INTERNET ACADEMY

Institute of Web Design & Software Services

# C#2

## インターネット・アカデミー

# C#　目次

# 1.条件分岐

# 条件分岐とは

特定の条件が成り立つ（成り立たない）場合にプログラムの流れを変えること。

## 条件分岐の種類

・**if**　…　ある条件が成り立つ・成り立たないでプログラムの処理を変える

・**switch**　…　変数の値の取りうる値によってプログラムの処理を変える

# 条件分岐(if文)

（）内に記述した条件がなりたてば{}内の処理が実行される

**▼記述例**

```
int a = 10;    //   いろいろな値に変えてみる
if(a > 0){
      Console.WriteLine("aは正の数です。");  //   正の数だった場合に実行
}
```

**▼実行結果**

```
aは正の数です。
```

# 条件分岐(if文)

（）内に記述した条件がなりたてば{}内の処理が実行される

**▼記述例**

```
int a = 10;    //   いろいろな値に変えてみる
if(a > 0){
    Console.WriteLine("aは正の数です。");  //  正の数だった場合に実行
}
```

**a > 0が成り立つ（true）の場合実行**

**▼実行結果**

```
aは正の数です。
```

**条件が成り立ったので{}内の処理が実行された**

# 条件分岐(if文)

（）内に記述した条件がなりたてば{}内の処理が実行される

**▼記述例**

```
int a = -10;  //   いろいろな値に変えてみる
if(a > 0){
    Console.WriteLine("aは正の数です。");  //  正の数だった場合に実行
}
```

**▼実行結果**

# 条件分岐(if文)

（）内に記述した条件がなりたてば{}内の処理が実行される

**▼記述例**

```
int a = -10;  //   いろいろな値に変えてみる
if(a > 0){
    Console.WriteLine("aは正の数です。");  //  正の数だった場合に実行
}
```

**▼実行結果**

```

```

**条件がなりたたないので{}内の処理が実行されない**

# 条件分岐(if文)

if文で用いられる条件を記述するための比較演算子の一覧

| 演算子 | 意味 | 記述例 |
| --- | --- | --- |
| > | より大きい | a > 0 |
| >= | 以上 | a >= 0 |
| < | より小さい | a < 0 |
| <= | 以下 | a <= 0 |
| == | 等しい | a == 0 |
| != | 等しくない | a != 0 |

# 条件分岐(if~else文)

if文にelseをつけると条件が成り立たなかった場合の処理も記述できる

## 利用例１：条件が成り立つ場合

**▼記述例**

```
int a = 10;
if (a > 0){
    Console.WriteLine("aは正の数です。");
}else{
    Console.WriteLine("aは0以下です。");
}
```

**▼実行結果**

```
aは正の数です。
```

# 条件分岐(if~else文)

if文にelseをつけると条件が成り立たなかった場合の処理も記述できる

**利用例１：条件が成り立つ場合**

**▼記述例**

```
int a = 10;

if (a > 0){
    Console.WriteLine("aは正の数です。");
}else{
    Console.WriteLine("aは0以下です。");
}
```

**a > 0が成り立つ（true）ので実行**

**▼実行結果**

```
aは正の数です。
```

# 条件分岐(if~else文)

if文にelseをつけると条件が成り立たなかった場合の処理も記述できる

## 利用例2：条件が成り立たない場合

**▼記述例**

```
int a = -10;
if (a > 0){
    Console.WriteLine("aは正の数です。");
}else{
    Console.WriteLine("aは0以下です。");
}
```

**a > 0が成り立たない（false）ので実行**

**▼実行結果**

```
aは0以下です。
```

# 条件分岐(if~else if~else文)

if~else if~elseを用いると複数の条件分岐を記述できる

**▼記述例**

```
int a = 0; //   いろいろな値に変えてみる
if (a > 0){
    Console.WriteLine("aは正の数です。");
}else if(a == 0){
    Console.WriteLine("aは0です。");
}else{
    Console.WriteLine("aは負の数です。");
}
```

**▼実行結果**

```
aは0です。
```

# 条件分岐(if~else if~else文)

if~else if~elseを用いると複数の条件分岐を記述できる

**▼記述例**

```
int a = 0; //   いろいろな値に変えてみる
if (a > 0){
    Console.WriteLine("aは正の数です。");
}else if(a == 0){
    Console.WriteLine("aは0です。");
}else{
    Console.WriteLine("aは負の数です。");
}
```

**▼実行結果**

```
aは0です。
```

**a ==0が成り立つ（true）ので実行**

else ifはif~elseの間に何個記述してもよい。

# 条件分岐(if~else if~else文)

if~else if~elseを用いると複数の条件分岐を記述できる

**▼記述例**

```
int a = 0; //   いろいろな値に変えてみる
if (a > 0){
    Console.WriteLine("aは正の数です。");
}else if(a == 0){
    Console.WriteLine("aは0です。");
}else{
    Console.WriteLine("aは負の数です。");
}
```

**▼実行結果**

```
aは0です。
```

**a ==0が成り立つ（true）ので実行**

else ifはif~elseの間に何個記述してもよい。

# 条件分岐(複数の条件)

「**&&**」(and)＝複数の条件が同時に成り立つかを判定する。

**▼記述例**

```
int a = 10;

int b = 10;

if(a == 10 && b == 10){

    Console.WriteLine("aもbも10です");

}
```

**▼実行結果**

```
aもbも10です
```

aが10でありかつbが10の場合のみ成立

**▼条件と組み合わせ**

| 条件 | True or False |
|---|---|
| a = 10,b = 10 | True |
| a = 10,b = 0 | False |
| a = 0,b = 10 | False |
| a = 0,b = 0 | False |

# 条件分岐(論理の反転)

「**!**」(not)＝条件式の内容を逆転させる。

**▼記述例**

```
int a = 0;
if (!(a == 10))
{
    Console.WriteLine("aは10ではありません。");
}
```

**▼実行結果**

```
aは10ではありません。
```

# 条件分岐(論理の反転)

「**!**」(not)＝条件式の内容を逆転させる。

**▼記述例**

```
int a = 0;
if (!(a == 10))
{
    Console.WriteLine("aは10ではありません。");
}
```

**a ==0ではない　→「a != 0」と同じ意味**

**▼実行結果**

```
aは10ではありません。
```

# 条件分岐(論理の反転)

「**!**」(not)＝条件式の内容を逆転させる。

**▼記述例**

```
string s = "world";   //   いろいろな値に変えてみる
if (!s.Equals("hello")){
    Console.WriteLine("sは「world」ではありません。");
}
```

**▼実行結果**

```
sは「world」ではありません。
```

# 条件分岐(複数の条件)

「||」(or)＝複数の条件のどれかが成り立つかを判定する。

**▼記述例**

```
int a = 10;

int b = 10;

if(a == 10 || b == 10){

    Console.WriteLine("aかbが10です");

}
```

**▼実行結果**

```
aもbも10です
```

aが10かもしくはbが10の場合のみ成立

**▼条件と組み合わせ**

| 条件 | True or False |
|---|---|
| a = 10,b = 10 | True |
| a = 10,b = 0 | False |
| a = 10,b = 0 | False |
| a = 0,b = 0 | False |

# 条件分岐(switch文)

switch文を用いると複数の条件分岐を記述できる

**▼記述例**

```
int num = 1;
switch(num){
case 1:
    Console.WriteLine("one");    //  numが1の場合
    break;
case 2:
    Console.WriteLine("two");    // numが2の場合
    break;
default:
    Console.WriteLine("不適切な値です。"); //  それ以外の値
    break;
}
```

**▼実行結果**

```
one
```

caseはいくつでもつけることができる

各処理の最後にbreakを付ける

# 条件分岐(switch文)

switch文を用いると複数の条件分岐を記述できる

**▼記述例**

```
int num = 1;
switch(num){
case 1:
    Console.WriteLine("one");    //  numが1の場合
    break;
case 2:
    Console.WriteLine("two");    // numが2の場合
    break;
default:
    Console.WriteLine("不適切な値です。"); //  それ以外の値
    break;
}
```

**numの値が「1」の場合に実行**

**▼実行結果**

```
one
```

caseはいくつでもつけることができる

各処理の最後にbreakを付ける

# 条件分岐(switch文)

switch文を用いると複数の条件分岐を記述できる

**▼記述例**

```
int num = 2;
switch(num){
case 1:
    Console.WriteLine("one");    //  numが1の場合
    break;
case 2:
    Console.WriteLine("two");    // numが2の場合
    break;
default:
    Console.WriteLine("不適切な値です。"); //  それ以外の値
    break;
}
```

**▼実行結果**

```
two
```

# 条件分岐(switch文)

switch文を用いると複数の条件分岐を記述できる

**▼記述例**

```
int num = 2;
switch(num){
case 1:
    Console.WriteLine("one");    //  numが1の場合
    break;
case 2:
    Console.WriteLine("two");    // numが2の場合
    break;
default:
    Console.WriteLine("不適切な値です。"); //  それ以外の値
    break;
}
```

**numの値が「2」の場合に実行**

**▼実行結果**

```
two
```

# 条件分岐(switch文)

switch文を用いると複数の条件分岐を記述できる

**▼記述例**

```
int num = 3;
switch(num){
case 1:
    Console.WriteLine("one");    //  numが1の場合
    break;
case 2:
    Console.WriteLine(“two”);    //  numが2の場合
    break;
default:
    Console.WriteLine("不適切な値です。"); //  それ以外の値
    break;
}
```

**▼実行結果**

```
不適切な値です。
```

defaultはcaseのいずれにも当てはまらない場合に実行される部分

# 条件分岐(switch文)

switch文を用いると複数の条件分岐を記述できる

**▼記述例**

```
int num = 3;
switch(num){
case 1:
    Console.WriteLine("one");    //  numが1の場合
    break;
case 2:
    Console.WriteLine(“two”);    //  numが2の場合
    break;
default:
    Console.WriteLine("不適切な値です。"); //  それ以外の値
    break;
}
```

**▼実行結果**

```
不適切な値です。
```

defaultはcaseのいずれにも当てはまらない場合に実行される部分

**numの値が「1」「2」以外の場合に実行**

# 条件分岐(switch文)

switch文をif文で置き換えると次のように

**▼switch文での記述例**

```
int num = 1;
switch(num){
case 1:
    Console.WriteLine("one");    //  numが1の場合
    break;
case 2:
    Console.WriteLine(“two”);    //  numが2の場合
    break;
default:
    Console.WriteLine("不適切な値です。"); //  それ以外の値
    break;
}
```

**▼if文での記述例**

```
int num = 1;
if(num == 1){
      Console.WriteLine("one");    //  numが1だった場合
}else if(num == 2){
      Console.WriteLine("two");    //  numが2だった場合
}else{
      Console.WriteLine("不適切な値です。"); //  それ以外の値
}
```

# 条件分岐(switch文)

switch文をif文で置き換えると次のように

**▼switch文での記述例**

```
int num = 1;
switch(num){
case 1:
    Console.WriteLine("one");    //  numが1の場合
    break;
case 2:
    Console.WriteLine(“two”);    //  numが2の場合
    break;
default:
    Console.WriteLine("不適切な値です。"); //  それ以外の値
    break;
}
```

対応関係

**▼if文での記述例**

```
int num = 1;
if(num == 1){
      Console.WriteLine("one");    //  numが1だった場合
}else if(num == 2){
      Console.WriteLine("two");    //  numが2だった場合
}else{
      Console.WriteLine("不適切な値です。"); //  それ以外の値
}
```

# 条件分岐(switch文)

switch文をif文で置き換えると次のように

**▼switch文での記述例**

```
int num = 1;
switch(num){
case 1:
    Console.WriteLine("one");    //  numが1の場合
    break;
case 2:
    Console.WriteLine(“two”);    //  numが2の場合
    break;
default:
    Console.WriteLine("不適切な値です。"); //  それ以外の値
    break;
}
```

対応関係

**▼if文での記述例**

```
int num = 1;
if(num == 1){
      Console.WriteLine("one");    //  numが1だった場合
}else if(num == 2){
      Console.WriteLine("two");    //  numが2だった場合
}else{
      Console.WriteLine("不適切な値です。"); //  それ以外の値
}
```

# 条件分岐(switch文)

switch文をif文で置き換えると次のように

**▼switch文での記述例**

```
int num = 1;
switch(num){
case 1:
    Console.WriteLine("one");    //  numが1の場合
    break;
case 2:
    Console.WriteLine(“two”);    //  numが2の場合
    break;
default:
    Console.WriteLine("不適切な値です。"); //  それ以外の値
    break;
}
```

対応関係

**▼if文での記述例**

```
int num = 1;
if(num == 1){
      Console.WriteLine("one");    //  numが1だった場合
}else if(num == 2){
      Console.WriteLine("two");    //  numが2だった場合
}else{
      Console.WriteLine("不適切な値です。"); //  それ以外の値
}
```

# 2.複雑なループ処理

# breakとcontinue

## break：ループを終了する

▼breakの記述例

```
for(int i = 0; i < 5; i++)
{
    if(i == 3)
    {
        break;  //  ループから抜ける
     }
     Console.WriteLine(i);
 }
```

▼実行結果

```
0
1
2
```

# breakとcontinue

## break：ループを終了する

▼breakの記述例

```
for(int i = 0; i < 5; i++)
{
    if(i == 3)
    {
        break;  //  ループから抜ける
    }
    Console.WriteLine(i);
}
```

iが3でループから抜ける

▼実行結果

```
0
1
2
```

「2」まで表示して終了

# breakとcontinue

## continue：その回のループをスキップする

▼ continueの記述例

```
for(int i = 0; i < 5; i++)
{
    if(i == 3)
    {
        continue;  //  先頭に戻る
     }
     Console.WriteLine(i);
}
```

▼実行結果

```
0
1
2
4
```

# breakとcontinue

## continue：その回のループをスキップする

▼ continueの記述例

```
for(int i = 0; i < 5; i++)
{
    if(i == 3)
    {
        continue;  //  先頭に戻る
    }
    Console.WriteLine(i);
}
```

iが3でループの先頭に戻る

▼実行結果

```
0
1
2
4
```

「3」の表示が飛ばされる

# breakとcontinue

## whileループでもbreakは利用可能

▼breakの記述例

```
int i = 0;
while(i < 5){
    if(i ==3){
        break;
    }
    Console.WriteLine(i);
    i++;
}
```

▼実行結果

```
0
1
2
```

# breakとcontinue

## whileループでもcontinueは利用可能

▼ continueの記述例

```
while(i < 5){
  i++;
  if(i ==3){
      continue;
  }
  Console.WriteLine(i);
}
```

▼実行結果

```
1
2
4
5
```

# breakとcontinue(応用)

## 準備：乱数（でたらめな数を発生させる）

▼ 記述例

```
// 乱数の初期設定
Random rnd = new Random();
// 1以上7未満の乱数を発生させる（1から6）
int dice = rnd.Next(1, 7);
Console.WriteLine(dice);
```

▼実行結果（実行するたび違う値）

```
1
```

# breakとcontinue(応用)

## 準備：乱数（でたらめな数を発生させる）

▼ 記述例

```
// 乱数の初期設定
Random rnd = new Random();
// 1以上7未満の乱数を発生させる（1から6）
int dice = rnd.Next(1, 7);
Console.WriteLine(dice);
```

1から6(=7-1)までの乱数発生させる

▼実行結果（実行するたび違う値）

```
1
```

1から6までの数が得られる

**ポイント**

Next(m,n)で、mからn-1までの整数の乱数を発生させることができる。

# breakとcontinue(応用)

## 無限ループ（乱数によって終了させられる）

▼ 記述例

```
Random rnd = new Random();
while(true){
  int dice = rnd.Next(1, 7);
  Console.WriteLine(dice);
  if(dice == 6){
    break;  // ループから抜ける
  }
}
```

▼実行結果（実行するたび違う値）

```
3
4
4
6
```

# breakとcontinue(応用)

## 無限ループ（乱数によって終了させられる）

▼ 記述例

```
Random rnd = new Random();
while(true){
  int dice = rnd.Next(1, 7);
  Console.WriteLine(dice);
  if(dice == 6){
    break;  // ループから抜ける
  }
}
```

▼実行結果（実行するたび違う値）

```
3
4
4
6
```

無限ループ（無限に繰り返し続ける）

# breakとcontinue(応用)

## 無限ループ（乱数によって終了させられる）

▼ 記述例

```
Random rnd = new Random();
while(true){
  int dice = rnd.Next(1, 7);
  Console.WriteLine(dice);
  if(dice == 6){
    break;  // ループから抜ける
  }
}
```

▼実行結果（実行するたび違う値）

```
3
4
4
6
```

diceの値が6ならループから抜ける

# foreachループ

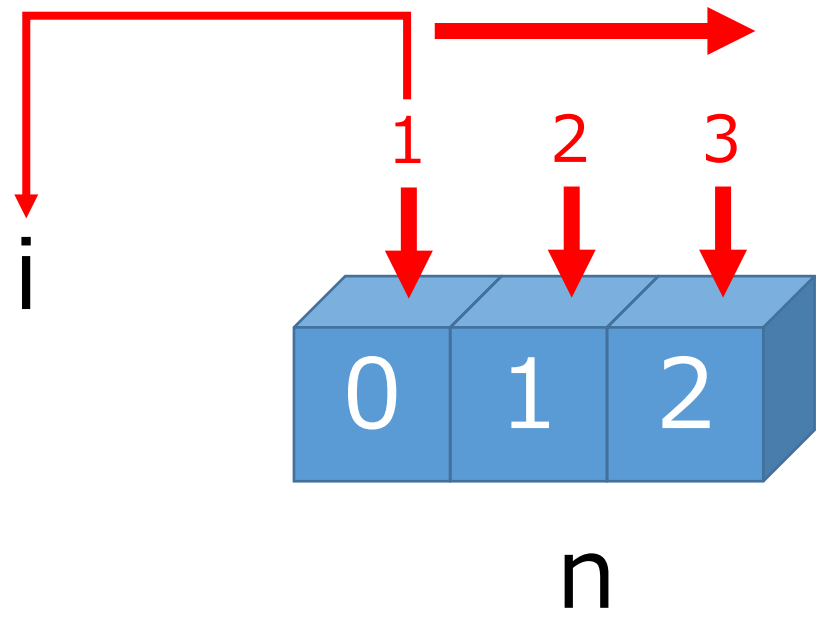
配列の値を取得するには**foreach**ループを用いることもできる。

**▼記述例**

```
int[] n = { 1, 2, 3 };

foreach(int i in n){

    Console.WriteLine(i);

}
```

**▼実行結果**

```
1
2
3
```




配列nの値を先頭から変数iに代入していき、
最後まで来たらループを終了する。

# 【参考】forの二重ループ

ループの中にループが入ったものを**二重ループ**という。

**▼記述例**

```
for(int i = 1; i <= 2; i++){

  // 内側のループ

  for(int j = 1; j <=3 ; j++){

      System.Console.WriteLine("i:"+i+" / j:"+j);

   }

    System.Console.WriteLine("---");

}
```

**▼実行結果**

```
i:1 / j:1
i:1 / j:2
i:1 / j:3
---
i:2 / j:1
i:2 / j:2
i:2 / j:3
---
```

# 【参考】forの二重ループ

ループの中にループが入ったものを**二重ループ**という。

**▼記述例**

```
for(int i = 1; i <= 2; i++){

  // 内側のループ

  for(int j = 1; j <=3 ; j++){

     System.Console.WriteLine("i:"+i+" / j:"+j);

  }

   System.Console.WriteLine("---");

}
```

内側のループ

**▼実行結果**

```
i:1 / j:1
i:1 / j:2
i:1 / j:3
---
i:2 / j:1
i:2 / j:2
i:2 / j:3
---
```

外側のループで内側のループ繰り返す

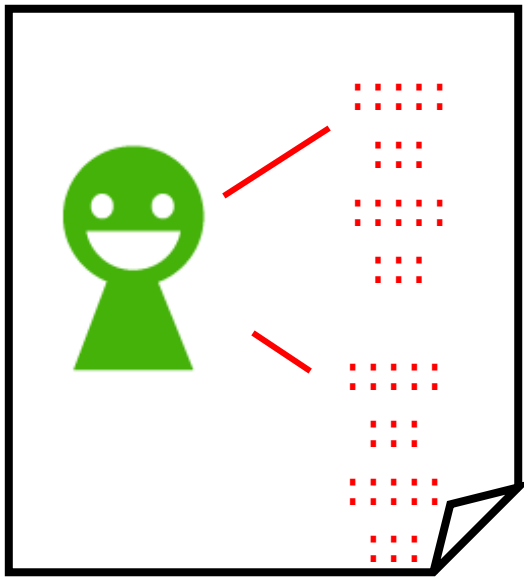
外側のループ

# 3.クラスとオブジェクト

# クラスとインスタンス

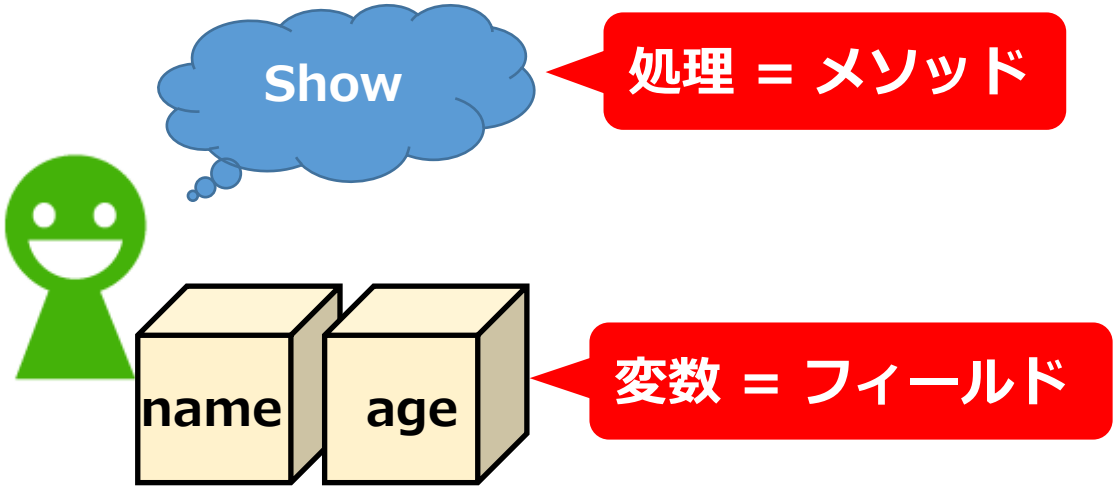
クラスとは独自のオブジェクト（インスタンス）を作るための設計図のようなもの

**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）**

# 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。

# 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;
}
```

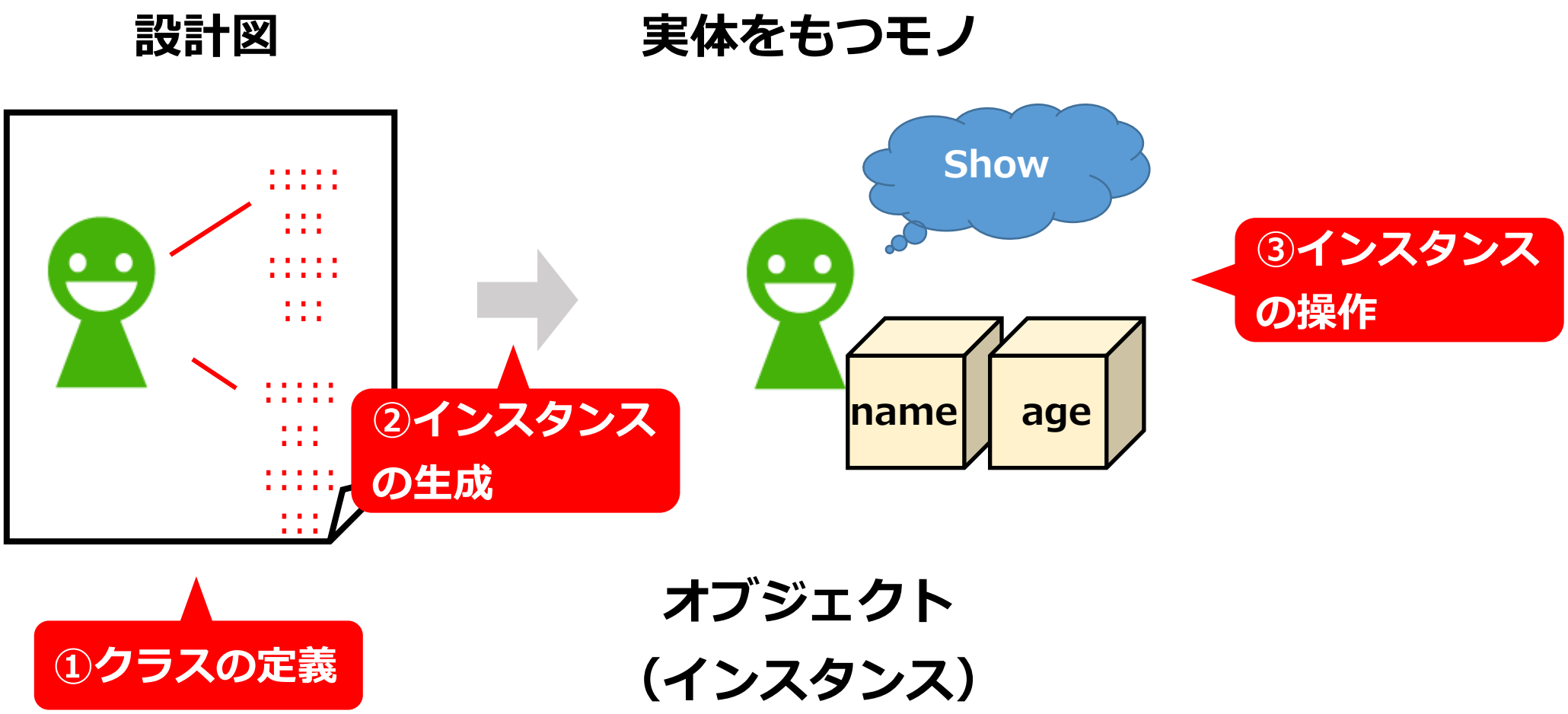
クラス名

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。

# 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



# 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
```

# 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);
```

インスタンスの生成

## ▼実行結果

```
名前:太郎 年齢:18
```

# 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
```

# 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
```

# VisualStudioの場合

クラスを作るためのプログラムをプロジェクトの作成からやってみる

## プロジェクト作成

# VisualStudioの場合

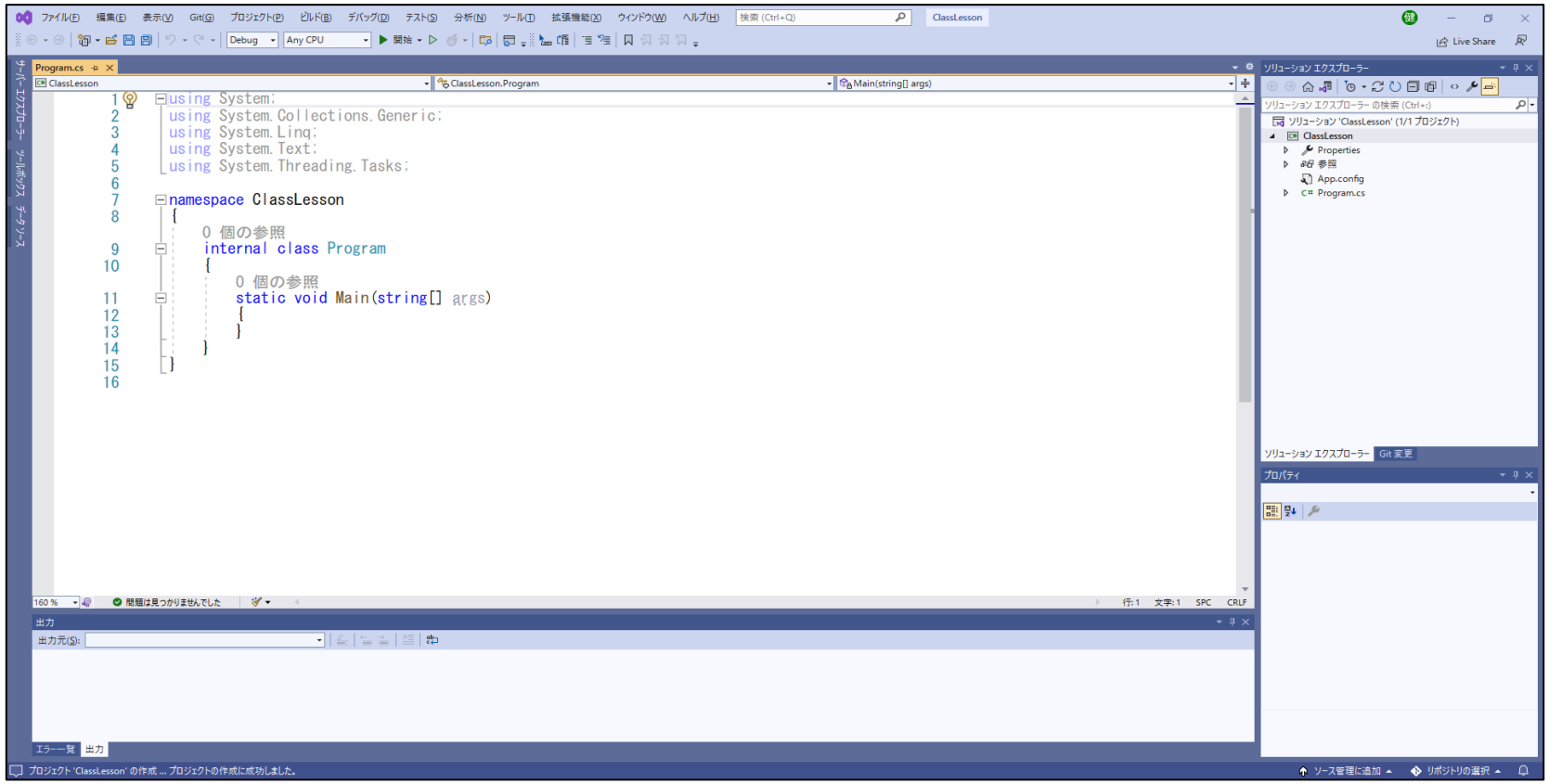
作成するプロジェクトの種類を選択する必要がある「**コンソールアプリ（.NETFramework）**」を選択し、「次へ」を選択する

# プロジェクトの作成と構造

プロジェクト名を入力し「次へ」ボタンをクリックし次の画面で「作成」をクリックする

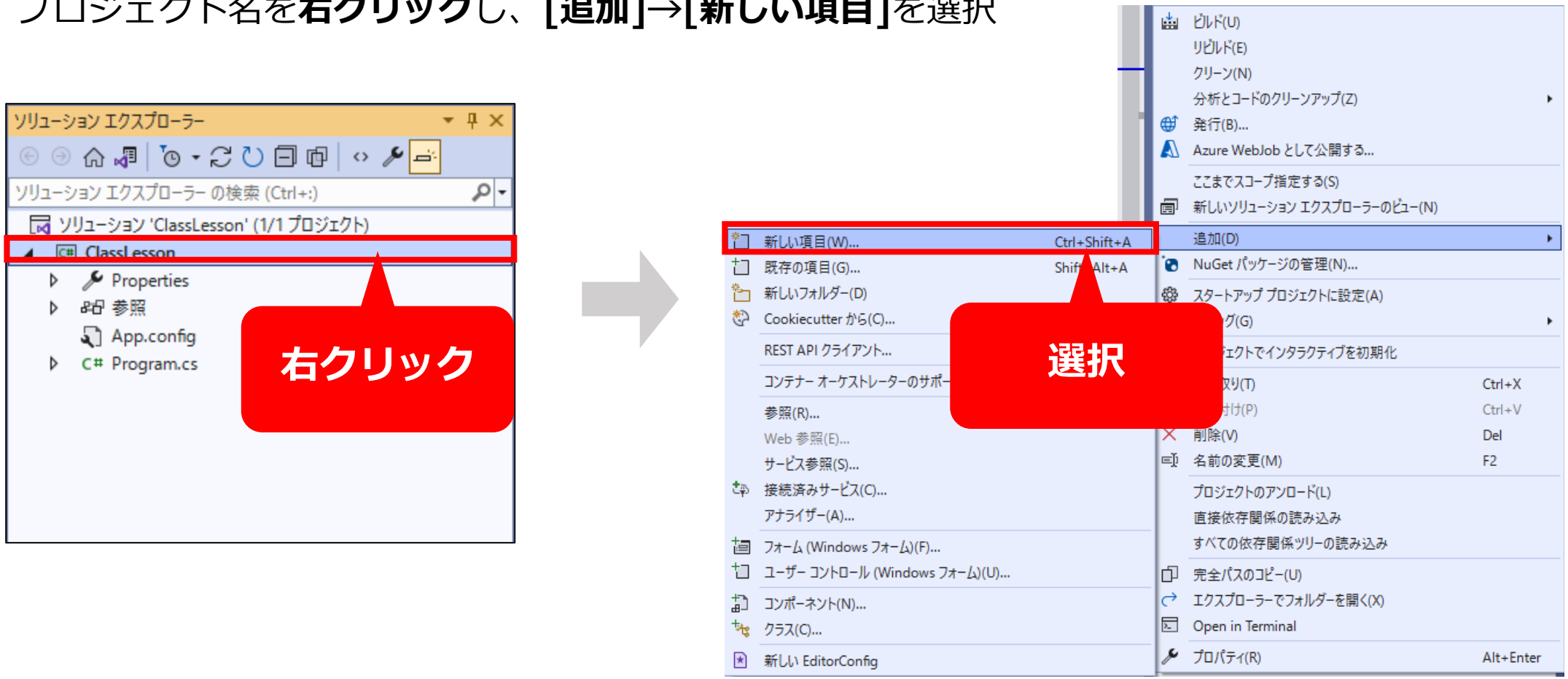
# VisualStudioの場合

プロジェクトが完成する。



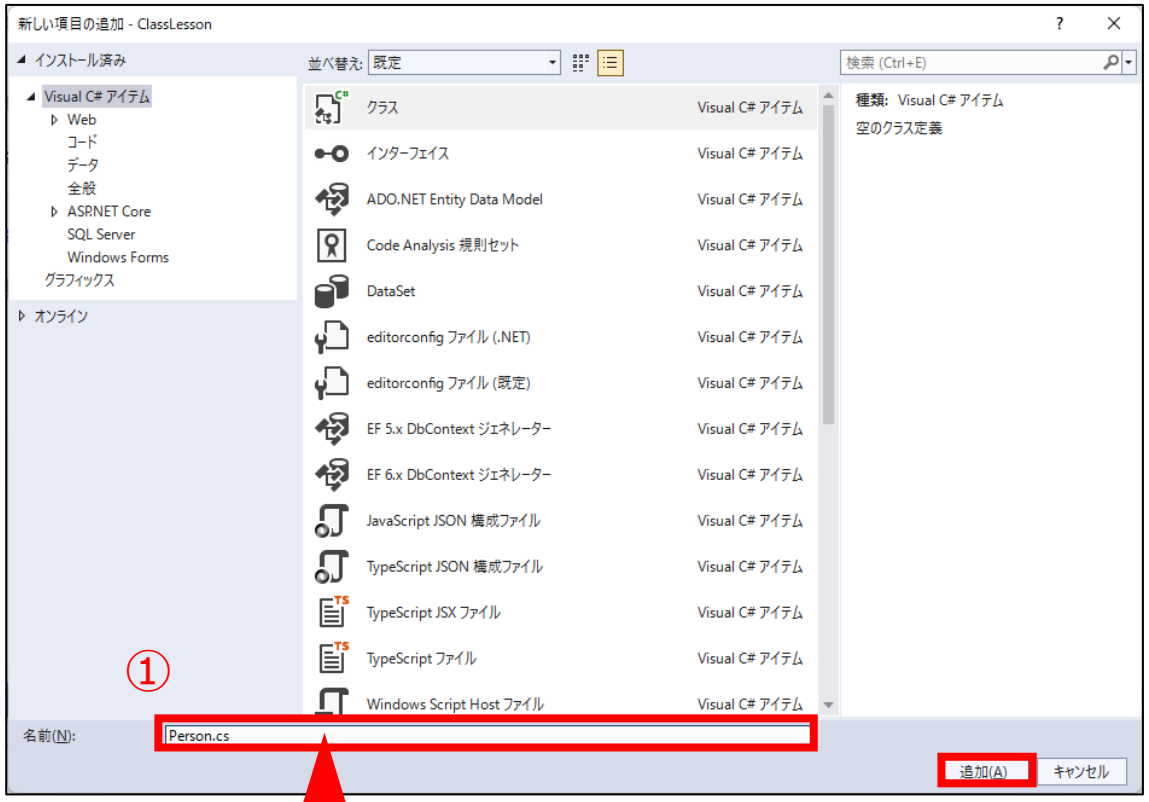
# VisualStudioの場合

プロジェクト名を**右クリック**し、**[追加]→[新しい項目]**を選択

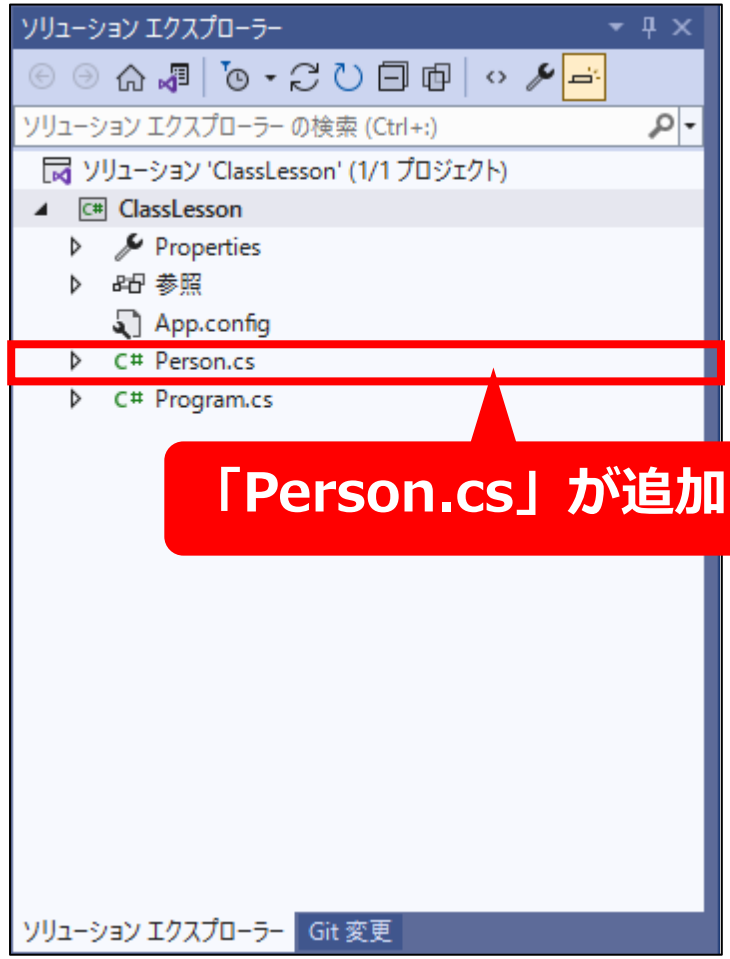


# VisualStudioの場合

名前に「**Person.cs**」（ファイル名はクラス名と一致させる）として「追加」をクリックする。



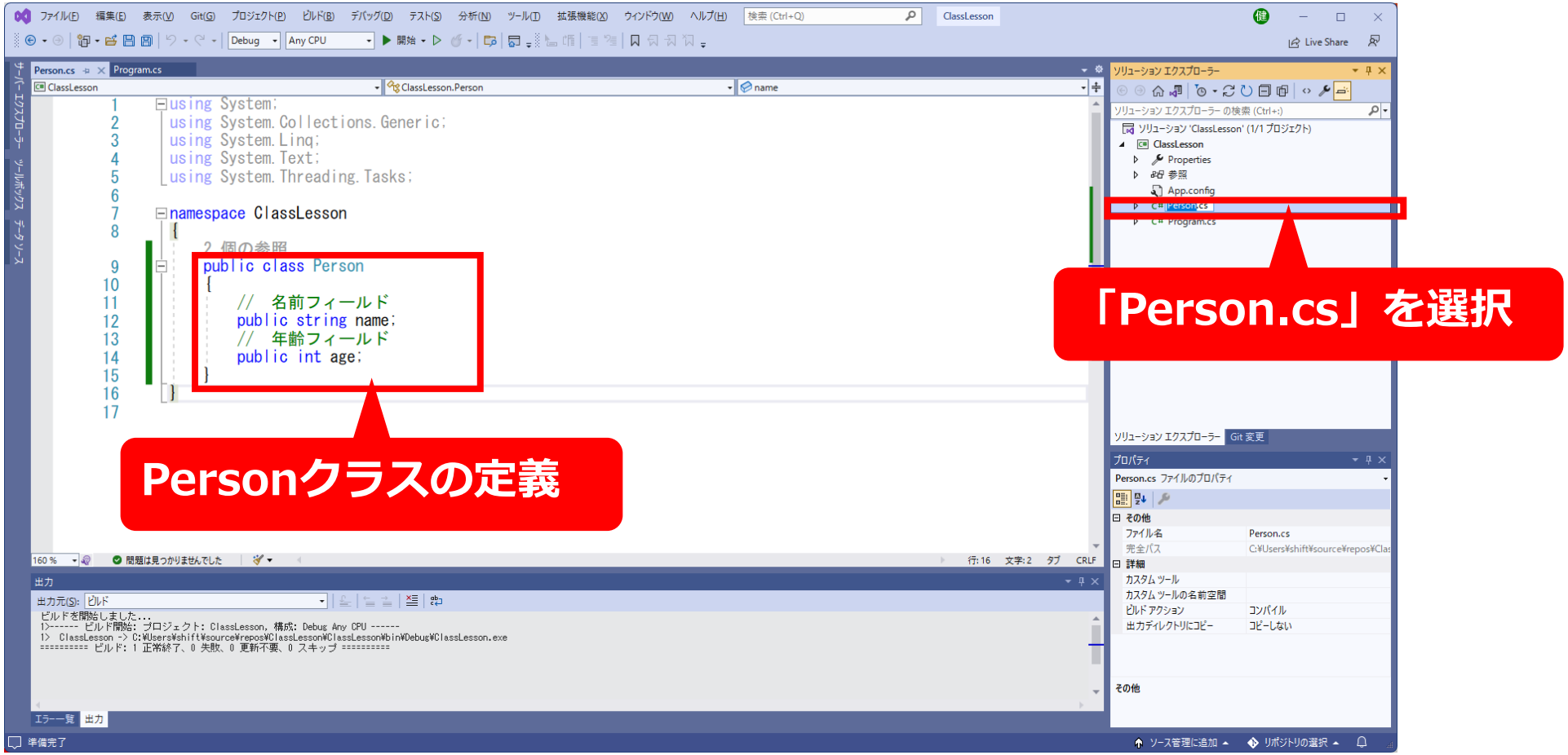

「Person.cs」と入力




「Person.cs」が追加される

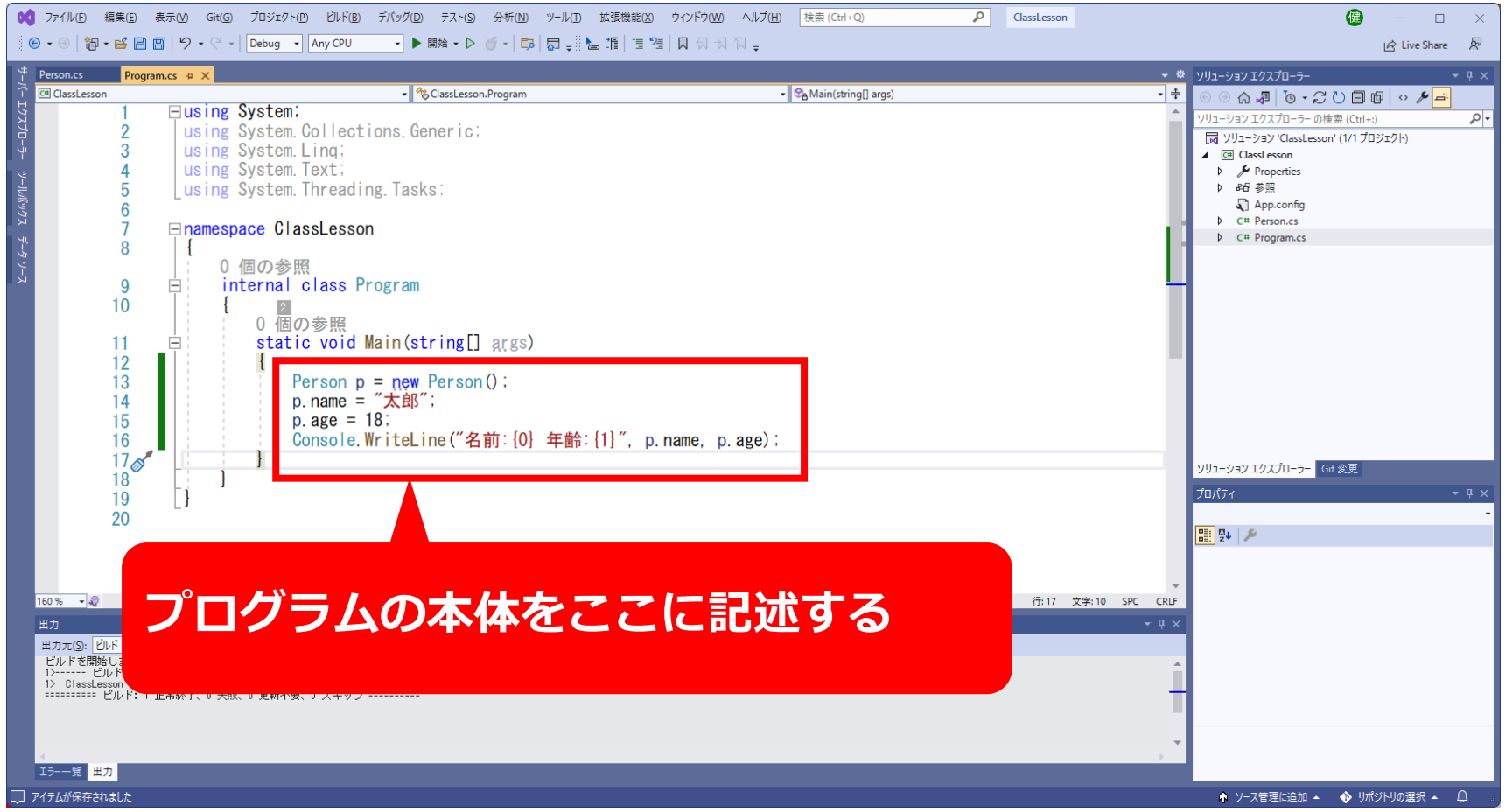
# VisualStudioの場合

「Person.cs」を選択しクラスの定義を記述する。



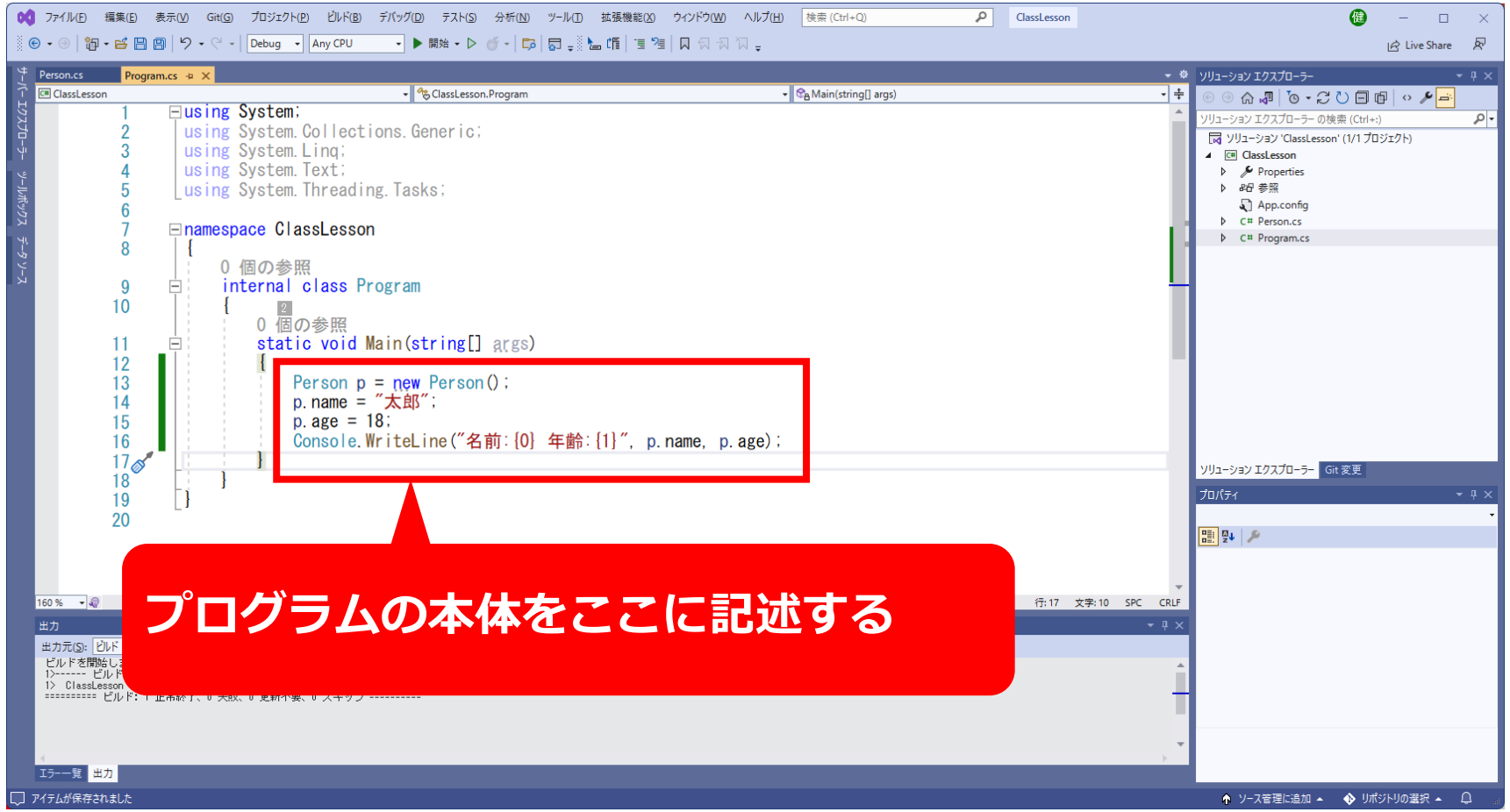
# VisualStudioの場合

「Program.cs」を選択しMainメソッドに処理を記述する。



# VisualStudioの場合

「**Program.cs**」を選択しMainメソッドに処理を記述する。



# Paizaの場合

「**Main.cs**」の横の＋をクリックし出現したタブに「**Person.cs**」と名前を付ける

# Paizaの場合

「**Main.cs**」にメインの処理、「**Person.cs**」にPersonクラスの定義を記述する。

Main.cs Person.cs

```
1  using System;
2
3  public class Hello{
4      public static void Main(){
5          Person p = new Person();
6          p.name = "太郎";
7          p.age = 18;
8          Console.WriteLine("名前:{0} 年齢:{1}", p.name, p.age);
9      }
10 }
11
```

Main.cs Person.cs

```
1  using System;
2
3  public class Person
4  {
5      //  名前フィールド
6      public string name;
7      //  年齢フィールド
8      public int age;
9  }
```

# Paizaの場合

「**Main.cs**」にメインの処理、「**Person.cs**」にPersonクラスの定義を記述する。

Main.cs ✖ Person.cs ✖ ✚

```
1  using System;
2
3  public class Hello{
4      public static void Main(){
5          Person p = new Person();
6          p.name = "太郎";
7          p.age = 18;
8          Console.WriteLine("名前:{0} 年齢:{1}", p.name, p.age);
9      }
10 }
11
```

Main.cs ✖ Person.cs ✖ ✚

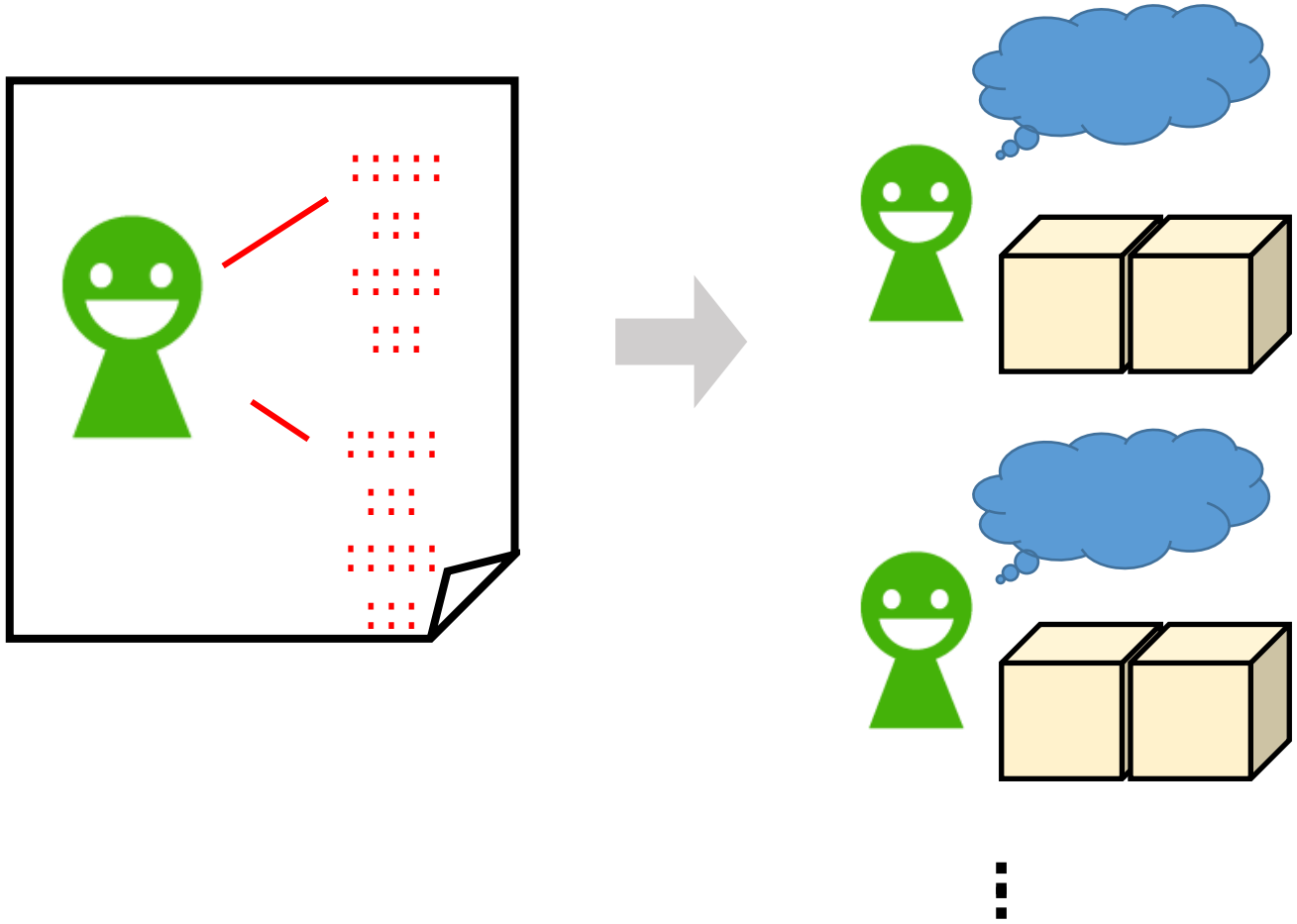
```
1  using System;
2
3  public class Person
4  {
5      //  名前フィールド
6      public string name;
7      //  年齢フィールド
8      public int age;
9  }
```

**! ポイント**

**それぞれの先頭に「using System;」をつける**

# 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

# 複数のインスタンスの生成

**▼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
```

# 複数のインスタンスの生成

## ▼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
```

# 複数のインスタンスの生成

## ▼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
```

# 演習

# 演習問題１：名前を保持するクラスの作成

以下のクラスをC#で定義してみましょう。

**・クラス名は「Student」（ファイル名はStudent.cs）**

**・名前を表すstring型のフィールドnameを持つ**

**・学年を表すint型のフィールドgradeを持つ**

**▼Student.cs**

# 演習問題１：名前を保持するクラスの作成

以下のクラスをC#で定義してみましょう。

**・クラス名は「Student」（ファイル名はStudent.cs）**

**・名前を表すstring型のフィールドnameを持つ**

**・学年を表すint型のフィールドgradeを持つ**

**▼Student.cs（正解）**

```
public class Student
{
    public string name;
    public int grade;
}
```

# 演習問題１：名前を保持するクラスの作成

以下のクラスをC#で定義してみましょう。

**・クラス名は「Student」（ファイル名はStudent.cs）**

**・名前を表すstring型のフィールドnameを持つ**

**・学年を表すint型のフィールドgradeを持つ**

**▼Student.cs（正解）**

```
public class Student
{
    public string name;
    public int grade;
}
```

# 演習問題 2：前を保持するクラスのオブジェクト生成

演習問題１で定義したStudentクラスを用いて以下の処理を記述しましょう

- **Studentクラスのインスタンスを生成し変数sに代入する**
- **生成したインスタンスのnameフィールドに「鈴木ひまり」と代入する**
- **生成したインスタンスのgradeフィールドに「3」と代入する**
- **提示された実行結果のように代入した名前と学年を出力する**

**▼実行結果**

```
名前:鈴木ひまり 学年:3
```

# 演習問題 2：前を保持するクラスのオブジェクト生成

演習問題１で定義したStudentクラスを用いて以下の処理を記述しましょう

**▼正解コード**

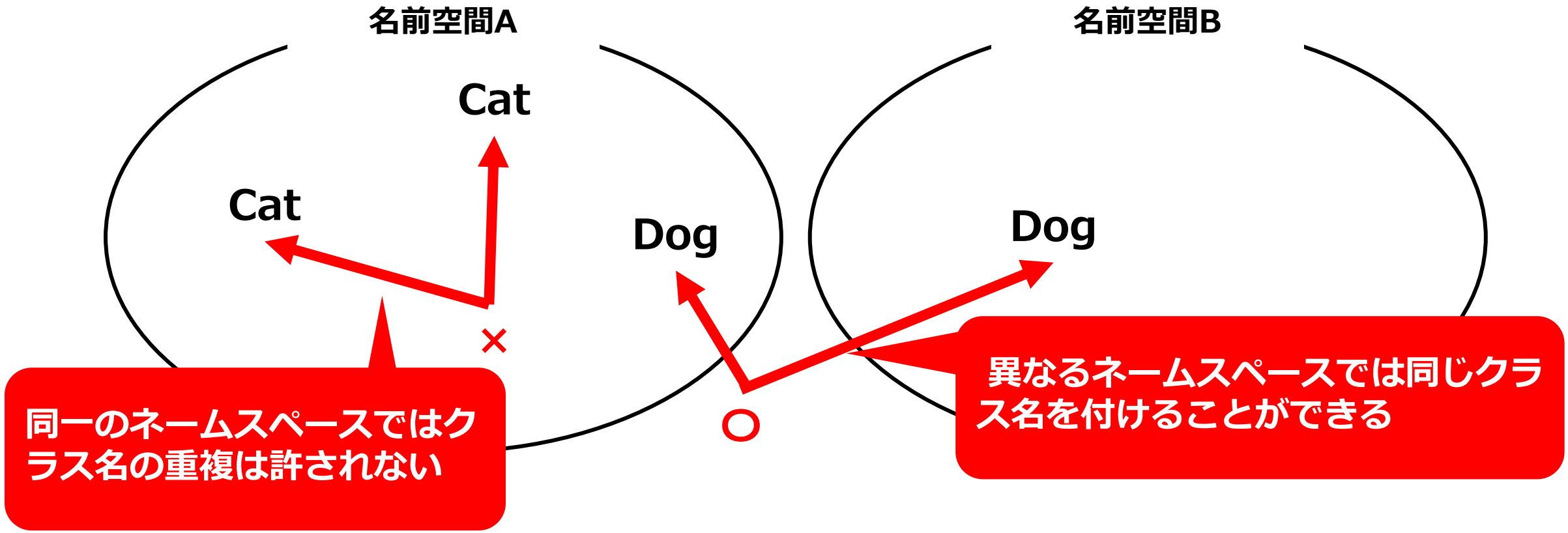
```
Student s = new Student();
s.name = "鈴木ひまり";
s.grade = 3;
Console.WriteLine("名前:{0} 学年:{1}",s.name,s.grade);
```

# 【参考】名前空間

# 名前空間とは

ネームスペースとも言い、同じクラス名等を使っても競合しないように区分けする仕組み

## 名前空間の考え方



# 名前空間とは

**ネームスペース（namespace）**の定義の中にクラスの定義を行う。

```
namespace ClassLesson
{
    0 個の参照
    internal class Program
    {
        0 個の参照
        static void Main(string[] args)
        {
            Person p = new Person();
            p.name = "太郎";
            p.age = 18;
            Console.WriteLine("名前:{0} 年齢:{1}", p.name, p.age);
        }
    }
}
```

```
namespace ClassLesson
{
    2 個の参照
    public class Person
    {
        //  名前フィールド
        public string name;
        //  年齢フィールド
        public int age;
    }
}
```

namespace （ネームスペースの名前） { … } の中にクラスを定義する

# 名前空間と正式なクラス名

正式なクラス名は、「**名前空間名.クラス名**」となる。

```
namespace ClassLesson
{
    0 個の参照
    internal class Program
    {
        0 個の参照
        static void Main(string[] args)
        {
            Person p = new Person();
            p.name = "太郎";
            p.age = 18;
            Console.WriteLine("名前:{0} 年齢:{1}", p.name, p.age);
        }
    }
}
```

**同一名前空間なので「Classname.」は不要**

```
namespace ClassLesson
{
    2 個の参照
    public class Person
    {
        //  名前フィールド
        public string name;
        //  年齢フィールド
        public int age;
    }
}
```

**正式なクラス名「ClassLesson.Person」**

同一の名前空間内ではクラス名のみで利用可能。

# usingディレクティブ

usingで名前空間名を指定すると利用するクラスの名前空間名を省略できる。

**▼usingディレクテイブの利用**

```
using Hoge;
```

**usingディレクテイブなし**

**▼インスタンスの生成（名前空間名なし）**

```
Fuga f = new Fuga();
```

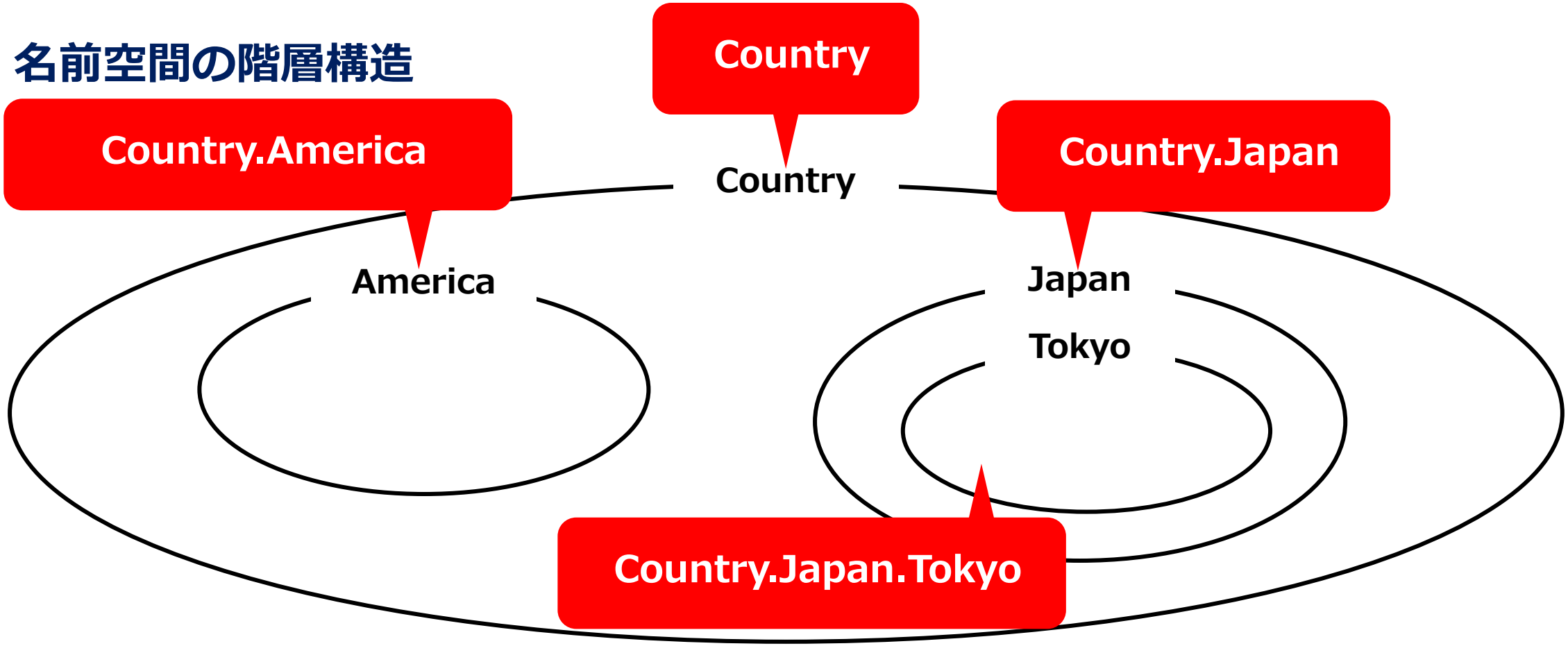
**▼インスタンスの生成（名前空間あり）**

```
Hoge.Fuga f = new Hoge.Fuga();
```

名前空間**Hoge**にクラス**Fuga**が存在した場合

# 名前空間の階層構造

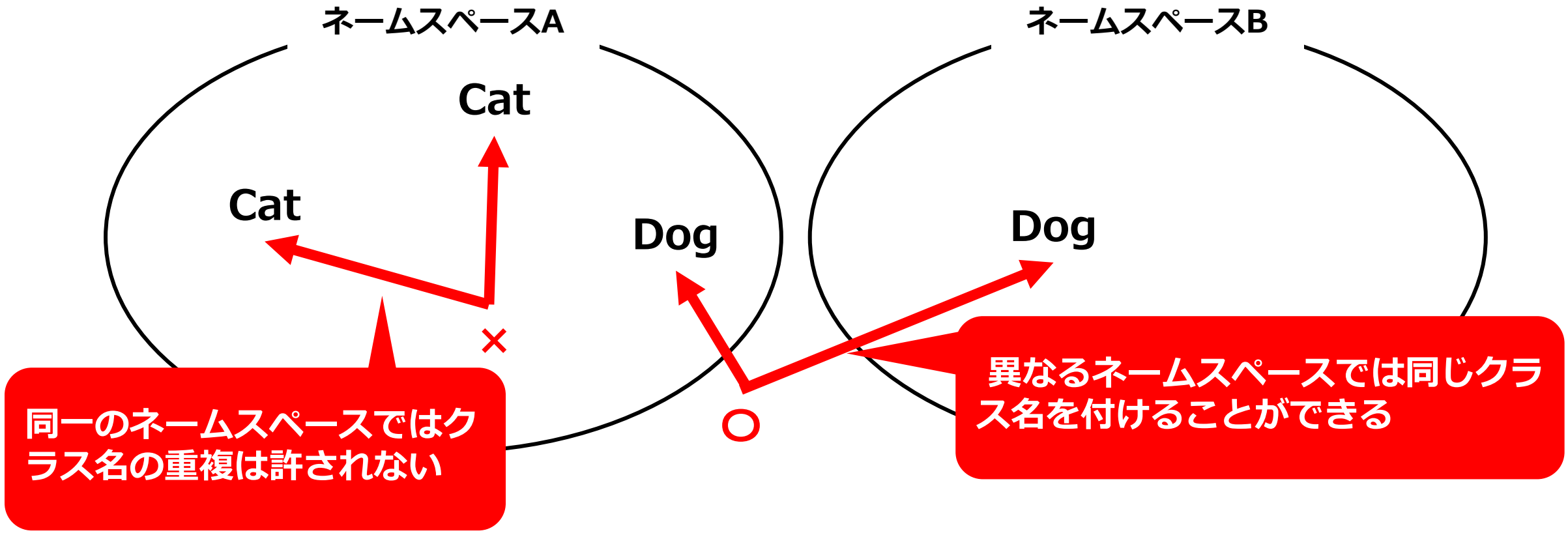
名前空間は階層構造を持つことができ、階層間は「.」（ピリオド）で区切る。



# ネームスペースSystem

C#にはもともと「System」という名前の名前空間が存在している。

## ネームスペース「System」



# 名前空間System

C#にはもともと「System」という名前の名前空間が存在している。

**▼usingディレクテイブの利用**

```
using System;
```

**▼Consoleクラスの利用**

```
Console.WriteLine(“Hello,World.”);
```

**▼usingディレクテイブなしでのConsoleクラスの利用**

```
System.Console.WriteLine(“Hello,World.”);
```

Consoleクラスなど、通常は名前空間Systemに属しているクラスを利用するには、通常は名前空間から始まる「System.Console…」とする必要があるが、「using System;」とすることにより通常は名前空間が省略されている。

# 名前空間System

名前空間Systemには階層構造があり目的に応じて使い分けられる。

## Systemの階層構造