

INTERNET ACADEMY

Institute of Web Design & Software Services

C#3

インターネット・アカデミー

C# 目次

1. 【復習】 クラスとインスタンス
2. メソッドの利用
3. コンストラクター
4. アクセス修飾子とカプセル化
5. プロパティ
6. 【参考】 静的クラス
7. 演習

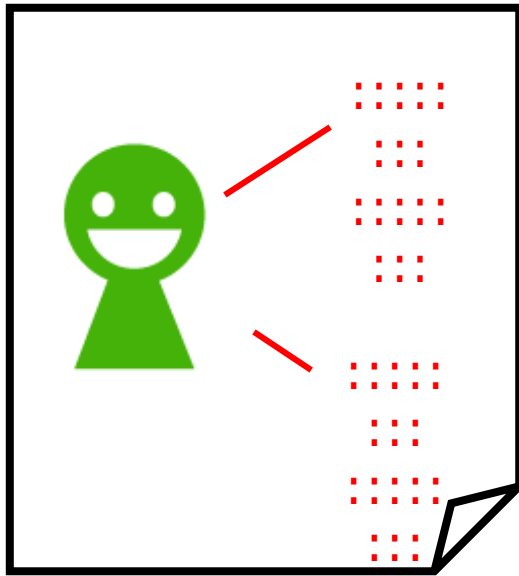


1. 【復習】クラスとインスタンス

【復習】クラスとインスタンス

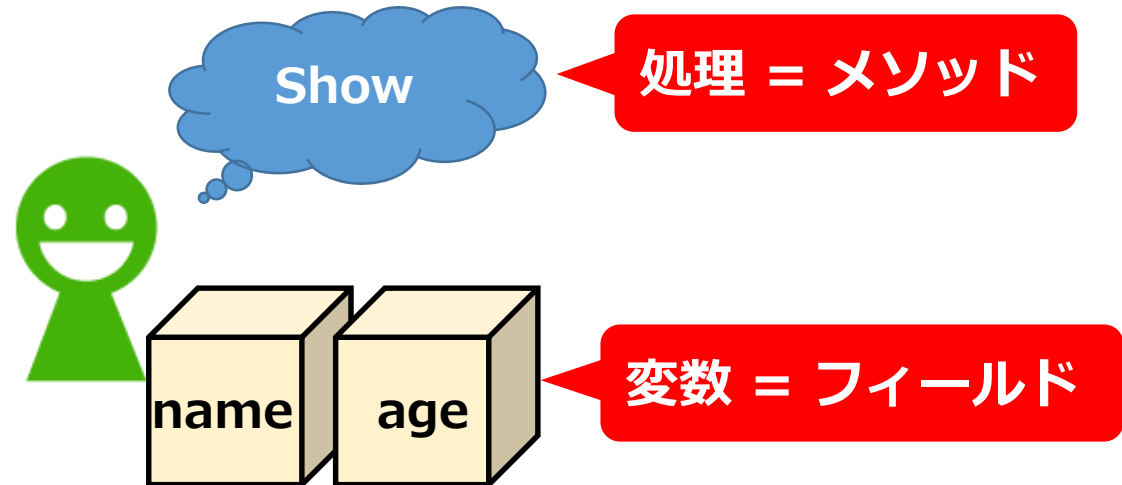
クラスとは独自のオブジェクト（インスタンス）を作るための設計図のようなもの

設計図



クラス

実体をもつモノ



オブジェクト
(インスタンス)

【復習】クラスとインスタンス

▼作成するクラス（Personクラス）

```
public class Person
{
    // 名前フィールド
    public string name;
    // 年齢フィールド
    public int age;
}
```

人物の情報を表すクラス。フィールドとして名前（name）と年齢（age）を持つ。

【復習】クラスとインスタンス

▼作成するクラス（Personクラス）

```
public class Person
```

クラス名

```
{
```

```
// 名前フィールド  
public string name;  
// 年齢フィールド  
public int age;
```

フィールド

```
}
```

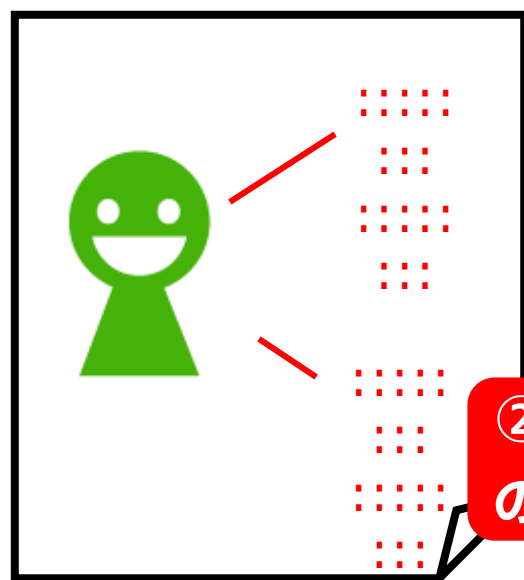
人物の情報を表すクラス。フィールドとして名前（name）と年齢（age）を持つ。

【復習】クラスとインスタンス

クラスとは独自のオブジェクト（インスタンス）を作るための設計図のようなもの

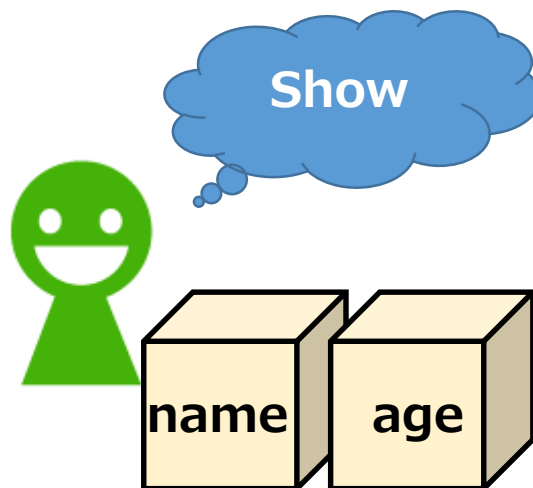
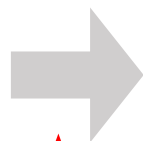
設計図

実体をもつモノ



②インスタンスの生成

①クラスの定義



③インスタンスの操作

オブジェクト
(インスタンス)

【復習】クラスとインスタンス

▼ Personクラスのインスタンスの生成とインスタンスの操作

```
Person p = new Person();
```

インスタンスの生成

```
p.name = "太郎";
```

```
p.age = 18;
```

```
Console.WriteLine("名前:{0} 年齢:{1}", p.name, p.age);
```

▼ 実行結果

名前:太郎 年齢:18

【復習】クラスとインスタンス

▼ Personクラスのインスタンスの生成とインスタンスの操作

```
Person p = new Person();  
p.name = "太郎";  
p.age = 18;  
Console.WriteLine("名前:{0} 年齢:{1}", p.name, p.age);
```

フィールドの操作(値の代入)


▼ 実行結果

名前:太郎 年齢:18

【復習】クラスとインスタンス

▼ Personクラスのインスタンスの生成とインスタンスの操作

```
Person p = new Person();  
p.name = "太郎";  
p.age = 18;  
Console.WriteLine("名前:{0} 年齢:{1}", p.name, p.age);
```

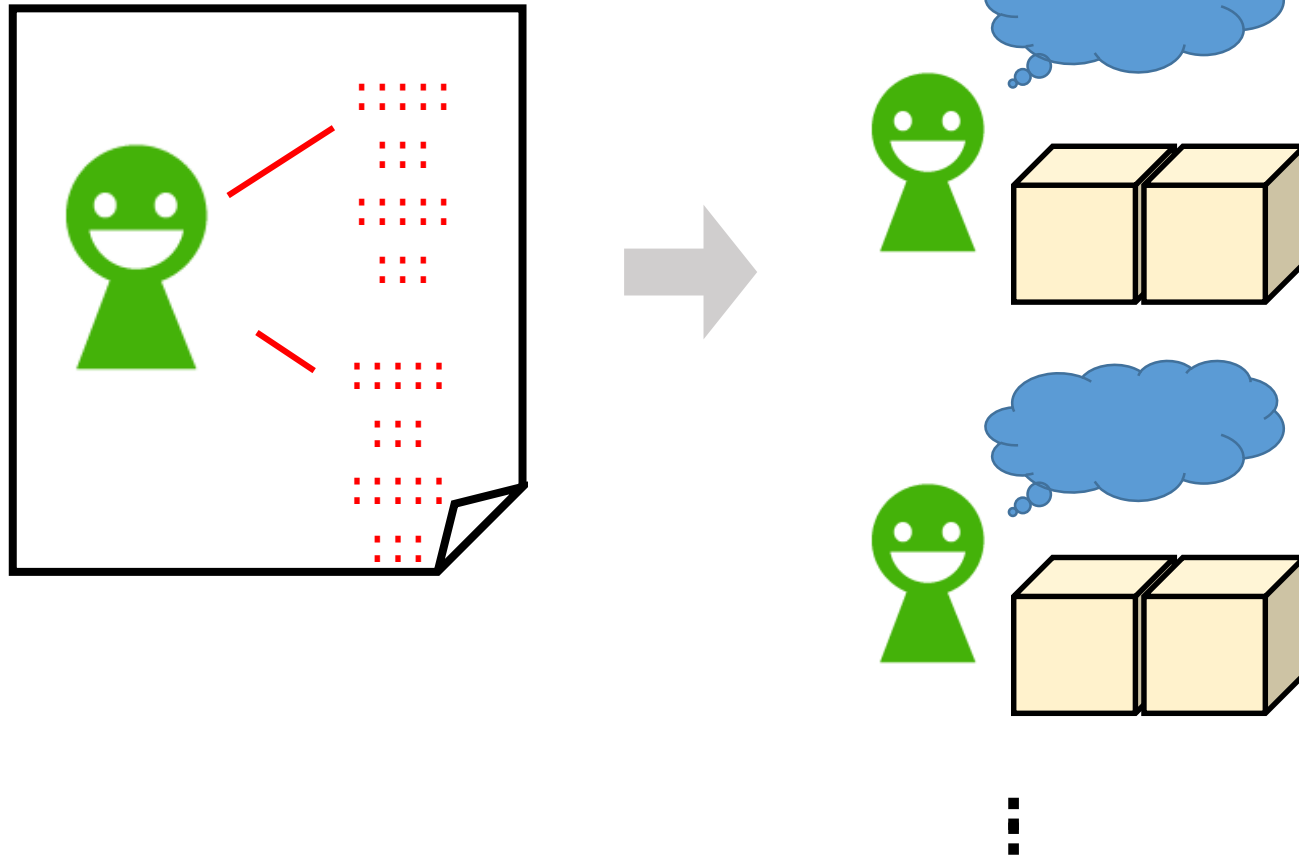


▼ 実行結果

名前:太郎 年齢:18

【復習】クラスとインスタンス

クラスとは独自のオブジェクト（インスタンス）を作るための設計図のようなもの



- ・ 同じ性質を持つオブジェクトを量産することが簡単
- ・ 修正や変更をする際に設計図となるクラス1つを変更するだけで済むため便利

【復習】クラスとインスタンス

▼ Personクラスのインスタンスの生成

```
Person p1 = new Person();  
p1.name = "太郎";  
p1.age = 18;  
Console.WriteLine("名前:{0} 年齢:{1}", p1.name, p1.age);  
Person p2 = new Person();  
p2.name = "花子";  
p2.age = 16;  
Console.WriteLine("名前:{0} 年齢:{1}", p2.name, p2.age);
```

▼ 実行結果

```
名前:太郎 年齢18  
名前:花子 年齢16
```

【復習】クラスとインスタンス

▼ Personクラスのインスタンスの生成

```
Person p1 = new Person();  
p1.name = "太郎";  
p1.age = 18;  
Console.WriteLine("名前:{0} 年齢:{1}", p1.name, p1.age);  
Person p2 = new Person();  
p2.name = "花子";  
p2.age = 16;  
Console.WriteLine("名前:{0} 年齢:{1}", p2.name, p2.age);
```

▼ 実行結果

```
名前:太郎 年齢18  
名前:花子 年齢16
```

何度も同じ処理を記述するのは効率が悪い



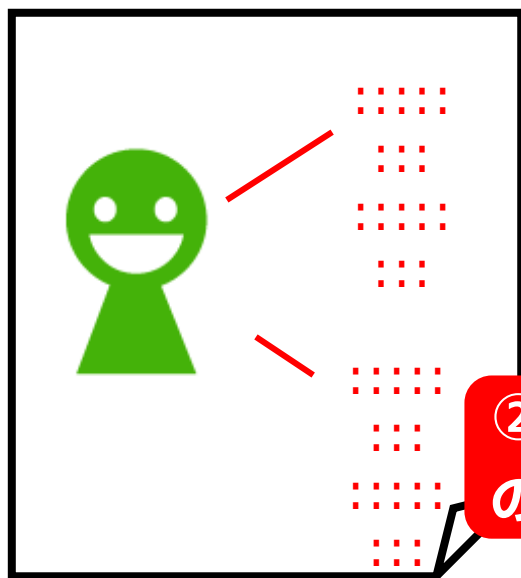
2.メソッドの利用

メソッドの利用

インスタンスを操作する方法としてメソッド (method) が存在する

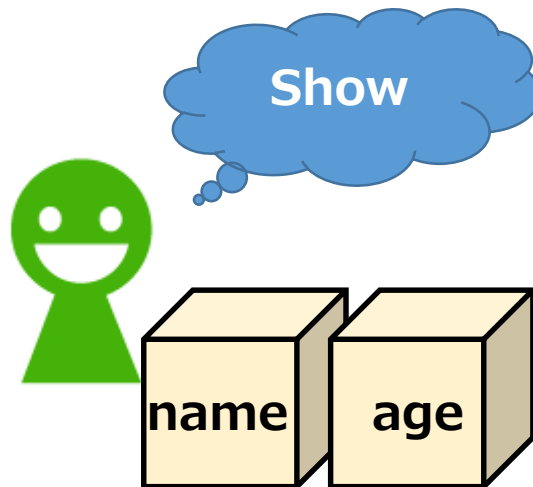
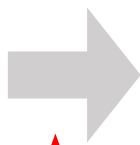
設計図

実体をもつモノ



②インスタンスの生成

①クラスの定義



③メソッドの利用

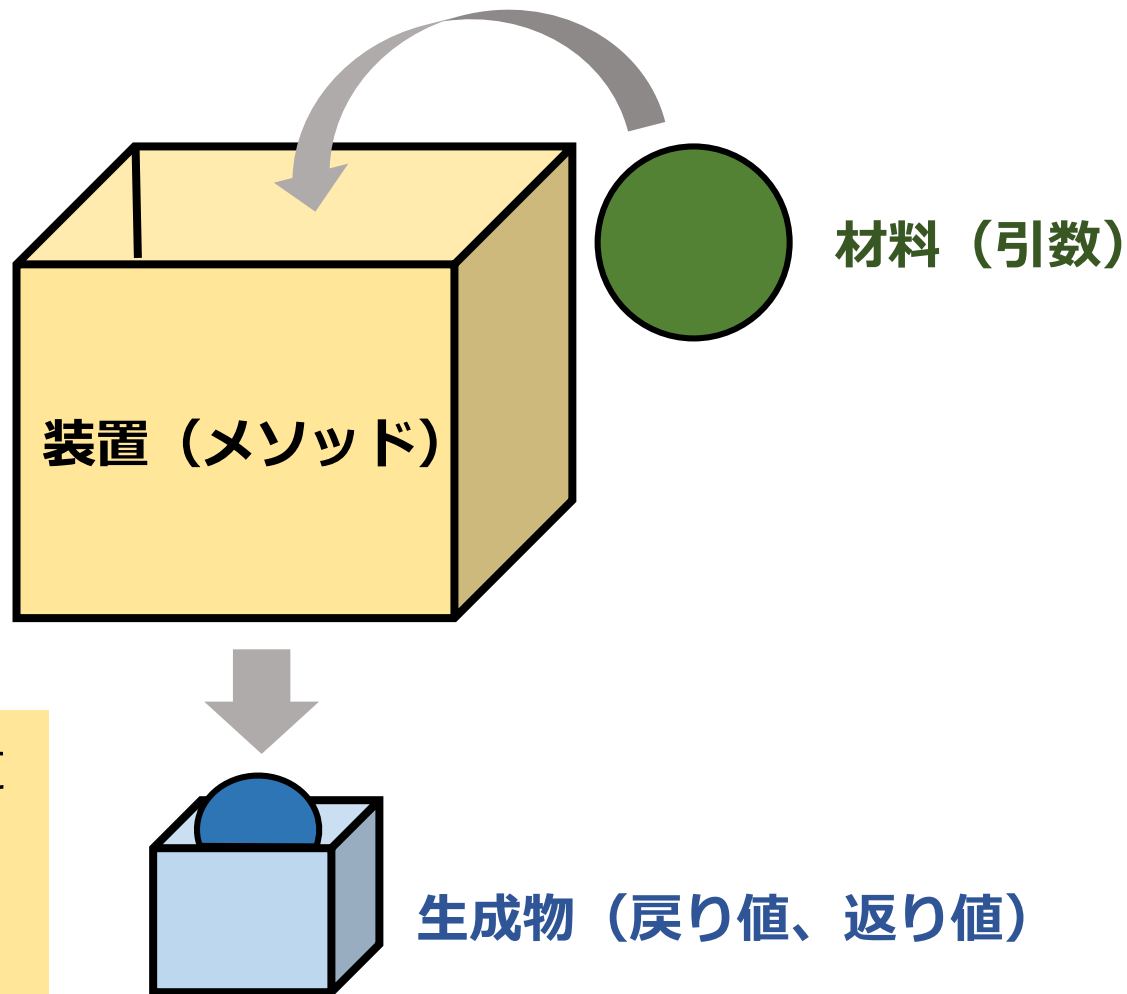
オブジェクト
(インスタンス)

メソッドとは

- ・ **メソッドの概念を把握する**
- ・ **記述方法を理解する**

メソッドとは

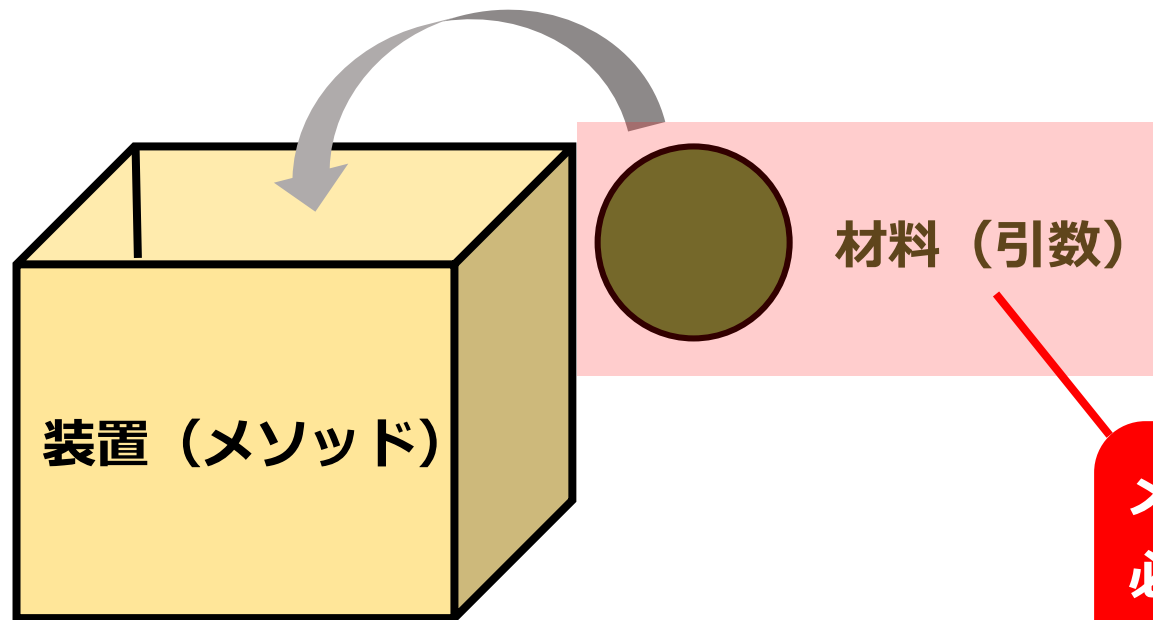
ある処理のまとまりに名前を付け、その実行命令が来たら実行する仕組み



メソッドはクラスに
いくつ追加しても
かまわない

メソッドとは

ある処理のまとまりに名前を付け、その実行命令が来たら実行する仕組み



メソッドを作るときには、必ずしも引数や戻り値が必要というわけではない

メソッドはクラスにいくつ追加してもかまわない

生成物 (戻り値、返り値)

メソッドの記述方法のパターン

- ・ 引数なし、戻り値なし
- ・ 引数なし、戻り値あり
- ・ 引数あり、戻り値なし
- ・ 引数あり、戻り値あり

メソッドの記述方法のパターン

- ・ 引数なし、戻り値なし
- ・ 引数なし、戻り値あり
- ・ 引数あり、戻り値なし
- ・ 引数あり、戻り値あり

一番簡単なのでまずは
このパターンから学習を
はじめます。

引数なし・戻り値なしのメソッドの書式

▼引数なし・戻り値なしのメソッドの書式

```
void メソッド名(){  
    処理  
    return;  
}
```

- ・「void」はメソッドに戻り値が存在しないことを意味する
- ・()の中にはなにも記述しない（引数が存在しないことを意味する）
- ・「return」で処理を終了する（最後のreturnは省略することも多い）

Personクラスにメソッドを追加

▼作成するクラス (Personクラス)

```
public class Person
{
    // 名前フィールド
    public string name;
    // 年齢フィールド
    public int age;
    // 情報を表示するメソッド
    public void Show()
    {
        Console.WriteLine("名前:{0} 年齢{1}", this.name, this.age);
    }
}
```

Personクラスに名前と年齢を表示するShowメソッドを追加。

Personクラスにメソッド（引数なし、戻り値なし）を追加

▼作成するクラス（Personクラス）

```
public class Person
{
    // 名前フィールド
    public string name;
    // 年齢フィールド
    public int age;
    // 情報を表示するメソッド
    public void Show()
    {
        Console.WriteLine("名前:{0} 年齢{1}", this.name, this.age);
    }
}
```

追加されたメソッド

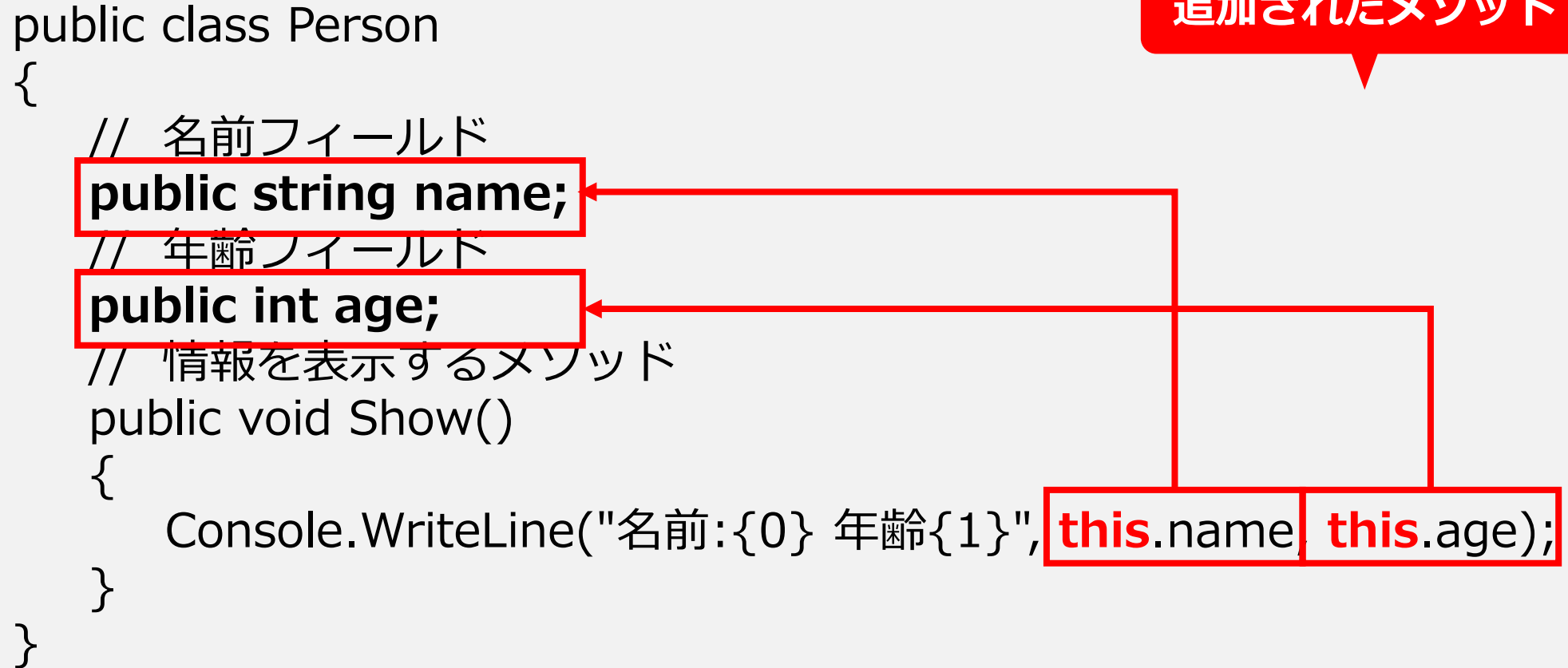
Personクラスに名前と年齢を表示するShowメソッドを追加。

Personクラスにメソッド（引数なし、戻り値なし）を追加

▼作成するクラス（Personクラス）

```
public class Person
{
    // 名前フィールド
    public string name;
    // 年齢フィールド
    public int age;
    // 情報を表示するメソッド
    public void Show()
    {
        Console.WriteLine("名前:{0} 年齢{1}", this.name, this.age);
    }
}
```

追加されたメソッド



「this」は自分自身のインスタンスを表す

メソッドの利用

▼ Personクラスのインスタンスの生成とメソッドの実行

```
Person p = new Person();  
p.name = "太郎";  
p.age = 18;  
p.Show();
```

▼ 実行結果

名前:太郎 年齢:18

メソッドの利用

▼ Personクラスのインスタンスの生成とメソッドの実行

```
Person p = new Person();  
p.name = "太郎";  
p.age = 18;  
p.Show();
```

メソッドの実行

▼ 実行結果

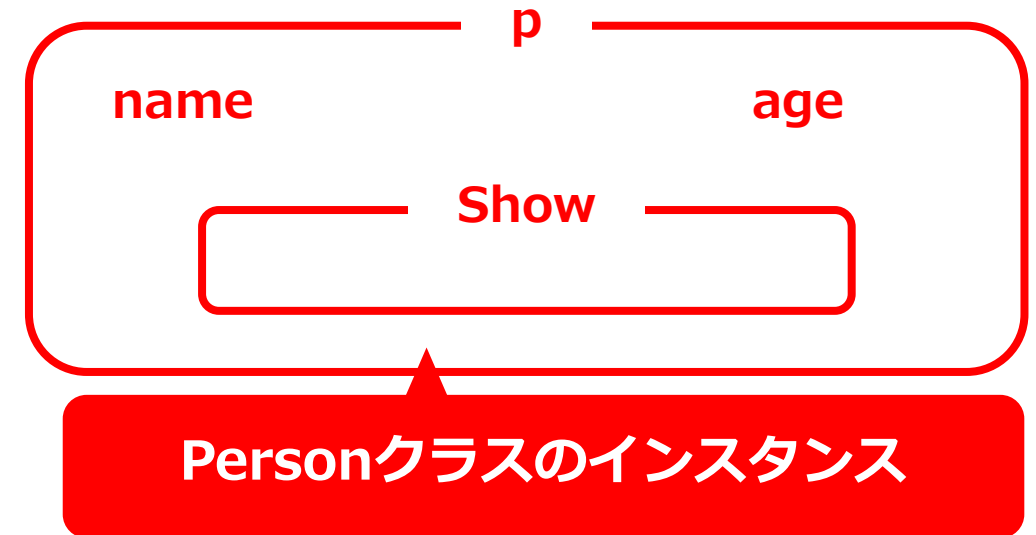
名前:太郎 年齢:18

インスタンス生成からメソッドの呼び出しまで

①インスタンスの生成

▼ メインの処理

```
Person p = new Person();  
p.name = "太郎";  
p.age = 18;  
p.Show();
```

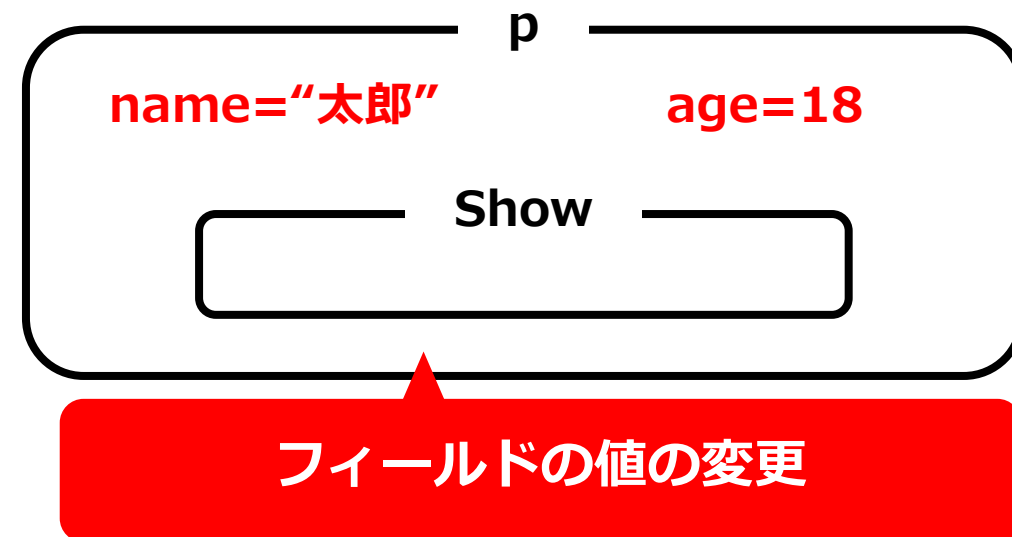


インスタンス生成からメソッドの呼び出しまで

②フィールドへの値の代入

▼ メインの処理

```
Person p = new Person();  
p.name = "太郎";  
p.age = 18;  
p.Show();
```



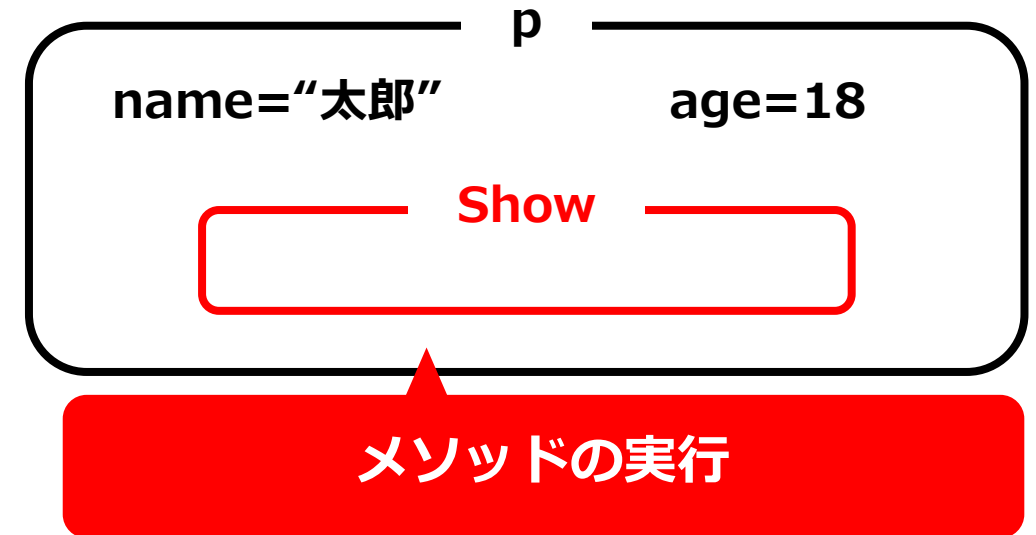
インスタンス生成からメソッドの呼び出しまで

③メソッドの呼び出し

▼ メインの処理

```
Person p = new Person();  
p.name = "太郎";  
p.age = 18;  
p.Show();
```

```
public void Show()  
{  
    Console.WriteLine("名前:{0} 年齢{1}", this.name, this.age);  
}
```



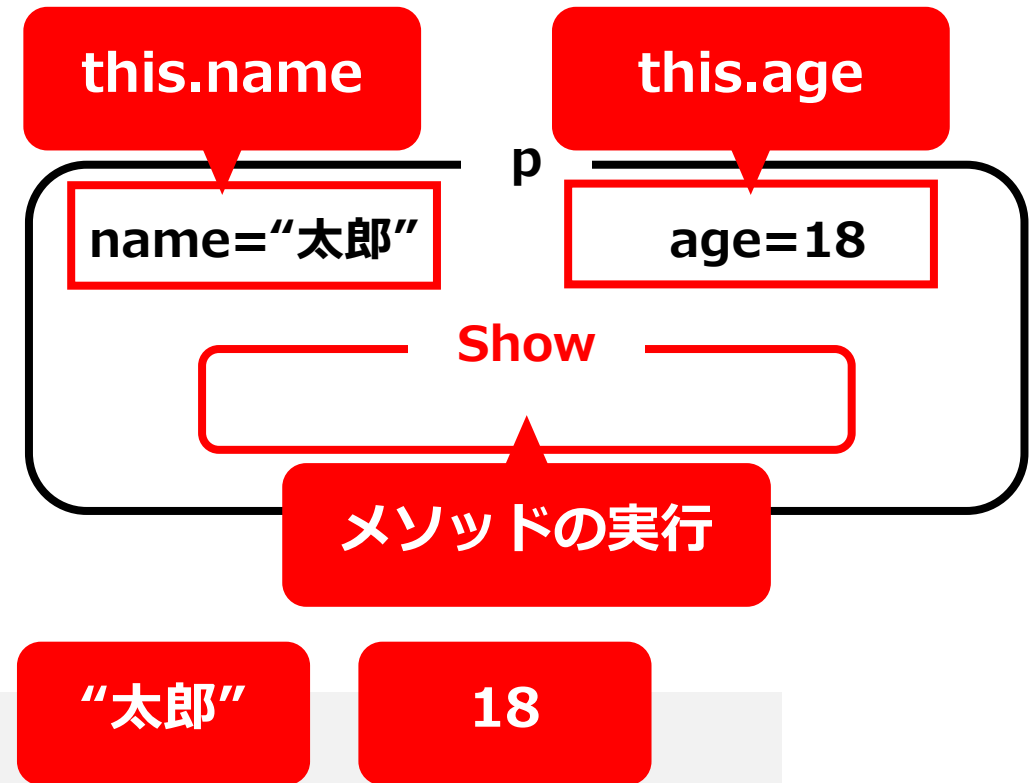
インスタンス生成からメソッドの呼び出しまで

③メソッドの呼び出し

▼ メインの処理

```
Person p = new Person();  
p.name = "太郎";  
p.age = 18;  
p.Show();
```

```
public void Show()  
{  
    Console.WriteLine("名前:{0} 年齢{1}", this.name, this.age);  
}
```



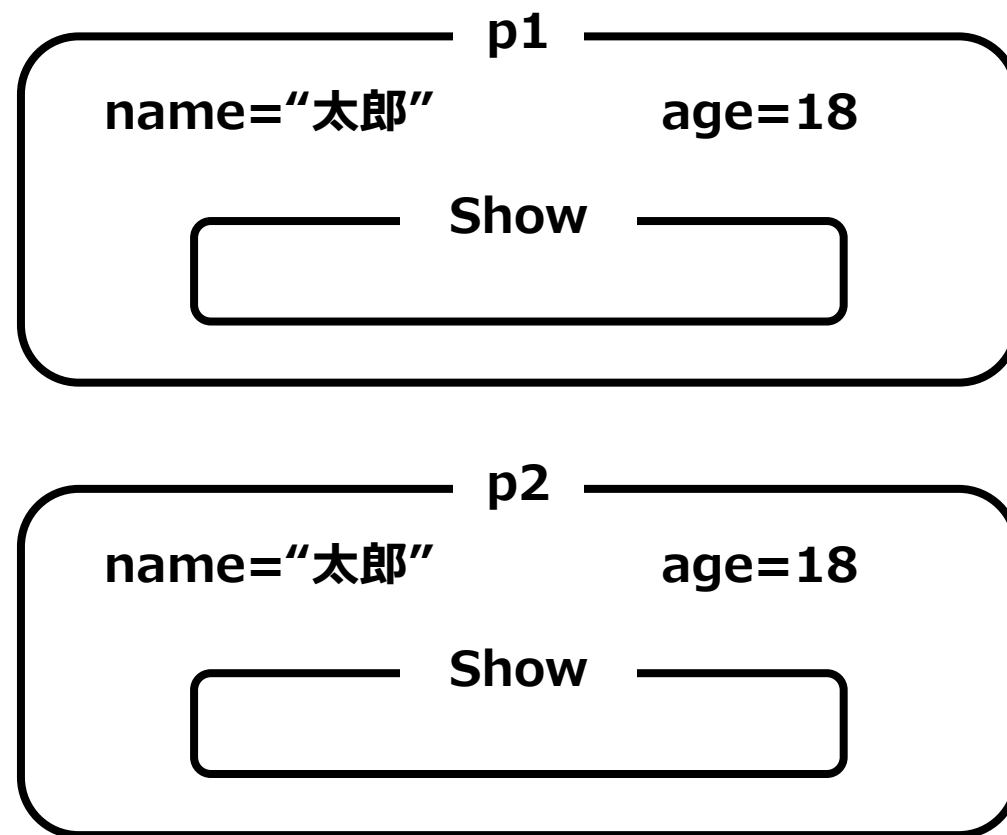
複数のインスタンスの場合のメソッド呼び出し

▼ Personクラスのインスタンスの生成

```
Person p1 = new Person();  
Person p2 = new Person();  
p1.name = "太郎";  
p1.age = 18;  
p1.Show();  
p2.name = "花子";  
p2.age = 16;  
p2.Show();
```

▼ 実行結果

名前:太郎 年齢18
名前:花子 年齢16



複数のインスタンスの場合のメソッド呼び出し

▼ Personクラスのインスタンスの生成

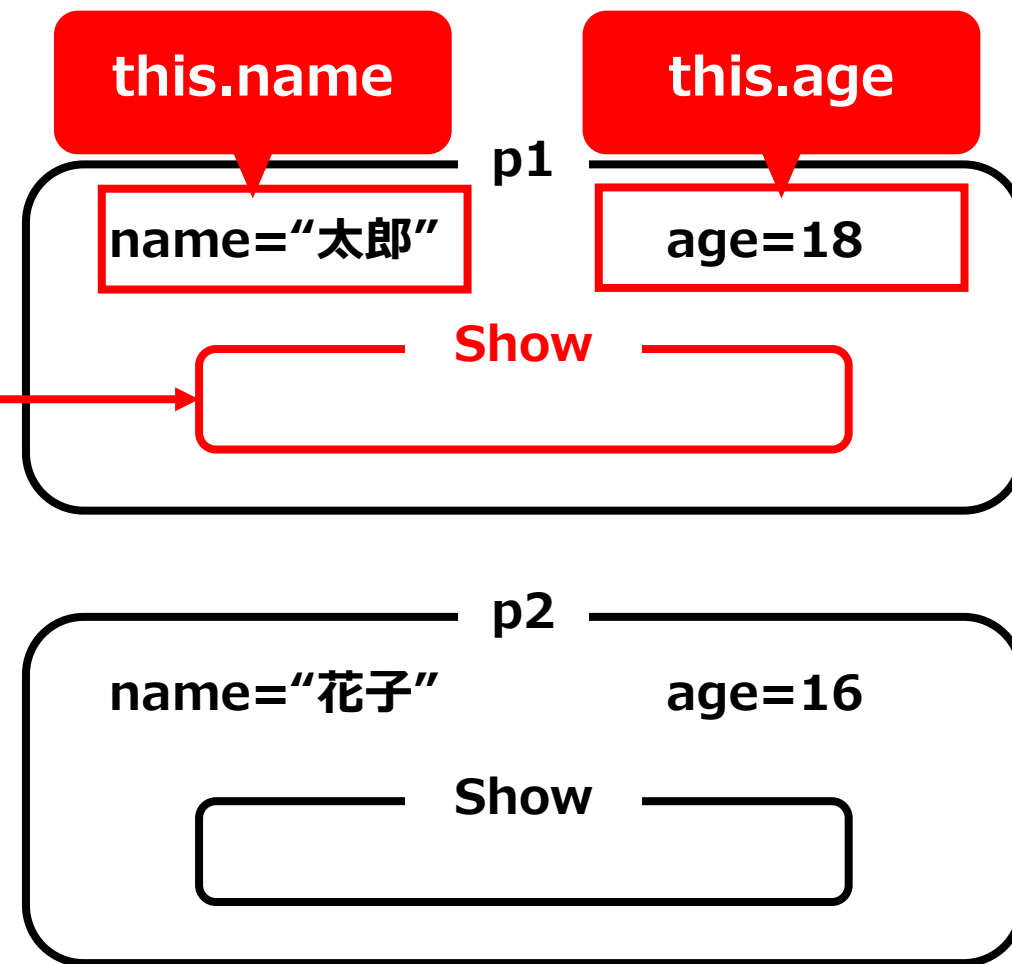
```
Person p1 = new Person();  
Person p2 = new Person();  
p1.name = "太郎";  
p1.age = 18;  
p1.Show();  
p2.name = "花子";  
p2.age = 16;  
p2.Show();
```

メソッドの実行

▼ 実行結果

名前:太郎 年齢18
名前:花子 年齢16

メソッドの実行



複数のインスタンスの場合のメソッド呼び出し

▼ Personクラスのインスタンスの生成

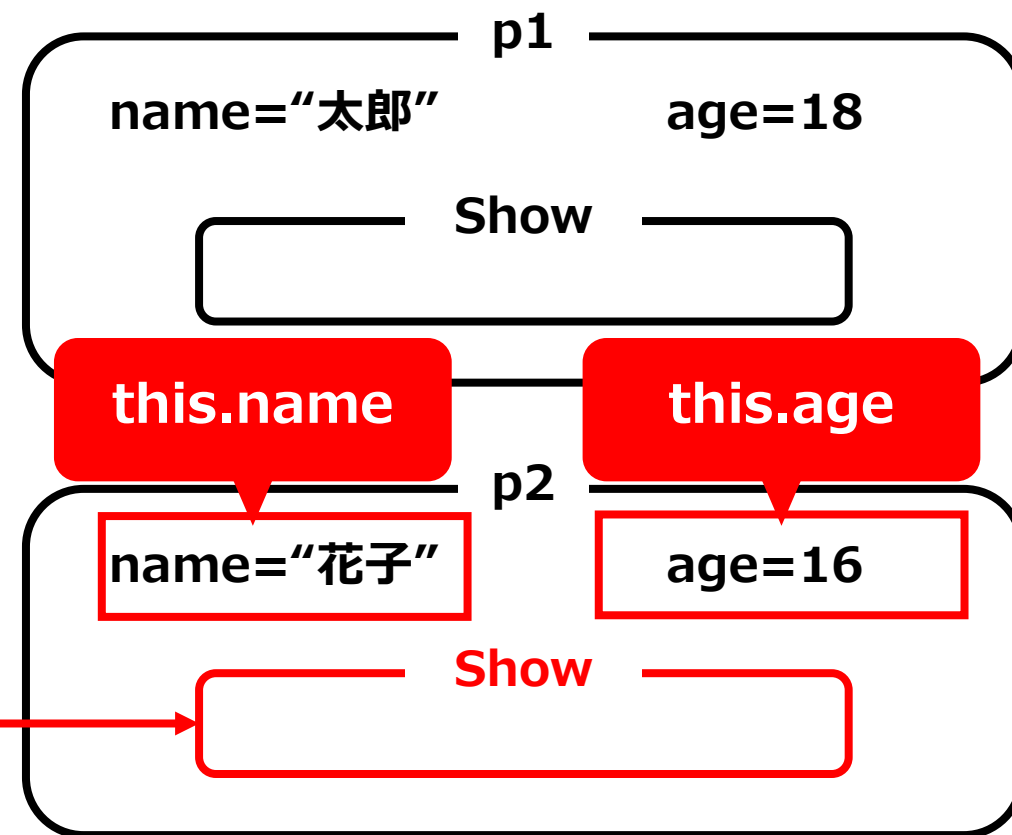
```
Person p1 = new Person();  
Person p2 = new Person();  
p1.name = "太郎";  
p1.age = 18;  
p1.Show();  
p2.name = "花子";  
p2.age = 16;  
p2.Show();
```

メソッドの実行

▼ 実行結果

```
名前:太郎 年齢18  
名前:花子 年齢16
```

メソッドの実行



メソッドの記述方法のパターン

- ・ 引数なし、戻り値なし
- ・ 引数なし、戻り値あり
- ・ 引数あり、戻り値なし
- ・ 引数あり、戻り値あり

引数なし・戻り値ありのメソッドの書式

▼引数なし・戻り値ありのメソッドの書式

```
戻り値の型 メソッド名(){  
    処理  
    return 戻り値;  
}
```

- ・「return」の後に戻り値を記述すると戻り値を返し処理を終了する
- ・戻り値がある場合「return」は省略できない
- ・戻り値の型と戻り値は一致させる（戻り値がintならintと記述するなど）

メソッドの呼び出し（引数なし・戻り値あり）

▼Productクラス(商品クラス)

```
public class Product
{
    public int price;        // 価格
    public int tax_rate;    // 税率
    // 税込み価格の取得
    public int CalcPrice()
    {
        return price + price * tax_rate / 100;
    }
}
```

価格（price）と税率（tax_rate）を設定してCalcPriceで税込み価格を取得する

メソッドの呼び出し（引数なし・戻り値あり）

▼Productクラス(商品クラス)

```
public class Product
{
    public int price;        // 価格
    public int tax_rate;    // 税率
    // 税込み価格の取得
    public int CalcPrice()
    {
        return price + price * tax_rate / 100;
    }
}
```

戻り値の型

戻り値を返す

価格（price）と税率（tax_rate）を設定してCalcPriceで税込み価格を取得する

メソッドの呼び出し（引数なし・戻り値あり）

▼Productクラス

```
Product p = new Product();  
p.price = 1000;    // 価格1000円  
p.tax_rate = 10;   // 税率10%  
Console.WriteLine("税込み価格:{0}円", p.CalcPrice());
```

▼実行結果

税込み価格:1100円

メソッドの呼び出し（引数なし・戻り値あり）

▼Productクラス

```
Product p = new Product();  
p.price = 1000;    // 価格1000円  
p.tax_rate = 10;   // 税率10%  
Console.WriteLine("税込み価格:{0}円", p.CalcPrice());
```

戻り値（計算結果）

▼実行結果

税込み価格:**1100**円

メソッドの記述方法のパターン

- ・ 引数なし、戻り値なし
- ・ 引数なし、戻り値あり
- ・ 引数あり、戻り値なし
- ・ 引数あり、戻り値あり

引数あり・戻り値なしのメソッドの書式

▼引数あり・戻り値なしのメソッドの書式

```
void メソッド名(引数1の型 引数1の変数,引数2の型 引数2の変数,...){  
    処理  
    return;  
}
```

- ・ 引数の型とその変数（例：int a）を記述する
- ・ 引数の数が複数ある場合には「,」で区切る
- ・ 呼び出した際に戻り値が与えられると、引数の変数に代入される

メソッドの呼び出し（引数あり・戻り値なし）

▼EMoneyクラス(電子マネークラス)

```
public class EMoney
{
    public int money = 0; // 金額
    public void Charge(int money)
    {
        this.money += money; // お金をチャージ
        Console.WriteLine("残高:{0}円",this.money);
    }
    public void Spend(string product,int price)
    {
        this.money -= price;
        Console.WriteLine("商品:{0} 価格{1}円",product,price);
        Console.WriteLine("残高:{0}円",this.money);
    }
}
```

電子マネーに
お金をチャージし、
買い物ができる

メソッドの呼び出し（引数あり・戻り値なし）

▼EMoneyクラス(電子マネークラス)

```
public class EMoney
{
    public int money = 0; // 金額
    public void Charge(int money)
    {
        this.money += money; // お金をチャージ
        Console.WriteLine("残高:{0}円",this.money);
    }
    public void Spend(string product,int price)
    {
        this.money -= price;
        Console.WriteLine("商品:{0} 価格{1}円",product,price);
        Console.WriteLine("残高:{0}円",this.money);
    }
}
```

引数が1つのメソッド

電子マネーに
お金をチャージし、
買い物ができる

メソッドの呼び出し（引数あり・戻り値なし）

▼EMoneyクラス(電子マネークラス)

```
public class EMoney
{
    public int money = 0; // 金額
    public void Charge(int money)
    {
        this.money += money; // お金をチャージ
        Console.WriteLine("残高:{0}円",this.money);
    }
    public void Spend(string product,int price)
    {
        this.money -= price;
        Console.WriteLine("商品:{0} 価格{1}円",product,price);
        Console.WriteLine("残高:{0}円",this.money);
    }
}
```

引数の変数名とフィールドの変数名が一致する場合フィールドに「this.」を付けて区別する

電子マネーに
お金をチャージし、
買い物ができる

メソッドの呼び出し（引数あり・戻り値なし）

▼EMoneyクラス(電子マネークラス)

```
public class EMoney
{
    public int money = 0; // 金額
    public void Charge(int money)
    {
        this.money += money; // お金をチャージ
        Console.WriteLine("残高:{0}円",this.money);
    }
    public void Spend(string product,int price)
    {
        this.money -= price;
        Console.WriteLine("商品:{0} 価格{1}円",product,price);
        Console.WriteLine("残高:{0}円",this.money);
    }
}
```

引数が2つのメソッド

電子マネーに
お金をチャージし、
買い物ができる

メソッドの呼び出し（引数あり・戻り値なし）

▼Productクラス

```
EMoney m = new EMoney();  
m.Charge(5000);           // 5000円チャージ  
m.Spend("ラーメン",1000); // ラーメン購入
```

▼実行結果

残高:5000円
商品:ラーメン 価格1000円
残高:4000円

メソッドの呼び出し（引数あり・戻り値なし）

▼Productクラス

```
EMoney m = new EMoney();  
m.Charge(5000); // 5000円チャージ  
m.Spend("ラーメン",1000); // ラーメン購入
```

▼実行結果

残高:5000円

残高をチャージ

商品:ラーメン 価格1000円

残高:4000円

Chargeメソッドの引数moneyに5000が代入される

メソッドの呼び出し（引数あり・戻り値なし）

▼Productクラス

```
EMoney m = new EMoney();  
m.Charge(5000);           // 5000円チャージ  
m.Spend("ラーメン",1000); // ラーメン購入
```

▼実行結果

残高:5000円

商品:ラーメン 価格1000円

残高:4000円

支払いと残高表示

Spendメソッドのproductに「ラーメン」、priceに「1000」が代入される

メソッドの記述方法のパターン

- ・ 引数なし、戻り値なし
- ・ 引数なし、戻り値あり
- ・ 引数あり、戻り値なし
- ・ 引数あり、戻り値あり

引数あり・戻り値ありのメソッドの書式

▼引数あり・戻り値なしのメソッドの書式

```
戻り値の型 メソッド名(引数1の型 引数1の変数,...){  
    処理  
    return 戻り値;  
}
```

- ・ 引数の型と値の指定を複数指定可能（複数の場合「,」で区切る
- ・ 戻り値の型と戻り値の値の型は一致させる

メソッドの呼び出し（引数あり・戻り値あり）

▼Buyクラス(購入クラス)

```
public class Buy
{
    public int unit_price;    // 単価

    public int Price(int number,int tax_rate)
    {
        int price = unit_price * number; // 商品の総額
        price += (price * tax_rate)/100; // 消費税を追加
        return price; // 税込みの総額
    }
}
```

商品の単価を設定しておく、数と税率を設定することにより税込みの総額を求められる

メソッドの呼び出し（引数あり・戻り値あり）

▼Buyクラス(購入クラス)

```
public class Buy
{
    public int unit_price;    // 単価

    public int Price(int number,int tax_rate)
    {
        int price = unit_price * number;    // 商品の総額
        price += (price * tax_rate)/100;    // 消費税を追加
        return price;    // 税込みの総額
    }
}
```

引数が2つ

戻り値がint

商品の単価を設定しておく、数と税率を設定することにより税込みの総額を求められる

メソッドの呼び出し（引数あり・戻り値あり）

▼Productクラス

```
Buy product = new Buy();  
product.unit_price = 100;           // 商品単価100円  
int price = product.Price(50, 10);  // 50個を税率10%で購入  
Console.WriteLine("購入価格:{0}円", price); // 総額表示
```

▼実行結果

購入価格:5500円

メソッドの呼び出し（引数あり・戻り値あり）

▼Productクラス

```
Buy product = new Buy();  
product.unit_price = 100; // 商品単価100円  
int price = product.Price(50, 10); // 50個を税率10%で購入  
Console.WriteLine("購入価格:{0}円", price); // 総額表示
```

メソッド呼び出し

戻り値を代入

▼実行結果

購入価格:5500円



3.コンストラクター

コンストラクターとは

- ・ インスタンスを生成するときに一度だけ実行される特殊なメソッド
- ・ クラス名と同じ名前となる
- ・ 引数を指定できるが戻り値は存在しない

コンストラクターとは

▼ Personクラスにコンストラクターを追加（引数なし）

```
public class Person
{
    public string name;    // 名前フィールド
    public int age;    // 年齢フィールド
    public Person()
    {
        Console.WriteLine("Personクラスのインスタンスの生成");
    }
    public void Show()
    {
        Console.WriteLine("名前:{0} 年齢{1}", this.name, this.age);
    }
}
```

コンストラクターとは

▼ Personクラスにコンストラクターを追加（引数なし）

```
public class Person
{
    public string name;    // 名前フィールド
    public int age;        // 年齢フィールド
    public Person()
    {
        Console.WriteLine("Personクラスのインスタンスの生成");
    }
    public void Show()
    {
        Console.WriteLine("名前:{0} 年齢{1}", this.name, this.age);
    }
}
```

コンストラクター

コンストラクターとは

▼コンストラクターの呼び出し

```
Person p = new Person();  
p.name = "太郎";  
p.age = 18;  
p.Show();
```

▼実行結果

Personクラスのインスタンスの生成
名前:太郎 年齢:18

コンストラクターとは

▼コンストラクターの呼び出し

```
Person p = new Person();
```

インスタンスの生成

```
p.name = "太郎";  
p.age = 18;  
p.Show();
```

▼実行結果

Personクラスのインスタンスの生成

コンストラクターの実行

名前:太郎 年齢:18

引数のついたコンストラクタ

▼ Personクラスにコンストラクターを追加（引数あり）

```
public class Person
{
    public string name;
    public int age;
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
    public void Show()
    {
        Console.WriteLine("名前:{0} 年齢:{1}", this.name, this.age);
    }
}
```

引数のついたコンストラクタ

▼ Personクラスにコンストラクターを追加（引数あり）

```
public class Person
{
    public string name;
    public int age;
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
    public void Show()
    {
        Console.WriteLine("名前:{0} 年齢:{1}", this.name, this.age);
    }
}
```

引数の値をフィールドに代入

コンストラクターとは

▼コンストラクターの呼び出し

```
Person p = new Person("太郎", 18);  
p.Show();
```

▼実行結果

名前:太郎 年齢:18

コンストラクターでフィールドを設定しているのでフィールドの設定を省略できる。



4. アクセス修飾子とカプセル化

アクセス修飾子とは？

- ・ メンバのアクセスおよびクラスのアクセスレベルを決める修飾子
- ・ メンバの前につけてアクセス修飾子を決定する
- ・ public, protected, private等といった種類がある
- ・ 省略することも可能

主なアクセス修飾子の種類と意味

- **public** ... 無制限のアクセスが可能（クラスの先頭につけることも可能）
- **protected** ... クラス内、もしくは派生クラス内でアクセス可能
- **private** ... クラス内からのみアクセス可能
- メンバのアクセス修飾子を省略した場合には、privateと同じ扱いになる

主なアクセス修飾子の種類と意味

- ・ **public** ... 無制限のアクセスが可能（クラスの先頭につけることも可能）
- ・ **protected** ... クラス内、もしくは派生クラス内でアクセス可能
- ・ **private** ... クラス内からのみアクセス可能
- ・ メンバのアクセス修飾子を省略した場合には、privateと同じ扱いになる

！ ポイント

protectedについてはC#No5の継承の時に学習します。

public修飾子の問題点

▼Personクラス

```
public class Person
{
    public string name;
    public int age;
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
    public void Show()
    {
        Console.WriteLine("名前:{0} 年齢:{1}", this.name, this.age);
    }
}
```

public修飾子の問題点

▼コンストラクターの呼び出し

```
Person p = new Person("太郎", 18);  
p.name = "花子";  
p.age = 16;  
p.Show();
```

▼実行結果

名前:花子 年齢:16

publicなフィールドは誰でも簡単に変更できてしまう

public修飾子の問題点

▼コンストラクターの呼び出し

```
Person p = new Person("太郎", 18);  
p.name = "花子";  
p.age = 16;  
p.Show();
```

外部からフィールドの値が変更できてしまう

▼実行結果

名前:花子 年齢:16

publicなフィールドは誰でも簡単に変更できてしまう

private修飾子

▼Personクラス

```
public class Person
{
    private string name;
    private int age;
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
    public void Show()
    {
        Console.WriteLine("名前:{0} 年齢:{1}", this.name, this.age);
    }
}
```

private修飾子

▼Personクラス

```
public class Person
{
    private string name;
    private int age;
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
    public void Show()
    {
        Console.WriteLine("名前:{0} 年齢{1}", this.name, this.age);
    }
}
```

クラスの外部からのアクセスを禁止する

private修飾子

▼コンストラクターの呼び出し

```
Person p = new Person("太郎", 18);  
//p.name = "花子";  
//p.age = 16;  
p.Show();
```

▼実行結果

名前:太郎 年齢:花子

privateなメンバは外部からのアクセスができなくなる

private修飾子

▼コンストラクターの呼び出し

```
Person p = new Person("太郎", 18);  
//p.name = "花子";  
//p.age = 16;  
p.Show();
```

コメントを外すとエラーになる（変更不可能）

▼実行結果

名前:太郎 年齢:花子

privateなメンバは外部からのアクセスができなくなる

カプセル化

カプセル化とは？

- ・フィールドは原則的に外部からのアクセスを不可能にする
- ・値の取得・変更をするフィールドには**アクセスメソッド**をつける
- ・フィールドに値を設定するメソッドを**セッター**という
- ・フィールドの値を取得するメソッドを**ゲッター**という

カプセル化の例

▼Carクラス（カプセル化されたクラス）

```
public class Car
{
    private int fuel;
    public void SetFuel(int fuel)
    {
        this.fuel = fuel;
    }
    public int GetFuel()
    {
        return this.fuel;
    }
}
```

カプセル化の例

▼Carクラス（カプセル化されたクラス）

```
public class Car
```

```
{
```

```
    private int fuel;
```

```
    public void SetFuel(int fuel)
```

```
    {
```

```
        this.fuel = fuel;
```

```
    }
```

```
    public int GetFuel()
```

```
    {
```

```
        return this.fuel;
```

```
    }
```

```
}
```

隠蔽されたフィールド

セッター（値の設定）

ゲッター（値の取得）

カプセル化の例

▼カプセル化の例

```
Car c = new Car();  
c.SetFuel(10);  
Console.WriteLine("燃料:{0}", c.GetFuel());
```

▼実行結果

燃料:10

アクセスメソッドを使って値の設定と取得を行う

カプセル化の例

▼カプセル化の例

```
Car c = new Car();  
c.SetFuel(10);  
Console.WriteLine("燃料:{0}", c.GetFuel());
```

フィールドfuelに10を設定

▼実行結果

燃料:10

アクセスメソッドを使って値の設定と取得を行う

カプセル化の例

▼カプセル化の例

```
Car c = new Car();  
c.SetFuel(10);  
Console.WriteLine("燃料:{0}", c.GetFuel());
```

フィールドfuelの値を取得

▼実行結果

燃料:10

アクセスメソッドを使って値の設定と取得を行う

アクセスメソッドの命名方法

アクセスメソッドはフィールドの名前に由来したものを付ける

- ・フィールド : age ... セッター : SetAge() ゲッター : GetAge()
- ・フィールド : name ... セッター : SetName() ゲッター : GetName()
- ・フィールド : tax ... セッター : SetTax() ゲッター : GetTax()

演習問題 1 : 計算をするクラスの作成

以下のクラスをC#で定義してみましょう。

- ・ クラス名は「Calc」（ファイル名はCalc.cs）
- ・ 数値を表すint型のフィールドnum1とnum2を持つ(0で初期化)
- ・ num1とnum2の和を戻り値として返すメソッドAddを持つ
- ・ Addメソッドには引数がない

演習問題 1 : 計算をするクラスの作成 (正解)

▼Calcクラス

```
public class Calc{  
    public int num1 = 0; // 1つ目の数  
    public int num2 = 0; // 2つ目の数  
    // 足し算の結果を得るメソッド  
    public int Add(){  
        return this.num1 + this.num2;  
    }  
}
```

演習問題2：計算をするクラスの利用

演習問題 1 で作ったCalcクラスを利用してみましょう。

- ・ Calcクラスのインスタンスを生成する
- ・ num1に10、num2に20を代入する
- ・ Addメソッドを利用して計算結果を出力する

▼期待される実行結果

10+20=30

演習問題2：計算をするクラスの利用（解答）

▼正解

```
Calc c = new Calc();  
c.num1 = 10;  
c.num2 = 20;  
Console.WriteLine("{0}+{1}={2}",c.num1,c.num2,c.Add());
```

演習問題3：計算をするクラスへのメソッドの追加

演習問題 1 で作ったCalcクラスに以下のメソッドを追加しましょう。

- ・ **Subメソッド** ... num1とnum2の差を返す（引数なし）
- ・ **Mulメソッド** ... num1とnum2の積を返す（引数なし）

演習問題 3 : 計算をするクラスの作成 (正解)

▼Calcクラス

```
public class Calc{  
    public int num1 = 0; // 1つ目の数  
    public int num2 = 0; // 2つ目の数  
    public int Add(){  
        return this.num1 + this.num2;  
    }  
    public int Sub(){  
        return this.num1 - this.num2;  
    }  
    public int Mul(){  
        return this.num1 * this.num2;  
    }  
}
```


演習問題4：計算をするクラスの利用2

演習問題3で作ったCalcクラスを利用してみましょう。

- ・ Calcクラスのインスタンスを生成する
- ・ num1に10、num2に20を代入する
- ・ Add,Sub,Mulメソッドを利用して計算結果を出力する

▼期待される実行結果

```
10+20=30  
10-20=-10  
10*20=200
```

演習問題4：計算をするクラスの利用2（解答）

▼正解

```
Calc c = new Calc();  
c.num1 = 10;  
c.num2 = 20;  
Console.WriteLine("{0}+{1}={2}", c.num1, c.num2, c.Add());  
Console.WriteLine("{0}-{1}={2}", c.num1, c.num2, c.Sub());  
Console.WriteLine("{0}*{1}={2}", c.num1, c.num2, c.Mul());
```


プロパティとは？

- ・ C#に用意されているクラスのカプセル化を容易にする仕組み
- ・ アクセスメソッドに相当するものを簡単に作ることができる
- ・ C#ではアクセスメソッドの代わりに用いる
- ・ 読み取り専用・書き込み専用をつくることもできる

プロパティの書式

▼プロパティの書式

```
戻り値の型 プロパティ名 {  
    set{ フィールド名 = value; };  
    get{ return フィールド名; };  
}
```

- setはセッター、getはゲッターに相当する処理を記述する
- set内の変数「value」は設定された値を表す特殊な変数
- setのみ、getのみを設定してもよい

プロパティの例

▼Carクラス（アクセスメソッドをプロパティに置き換える）

```
public class Car
{
    private int fuel;
    public int Fuel
    {
        set { fuel = value; }
        get { return fuel; }
    }
}
```

プロパティの例

▼Carクラス（アクセスメソッドをプロパティに置き換える）

```
public class Car
```

```
{
```

```
    private int fuel;
```

```
    public int Fuel
```

```
{
```

```
        set { fuel = value; }
```

```
        get { return fuel; }
```

```
    }
```

```
}
```

フィールド

フィールドfuelのプロパティ

プロパティの例

▼Carクラス（アクセスメソッドをプロパティに置き換える）

```
public class Car
{
    private int fuel;
    public int Fuel
    {
        set { fuel = value; }
        get { return fuel; }
    }
}
```

フィールド名はフィールドの最初を大文字に

プロパティの例

▼Carクラス（アクセスメソッドをプロパティに置き換える）

```
public class Car
{
    private int fuel;
    public int Fuel
    {
        set { fuel = value; }
        get { return fuel; }
    }
}
```

値の設定（valueは設定された値）

set { fuel = value; }

プロパティの例

▼Carクラス（アクセスメソッドをプロパティに置き換える）

```
public class Car
{
    private int fuel;
    public int Fuel
    {
        set { fuel = value; }
        get { return fuel; }
    }
}
```

値の取得

プロパティの例

▼コンストラクターの呼び出し

```
Car c = new Car();  
c.Fuel = 10;  
Console.WriteLine("燃料:{0}", c.Fuel);
```

▼実行結果

燃料:10

プロパティFuelで値の設定・取得ができる

```
public class Car  
{  
    private int fuel;  
    public int Fuel  
    {  
        set { fuel = value; }  
        get { return fuel; }  
    }  
}
```

プロパティの例

▼コンストラクターの呼び出し

```
Car c = new Car();  
c.Fuel = 10;  
Console.WriteLine("燃料:{0}", c.Fuel);
```

▼実行結果

燃料:10

プロパティFuelで値の設定・取得ができる

```
public class Car  
{  
    private int fuel;  
    public int Fuel  
    {  
        set { fuel = value; }  
        get { return fuel; }  
    }  
}
```

fuel:10

value:10

プロパティの例

▼コンストラクターの呼び出し

```
Car c = new Car();  
c.Fuel = 10;  
Console.WriteLine("燃料:{0}", c.Fuel);
```

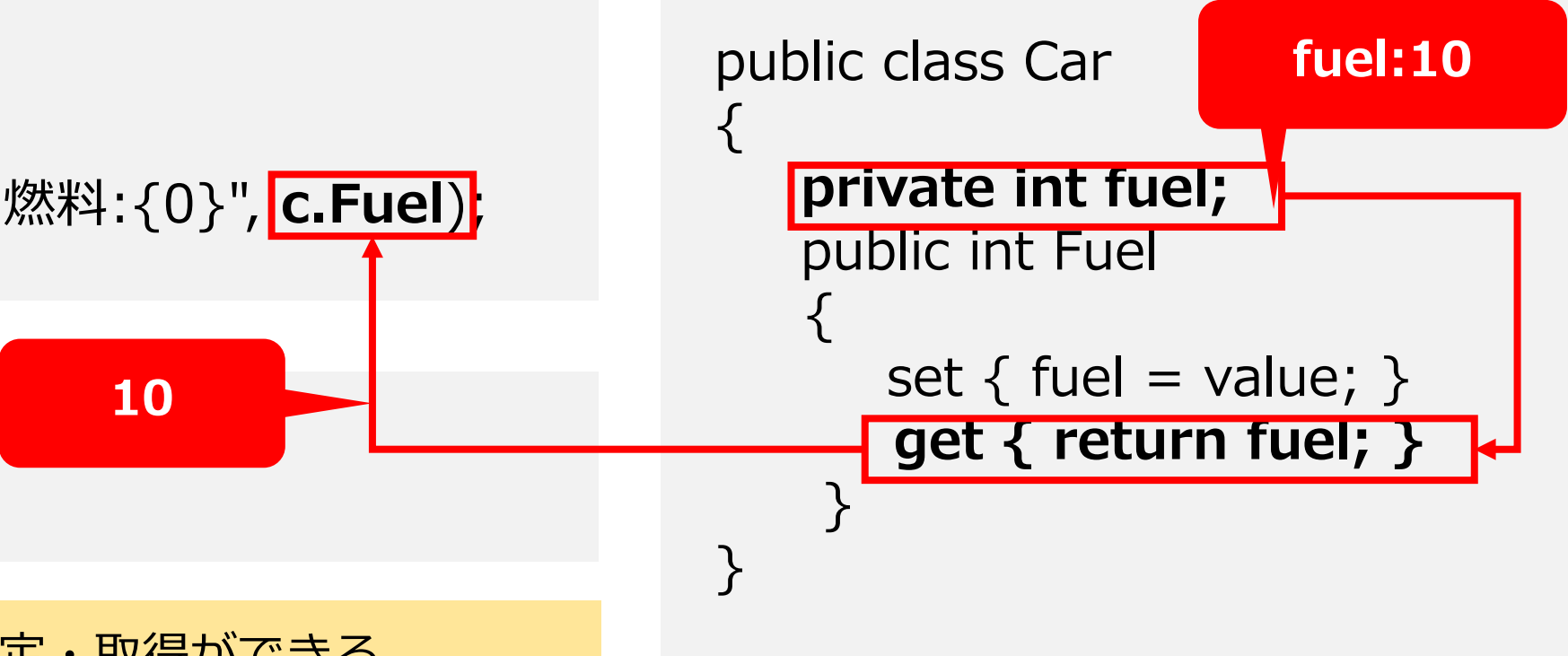
▼実行結果

燃料:10

10

プロパティFuelで値の設定・取得ができる

```
public class Car  
{  
    private int fuel;  
    public int Fuel  
    {  
        set { fuel = value; }  
        get { return fuel; }  
    }  
}
```



読み取り専用のプロパティ

▼コンストラクターの呼び出し

```
City c = new City("東京都");  
//c.Name = "大阪市"; // エラーになる  
Console.WriteLine(c.Name);
```

▼実行結果

東京都

getプロパティのみを付けると読み取り専用になる

```
class City{  
    private string name;  
    public City(string name)  
    {  
        this.name = name;  
    }  
    public string Name  
    {  
        get { return name; }  
    }  
}
```

読み取り専用のプロパティ

▼コンストラクターの呼び出し

```
City c = new City("東京都");  
//c.Name = "大阪市"; // エラーになる  
Console.WriteLine(c.Name);
```

コンストラクタでname設定

“東京都”

▼実行結果

東京都

getプロパティのみを付けると読み取り専用になる

```
class City{  
    private string name;  
    public City(string name)  
    {  
        this.name = name;  
    }  
    public string Name  
    {  
        get { return name; }  
    }  
}
```

読み取り専用のプロパティ

▼コンストラクターの呼び出し

```
City c = new City("東京都");  
//c.Name = "大阪市"; // エラーになる  
Console.WriteLine(c.Name);
```

▼実行結果

東京都

getプロパティのみを付けると読み取り専用になる

```
class City{  
    private string name;  
    public City(string name)  
    {  
        this.name = name;  
    }  
    public string Name  
    {  
        get { return name; }  
    }  
}
```

“東京都”

演習問題5：単純なRPGの作成 1

RPGのゲームのキャラクタークラスGameCharをC#で定義してみましょう。

- ①privateなフィールドname（名前）とhp（ヒットポイント）を持つ
- ②name・hpの初期値はコンストラクタの引数で指定する
- ③name・hpは読み取り専用プロパティName、HPを持つ
- ④戻り値無しの攻撃メソッドAttackで整数型の引数でダメージを受ける
- ⑤Attackで受けたダメージの分だけhpが減る（0以下にならない）
- ⑥Attackでhpがマイナスになっても0とするようにする
- ⑦Attackでhpが0になったら「〇〇は倒れた」と表示（〇〇はnameの値）
- ⑧ShowHPメソッド(引数・戻り値無し)でキャラクターのHPを表示

演習問題5：単純なRPGの作成 1（解答例）

▼ GameCharクラス1（フィールドとコンストラクタ）

```
public class GameChar
{
    private string name; // 名前
    private int hp; // ヒットポイント
    public GameChar(string name,int hp)
    {
        this.name = name;
        this.hp = hp;
    }
}
```

演習問題5：単純なRPGの作成 1（解答例）

▼ GameCharクラス1（フィールドとコンストラクタ）

```
public class GameChar
{
  ① private string name; // 名前
    private int hp; // ヒットポイント
    public GameChar(string name,int hp)
    {
      ② this.name = name;
        this.hp = hp;
    }
}
```

privateで外部から隠蔽

「this」が付くとフィールドを意味する

演習問題5：単純なRPGの作成 1（解答例）

▼ GameCharクラス2（メソッド）

```
public void Attack(int damage)
{
    hp -= damage;
    if (hp <= 0)
    {
        hp = 0;
        Console.WriteLine("{0}は倒れた", name);
    }
}
public void ShowHP()
{
    Console.WriteLine("{0}のHP:{1}", name, hp);
}
```

演習問題5：単純なRPGの作成 1（解答例）

▼ GameCharクラス2（メソッド）

```
public void Attack(int damage)
{
    hp -= damage;
    if (hp <= 0)
    {
        hp = 0;
        Console.WriteLine("{0}は倒れた", name);
    }
}

public void ShowHP()
{
    Console.WriteLine("{0}のHP:{1}", name, hp);
}
```

演習問題5：単純なRPGの作成 1（解答例）

▼ GameCharクラス3（プロパティ）

```
public string Name
{
    get { return name; }
}
public int HP
{
    get { return hp; }
}
}
```

演習問題5：単純なRPGの作成 1（解答例）

▼ GameCharクラス3（プロパティ）

```
③ public string Name
    {
        get { return name; }
    }
    public int HP
    {
        get { return hp; }
    }
}
```

「get」だけにすれば読み取り専用

演習問題6 : GameCharクラスのインスタンスの生成1

演習問題 5 のGameCharクラスのインスタンスを利用してみましょう

- ①最初に「RPGゲーム」と表示する
- ②GameCharクラスのインスタンスを生成する
- ③キャラクター名は「勇者」、HPは10とする
- ④ShowHPメソッドでヒットポイントを表示する

▼期待される実行結果

```
RPGゲーム  
勇者のHP:10
```

演習問題6 : GameCharクラスのインスタンスの生成1 (解答例)

▼正解

```
Console.WriteLine("RPGゲーム");  
GameChar c = new GameChar("勇者",10);  
c.ShowHP();
```

演習問題7 : GameCharクラスのインスタンスの生成2

演習問題6に以下の処理を追加してみましょう（GameCharクラスは変更しない）

- ① 乱数で決めた1~10の敵の攻撃とする
- ② 「○は▲の攻撃を受けた」と表示（○：キャラの名前▲：①の乱数）
- ③ Attackメソッドの引数①として呼び出す
- ④ ShowHPメソッドでヒットポイントを表示する

▼期待される実行結果（乱数が6だったケース）

```
RPGゲーム  
勇者のHP:10  
勇者は6の攻撃を受けた  
勇者のHP:4
```

演習問題7 : GameCharクラスのインスタンスの生成2 (解答例)

▼正解

```
Console.WriteLine("RPGゲーム");
GameChar c = new GameChar("勇者", 10);
c.ShowHP();
/* 追加した処理 */
Random r = new Random();
int attack = r.Next(1, 10);
Console.WriteLine("{0}は{1}の攻撃を受けた", c.Name, attack);
c.Attack(attack); // 敵の攻撃
c.ShowHP();
```

演習問題7 : GameCharクラスのインスタンスの生成2 (解答例)

▼正解

```
Console.WriteLine("RPGゲーム");  
GameChar c = new GameChar("勇者", 10);  
c.ShowHP();  
/* 追加した処理 */
```

- ① Random r = new Random();
int attack = r.Next(1, 10);
- ② Console.WriteLine("{0}は{1}の攻撃を受けた", c.Name, attack);
- ③ c.Attack(attack); // 敵の攻撃
- ④ c.ShowHP();

nameを取得するにはプロパティを用いる

演習問題8 : GameCharクラスのインスタンスの生成3

演習問題7で追加した処理を以下のように変更しましょう

- ①演習問題7で追加した部分をhpが0になるまで繰り返す
- ②hpが0になったら「**ゲームオーバー**」と表示し終了する

▼期待される実行結果

```
RPGゲーム  
勇者のHP:10  
勇者は7の攻撃を受けた  
勇者のHP:3  
勇者は7の攻撃を受けた  
勇者は倒れた  
勇者のHP:0  
**ゲームオーバー**
```

演習問題8 : GameCharクラスのインスタンスの生成2 (解答)

▼正解

```
Console.WriteLine("RPGゲーム");
GameChar c = new GameChar("勇者",10);
c.ShowHP();
// 変更した処理
Random r = new Random();
while(c.HP != 0)
{
    int attack = r.Next(1, 10);
    Console.WriteLine("{0}は{1}の攻撃を受けた", c.Name, attack);
    c.Attack(attack); // 敵の攻撃
    c.ShowHP();
}
Console.WriteLine("**ゲームオーバー**");
```

演習問題8 : GameCharクラスのインスタンスの生成2 (解答)

▼正解

```
Console.WriteLine("RPGゲーム");  
GameChar c = new GameChar("勇者",10);  
c.ShowHP();  
// 変更した処理  
Random r = new Random();
```

```
while(c.HP != 0)  
{
```

Hpが0かどうかプロパティで判断

```
    int attack = r.Next(1, 10);
```

```
    Console.WriteLine("{0}は{1}の攻撃を受けた", c.Name, attack);
```

```
    c.Attack(attack); // 敵の攻撃
```

```
    c.ShowHP();
```

```
}
```

```
Console.WriteLine("**ゲームオーバー**");
```