", 100);
    Item item2 = new Item("バナナ", 200);

    // showInfo()メソッドの呼び出し
    item1.showInfo();
    item2.showInfo();
  }
}
```

コンストラクタのオーバーロード

- 通常のメソッドと同様に、コンストラクタもオーバーロードが可能

例

```
// 名前だけを指定するコンストラクタ
public Item(String name) {
  this.name = name;
}

// 名前と金額を指定するコンストラクタ
public Item(String name, int price) {
  this.name = name;
  this.price = price;
}
```

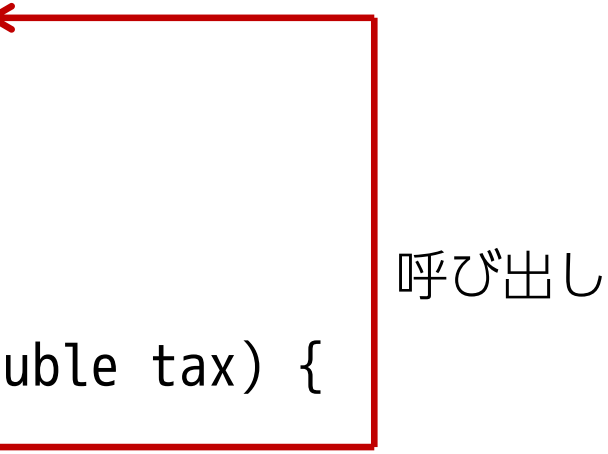
コンストラクタの呼び出し

- コンストラクタ内で別のコンストラクタを呼び出す場合は、**this()** を記述する
  - this()はコンストラクタ内の先頭行に記述しなくてはいけない

例

```
public Item(String name, int price) {
  this.name = name;
  this.price = price;
}

public Item(String name, int price, double tax) {
  this(name, price);
  this.tax = tax;
}
```



デフォルトコンストラクタ

- クラス定義時にコンストラクタを省略するとデフォルトコンストラクタが暗黙的に定義される
  - コンパイル時に自動生成される
- デフォルトコンストラクタの特徴
  - アクセス修飾子はpublicで、引数や処理内容をもたない（以下のようなコンストラクタを記述するのと同じ）

```
public class Item {
  public Item() {
    // 処理内容なし
  }
}
```

デフォルトコンストラクタ

- クラス定義時に、引数をもつコンストラクタを定義すると、デフォルトコンストラクタは暗黙的には定義されなくなる
  - 引数なしでインスタンスの生成ができなくなる

```
public class Item {
  public Item(String name, int price) {
    …
  }
}
```

引数をもつコンストラクタのみが定義されている

```
public static void main(String[] args) {
   Item item1 = new Item("りんご", 100);
   Item item2 = new Item();
}
```

コンパイルエラーが発生する

フィールドの初期値

- 初期値の指定がないフィールドには、以下の値が自動的に初期値として設定される

型	例	初期値
boolean型	boolean	false
整数型	int, short, long 等	0
浮動小数点型	float, double	0.0
参照型	String, 配列 等	null

フィールドの初期値

- フィールドの初期値を設定している例

```
public class Item {
  public String name = "りんご";
  ...
```

- 基本的にオブジェクトの必須情報はコンストラクタの引数で渡すようにするとよい

練習

- 練習20-2

アクセッサ

- フィールドの代入と参照を行うメソッドのことをアクセッサ(アクセッサ・メソッド)と呼ぶ
  - フィールド代入用のメソッドを Setter と呼ぶ
  - フィールド参照用のメソッドを Getter と呼ぶ
- フィールドへの直接的なアクセスを制限する、**カプセル化**(後述)における重要なテクニック

アクセッサの例

- 例：nameフィールドへのアクセッサ

```
public class Item {
  private String name;

  // Getter : nameフィールドを参照するためのメソッド
  public String getName() {
    return name;
  }

  // Setter : nameフィールドに値を代入するためのメソッド
  public void setName(String name) {
    this.name = name;
  }
}
```

アクセッサとなるメソッド名は、
フィールド名の前にgetやsetをつけたものにする

アクセス修飾子

- クラスのメンバに付与される public や private といった記述はアクセス修飾子と呼ばれ、他のクラスからアクセスできるかどうかを決定する
  - 前頁のnameフィールドはprivateなので、自クラスからのみアクセス可能となっている

アクセス修飾子	説明
public	どのクラスからもアクセス可能
protected	自クラス、サブクラス、同パッケージ内のクラスからアクセス可能
アクセス修飾子なし (package-private)	自クラス、同パッケージ内のクラスからのみアクセス可能
private	自クラスからのみアクセス可能

アクセス修飾子の例

クラスの定義

```
public class Item {
  public String name;
  private int price;

  public void setPrice(int price){
    this.price = price;
  }
}
```

クラスの利用

```
Item item1 = new Item();
item1.name = "バナナ"; // publicなのでOK
item1.price = 200;     // privateなのでエラー
item1.setPrice(200);   // ⇒ publicなSetterを通じてアクセス
```

カプセル化

- オブジェクトの内部構造を隠し、公開メソッドからのみアクセスさせる手法
  - フィールドを `private` や `protected` にする
  - フィールドにアクセスするための `public` なメソッド(**アクセッサ**)を用意する
  - アクセッサ名は、フィールド名に`get/set`を付けたものにする
    例：`myName`フィールドの場合、`getMyName()`, `setMyName()`
  - **Getter**メソッドのみ作成することで、**Read Only**のフィールドを実装することができる
  - **Setter**メソッドを利用することで、フィールドに入れられる値をチェックすることができる(バリデーションが行える)

Eclipseでは簡単にGetter/Setterが作成できる
「ソース ≫ GetterおよびSetterの生成」

練習

- 練習20-3

静的フィールドの特徴①

- インスタンスを生成せずに利用することができる

Car.java

```
public class Car {
  public static int maxSpeed = 120;
}
```

MyApp.java

```
public class MyApp {
  public static void main(String[] args) {
    System.out.println(Car.maxSpeed);
  }
}
```

クラス名.フィールド

静的フィールドの特徴②

- 静的フィールドは各インスタンスではなく、クラスに属する
  - 共有の入れ物のようなイメージ

MyApp.java

```
public class MyApp {
  public static void main(String args[]) {
    Car car1 = new Car();
    Car car2 = new Car();
    System.out.println(car1.maxSpeed); //120と表示
    car2.maxSpeed = 180;
    System.out.println(car1.maxSpeed); //180と表示
  }
}
```

静的メソッドの特徴①

- インスタンスを生成せずに利用することができる

Car.java

```
public class Car {
  public static void moveForward() {
    System.out.println("前に進みます");
 }
}
```

MyApp.java

```
public class MyApp {
  public static void main(String args[]) {
    Car.moveForward();
  }
}
```

クラス名.メソッド()

静的メソッドの特徴②

- 静的メソッド内では、静的メンバのみ利用可能
  - 他クラスのインスタンスメンバは利用可能

```
public class MyApp {
  public static String message1 = "Hello";
  public String message2 = "How are you?";
  public static void sayHi1() { System.out.println("やあ"); }
  public void sayHi2() { System.out.println("元気ですか？"); }

  public static void main(String args[]) {
    System.out.println(message1);
    sayHi1();
    Car car = new Car();
    car.move();
  }
}
```

静的メンバなので利用可能
Message2やsayHi2()は利用不可

他クラスであれば、インスタンス
メンバであっても利用可能

練習

- 練習20-4

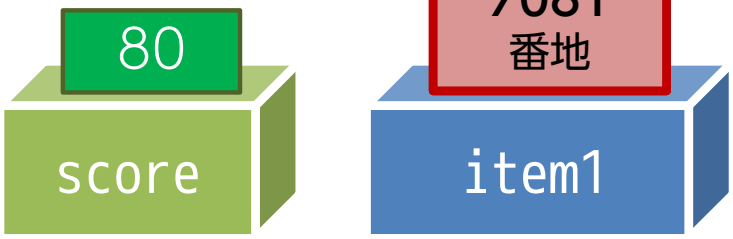
参照型

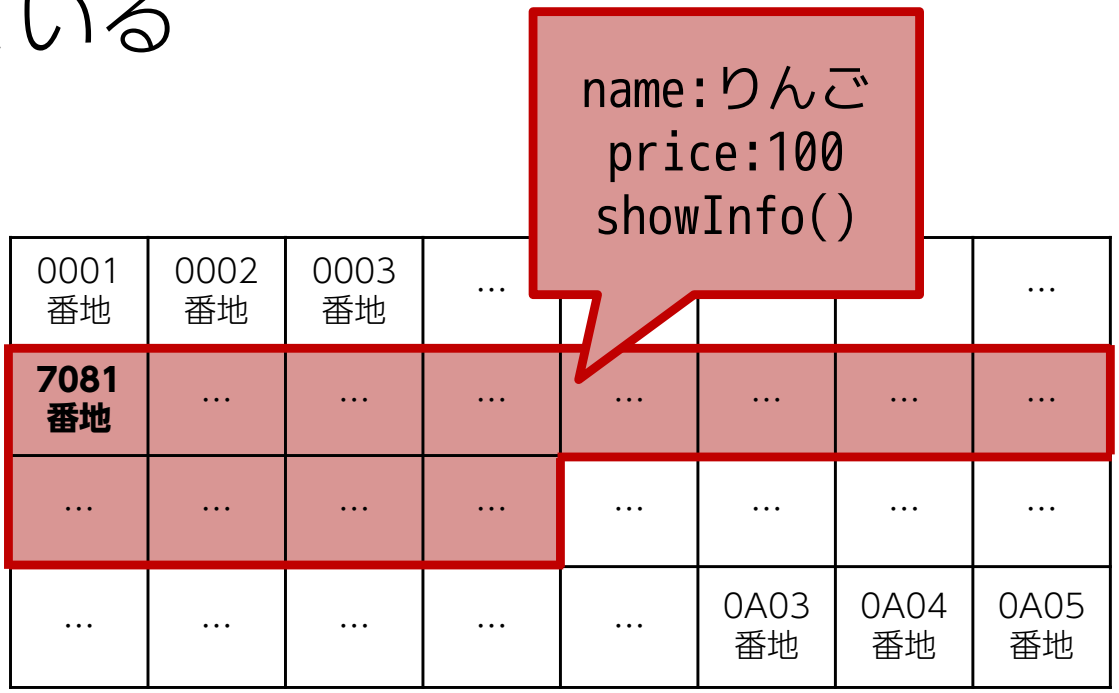
- プリミティブ型以外の型を、参照型と呼ぶ
- 参照型の種類
  - 配列型、列挙型、クラス型

型		例
プリミティブ型		`byte, char, short, int, long` `float, double, boolean`
参照型	配列型	`String[], int[]`
	列挙型	`enum`
	クラス型	`Arrays, Math` 自分で作成したクラスもクラス型になる

プリミティブ型と参照型の違い

- プリミティブ型の変数はデータの実体を格納している
- 参照型の変数はデータの実体があるメモリ上の場所情報を格納している

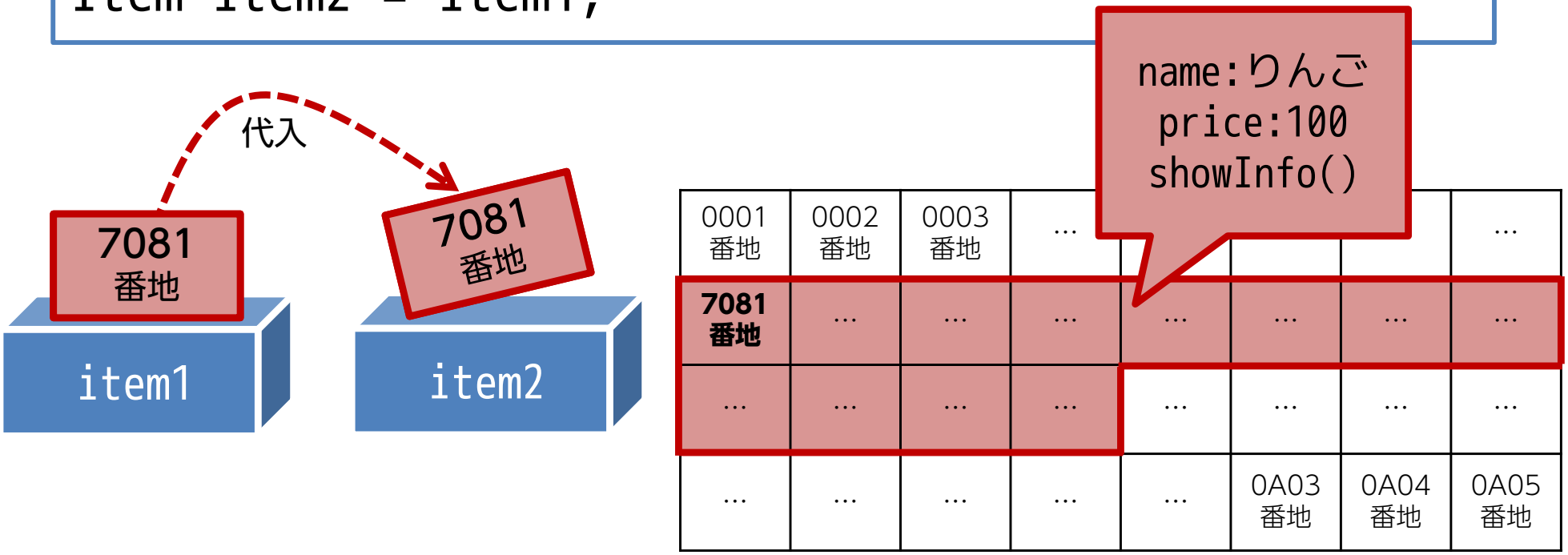




参照型の性質

- 参照型の代入はオブジェクトのコピーが作られるのではなく、**オブジェクトの場所を表す情報**がコピーされる

```
Item item1 = new Item("りんご", 100);
Item item2 = item1;
```



インスタンスの代入

- 配列の代入と同様の現象が起こるので注意する

Item.java

```
public class Item {
  public int price = 100;
  public void showInfo() {System.out.println("価格:"+price+"円");}
}
```

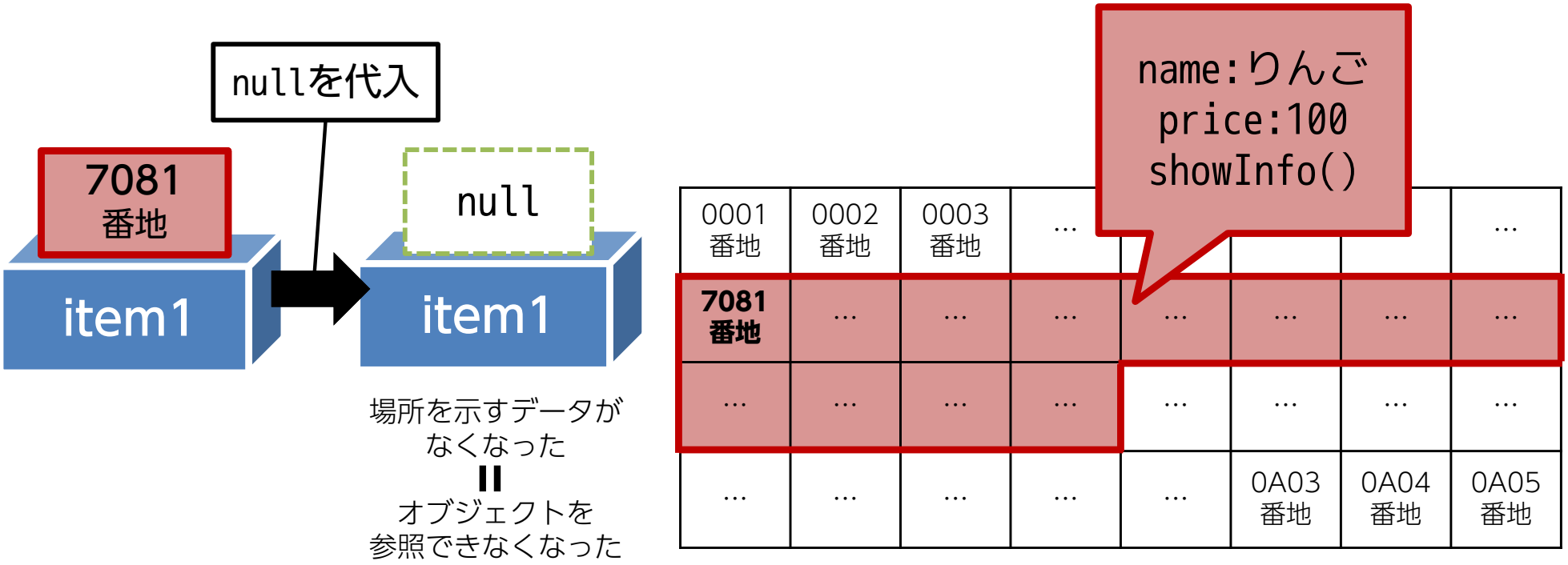
MyApp.java

```
public class MyApp {
  public static void main(String args[]) {
    Item item1 = new Item();
    Item item2 = item1;
    item2.price = 500;
    item1.showInfo(); //「価格:500円」と表示される
  }
}
```

参照型の性質

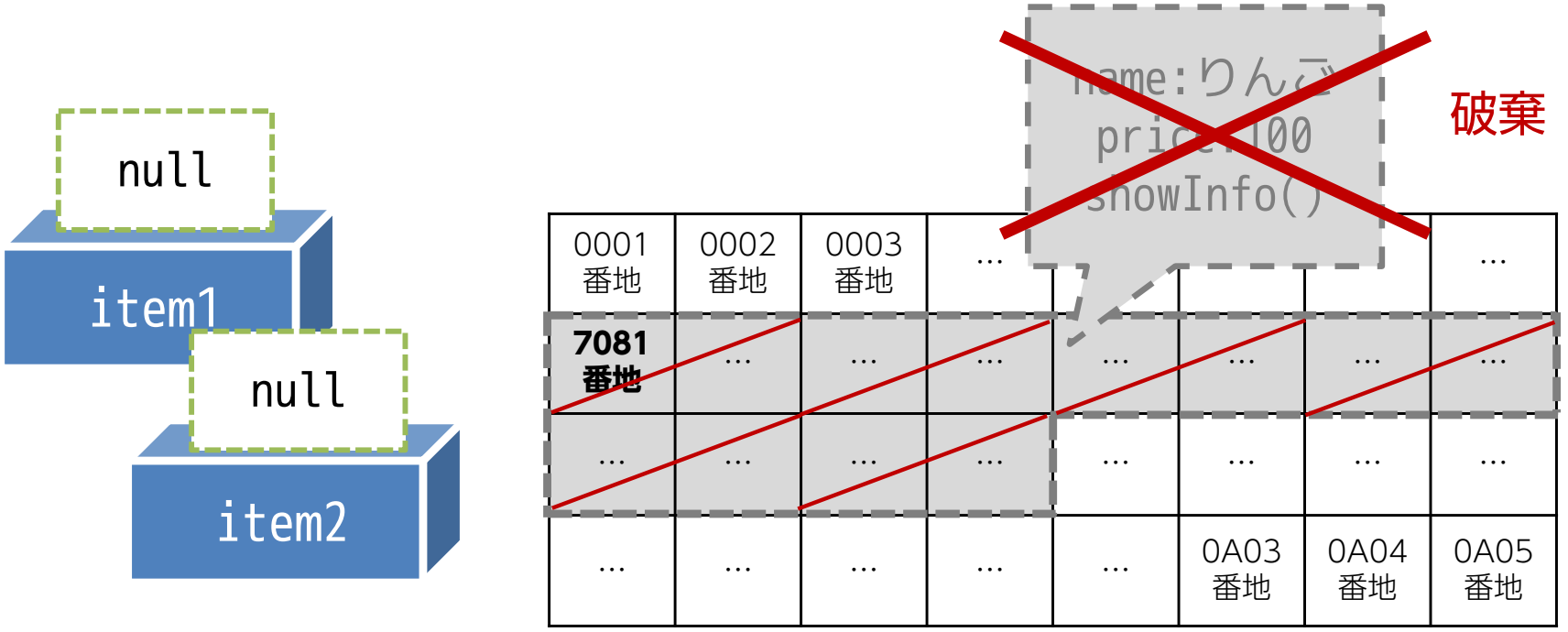
- 参照型変数にnullを代入すると、場所を指し示すデータがなくなるのでオブジェクトを参照できなくなる

```
item1 = null;
```



ガベージコレクション

- 変数にnullが代入されるなどしてオブジェクトがどこからも参照されなくなると、Java VMの判断によってオブジェクトが破棄される
- この仕組みをGC (Garbage Collection)という



練習

- 練習20-5

varによる変数宣言

- 変数の型を明示的に記述する代わりに、varを記述することもできる(型推論)
  - 冗長な型を記述しなくてもよい反面、バグの原因になってしまうことも考えられる

    ⇒ ルールを決めて運用するのが望ましい

例：UserPersonalInformationクラスの利用

```
// 従来の書き方
UserPersonalInformation upi1 = new UserPersonalInformation();

// varを使った書き方
var upi2 = new UserPersonalInformation();
```

依存関係

- 以下のPersonクラスはHello型のフィールドをもっており、introduce〇〇メソッド内で利用している
  ⇒ Helloクラスに**依存**している

Person.java

```
public class Person {
  private String name;
  private Hello hello;

  public void introduceInJapanese() {
    hello.japanese();
    System.out.println(name + "です");
  }

  public void introduceInEnglish() {
    hello.English();
    System.out.println("I am " + name);
  }
}
```

Hello.java

```
public class Hello {
 public void japanese() {
  System.out.println("こんにちは!");
 }

 public void english() {
  System.out.println("Hello!")
 }
}
```

依存性の注入

- Personクラス内でHello型のフィールドを利用するには、コンストラクタやセッターを通じて、依存しているHelloオブジェクトを代入する必要がある
  ⇒ 依存性の注入(Dependency Injection)と呼ばれる

Person.java

```
public class Person {
  private String name;
  private Hello hello;

  //コンストラクタ・インジェクション用
  public Person(String name,
                Hello hello) {
    this.name = name;
    this.hello = hello;
  }

  //セッター・インジェクション用
  public void setHello(Hello hello) {
      this.hello = hello;
  }

  public void introduceInJapanese {
    hello.japanese();
    System.out.println(name + "です");
  }
  ...
}
```

依存性の注入

前頁のPersonクラスの利用

```
// 依存関係を解決していない
Person person1 = new Person();
person1.setName("山田太郎");
person1.introduceInJapanese(); // NullPointerExceptionが発生

// コンストラクタを通じて、依存関係を解決
Person person2 = new Person("鈴木次郎", new Hello());
person2.introduceInJapanese(); // 例外は発生しない

// セッターを通じて、依存関係を解決
Person person3 = new Person();
person3.setName("佐藤花子");
person3.setHello(new Hello());
person3.introduceInJapanese(); // 例外は発生しない
```

コンストラクタ・インジェクション

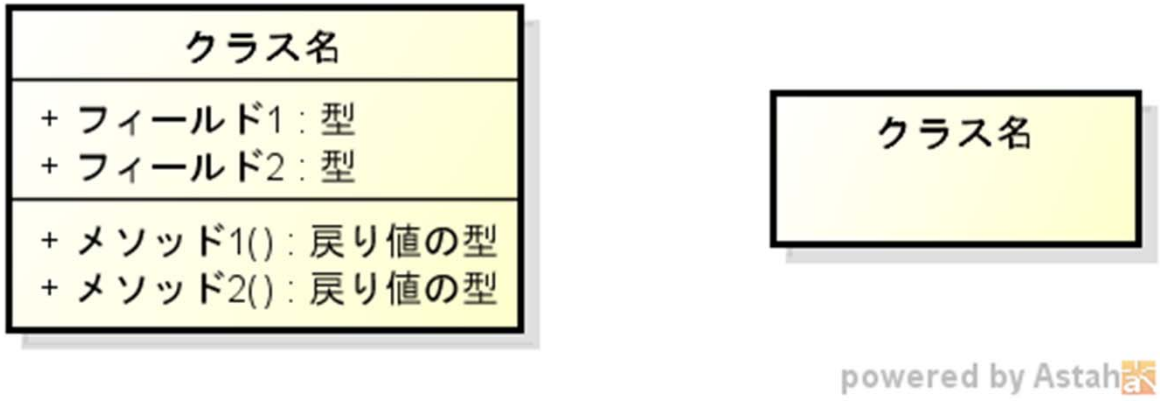
セッター・インジェクション

練習

- 練習20-6

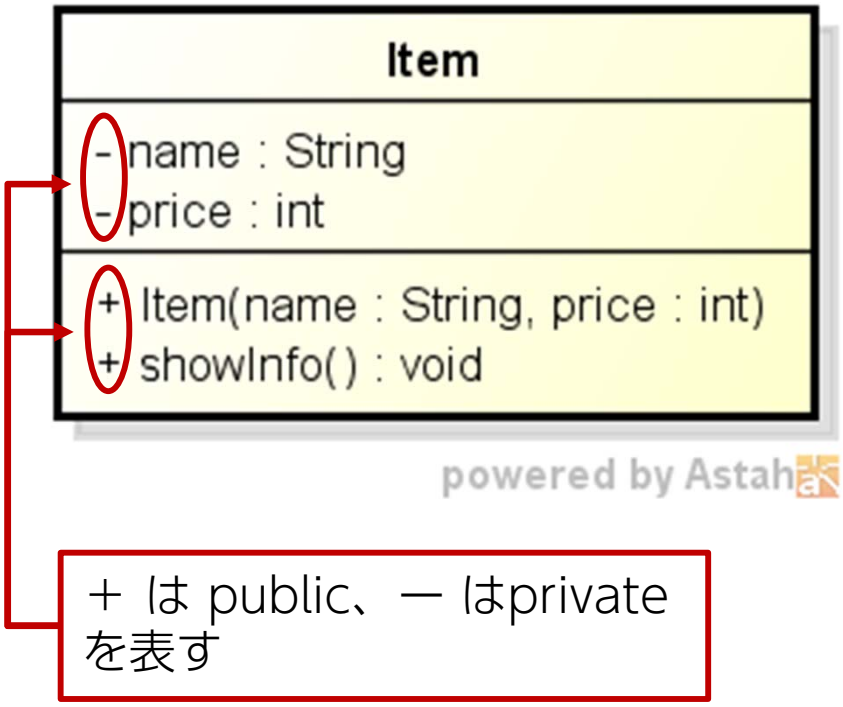
UML: クラス図

- クラスの種類と関連を表現する図
- クラスの表記方法
  - 四角を縦に3つに区切り、一番上にクラス名、中央にフィールド、一番下にメソッドを記述する
    - UMLではフィールドを**属性**、メソッドを**操作**と呼ぶ
  - フィールドやメソッドは省略可能(下図右)



UML: クラス図

- 例：Itemクラス

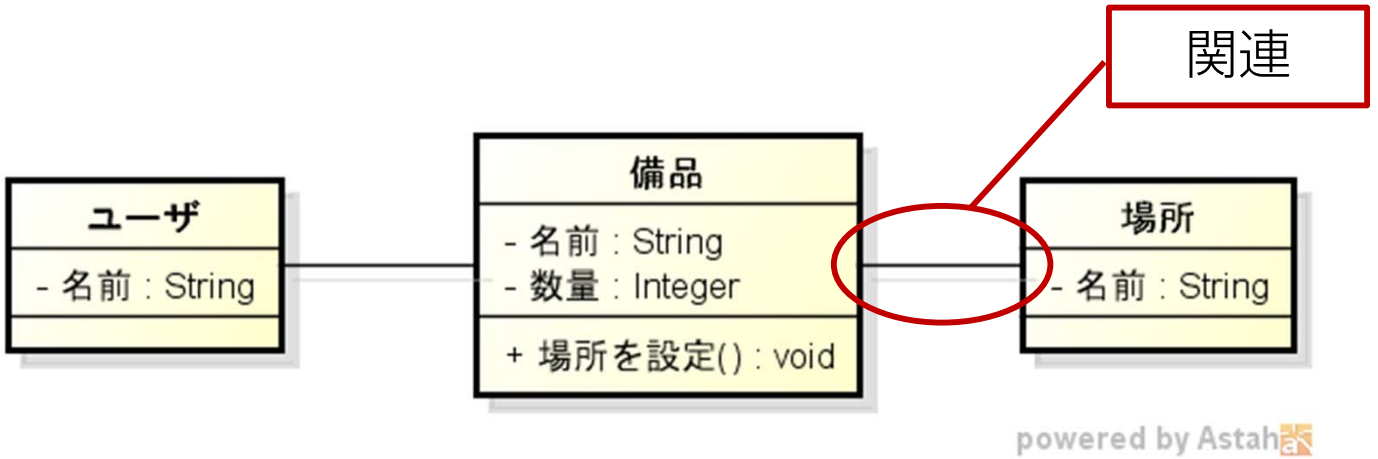



```
public class Item {
  private String name;
  private int price;

  public Item(String name, int price) {
    this.name = name;
    this.price = price;
  }
  public void showInfo() {
    ...
  }
}
```

UML: クラス図

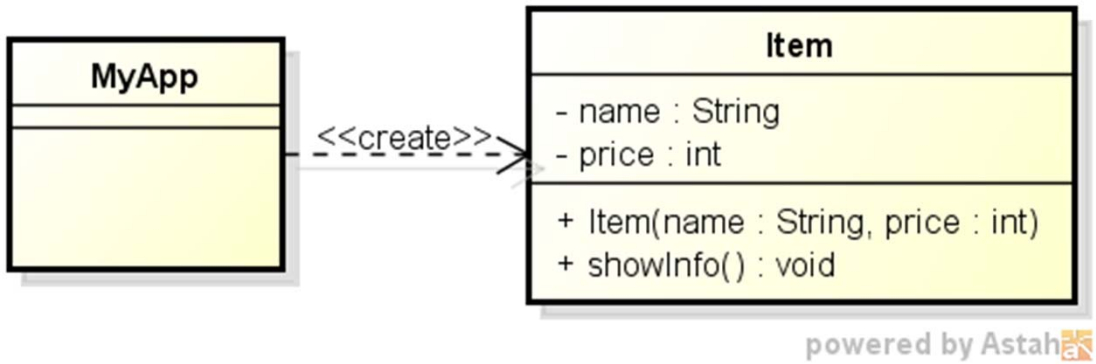
- クラス間を線で結ぶことで関連を示す
  - 関連の種類は様々あり、関連の種類によって線の種類も異なる(関連の種類を明示しない場合は単純な直線を使用する)



UML: クラス図

- 例：MyAppクラスとItemクラスの関連

```
public class MyApp {
  public static void main(String[] args) {
    Item item1 = new Item("りんご", 100);
    item1.showInfo();
  }
}
```



左の図では MyApp クラスが Item クラスのインスタンスを生成するということを表している