INTERNET ACADEMY

Institute of Web Design & Software Services

# C#6

## インターネット・アカデミー

# C#　目次

# 1.コレクションとは

# 配列変数の問題点

大量のデータを扱う配列変数にはいろいろと欠点がある

## 配列変数の問題点

・**サイズが変えられない** … 最初に定義した大きさから変更することができない。

・**データ管理の方法が画一的** … インデックスによる要素の管理しかできない。

・**データの重複などの確認が面倒** … データの重複をしたくない場合事前チェックが必要。

# コレクションとは

配列を高度にした機能を提供した大量のデータを扱うことを可能にしたクラス群のこと

## 主なコレクション

・**List(リスト)** … 配列に似ているが要素の追加・挿入・削除が可能。

・**Dictionary（ディクショナリー）** … 辞書のようにキー・値の組み合わせでデータ管理。

・**HashSet（ハッシセット）** … 重複のないデータの管理や集合演算が可能。

# 2.List

# Listの基本

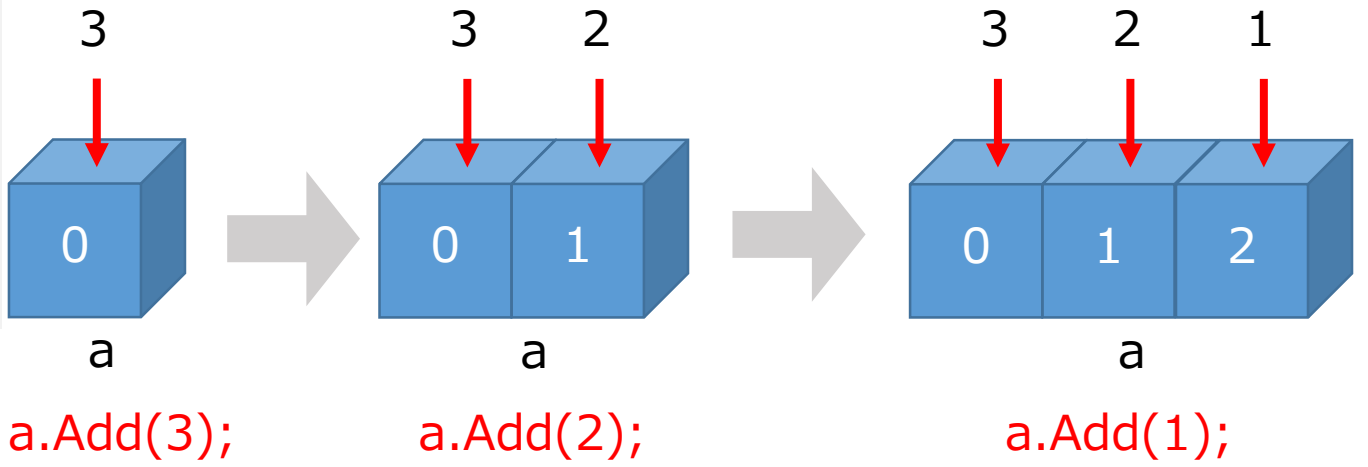
## 基本的なListの使い方

▼記述例

```
List<int> a = new List<int>();
a.Add(3);
a.Add(2);
a.Add(1);
Console.WriteLine(a[0]);
Console.WriteLine(a[1]);
Console.WriteLine(a[2]);
```

▼実行結果

```
3
2
1
```



# Listの基本

## 基本的なListの使い方

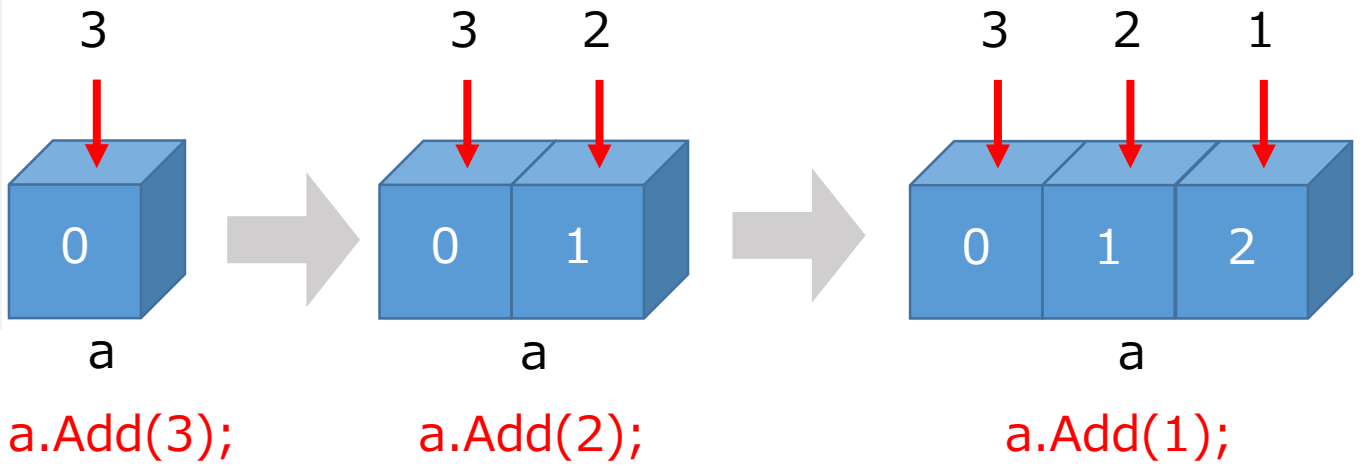
▼記述例

```
List<int> a = new List<int>();
a.Add(3);
a.Add(2);
a.Add(1);
Console.WriteLine(a[0]);
Console.WriteLine(a[1]);
Console.WriteLine(a[2]);
```

int型のリストの生成（中身は空）

▼実行結果

```
3
```



# Listの基本

## 基本的なListの使い方

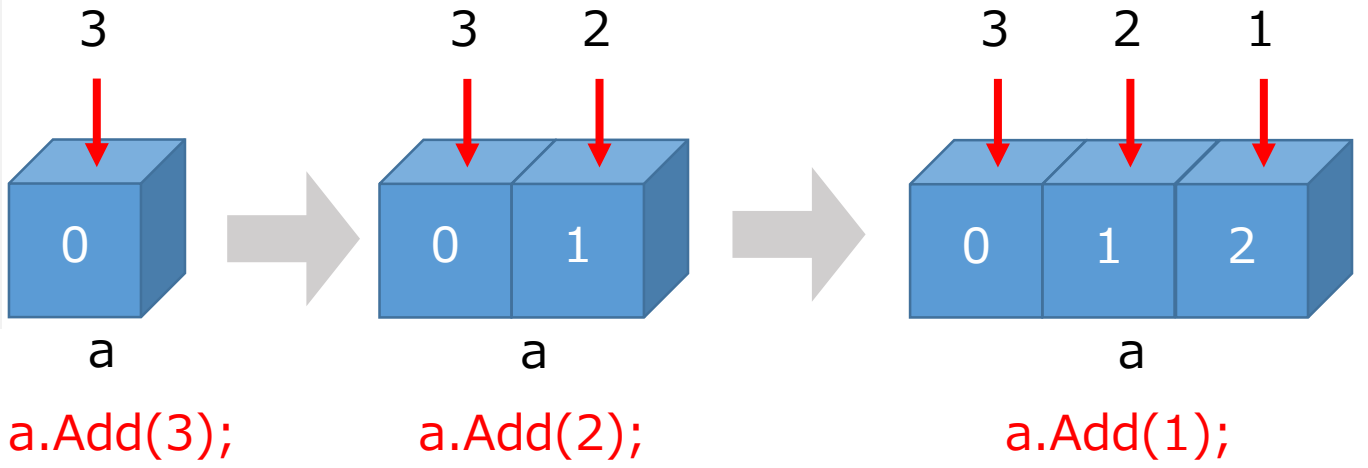
▼記述例

```
List<int> a = new List<int>();
a.Add(3);
a.Add(2);
a.Add(1);
Console.WriteLine(a[0]);
Console.WriteLine(a[1]);
Console.WriteLine(a[2]);
```

リストへのデータの追加

▼実行結果

```
3
2
1
```



# Listの基本

## 基本的なListの使い方

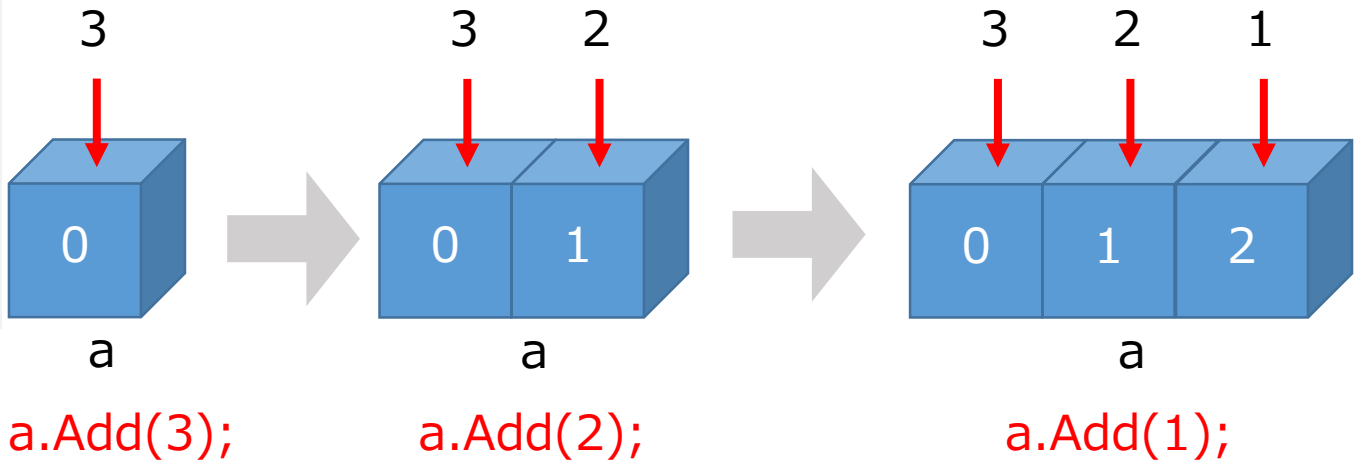
▼記述例

```
List<int> a = new List<int>();
a.Add(3);
a.Add(2);
a.Add(1);
Console.WriteLine(a[0]);
Console.WriteLine(a[1]);
Console.WriteLine(a[2]);
```

配列と同様にインデックスでアクセス

▼実行結果

```
3
2
1
```



# Listの基本

## 基本的なListの使い方

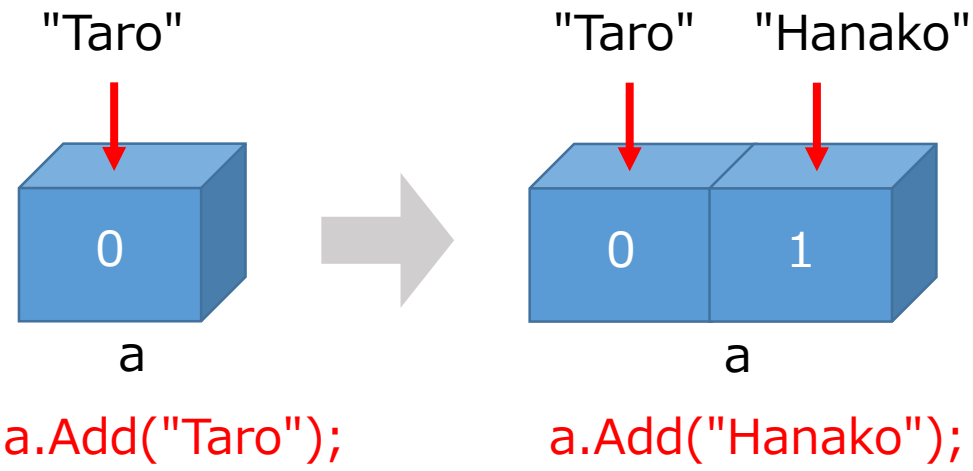
▼記述例

```
List<string> a = new List<string>();
a.Add("Taro");
a.Add("Hanako");
Console.WriteLine(a[0]);
Console.WriteLine(a[1]);
```

<>の中をstringに指定

▼実行結果

```
Taro
Hanako
```



# Listの基本

## 複数の値を一度に挿入

▼記述例

```
List<int> a = new List<int>{ 3,2,1 };
Console.WriteLine(a[0]);
Console.WriteLine(a[1]);
Console.WriteLine(a[2]);
```

{}の中に「,」で区切って代入

▼実行結果

```
3
2
1
```

# Listの応用

## Listの長さの取得

▼記述例

```
List<int> a = new List<int>{ 3,2,1 };
Console.WriteLine(a.Count);
```

→　**実行結果「3」**

▼記述例

```
List<string> a = new List<string>{ "Taro","Hanako" };
Console.WriteLine(a.Count);
```

→　**実行結果「2」**

# Listの応用

## Listとfor文

▼記述例

```
List<int> a = new List<int>{ 3,2,1 };
for(int i = 0; i < a.Count ; i++){
    Console.WriteLine("a[{0}]={1} ", i,a[i]);
}
```

▼実行結果

```
a[0]=3
a[1]=2
a[2]=1
```

# Listの応用

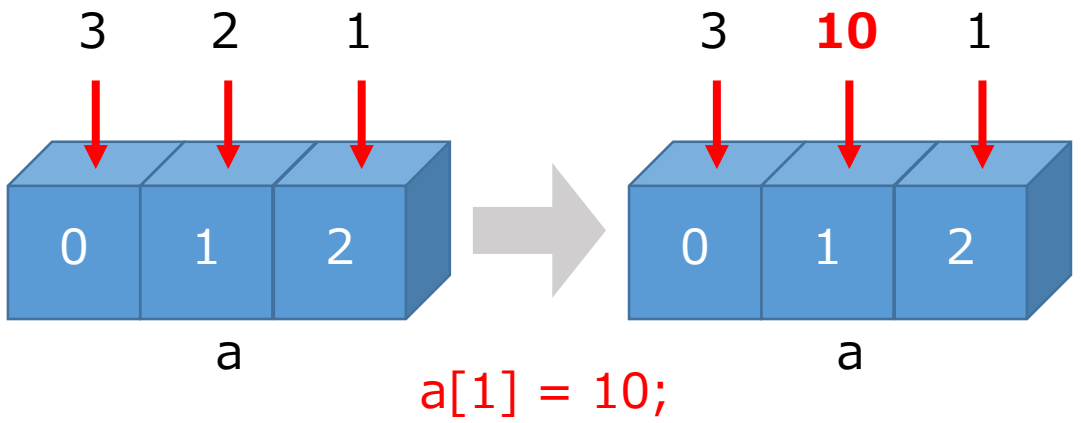
## Listの値の変更

▼記述例

```
List<int> a = new List<int>{ 3,2,1 };
a[1] = 10;  // 1番を10に変更
for(int i = 0; i < a.Count ; i++)
{
    Console.WriteLine("a[{0}]={1} ", i,a[i]);
}
```

▼実行結果

```
a[0]=3
a[1]=10
a[2]=1
```



# Listの応用

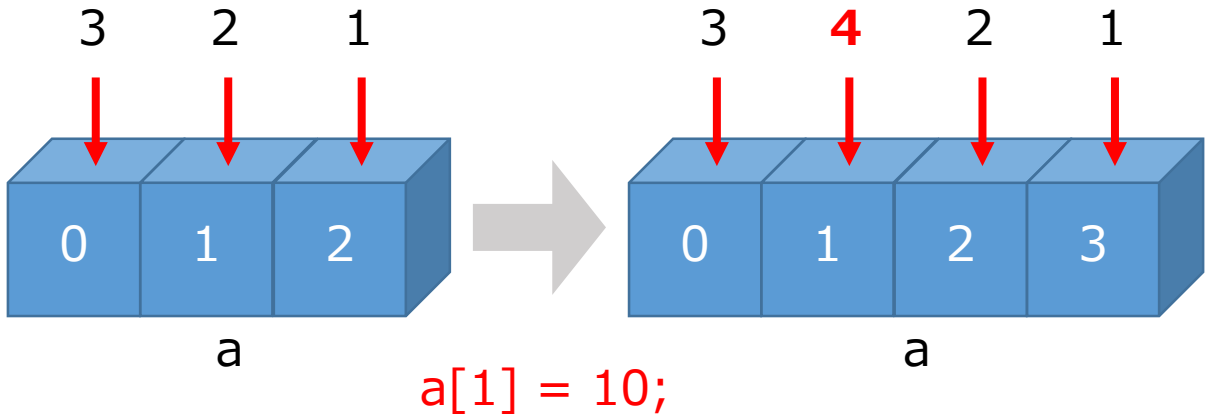
## Listへの値の挿入

▼記述例

```
List<int> a = new List<int>{ 3,2,1 };
a.Insert(1, 4);  // 1番目に4を挿入
for(int i = 0; i < a.Count ; i++)
{
    Console.WriteLine("a[{0}]={1} ", i,a[i]);
}
```

▼実行結果

```
a[0]=3
a[1]=4
a[2]=2
a[3]=1
```



# Listの応用

foreachとList

**▼記述例**

```
List<string> a = new List<string>{ "Taro","Hanako","Jiro" };
foreach (String s in a)
{
   Console.WriteLine(s);
}
```

**▼実行結果**

```
Taro
Hanako
Jiro
```

# Listの応用

## Listの要素の削除(Remove)

▼記述例

```
List<string> a = new List<string>{ "Taro","Hanako","Jiro" };
a.Remove("Taro");   //  "Taro"を削除
foreach (String s in a)
{
    Console.WriteLine(s);
}
```

▼実行結果

```
Hanako
Jiro
```

# Listの応用

## Listの要素の削除(RemoveAt)

▼記述例

```
List<string> a = new List<string>{ "Taro","Hanako","Jiro" };
a.RemoveAt(1);      //  1番目のデータを削除
foreach (String s in a)
{
    Console.WriteLine(s);
}
```

▼実行結果

```
Taro
Jiro
```

# Listの応用

## Listの検索

### ▼記述例

```
List<string> a = new List<string>{ "Taro","Hanako","Jiro" };
Console.WriteLine(a.IndexOf("Jiro"));
```

### ▼実行結果

```
2
```

# Listの使い方

## Listのクリア

### ▼記述例

```
List<int> a = new List<int>{ 3,2,1 };
a.Clear();  //  クリア
Console.WriteLine(a.Count);
```

### ▼実行結果

```
0
```

# 3.Dictionary

# Dictionaryの考え方

辞書（Dictionary）のように**キー（key）**と**値(Value)**の組み合わせでデータを管理

## Dictionaryの考え方

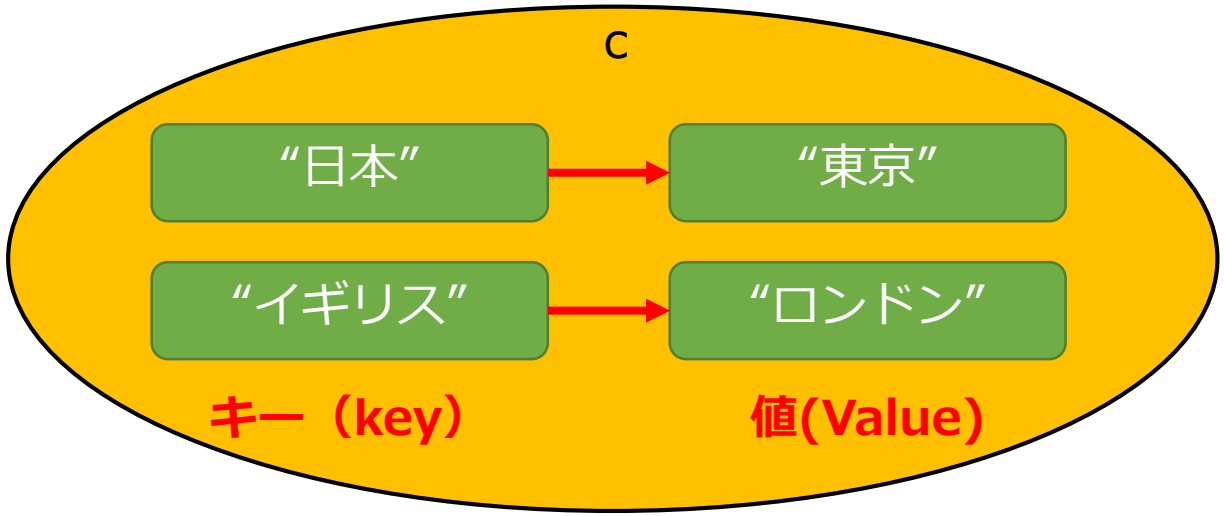
```
Dictionary<string, string> c = new Dictionary<string, string>();
```

**キーのデータ型**

**値のデータ型**

```
c["日本"] = "東京";
c["イギリス"] = "ロンドン";
```

**値の設定**



# Dictionaryの考え方

辞書（Dictionary）のように**キー（key）**と**値(Value)**の組み合わせでデータを管理

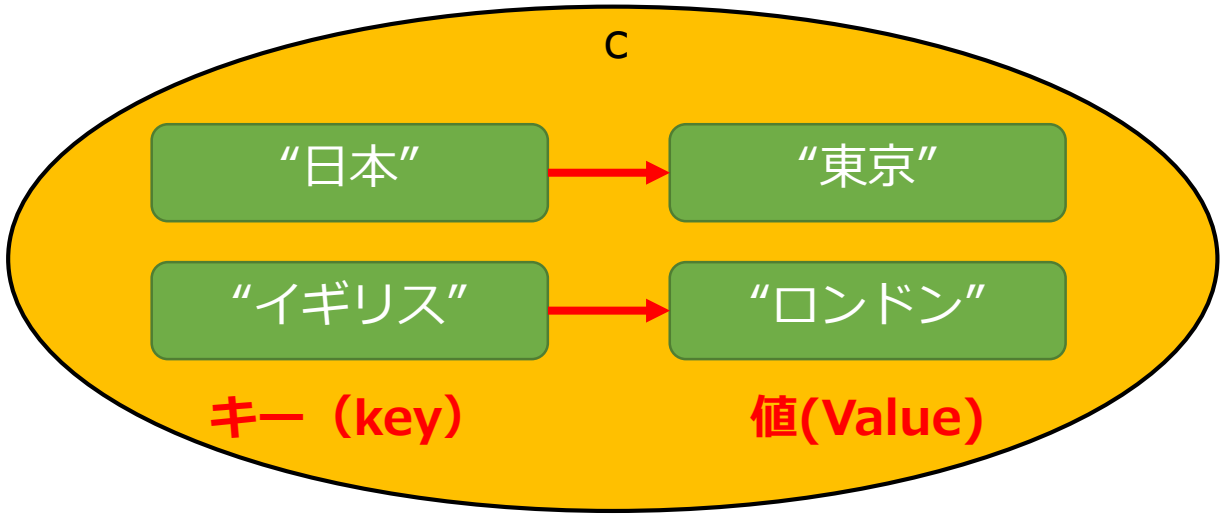
## Dictionaryの考え方

```
Dictionary<string, string> c = new Dictionary<string, string>();
```

**キーのデータ型**　**値のデータ型**

```
c["日本"] = "東京";
c["イギリス"] = "ロンドン";
```

**値の設定**




**ポイント**

**キー・値の型は自由に定義できる**

# Dictionaryの基本

## Dictionaryの基本的な使い方

**▼記述例**

```
Dictionary<string, string> c = new Dictionary<string, string>();
c["日本"] = "東京";
c["イギリス"] = "ロンドン";
Console.WriteLine(c["日本"]);
Console.WriteLine(c["イギリス"]);
```

**▼実行結果**

```
東京
ロンドン
```

# Dictionaryの基本

## Dictionaryのサイズの確認

▼記述例

```
Dictionary<string, string> c = new Dictionary<string, string>();
c["日本"] = "東京";
c["イギリス"] = "ロンドン";
c["フランス"] = "パリ";
Console.WriteLine(c.Count);
```

▼実行結果

```
3
```

# Dictionaryとループ

## Dictionaryのキーの取得

▼記述例

```
Dictionary<string, string> c = new Dictionary<string, string>();
c["日本"] = "東京";
c["イギリス"] = "ロンドン";
foreach (string s in c.Keys)
{
    Console.WriteLine(s);
}
```

▼実行結果

```
日本
イギリス
```

# Dictionaryとループ

## Dictionaryのキーの取得

▼記述例

```
Dictionary<string, string> c = new Dictionary<string, string>();
c["日本"] = "東京";
c["イギリス"] = "ロンドン";
foreach (string s in c.Keys)
{
    Console.WriteLine(s);
}
```

「Keys」プロパティでキーの一覧取得可能

▼実行結果

```
日本
イギリス
```

# Dictionaryとループ

## Dictionaryの値の取得

▼記述例

```
Dictionary<string, string> c = new Dictionary<string, string>();
c["日本"] = "東京";
c["イギリス"] = "ロンドン";
foreach (string s in c.Values)
{
    Console.WriteLine(s);
}
```

▼実行結果

```
東京
ロンドン
```

# Dictionaryとループ

## Dictionaryの値の取得

▼記述例

```
Dictionary<string, string> c = new Dictionary<string, string>();
c["日本"] = "東京";
c["イギリス"] = "ロンドン";
foreach (string s in c.Values)
{
    Console.WriteLine(s);
}
```

「Values」プロパティで値の一覧取得可能

▼実行結果

```
東京
ロンドン
```

# Dictionaryとループ

## Dictionaryのキー・値の取得

▼記述例

```
Dictionary<string, string> c = new Dictionary<string, string>();
c["日本"] = "東京";
c["イギリス"] = "ロンドン";
foreach(KeyValuePair<string, string> item in c) {
    Console.WriteLine("[{0}:{1}]", item.Key, item.Value);
}
```

▼実行結果

```
[日本:東京]
[イギリス:ロンドン]
```

# Dictionaryとループ

## Dictionaryの値の取得

▼記述例

```
Dictionary<string, string> c = new Dictionary<string, string>();
c["日本"] = "東京";
c["イギリス"] = "ロンドン";
foreach(KeyValuePair<string, string> item in c) {
    Console.WriteLine("[{0}:{1}]", item.Key, item.Value);
}
```

「KeyValuePair」型で要素の型を定義

キーの取得

値の取得

▼実行結果

```
[日本:東京]
[イギリス:ロンドン]
```

# Dictionaryとループ

## Dictionaryの要素の削除

▼記述例

```
Dictionary<string, string> capital = new Dictionary<string, string>();
capital["日本"] = "東京";
capital["イギリス"] = "ロンドン";
capital.Remove("イギリス"); //  削除
foreach(KeyValuePair<string, string> item in capital) {
    Console.WriteLine("[{0}:{1}]", item.Key, item.Value);
}
```

▼実行結果

```
[日本:東京]
```

# Dictionaryの応用

## Dictionaryの要素の削除

▼記述例

```
Dictionary<string, string> c = new Dictionary<string, string>();
c["日本"] = "東京";
c["イギリス"] = "ロンドン";
c.Remove("イギリス"); //  削除
foreach(KeyValuePair<string, string> item in c) {
    Console.WriteLine("[{0}:{1}]", item.Key, item.Value);
}
```

削除したい要素のキーを指定する

▼実行結果

```
[日本:東京]
```

# Dictionaryの応用

## Dictionaryのサイズの取得

**▼記述例**

```
Dictionary<string, string> c = new Dictionary<string, string>();
c["日本"] = "東京";
c["イギリス"] = "ロンドン";
Console.WriteLine(capital.Count);
```

**▼実行結果**

```
2
```

# Dictionaryの応用

## Dictionaryのクリア

**▼記述例**

```
Dictionary<string, string> c = new Dictionary<string, string>();
c["日本"] = "東京";
c["イギリス"] = "ロンドン";
c.Clear(); //  クリア
Console.WriteLine(c.Count);
```

**▼実行結果**

```
0
```

# Dictionaryのクリア

## Dictionaryのサイズの取得

**▼記述例**

```
Dictionary<string, string> c = new Dictionary<string, string>();
c["日本"] = "東京";
c["イギリス"] = "ロンドン";
c.Clear(); //  クリア
Console.WriteLine(c.Count);
```

**▼実行結果**

```
0
```

# 4.HashSet

# HashSetの考え方

ListのようにAddでデータの追加ができるがデータが重複せずインデックスの概念がない

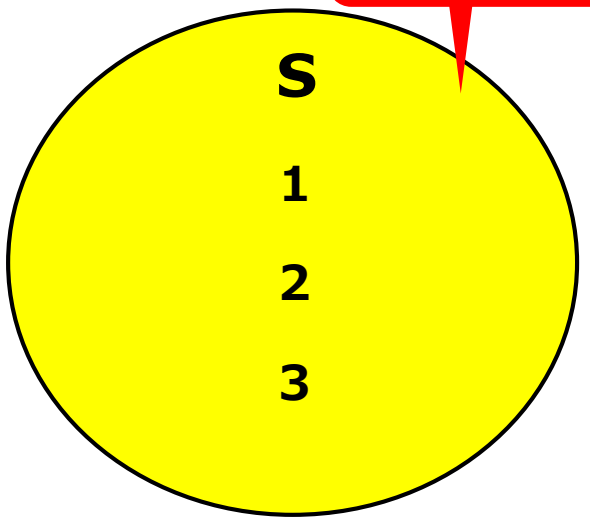
## HashSetの考え方

```
HashSet<int> s = new HashSet<int>();
```

キーのデータ型

同じ値を重複して登録できない

```
s.Add(1);
s.Add(1);
s.Add(2);
s.Add(3);
```



# HashSetの基本

## データの登録(Addメソッド)

▼記述例

```
HashSet<int> s = new HashSet<int>();
s.Add(1);
s.Add(1);
s.Add(2);
s.Add(3);
foreach (int i in s)
{
    Console.WriteLine(i);
}
```

▼実行結果

```
1
2
3
```

# HashSetの基本

## データの登録(Addメソッド)

▼記述例

```
HashSet<int> s = new HashSet<int>();
s.Add(1);
s.Add(1);
s.Add(2);
s.Add(3);
foreach (int i in s)
{
    Console.WriteLine(i);
}
```

データの登録はAddメソッド

▼実行結果

```
1
2
3
```

# HashSetの基本

## データの登録(Addメソッド)

▼記述例

```
HashSet<int> s = new HashSet<int>();
s.Add(1);
s.Add(1);
s.Add(2);
s.Add(3);
foreach (int i in s)
{
    Console.WriteLine(i);
}
```

インデックスの概念がないのでforeachで取得

▼実行結果

```
1
2
3
```

# HashSetの基本

## データの登録(Addメソッド)

▼記述例

```
HashSet<int> s = new HashSet<int>();
s.Add(1);
s.Add(1);
s.Add(2);
s.Add(3);
foreach (int i in s)
{
    Console.WriteLine(i);
}
```

▼実行結果

```
1
2
3
```

同じ値は重複できない

# HashSetの基本

## データを一括登録

▼記述例

```
HashSet<string> names = new HashSet<string>{ "Taro","Hanako","Jiro" };
foreach (string s in names)
{
    Console.WriteLine(s);
}
```

▼実行結果

```
Taro
Hanako
Jiro
```

# HashSetの基本

## データを一括登録

▼記述例

```
HashSet<string> names = new HashSet<string>{ "Taro","Hanako","Jiro" };
foreach (string s in names)
{
    Console.WriteLine(s);
}
```

{}で区切り「,」で区切る

▼実行結果

```
Taro
Hanako
Jiro
```

# HashSetの基本

## データのサイズの取得

**▼記述例**

```
HashSet<string> names = new HashSet<string>{ "Taro","Hanako","Jiro" };
Console.WriteLine(names.Count);
```

**▼実行結果**

```
3
```

# HashSetの集合演算

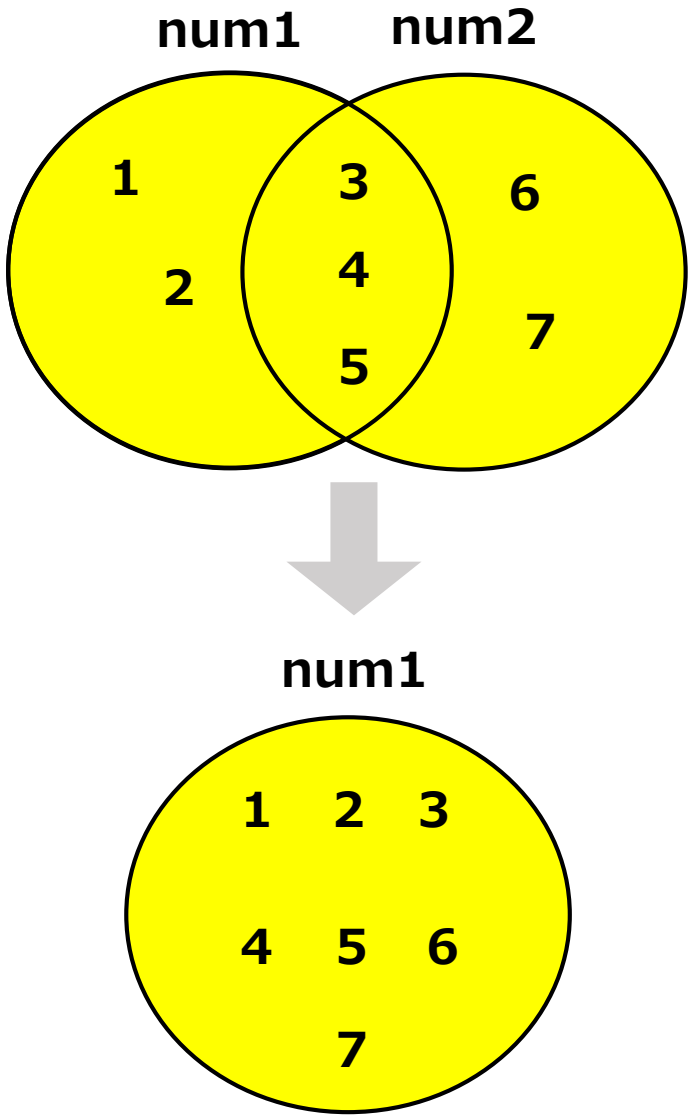
## 集合演算（和集合）

▼記述例

```
HashSet<int> nums1 = new HashSet<int>{ 1,2,3,4,5 };
HashSet<int> nums2 = new HashSet<int>{ 3,4,5,6,7 };
nums1.UnionWith(nums2); //  和集合
foreach (int s in nums1)
{
    Console.Write(s+" ");
}
```

▼実行結果

```
1 2 3 4 5 6 7
```



# HashSetの集合演算

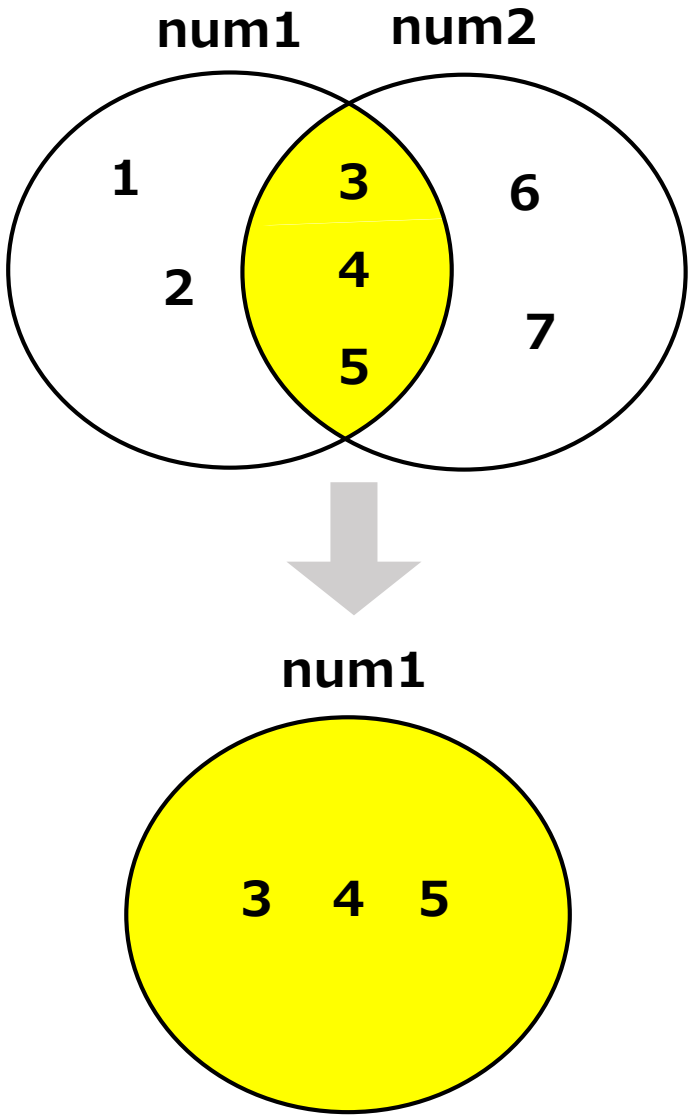
**集合演算（積集合）**

**▼記述例**

```
HashSet<int> nums1 = new HashSet<int>{ 1,2,3,4,5 };
HashSet<int> nums2 = new HashSet<int>{ 3,4,5,6,7 };
nums1.IntersectWith(nums2); //  積集合
foreach (int s in nums1)
{
    Console.Write(s+" ");
}
```

**▼実行結果**

```
3 4 5
```




nums1.IntersectWith(nums2);

# 5.複雑なコレクションの操作

# 複雑なコレクションの操作

Listの各要素がDictionaryのパターン

**・名簿のデータなどを保存するタイプの構造として便利**

**・データベースなどと組み合わせると便利**

**▼①リストの生成**

```
//  辞書のリスト
List<Dictionary<string, string>> list
                                = new List<Dictionary<string, string>>();
```

キーがstring型、値もstring型の辞書を値にもつリストを定義する

# 複雑なコレクションの操作

Dictionary型のデータを生成しリストに追加する

**▼②リストへのデータの追加**

```
//  辞書データの作成
Dictionary<string, string> data = new Dictionary<string, string>();
data["name"] =  "山田太郎";
data["age"] =  "18";
data["email"]="taro@mail.com";
```

名前・年齢・メールアドレスなどを登録

# 複雑なコレクションの操作

作成した辞書をリストに追加

| | |
|---|---|
| **name** | “山田太郎” |
| **age** | “18” |
| **email** | “taro@mail.com” |

| | |
|---|---|
| **name** | “佐藤花子” |
| **age** | “16” |
| **email** | “hanako@mail.com” |

……

**▼③データのリストへの追加**

```
//リストへ追加
list.Add(data);
```

キーがstring型、値もstring型の辞書を値にもつリストを定義する

# 複雑なコレクションの操作

**▼④登録したリストの出力**

```
foreach(Dictionary<string, string> item in list){
    Console.WriteLine("名前:{0}", item["name"]);
    Console.WriteLine("年齢:{0}", item["age"]);
    Console.WriteLine("メール:{0}", item["email"]);
}
```

**▼実行結果**

```
名前:山田太郎
年齢:18
メール:taro@mail.com
```

# 演習

# 演習問題１：変数と演算

以下の処理を実行するC#プログラムを考えてみましょう

- **整数型変数num1を用意し、10を代入する**
- **整数型変数num2を用意し、20を代入する**
- **num1とnum2の合計を整数型変数sumに代入し出力**

**▼実行結果**

```
30
```

# 演習問題１：変数と演算

以下の処理を実行するC#プログラムを考えてみましょう

- **整数型変数num1を用意し、10を代入する**
- **整数型変数num2を用意し、20を代入する**
- **num1とnum2の合計を整数型変数sumに代入し出力**

**▼正解コード**

```
int num1 = 10;
int num2 = 20;
int sum = num1 + num2;
Console.WriteLine(sum);
```

# 演習問題 2：ループ処理(for文)

以下の処理を実行するC#プログラムを考えてみましょう

**・2から5までを画面に出力する**

**※変数iの値を１つずつ増やしていくこと**

**▼実行結果**

```
2
3
4
5
```

# 演習問題 2：ループ処理(for文)

以下の処理を実行するC#プログラムを考えてみましょう

**・2から5までを画面に出力する**

**※変数iの値を１つずつ増やしていくこと**

**▼正解コード**

```
for(int i = 2; i < 6; i++){
    Console.WriteLine(i);
}
```

# 演習問題 2：ループ処理(for文)

以下の処理を実行するC#プログラムを考えてみましょう

**・2から5までを画面に出力する**

**※変数iの値を１つずつ増やしていくこと**

**▼正解コ**

開始値

継続条件

```
for(int i = 2; i < 6; i++){
    Console.WriteLine(i);
}
```

iの開始値を2、継続条件を「i<6」としてiに１を足しながらループを継続させる

# 6.例外処理

# 例外と例外処理

**例外 …** プログラムで起こってはいけないことが起こること

## 例外の発生と対処方法

・**例外が発生すると** … 通常はプログラムが異常終了をしてしまう

・**「例外処理」での対処** … 例外が発生した場合のプログラムのふるまいを記述すること

・**「例外処理」があると** … 例外が発生しても異常終了せずプログラムを続行できる

# 例外と例外処理

**例外処理** … 例外が発生してもプログラムが異常終了しないようにする

## 例外処理の記述方法

・**例外のキャッチ** … try ~ catchで例外をキャッチする

・**例外処理の実行** … 発生した例外に応じた処理を行う

・**finally内の処理の実行** … その他必要な事項がある場合には実行する（省略可能）

# 例外と例外処理

**例外クラス** … 例外発生時にインスタンス生成するクラス

## 主な例外と対応する例外クラスの種類

- **配列・リストの範囲外へのアクセス … IndexOutOfRangeException**

- **0出の割り算 … DivideByZeroException**

- **型変換の失敗 … FormatException**

# 例外と例外処理

**例外クラス** … 例外発生時にインスタンス生成するクラス

## 主な例外と対応する例外クラスの種類

- **配列・リストの範囲外へのアクセス … IndexOutOfRangeException**
- **0出の割り算 … DivideByZeroException**
- **型変換の失敗 … FormatException**

**ポイント**

**発生する例外の種類によって例外クラスは異なる**

# 例外の発生

## 例外が発生するパターン

▼記述例

```
int[] a = { 0, 1, 2 };
Console.WriteLine(a[3]);
```

▼実行結果

```
ハンドルされていない例外: System.IndexOutOfRangeException: インデックスが配列の境界外です。
場所 ConsoleApp1.Program.Main(String[] args) 場所 C:¥Users¥shift¥source¥repos¥ConsoleApp1¥ConsoleApp1¥Program.cs:行 15
```

# 例外の発生

## 例外が発生するパターン

▼記述例

```
int[] a = { 0, 1, 2 };
Console.WriteLine(a[3]);
```

配列の範囲外にアクセス

▼実行結果

```
ハンドルされていない例外: System.IndexOutOfRangeException: インデックスが配列の境界外です。
場所 ConsoleApp1.Program.Main(String[] args) 場所 C:¥Users¥shift¥source¥repos¥ConsoleApp1¥ConsoleApp1¥Program.cs:行 15
```

発生した例外クラス

例外が発生し終了

# 例外処理

## 例外処理の例（配列の範囲外へのアクセス）

▼記述例

```
try
{
    int[] a = { 0, 1, 2 };
    Console.WriteLine(a[3]);
}catch(IndexOutOfRangeException e) {
    //  配列の範囲外
    Console.WriteLine(e.Message);
}
```

▼実行結果

```
インデックスが配列の境界外です。
```

# 例外処理

## 例外処理の例（配列の範囲外へのアクセス）

▼記述例

```
try
{
    int[] a = { 0, 1, 2 };
    Console.WriteLine(a[3]);
}catch(IndexOutOfRangeException e) {
    //  配列の範囲外
    Console.WriteLine(e.Message);
}
```

例外が発生

▼実行結果

```
インデックスが配列の境界外です。
```

# 例外処理

## 例外処理の例（配列の範囲外へのアクセス）

▼記述例

```
try
{
    int[] a = { 0, 1, 2 };
    Console.WriteLine(a[3]);
}catch(IndexOutOfRangeException e) {
    //  配列の範囲外
    Console.WriteLine(e.Message);
}
```

例外をキャッチ

▼実行結果

```
インデックスが配列の境界外です。
```

# 例外処理

## 例外処理の例（配列の範囲外へのアクセス）

▼記述例

```
try
{
    int[] a = { 0, 1, 2 };
    Console.WriteLine(a[3]);
}catch(IndexOutOfRangeException e) {
    //  配列の範囲外
    Console.WriteLine(e.Message);
}
```

例外処理の実行

▼実行結果

```
インデックスが配列の境界外です。
```

# 例外処理

## 例外処理の例（配列の範囲外へのアクセス）

▼記述例

```
try
{
    int[] a = { 0, 1, 2 };
    Console.WriteLine(a[3]);
}catch(IndexOutOfRangeException e) {
    //  配列の範囲外
    Console.WriteLine(e.Message);
}
```

例外処理の実行

▼実行結果

```
インデックスが配列の境界外です。
```

e.Message

**ポイント**

例外処理は例外が発生した場合のみ実行される

# 例外処理

## 例外処理（ゼロでの割り算）

▼記述例

```
try
{
    int a = 10, b = 0;
    Console.WriteLine(a / b);
}catch(DivideByZeroException e) {
    //  ゼロでの割り算
     Console.WriteLine(e.Message);
}
```

▼実行結果

```
0 で除算しようとしました。
```

# 例外処理

## 例外処理（ゼロでの割り算）

▼記述例

```
try
{
    int a = 10, b = 0;
    Console.WriteLine(a / b);
}catch(DivideByZeroException e) {
    //  ゼロでの割り算
     Console.WriteLine(e.Message);
}
```

例外の種類により例外クラスは異なる

▼実行結果

```
0 で除算しようとしました。
```

# 例外処理

## Exceptionクラス

▼記述例

```
try
{
    int a = 10, b = 0;
    Console.WriteLine(a / b);
}catch(Exception e) {
    //  ゼロでの割り算
     Console.WriteLine(e.Message);
}
```

▼実行結果

```
0 で除算しようとしました。
```

すべてに例外クラスはExceptionクラスを継承しているので、Exceptionクラスですべての例外をキャッチできる

# 例外処理

## finally(例外が発生する場合)

▼記述例

```
try
{
    int a = 10,b = 0;
    Console.WriteLine(a / b);    // 例外発生
}catch(Exception e) {
     Console.WriteLine(e.Message);
}finally{
    Console.WriteLine("終了");
}
```

▼実行結果

```
0 で除算しようとしました。
終了
```

finallyの部分の処理は例外処理の実行後に実行される。

# 例外処理

## finally(例外が発生する場合)

▼記述例

```
try
{
    int a = 10,b = 0;
    Console.WriteLine(a / b);    // 例外発生
}catch(Exception e) {
     Console.WriteLine(e.Message);
}finally{
    Console.WriteLine("終了");
}
```

例外処理の後で実行

▼実行結果

```
0 で除算しようとしました。
終了
```

finallyの処理

finallyの部分の処理は例外処理の実行後に実行される。

# 例外処理

## finally(例外が発生する場合)

▼記述例

```
try
{
    int a = 10,b = 5;
    Console.WriteLine(a / b);    // 例外なし
}catch(Exception e) {
     Console.WriteLine(e.Message);
}finally{
    Console.WriteLine("終了");
}
```

▼実行結果

```
2
終了
```

finallyの部分の処理は例外が発生しなくても必ず最後に実行される。

# 例外処理

## finally(例外が発生する場合)

▼記述例

```
try
{
    int a = 10,b = 5;
    Console.WriteLine(a / b);    // 例外なし
}catch(Exception e) {
    Console.WriteLine(e.Message);
}finally{
    Console.WriteLine("終了");
}
```

実行されない

最後に実行される

▼実行結果

```
2
終了
```

finallyの部分の処理は例外が発生しなくても必ず最後に実行される。