C# 기초 콘솔

유건곤 강사 제작교안

수업시에 진행할 c#기초강의 내용을 제가 중요한부분 적어놨습니다.

기초내용을 교안으로 습득하고 추후 실습위주로 수업을 진행합니다.

핵심적인 기초내용들을 습득해서 이해하고 직접 실습을 같이하면서

더 빠르게 게임 개발을 할 수 있는 방법으로 진행할생각입니다.

1. 변수

설명:

- 변수는 데이터를 저장하는 이름이 있는 공간입니다.
- 데이터를 프로그램에서 동적으로 처리할 수 있게 합니다.
- 변수의 데이터 타입에 따라 저장할 수 있는 값의 종류와 크기가 결정됩니다.

변수는 데이터를 저장할 "그릇"과 같으며, 이 그릇은 미리 크기(데이터 타입)를 지정해야 한다

```
csharp
코드 복사
// 변수 선언: 데이터 타입과 변수 이름을 지정합니다.
int age; // 정수형 변수 age 선언
age = 25; // 변수에 값 저장

Console.WriteLine(age); // 변수에 저장된 값을 출력
```

2. 리터럴 사용하기

설명:

- 리터럴은 코드에서 고정된 값을 의미합니다.
- 예를 들어, 숫자(10), 문자('A'), 문자열("Hello") 등이 리터럴입니다.

리터럴은 "변하지 않는 값"으로, 변수에 할당하거나 직접 사용할 수 있다

```
csharp
코드 복사
// 리터럴: 코드에서 고정된 값을 의미합니다.
int number = 10;      // 정수형 리터럴
double pi = 3.14;     // 실수형 리터럴
char letter = 'A';    // 문자 리터럴
string name = "Alice"; // 문자열 리터럴

Console.WriteLine(number); // 출력: 10
Console.WriteLine(pi);     // 출력: 3.14
Console.WriteLine(letter); // 출력: A
Console.WriteLine(name);   // 출력: Alice
```

---

3. 변수를 만들어 값 저장 후 사용하기

설명:

- 변수를 선언하고 나중에 값을 저장해서 사용할 수 있습니다.
- 변수를 초기화하지 않으면 기본값이 저장됩니다.

값을 나중에 설정할 수 있다, 사용 전에 값을 초기화해야 한다

```
csharp
코드 복사
// 변수 선언 후 값 저장
string greeting; // 문자열 변수를 선언
greeting = "Hello, World!"; // 변수에 값을 저장

// 변수의 값을 사용
```

```
Console.WriteLine(greeting); // 출력: Hello, World!
```

---

4. 변수 선언과 동시에 초기화하기

설명:

- 변수 선언과 초기화를 한 번에 수행하면 코드가 더 간결해집니다.
- 이렇게 하면 초기값을 즉시 사용할 수 있습니다.

초기화는 변수 사용 시 오류를 방지하며, 선언과 초기화를 동시에 하는 것이 일반적

```
csharp
코드 복사
// 변수 선언과 초기화를 한 번에 수행
int score = 100; // 정수형 변수 선언과 동시에 100으로 초기화
double temperature = 36.5; // 실수형 변수 선언과 초기화
string city = "Seoul"; // 문자열 변수 선언과 초기화

// 변수 출력
Console.WriteLine(score); // 출력: 100
Console.WriteLine(temperature); // 출력: 36.5
Console.WriteLine(city); // 출력: Seoul
```

---

5. 형식이 같은 변수 여러 개를 한 번에 선언하기

설명:

- 같은 데이터 타입의 변수를 쉼표로 구분하여 선언할 수 있습니다.
- 가독성을 높이고, 관련 변수를 그룹화하는 데 유용합니다.

이 방식은 간단해 보이지만, 각 변수의 값이 독립적

```
csharp
코드 복사
// 같은 데이터 타입의 변수를 쉼표로 구분해 한 번에 선언
int x = 10, y = 20, z = 30; // 정수형 변수 x, y, z를 선언하고 각각 초기화
```

```
// 출력
Console.WriteLine(x); // 출력: 10
Console.WriteLine(y); // 출력: 20
Console.WriteLine(z); // 출력: 30
```

---

6. 상수 사용하기

설명:

- `const` 키워드를 사용하면 값을 변경할 수 없는 상수를 선언할 수 있습니다.
- 상수는 프로그램 내에서 변하지 않는 중요한 값을 정의할 때 사용합니다.

상수는 "고정된 값"으로, 중요한 값이 실수로 변경되는 것을 방지한다

```
csharp
코드 복사
// 상수: 값을 변경할 수 없는 변수
const double Pi = 3.14159; // 상수 Pi 선언 및 초기화
const int MaxScore = 100;  // 정수형 상수 선언

// 출력
Console.WriteLine("Pi: " + Pi); // 출력: Pi: 3.14159
Console.WriteLine("Max Score: " + MaxScore); // 출력: Max Score: 100

// Pi = 3.14; // 오류 발생! 상수는 값을 변경할 수 없습니다.
```

C#에서 숫자 데이터 형식을 사용

---

1. 숫자 데이터 형식

- 숫자 데이터 형식은 정수와 실수를 저장할 때 사용됩니다.
- C#은 다양한 숫자 데이터 형식을 제공하여 메모리 사용량과 값의 범위를 효과적으로 관리할 수 있습니다.

```
csharp
코드 복사
```

```
// 숫자 데이터 형식: 정수와 실수를 다룰 때 사용하는 다양한 타입
int integerNum = 10;      // 정수 데이터
float floatNum = 3.14f;   // 단정밀도 실수
double doubleNum = 3.14159; // 배정밀도 실수

Console.WriteLine(integerNum); // 출력: 10
Console.WriteLine(floatNum);   // 출력: 3.14
Console.WriteLine(doubleNum);  // 출력: 3.14159
```

---

2. 정수 데이터 형식

설명:

- 정수 데이터 형식은 소수점 없이 숫자를 저장하며, 음수와 양수를 모두 다룹니다.
- 대표적인 정수 형식: `int`, `long`.

```
csharp
코드 복사
// 정수 데이터 형식: 소수점이 없는 숫자를 표현
int intValue = -100;    // 4바이트 크기의 정수
long longValue = 1234567890L; // 8바이트 크기의 정수

Console.WriteLine(intValue);  // 출력: -100
Console.WriteLine(longValue); // 출력: 1234567890
```

---

3. 부호 있는 정수 데이터 형식

설명:

- 부호 있는 정수는 음수와 양수를 모두 표현합니다.
- `sbyte`, `short`, `int`, `long` 등은 부호 있는 정수 데이터 형식입니다.

```
csharp
코드 복사
// 부호 있는 정수: 음수와 양수를 모두 표현 가능
sbyte signedByte = -50;  // 1바이트 크기
```

```
short signedShort = -32000; // 2바이트 크기
int signedInt = -2000000000; // 4바이트 크기

Console.WriteLine(signedByte);   // 출력: -50
Console.WriteLine(signedShort);  // 출력: -32000
Console.WriteLine(signedInt);    // 출력: -2000000000
```

---

4. 부호 없는 정수 데이터 형식

설명:

- 부호 없는 정수는 0 이상의 양수만 저장할 수 있습니다.
- `byte`, `ushort`, `uint`, `ulong` 등이 해당됩니다.

```
csharp
코드 복사
// 부호 없는 정수: 0 이상의 정수만 표현 가능
byte unsignedByte = 255;   // 1바이트 크기
ushort unsignedShort = 65000; // 2바이트 크기
uint unsignedInt = 4000000000; // 4바이트 크기

Console.WriteLine(unsignedByte);   // 출력: 255
Console.WriteLine(unsignedShort);  // 출력: 65000
Console.WriteLine(unsignedInt);    // 출력: 4000000000
```

---

5. 실수 데이터 형식

설명:

- 실수 데이터 형식은 소수점을 포함한 숫자를 저장합니다.
- `float`, `double`, `decimal`이 주로 사용됩니다.
  - **`float`**: 단정밀도 (정밀도 낮음, 메모리 적게 사용).
  - **`double`**: 배정밀도 (정밀도 높음, 일반적으로 사용).
  - **`decimal`**: 고정밀도 (금융 계산에 사용).

```
csharp
코드 복사
// 실수 데이터 형식: 소수점을 포함한 숫자를 표현
float singlePrecision = 3.14f;    // 단정밀도 실수 (4바이트)
double doublePrecision = 3.1415926535; // 배정밀도 실수 (8바이트)
decimal highPrecision = 3.14159265358979323846264338 33m; // 고정밀도 (16바이트)

Console.WriteLine(singlePrecision); // 출력: 3.14
Console.WriteLine(doublePrecision); // 출력: 3.1415926535
Console.WriteLine(highPrecision);   // 출력: 3.14159265358979323846264338 33
```

---

6. 숫자 형식의 리터럴 값에 접미사 붙이기

설명:

- 숫자 리터럴은 접미사를 사용하여 데이터 형식을 명시할 수 있습니다.
  - `L` : `long` 형식
  - `f` : `float` 형식
  - `m` : `decimal` 형식

```
csharp
코드 복사
// 접미사 사용: 숫자의 데이터 형식을 명시
int integerValue = 100;     // 기본 정수형 (int)
long longValue = 100L;      // 정수형 (long)
float floatValue = 3.14f;   // 실수형 (float)
double doubleValue = 3.14;  // 기본 실수형 (double)
decimal decimalValue = 3.14m; // 고정밀도 실수형 (decimal)

Console.WriteLine(integerValue);  // 출력: 100
Console.WriteLine(longValue);     // 출력: 100
Console.WriteLine(floatValue);    // 출력: 3.14
Console.WriteLine(doubleValue);   // 출력: 3.14
```

```
Console.WriteLine(decimalValue);  // 출력: 3.14
```

요약

- **정수**: `int`, `long`, `byte`, `sbyte` 등.
- **실수**: `float`, `double`, `decimal`.
- **부호 유무**: 부호 있는 데이터(`int`, `sbyte`)와 부호 없는 데이터(`uint`, `byte`).
- **리터럴 접미사**: 숫자의 데이터 형식을 명시 (`f`, `L`, `m` 등).

위 코드를 직접 실행해 보며 데이터 형식별 특성을 이해하세요! 😊

숫자 이외의 데이터 형식을 다루는 C#

---

1. 문자 데이터 형식: `char`

설명

- `char`는 단일 문자를 저장하는 데이터 형식입니다.
- 2바이트 크기로 Unicode 표준을 사용합니다.

```
csharp
코드 복사
// char 형식: 단일 문자를 표현
char letter = 'A';  // 문자 'A' 저장
char symbol = '#';  // 특수 기호 저장
char number = '9';  // 숫자 형태의 문자 저장 (문자 '9', 숫자 9 아님)

Console.WriteLine(letter);  // 출력: A
Console.WriteLine(symbol);  // 출력: #
Console.WriteLine(number);  // 출력: 9
```

---

2. 문자열 데이터 형식: `string`

설명

- `string`은 텍스트를 저장하는 데이터 형식입니다.
- 0개 이상의 문자로 구성된 문자열을 다룹니다.

```
csharp
코드 복사
// string 형식: 여러 문자를 저장
string greeting = "Hello, World!";  // 문자열 저장
string name = "Alice";  // 이름 저장

Console.WriteLine(greeting); // 출력: Hello, World!
Console.WriteLine(name);     // 출력: Alice
```

---

3. 논리 데이터 형식: `bool`

설명

- `bool`은 참(`true`) 또는 거짓(`false`) 값을 저장합니다.
- 조건문 및 논리 연산에 사용됩니다.

```
csharp
코드 복사
// bool 형식: 참(True) 또는 거짓(False)
bool isRunning = true;  // 프로그램 실행 상태
bool isFinished = false; // 프로그램 종료 상태

Console.WriteLine(isRunning);  // 출력: True
Console.WriteLine(isFinished); // 출력: False
```

---

4. 변하지 않는 값: 상수

설명

- `const` 키워드는 값이 변경되지 않는 변수를 선언할 때 사용합니다.
- 상수는 한 번 초기화된 후 값을 변경할 수 없습니다.

```
csharp
코드 복사
// const: 변하지 않는 값을 정의
const double Pi = 3.14159;  // 원주율
const int MaxScore = 100;   // 최대 점수

Console.WriteLine(Pi);      // 출력: 3.14159
Console.WriteLine(MaxScore); // 출력: 100
// Pi = 3.14; // 오류! 상수 값은 변경할 수 없습니다.
```

---

5. 닷넷 데이터 형식

설명

- 닷넷은 `int`, `double`과 같은 기본 데이터 형식에 대한 **시스템 형식**을 제공합니다.
- 예: `System.Int32`, `System.Double`.

```
csharp
코드 복사
// 닷넷 형식: 기본 형식의 닷넷 표현
System.Int32 number = 123;   // int의 닷넷 형식
System.String text = "Hello"; // string의 닷넷 형식
System.Boolean flag = true;  // bool의 닷넷 형식

Console.WriteLine(number); // 출력: 123
Console.WriteLine(text);   // 출력: Hello
Console.WriteLine(flag);   // 출력: True
```

---

6. 래퍼 형식

설명

- 래퍼 형식은 기본 데이터 형식을 클래스 형태로 감싸서 객체로 취급할 수 있게 합니다.
- 예: `int`는 `System.Int32`로 사용 가능하며, 메서드와 속성을 제공합니다.

```
csharp
코드 복사
// int 래퍼 형식의 메서드 활용
int number = 123;
string numberAsString = number.ToString(); // 정수를 문자열로 변환

// bool 래퍼 형식의 메서드 활용
bool flag = true;
string flagAsString = flag.ToString(); // 논리값을 문자열로 변환

Console.WriteLine(numberAsString); // 출력: "123"
Console.WriteLine(flagAsString);   // 출력: "True"
```

강의 요약

1. `char`: 단일 문자 (예: 'A').
2. `string`: 문자열 데이터 (예: "Hello, World!").
3. `bool`: 논리값, `true` 또는 `false`.
4. **상수**: 변경 불가능한 값.
5. **닷넷 데이터 형식**: 모든 기본 형식의 닷넷 시스템 이름.
6. **래퍼 형식**: 기본 데이터 형식을 객체처럼 다루는 방법.

이 코드와 설명을 바탕으로 실습을 진행하며 데이터 형식을 익히세요! 😊

---

1. 문자열 입력 관련 메서드

설명:

- 사용자의 입력은 `Console.ReadLine()` 메서드를 사용하여 문자열로 받아옵니다.
- 입력된 값은 변수에 저장하여 처리할 수 있습니다.

예제:

```
csharp
코드 복사
// 사용자 입력을 문자열로 받기
Console.Write("이름을 입력하세요: ");
string userName = Console.ReadLine(); // 사용자로부터 입력 받기

Console.WriteLine($"안녕하세요, {userName}님!"); // 입력값 출력
```

`Console.ReadLine()`은 항상 문자열로 입력을 받으므로, 숫자 등 다른 데이터 형식으로 사용할 경우 변환이 필요

---

2. 형식 변환

설명:

- 입력된 문자열을 숫자 등 다른 데이터 형식으로 변환하려면 `Convert` 클래스나 `int.Parse()` 같은 메서드를 사용합니다.
- 올바른 변환을 위해 입력 값 검증이 필요합니다.

예제:

```
csharp
코드 복사
// 문자열을 정수로 변환
Console.Write("나이를 입력하세요: ");
string input = Console.ReadLine(); // 사용자로부터 입력 받기
int age = int.Parse(input); // 문자열을 정수로 변환

Console.WriteLine($"내년에는 {age + 1}살이 되겠군요!"); // 변환된 값 사용
```

---

3. 이진수 다루기

설명:

- 이진수 처리는 `Convert` 클래스를 사용해 문자열에서 이진수를 정수로 변환하거나, 정수를 이진수 문자열로 변환할 수 있습니다.

예제:

```
csharp
코드 복사
// 이진수를 정수로 변환
Console.Write("2진수를 입력하세요: ");
string binaryInput = Console.ReadLine();
int decimalValue = Convert.ToInt32(binaryInput, 2); // 2진수 -> 10진수 변환

// 정수를 이진수로 변환
string binaryOutput = Convert.ToString(decimalValue, 2); // 10진수 -> 2진수 변환

Console.WriteLine($"입력한 이진수: {binaryInput}");
Console.WriteLine($"10진수로 변환: {decimalValue}");
Console.WriteLine($"다시 이진수로 변환: {binaryOutput}");
```

이진수를 다루는 방법은 개발자의 효율적인 계산에 매우 유용

---

4. `var` 키워드로 암시적으로 형식화된 로컬 변수 만들기

설명:

- `var` 키워드는 컴파일러가 변수의 데이터 형식을 자동으로 추론하게 합니다.
- 선언 시 반드시 초기화해야 합니다.

예제:

```
csharp
코드 복사
// var를 사용하여 변수 선언
var name = "Alice"; // 문자열로 추론
var age = 25;       // 정수로 추론
```

```
var isStudent = true; // 논리값으로 추론

Console.WriteLine($"이름: {name}, 나이: {age}, 학생 여부: {isStudent}");
```

`var`는 읽기 쉽고 유연하지만, 남용하지 않도록 타입 추론을 명확히 이해

---

5. 변수의 기본값을 `default` 키워드로 설정하기

설명:

- `default` 키워드를 사용하면 데이터 형식에 따라 기본값을 설정할 수 있습니다.
- 예: 숫자는 `0`, 문자열은 `null`, 논리값은 `false`.

예제:

```
csharp
코드 복사
// default 키워드를 사용한 기본값 설정
int defaultInt = default;         // 기본값: 0
string defaultString = default;   // 기본값: null
bool defaultBool = default;       // 기본값: false

Console.WriteLine($"정수 기본값: {defaultInt}");      // 출력: 0
Console.WriteLine($"문자열 기본값: {defaultString}"); // 출력: (null)
Console.WriteLine($"논리값 기본값: {defaultBool}");   // 출력: False
```

강의 팁:

`default`는 변수를 초기화하지 않을 경우 발생하는 오류를 방지하는 데 유용

강의 요약

1. **문자열 입력 메서드**: 사용자로부터 값을 입력받고 처리.
2. **형식 변환**: 문자열을 정수 등으로 변환하여 숫자 연산 가능.
3. **이진수 다루기**: 이진수 변환으로 수학적 연산 확장.
4. `var` **키워드**: 타입 추론을 통해 선언과 초기화를 간결하게.

5. `default` **키워드**: 데이터 형식에 맞는 기본값을 설정.

Tip: 각 기능을 실습해 보면서 다양한 입력을 처리하는 방법을 익히세요! 😊

---

1. 연산자

설명:

- 연산자는 값을 계산하거나 조작할 때 사용됩니다.
- 단항, 산술, 관계형, 논리, 비트 연산자 등 다양한 종류가 있습니다.

예제:

```
csharp
코드 복사
int a = 5, b = 3;
int sum = a + b; // 산술 연산자 사용
bool isEqual = (a == b); // 관계형 연산자 사용

Console.WriteLine($"합: {sum}"); // 출력: 8
Console.WriteLine($"a와 b가 같은가? {isEqual}"); // 출력: False
```

---

2. 단항 연산자

설명:

- 단항 연산자는 피연산자 하나에 적용됩니다.
- `+`, `-`, `!`, `~` 등이 포함됩니다.

예제:

```
csharp
코드 복사
int number = 5;
Console.WriteLine(+number); // 양수 출력: 5
Console.WriteLine(-number); // 음수 출력: -5
```

```
bool flag = true;
Console.WriteLine(!flag); // 논리 부정: False
```

---

3. 변환 연산자: `()` 기호로 데이터 형식 변환하기

설명:

- `()`를 사용해 데이터 형식을 명시적으로 변환합니다.

예제:

```
csharp
코드 복사
double pi = 3.14;
int integerPi = (int)pi; // 실수를 정수로 변환

Console.WriteLine(integerPi); // 출력: 3
```

Tip: 실수에서 정수로 변환 시 소수점이 버려짐을 주의하세요.

---

4. 산술 연산자

설명:

- `+`, `-`, `*`, `/`, `%`로 덧셈, 뺄셈, 곱셈, 나눗셈, 나머지 연산을 수행합니다.

예제:

```
csharp
코드 복사
int a = 10, b = 3;
Console.WriteLine(a + b); // 덧셈: 13
Console.WriteLine(a - b); // 뺄셈: 7
Console.WriteLine(a * b); // 곱셈: 30
Console.WriteLine(a / b); // 나눗셈: 3
Console.WriteLine(a % b); // 나머지: 1
```

---

5. 문자열 연결 연산자

설명:

- `+`를 사용해 문자열을 연결할 수 있습니다.

예제:

```
csharp
코드 복사
string firstName = "Alice";
string lastName = "Smith";

Console.WriteLine(firstName + " " + lastName); // 출력: Alice Smith
```

---

6. 할당 연산자와 증감 연산자 사용하기

7. 할당 연산자

설명:

- 값을 변수에 저장합니다. `=`, `+=`, `=`, `=`, `/=`, `%=` 등이 포함됩니다.

예제:

```
csharp
코드 복사
int a = 10;
a += 5; // a = a + 5
Console.WriteLine(a); // 출력: 15
```

8. 증감 연산자

설명:

- `++`와 `-`를 사용해 값을 1씩 증가하거나 감소시킵니다.

예제:

```
csharp
코드 복사
```

```
int b = 3;
b++; // 후위 증가
Console.WriteLine(b); // 출력: 4

--b; // 전위 감소
Console.WriteLine(b); // 출력: 3
```

---

9. 관계형 연산자와 논리 연산자 사용하기

10. 관계형 연산자

설명:

- 두 값을 비교하여 관계를 평가합니다. (`==`, `!=`, `<`, `>`, `<=`, `>=`)

예제:

```
csharp
코드 복사
int x = 5, y = 10;
Console.WriteLine(x < y); // 출력: True
Console.WriteLine(x == y); // 출력: False
```

11. 논리 연산자

설명:

- 논리값을 결합하거나 반전합니다. (`&&`, `||`, `!`)

예제:

```
csharp
코드 복사
bool a = true, b = false;
Console.WriteLine(a && b); // AND: False
Console.WriteLine(a || b); // OR: True
Console.WriteLine(!a);     // NOT: False
```

---

12. 비트 연산자와 시프트 연산자로 비트 단위 연산하기

13. 비트 연산자

설명:

- 비트 단위로 AND(`&`), OR(`|`), XOR(`^`), NOT(`~`)를 수행합니다.

예제:

```
csharp
코드 복사
int x = 5; // 0101
int y = 3; // 0011

Console.WriteLine(x & y); // AND: 1 (0001)
Console.WriteLine(x | y); // OR: 7 (0111)
Console.WriteLine(x ^ y); // XOR: 6 (0110)
Console.WriteLine(~x);    // NOT: -6
```

14. 시프트 연산자

설명:

- 비트를 좌우로 이동시킵니다. (`<<`, `>>`)

예제:

```
csharp
코드 복사
int value = 4; // 0100
Console.WriteLine(value << 1); // 왼쪽 이동: 8 (1000)
Console.WriteLine(value >> 1); // 오른쪽 이동: 2 (0010)
```

---

15. 기타 연산자

설명:

- `?:` (삼항 연산자): 조건문을 간단히 표현.
- `is`: 객체가 특정 형식인지 확인.

- `??`: null 병합 연산자.

예제:

```
csharp
코드 복사
int a = 10, b = 20;
int max = (a > b) ? a : b; // 삼항 연산자
Console.WriteLine(max); // 출력: 20
```

---

16. 연산자 우선순위

설명:

- 연산자는 우선순위에 따라 계산됩니다.
- 괄호를 사용하여 우선순위를 명시적으로 지정할 수 있습니다.

예제:

```
csharp
코드 복사
int result = 10 + 2 * 5; // 곱셈이 덧셈보다 우선
Console.WriteLine(result); // 출력: 20

int adjustedResult = (10 + 2) * 5; // 괄호로 우선순위 변경
Console.WriteLine(adjustedResult); // 출력: 60
```

강의 요약

1. **연산자 개념 이해**: 기본 산술 및 논리 연산부터 시작.
2. **단항/이항/기타 연산자**: 각 연산자의 역할 강조.
3. **비트 연산자**: 저수준 프로그래밍에 유용.
4. **우선순위**: 복잡한 연산은 괄호로 명확히 표현.

Tip: 실습을 통해 각 연산자의 결과를 출력하며 설명하세요! 😊

아래는 **제어문 소개 및 if/else문부터 반복문 제어까지** 다룬 설명과 예제입니다. 주제별로 코드를 제공하며, 강의 중 실습과 흐름도를 함께 활용하면 이해도를 높일 수 있습니다.

---

1. 제어문

설명:

제어문은 프로그램의 흐름을 제어하는 데 사용됩니다. 순차 실행, 조건 분기, 반복 실행 등이 포함됩니다.

---

2. 순차문: 순서대로 실행하기

설명:

순차문은 코드가 위에서 아래로 순서대로 실행됩니다.

예제:

```
Console.WriteLine("1단계: 시작");
Console.WriteLine("2단계: 진행 중");
Console.WriteLine("3단계: 종료");
```

---

3. 조건문: if 문과 가지치기

설명:

조건에 따라 코드의 실행 여부를 결정합니다.

예제:

```
int score = 85;
if (score >= 90)
{
    Console.WriteLine("A 학점");
}
else
{
    Console.WriteLine("90점 미만");
}
```

---

4. else 문

설명:

`if` 조건이 거짓일 때 실행됩니다.

예제:

```
int number = 10;
if (number > 15)
{
    Console.WriteLine("15보다 큽니다");
}
else
{
    Console.WriteLine("15보다 작거나 같습니다");
}
```

---

5. else if 문 (다중 조건 처리)

설명:

여러 조건을 순차적으로 검사합니다.

예제:

```
int score = 75;
if (score >= 90)
{
    Console.WriteLine("A 학점");
}
else if (score >= 80)
{
    Console.WriteLine("B 학점");
}
else if (score >= 70)
{
    Console.WriteLine("C 학점");
}
else
{
```

```
    Console.WriteLine("F 학점");
}
```

6. 조건문 (if 문) 정리

설명:

조건문은 중첩되거나 다중 조건을 처리하며, 프로그램 흐름을 세밀하게 제어할 수 있습니다.

7. 조건문: switch 문으로 다양한 조건 처리하기

8. switch 문 소개

설명:

`switch` 문은 하나의 값에 대해 여러 케이스를 처리할 때 사용합니다.

9. switch 문 사용하기

예제:

```
int day = 3;
switch (day)
{
    case 1:
        Console.WriteLine("월요일");
        break;
    case 2:
        Console.WriteLine("화요일");
        break;
    case 3:
        Console.WriteLine("수요일");
        break;
    default:
        Console.WriteLine("유효하지 않은 요일");
        break;
}
```

10. 반복문: for 문을 사용하여 구간 반복하기

11. for 문으로 반복하기

설명:

`for` 문은 고정된 횟수를 반복할 때 사용합니다.

예제:

```
for (int i = 1; i <= 5; i++)
{
    Console.WriteLine($"숫자: {i}");
}
```

---

12. 무한 루프

설명:

종료 조건이 없는 반복문입니다. 적절한 조건으로 종료해야 합니다.

예제:

```
int count = 0;
while (true)
{
    Console.WriteLine("무한 루프 실행");
    count++;
    if (count == 3) break; // 무한 루프 탈출 조건
}
```

---

13. for 문으로 1부터 4까지 팩토리얼 값 출력하기

설명:

팩토리얼은 `n! = n * (n-1) * ... * 1`입니다.

- 팩토리얼(Factorial)**은 **자연수에서 해당 숫자까지의 모든 양의 정수를 곱한 값**을 의미합니다.

수학적으로는 다음과 같이 정의됩니다:

$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$

$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$

여기서 n!n!n!은 "n 팩토리얼"이라고 읽습니다. 팩토리얼은 주로 조합, 확률, 순열 계산 등에서 자주 사용됩니다.

팩토리얼의 특징

1. n=0n = 0n=0일 때, 0!=1로 정의됩니다.

   0!=10! = 1

2. 양의 정수에서만 사용하며, 음수에는 적용되지 않습니다.

예제

- 3!=3×2×1=63! = 3 \times 2 \times 1 = 63!=3×2×1=6
- 5!=5×4×3×2×1=1205! = 5 \times 4 \times 3 \times 2 \times 1 = 1205!=5×4×3×2×1=120

예제:

```
for (int i = 1; i <= 4; i++)
{
    int factorial = 1;
    for (int j = 1; j <= i; j++)
    {
        factorial *= j;
    }
    Console.WriteLine($"{i}! = {factorial}");
}
```

---

14. 구구단을 가로로 출력하기

설명:

`for` 중첩문으로 구구단을 출력합니다.

예제:

```
for (int i = 1; i <= 9; i++)
{
    for (int j = 1; j <= 9; j++)
    {
        Console.Write($"{i}x{j}={i * j}\t");
```

```
    }
    Console.WriteLine();
}
```

---

15. while 문과 do 문, foreach 문으로 반복 처리하기

16. while 문

설명:

조건이 참인 동안 반복 실행합니다.

예제:

```
int n = 1;
while (n <= 5)
{
    Console.WriteLine($"숫자: {n}");
    n++;
}
```

17. 피보나치 수열을 while 문으로 표현하기

설명:

피보나치 수열은 각 숫자가 이전 두 숫자의 합입니다.

예제:

```
int a = 0, b = 1, count = 10;
Console.Write($"{a} {b} ");
while (count > 2)
{
    int temp = a + b;
    Console.Write($"{temp} ");
    a = b;
    b = temp;
    count--;
}
```

18. do while 반복문으로 최소 한 번은 실행하기

설명:

`do-while`은 조건과 관계없이 최소 1회 실행합니다.

예제:

```
int x = 5;
do
{
    Console.WriteLine("최소 한 번 실행됩니다");
    x--;
} while (x > 0);
```

19. foreach 문으로 배열 반복하기

설명:

컬렉션의 모든 요소를 반복할 때 사용합니다.

예제:

```
string[] fruits = { "사과", "바나나", "체리" };
foreach (string fruit in fruits)
{
    Console.WriteLine(fruit);
}
```

---

20. break, continue, goto로 반복문 제어하기

21. break 문

설명:

반복문을 중단합니다.

예제:

```
for (int i = 1; i <= 10; i++)
{
    if (i == 5) break;
    Console.WriteLine(i);
}
```

22. continue 문으로 코드 건너뛰기

설명:

현재 반복을 건너뛰고 다음 반복으로 넘어갑니다.

예제:

```
for (int i = 1; i <= 10; i++)
{
    if (i % 2 == 0) continue;
    Console.WriteLine(i); // 홀수만 출력
}
```

23. goto로 프로그램 흐름을 원하는 대로 바꾸기

설명:

`goto`는 레이블로 이동합니다. 신중히 사용해야 합니다.

예제:

```
int n = 1;
start:
if (n <= 5)
{
    Console.WriteLine(n);
    n++;
    goto start; // 레이블로 이동
}
```

강의 요약

1. **조건문**: 프로그램 흐름을 분기하는 방법.
2. **반복문**: 특정 코드를 반복 실행하는 구조.
3. **제어문 활용**: break/continue/goto로 흐름 제어.

Tip: 강의 시 흐름도를 함께 그려주며 설명하면 효과적입니다. 😊

배열

1. **컬렉션**

- C#에서 컬렉션은 데이터를 저장하고 관리하는 자료구조입니다.
- 배열은 가장 기본적인 형태의 컬렉션입니다.

1. **배열**

- 동일한 데이터 타입의 데이터들을 연속된 메모리 공간에 저장합니다.
- 크기가 고정되어 있어 한번 생성하면 크기를 변경할 수 없습니다.

1. **배열 선언하기**

- 다양한 방법으로 배열을 선언할 수 있습니다.
- 크기만 지정하거나, 초기값과 함께 선언할 수 있습니다.

1. **1차원 배열**

- 가장 기본적인 형태의 배열입니다.
- 인덱스를 통해 개별 요소에 접근할 수 있습니다.

1. **다차원 배열**

- 2차원 이상의 배열을 만들 수 있습니다.
- 행렬과 같은 데이터 구조를 표현할 때 유용합니다.

1. **가변 배열**

- 각 행의 길이가 다른 배열을 만들 수 있습니다.
- 비정형 데이터를 저장할 때 유용합니다.

1. **var 키워드**

- 컴파일러가 자동으로 타입을 추론하게 할 수 있습니다.
- 코드를 더 간결하게 만들 수 있습니다.

```
using System;

class ArrayExamples
{
    static void Main()
```

```
{
    // 1. 컬렉션 - 데이터를 저장하는 자료구조
    // 배열은 가장 기본적인 컬렉션 형태입니다.

    // 2. 배열 - 동일한 데이터 타입의 연속된 집합
    Console.WriteLine("=== 기본 배열 ===");
    string[] fruits = { "사과", "바나나", "오렌지" };

    // 3. 배열 선언하기 - 다양한 방법
    Console.WriteLine("\n=== 배열 선언 방법 ===");
    int[] numbers1 = new int[3];          // 크기만 지정
    int[] numbers2 = new int[] { 1, 2, 3 };// 초기화와 함께 선언
    int[] numbers3 = { 1, 2, 3 };         // 간단한 선언과 초기화

    // 4. 1차원 배열 사용
    Console.WriteLine("\n=== 1차원 배열 사용 ===");
    int[] scores = new int[3];
    scores[0] = 90;
    scores[1] = 85;
    scores[2] = 88;

    // 배열 순회하기
    for (int i = 0; i < scores.Length; i++)
    {
        Console.WriteLine($"점수 {i + 1}: {scores[i]}");
    }

    // 5. 다차원 배열
    Console.WriteLine("\n=== 다차원 배열 ===");
    // 2차원 배열 선언
    int[,] matrix = new int[2, 3] {
        { 1, 2, 3 },
        { 4, 5, 6 }
    };

    // 2차원 배열 순회
    for (int i = 0; i < 2; i++)
    {
```

```
            for (int j = 0; j < 3; j++)
            {
                Console.Write($"{matrix[i,j]} ");
            }
            Console.WriteLine();
        }

        // 6. 가변 배열 (Jagged Array)
        Console.WriteLine("\n=== 가변 배열 ===");
        int[][] jaggedArray = new int[3][];
        jaggedArray[0] = new int[] { 1, 2 };
        jaggedArray[1] = new int[] { 3, 4, 5 };
        jaggedArray[2] = new int[] { 6 };

        // 가변 배열 순회
        for (int i = 0; i < jaggedArray.Length; i++)
        {
            for (int j = 0; j < jaggedArray[i].Length; j++)
            {
                Console.Write($"{jaggedArray[i][j]} ");
            }
            Console.WriteLine();
        }

        // 7. var 키워드로 배열 선언하기
        Console.WriteLine("\n=== var 키워드 사용 ===");
        var numbers = new[] { 1, 2, 3, 4, 5 };
        Console.WriteLine($"배열 타입: {numbers.GetType()}");
    }
}
```

1. 함수

설명: 함수는 특정 작업을 수행하기 위해 작성된 코드 블록입니다. 재사용 가능하며 가독성을 높여줍니다.

```
// 함수 정의
void SayHello()
{
    Console.WriteLine("Hello, World!");
}

// 함수 호출
SayHello();
```

---

2. 함수 정의하고 사용하기

설명: 함수를 정의한 후에는 호출하여 실행할 수 있습니다.

```
void Greet()
{
    Console.WriteLine("Welcome to C# programming!");
}

Greet(); // 함수 호출
```

---

3. 매개변수와 반환값

설명: 함수는 입력값(매개변수)을 받고, 결과를 반환할 수 있습니다.

```
int Add(int a, int b)
{
    return a + b;
}

int result = Add(5, 3);
Console.WriteLine($"Result: {result}");
```

---

4. 매개변수가 있는 함수

설명: 매개변수를 통해 함수에 데이터를 전달합니다.

```
void PrintMessage(string message)
{
    Console.WriteLine(message);
}

PrintMessage("Hello, with parameter!");
```

---

5. 반환값이 있는 함수

설명: 함수는 작업 결과를 반환할 수 있습니다.

```
double Multiply(double x, double y)
{
    return x * y;
}

double product = Multiply(2.5, 4.0);
Console.WriteLine($"Product: {product}");
```

---

6. 함수를 사용하여 큰 값과 작은 값, 절댓값 구하기

설명: 함수를 사용하여 수학적 연산을 수행할 수 있습니다.

```
int MaxValue(int a, int b)
{
    return (a > b) ? a : b;
}

int MinValue(int a, int b)
{
    return (a < b) ? a : b;
}

int AbsoluteValue(int number)
{
    return (number < 0) ? -number : number;
}
```

```
Console.WriteLine($"Max: {MaxValue(3, 7)}");
Console.WriteLine($"Min: {MinValue(3, 7)}");
Console.WriteLine($"Absolute: {AbsoluteValue(-15)}");
```

---

7. XML 문서 주석을 사용하여 함수 설명 작성하기

설명: XML 주석은 함수의 용도를 설명합니다.

```
/// <summary>
/// 두 수를 더합니다.
/// </summary>
/// <param name="x">첫 번째 숫자</param>
/// <param name="y">두 번째 숫자</param>
/// <returns>두 숫자의 합</returns>
int Add(int x, int y)
{
    return x + y;
}
```

---

8. 기본 매개변수

설명: 기본값을 지정하여 매개변수가 생략될 때 사용할 값을 설정합니다.

```
void PrintName(string name = "Guest")
{
    Console.WriteLine($"Hello, {name}!");
}

PrintName(); // "Hello, Guest!"
PrintName("Alice"); // "Hello, Alice!"
```

---

9. 명명된 매개변수

설명: 매개변수 이름을 지정하여 순서와 상관없이 값을 전달할 수 있습니다.

```
void DisplayInfo(string name, int age)
{
```

```
    Console.WriteLine($"Name: {name}, Age: {age}");
}

DisplayInfo(age: 25, name: "John");
```

10. 함수 오버로드: 다중 정의

설명: 동일한 이름의 함수를 매개변수 타입이나 개수에 따라 정의할 수 있습니다.

```
void PrintValue(int value)
{
    Console.WriteLine($"Integer: {value}");
}

void PrintValue(string value)
{
    Console.WriteLine($"String: {value}");
}

PrintValue(10);
PrintValue("Overloading example");
```

11. 재귀 함수

설명: 함수가 자기 자신을 호출할 수 있습니다.

```
int Factorial(int n)
{
    if (n <= 1) return 1;
    return n * Factorial(n - 1);
}

Console.WriteLine($"Factorial(5): {Factorial(5)}");
```

12. 함수 범위: 전역 변수와 지역 변수

설명: 변수는 선언된 위치에 따라 사용할 수 있는 범위가 달라집니다.

```
int globalVar = 10; // 전역 변수

void DisplayVariables()
{
    int localVar = 5; // 지역 변수
    Console.WriteLine($"Global: {globalVar}, Local: {localVar}");
}

DisplayVariables();
```

13. 화살표 함수: `=>`

설명: 간단한 표현식을 화살표를 사용하여 작성할 수 있습니다.

```
int Square(int x) ⇒ x * x;
Console.WriteLine($"Square(4): {Square(4)}");
```

14. 식 본문 메서드

설명: 메서드 본문을 간결하게 표현할 수 있습니다.

```
void Greet() ⇒ Console.WriteLine("Hello, concise method!");
Greet();
```

15. 로컬 함수

설명: 함수 내부에 정의된 함수로, 코드 구조를 깔끔하게 유지합니다.

```
void OuterFunction()
{
    void InnerFunction()
    {
        Console.WriteLine("This is a local function.");
    }

    InnerFunction();
}
```

```
OuterFunction();
```

---

16. Main 메서드의 명령줄 인수

설명: 프로그램 실행 시 명령줄에서 데이터를 받을 수 있습니다.

```
static void Main(string[] args)
{
    foreach (var arg in args)
    {
        Console.WriteLine($"Argument: {arg}");
    }
}
```

C# 인터렉티브로 복습하기

설명: C# Interactive(CSI)에서 코드 실행 및 테스트가 가능합니다. Visual Studio에서 `Ctrl+.` 으로 접근 가능합니다.

```
> int Add(int x, int y) => x + y;
> Console.WriteLine(Add(3, 4)); // 결과: 7
```

이 자료를 활용해 강의를 준비하고 질문을 유도하면 좋습니다! 😊

---

1. 클래스, 구조체, 열거형, 네임스페이스

설명: C#의 클래스, 구조체, 열거형, 네임스페이스는 코드 조직화와 설계에 사용됩니다.

```
using System;

namespace MyNamespace
{
    class MyClass
```

```
    {
        public string Name { get; set; }
        public void Greet() => Console.WriteLine($"Hello, {Name}!");
    }
}

class Program
{
    static void Main(string[] args)
    {
        var obj = new MyNamespace.MyClass { Name = "Alice" };
        obj.Greet();
    }
}
```

2. Math 클래스 사용하기

설명: `Math` 클래스는 수학적 계산을 위한 정적 메서드를 제공합니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine($"Pi: {Math.PI}");
        Console.WriteLine($"Square root of 25: {Math.Sqrt(25)}");
        Console.WriteLine($"Power (2^3): {Math.Pow(2, 3)}");
        Console.WriteLine($"Round(3.75): {Math.Round(3.75)}");
    }
}
```

3. `nameof` 연산자

설명: 클래스, 메서드 등의 이름을 문자열로 가져옵니다.

```
using System;

class SampleClass
{
    public void SampleMethod() { }
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine($"Class name: {nameof(SampleClass)}");
        Console.WriteLine($"Method name: {nameof(SampleClass.SampleMet
hod)}");
    }
}
```

구조체 사용하기

1. 구조체란?

설명: 구조체는 값 타입으로 데이터와 메서드를 캡슐화합니다.

```
using System;

struct Point
{
    public int X;
    public int Y;
}

class Program
{
    static void Main(string[] args)
```

```
    {
        Point p = new Point { X = 5, Y = 10 };
        Console.WriteLine($"Point: ({p.X}, {p.Y})");
    }
}
```

2. 구조체 만들기

설명: 구조체는 필드, 속성, 메서드를 가질 수 있습니다.

```
using System;

struct Rectangle
{
    public int Width;
    public int Height;

    public int GetArea() => Width * Height;
}

class Program
{
    static void Main(string[] args)
    {
        var rect = new Rectangle { Width = 5, Height = 4 };
        Console.WriteLine($"Area: {rect.GetArea()}");
    }
}
```

3. 구조체 배열

설명: 구조체는 인스턴스를 만들어 사용할 수 있습니다.

```
using System;

struct Point
{
    public int X;
    public int Y;
}

class Program
{
    static void Main(string[] args)
    {
        Point[] points = new Point[2];
        points[0] = new Point { X = 1, Y = 2 };
        points[1] = new Point { X = 3, Y = 4 };

        foreach (var point in points)
        {
            Console.WriteLine($"Point: ({point.X}, {point.Y})");
        }
    }
}
```

---

4. 구조체 매개변수

설명: 함수의 매개변수로 구조체를 전달할 수 있습니다.

```
using System;

struct Point
{
    public int X;
    public int Y;
}
```

```
class Program
{
    static void PrintPoint(Point p)
    {
        Console.WriteLine($"Point: ({p.X}, {p.Y})");
    }

    static void Main(string[] args)
    {
        var myPoint = new Point { X = 10, Y = 20 };
        PrintPoint(myPoint);
    }
}
```

---

6. 내장형 구조체

설명: .NET에는 `DateTime`, `TimeSpan`과 같은 내장 구조체가 있습니다.

```
csharp
코드 복사
DateTime now = DateTime.Now;
Console.WriteLine($"Current Date and Time: {now}");

TimeSpan duration = new TimeSpan(1, 30, 0); // 1시간 30분
Console.WriteLine($"Duration: {duration}");
```

열거형 형식 사용하기

1. 열거형 형식 사용하기

설명: 열거형은 상수 값의 집합입니다.

```
using System;

enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Sat
```

```
urday }

class Program
{
    static void Main(string[] args)
    {
        Days today = Days.Wednesday;
        Console.WriteLine($"Today is {today}");
    }
}
```

2. 열거형 항목에 상수 값 주기

설명: 열거형 항목에 숫자 값을 지정할 수 있습니다.

```
using System;

enum StatusCode { Success = 200, NotFound = 404, ServerError = 500 }

class Program
{
    static void Main(string[] args)
    {
        StatusCode code = StatusCode.NotFound;
        Console.WriteLine($"Code: {(int)code}");
    }
}
```

3. 열거형 관련 클래스 사용하기

설명: 열거형 값을 문자열로 변환하거나 문자열에서 값을 가져올 수 있습니다.

```
using System;
```

```
enum Colors { Red, Green, Blue }

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine($"Color name: {Enum.GetName(typeof(Colors),
1)}"); // Green

        foreach (var color in Enum.GetValues(typeof(Colors)))
        {
            Console.WriteLine(color);
        }
    }
}
```

---

1. 클래스 소개하기

설명: 클래스는 객체 지향 프로그래밍의 기본 단위로, 데이터(필드)와 동작(메서드)을 캡슐화합니다.

```
using System;

class Person
{
    public string Name { get; set; }
    public void Greet() ⇒ Console.WriteLine($"Hello, my name is {Name}.");
}

class Program
{
    static void Main(string[] args)
    {
        var person = new Person { Name = "Alice" };
        person.Greet();
```

```
    }
}
```

2. 클래스 만들기

설명: 사용자 정의 클래스를 통해 원하는 데이터를 캡슐화할 수 있습니다.

```
using System;

class Rectangle
{
    public int Width { get; set; }
    public int Height { get; set; }

    public int GetArea() => Width * Height;
}

class Program
{
    static void Main(string[] args)
    {
        var rect = new Rectangle { Width = 5, Height = 10 };
        Console.WriteLine($"Area: {rect.GetArea()}");
    }
}
```

3. 클래스 여러 개 만들기

설명: 여러 클래스를 정의하고 각 역할에 맞게 활용할 수 있습니다.

```
using System;

class Person
{
    public string Name { get; set; }
    public void Introduce() => Console.WriteLine($"Hi, I'm {Name}.");
}
```

```
class Job
{
    public string Title { get; set; }
    public void Describe() => Console.WriteLine($"I work as a {Title}.");
}

class Program
{
    static void Main(string[] args)
    {
        var person = new Person { Name = "Alice" };
        var job = new Job { Title = "Developer" };

        person.Introduce();
        job.Describe();
    }
}
```

4. 클래스 시그니처

설명: 클래스의 시그니처는 이름, 접근 제한자, 상속 여부 등을 나타냅니다.

```
using System;

public class MyClass
{
    public string Message { get; set; }

    public void ShowMessage() => Console.WriteLine(Message);
}

class Program
{
    static void Main(string[] args)
    {
        var obj = new MyClass { Message = "Hello, Class!" };
        obj.ShowMessage();
```

```
    }
}
```

5. 자주 사용하는 내장 클래스

설명: C#에는 다양한 내장 클래스가 있으며, 예를 들어 `DateTime` 클래스는 시간과 날짜를 다룹니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        DateTime now = DateTime.Now;
        Console.WriteLine($"Current Date and Time: {now}");
    }
}
```

6. `Environment` 클래스로 프로그램 강제 종료하기

설명: `Environment.Exit` 메서드를 사용하여 프로그램을 강제 종료할 수 있습니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Exiting the program...");
        Environment.Exit(0);
    }
}
```

7. 환경 변수 사용하기

설명: 환경 변수를 읽고 쓸 수 있습니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        string path = Environment.GetEnvironmentVariable("PATH");
        Console.WriteLine($"PATH: {path}");
    }
}
```

8. EXE 파일 실행하기

설명: `System.Diagnostics.Process` 클래스를 사용하여 외부 프로그램을 실행합니다.

```
using System.Diagnostics;

class Program
{
    static void Main(string[] args)
    {
        Process.Start("notepad.exe");
    }
}
```

9. `Random` 클래스

설명: `Random` 클래스를 사용하여 난수를 생성합니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Random random = new Random();
        int randomNumber = random.Next(1, 101); // 1부터 100까지
```

```
        Console.WriteLine($"Random Number: {randomNumber}");
    }
}
```

10. 프로그램 실행 시간 구하기

설명: 프로그램 실행 시간을 측정할 수 있습니다.

```
using System;
using System.Diagnostics;

class Program
{
    static void Main(string[] args)
    {
        Stopwatch stopwatch = Stopwatch.StartNew();

        // 실행 코드
        for (int i = 0; i < 1000000; i++) { }

        stopwatch.Stop();
        Console.WriteLine($"Execution Time: {stopwatch.ElapsedMillisecond
s} ms");
    }
}
```

11. 정규식

설명: `Regex` 클래스를 사용하여 문자열 검색 및 검사를 수행할 수 있습니다.

```
using System;
using System.Text.RegularExpressions;

class Program
{
    static void Main(string[] args)
    {
        string input = "123-456-7890";
```

```
        string pattern = @"^\d{3}-\d{3}-\d{4}$";
        bool isMatch = Regex.IsMatch(input, pattern);

        Console.WriteLine($"Is valid phone number: {isMatch}");
    }
}
```

13. 값 형식과 참조 형식

설명: 값 형식은 스택에 저장되고, 참조 형식은 힙에 저장됩니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int valueType = 10;
        object referenceType = valueType;

        valueType = 20;

        Console.WriteLine($"ValueType: {valueType}"); // 20
        Console.WriteLine($"ReferenceType: {referenceType}"); // 10
    }
}
```

14. 박싱과 언박싱

설명: 값 형식을 참조 형식으로 변환(박싱), 다시 값 형식으로 변환(언박싱)합니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int value = 42;
```

```
        object boxed = value; // 박싱
        int unboxed = (int)boxed; // 언박싱

        Console.WriteLine($"Boxed: {boxed}, Unboxed: {unboxed}");
    }
}
```

15. `is` 연산자로 형식 비교하기

설명: 객체가 특정 형식인지 확인할 수 있습니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        object obj = "Hello";
        Console.WriteLine(obj is string); // true
        Console.WriteLine(obj is int); // false
    }
}
```

16. `as` 연산자로 형식 변환하기

설명: `as` 연산자를 사용해 안전하게 형 변환을 수행합니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        object obj = "Hello";
        string str = obj as string;

        Console.WriteLine(str ?? "Conversion failed");
```

```
    }
}
```

17. 패턴 매칭: `if` 문과 `is` 연산자 사용하기

설명: `is` 연산자를 사용한 패턴 매칭입니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        object obj = 42;

        if (obj is int number)
        {
            Console.WriteLine($"Number: {number}");
        }
        else
        {
            Console.WriteLine("Not a number.");
        }
    }
}
```

문자열 다루기

1. 문자열 다루기

설명: 문자열은 `string` 키워드를 사용해 선언하며 다양한 방법으로 처리할 수 있습니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
```

```
        string greeting = "Hello";
        string name = "Alice";

        string message = greeting + ", " + name + "!";
        Console.WriteLine(message); // Hello, Alice!

        Console.WriteLine($"Length of name: {name.Length}"); // 문자열 길이
        Console.WriteLine($"To Upper: {name.ToUpper()}"); // 대문자 변환
        Console.WriteLine($"Substring: {name.Substring(1)}"); // 부분 문자열
    }
}
```

2. 문자열 처리 관련 주요 API 살펴보기

설명: `string` 클래스에는 문자열 처리를 위한 다양한 메서드가 있습니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        string text = "C# is awesome!";
        Console.WriteLine($"Contains 'awesome': {text.Contains("awesom
e")}");
        Console.WriteLine($"Starts with 'C#': {text.StartsWith("C#")}");
        Console.WriteLine($"Ends with '!': {text.EndsWith("!")}");
        Console.WriteLine($"Index of 'is': {text.IndexOf("is")}");
        Console.WriteLine($"Replace 'awesome' with 'great': {text.Replace("a
wesome", "great")}");
    }
}
```

3. `StringBuilder` 클래스를 사용하여 문자열 연결하기

설명: `StringBuilder`는 문자열 연결 작업에서 성능이 뛰어납니다.

```
using System;
using System.Text;

class Program
{
    static void Main(string[] args)
    {
        StringBuilder sb = new StringBuilder("Hello");
        sb.Append(", ");
        sb.Append("World!");
        Console.WriteLine(sb.ToString()); // Hello, World!
    }
}
```

---

4. `String`과 `StringBuilder` 클래스의 성능 차이 비교하기

설명: 반복적으로 문자열을 수정할 때 `StringBuilder`가 더 효율적입니다.

```
using System;
using System.Diagnostics;
using System.Text;

class Program
{
    static void Main(string[] args)
    {
        int iterations = 10000;

        Stopwatch sw = Stopwatch.StartNew();
        string text = "";
        for (int i = 0; i < iterations; i++)
        {
            text += "a";
        }
        sw.Stop();
        Console.WriteLine($"String: {sw.ElapsedMilliseconds} ms");
```

```
        sw.Restart();
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < iterations; i++)
        {
            sb.Append("a");
        }
        sw.Stop();
        Console.WriteLine($"StringBuilder: {sw.ElapsedMilliseconds} ms");
    }
}
```

예외 처리하기

1. 예외와 예외 처리

설명: 예외는 프로그램 실행 중 발생하는 오류입니다. 예외를 처리하면 프로그램이 중단되지 않고 실행을 계속할 수 있습니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        try
        {
            int[] numbers = { 1, 2, 3 };
            Console.WriteLine(numbers[5]); // 오류 발생
        }
        catch (IndexOutOfRangeException ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }
}
```

2. `try~catch~finally` 구문

설명: `finally` 블록은 예외 발생 여부와 상관없이 항상 실행됩니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        try
        {
            int number = int.Parse("NotANumber"); // 오류 발생
        }
        catch (FormatException ex)
        {
            Console.WriteLine($"Format Error: {ex.Message}");
        }
        finally
        {
            Console.WriteLine("Execution finished.");
        }
    }
}
```

---

3. `Exception` 클래스로 예외 처리하기

설명: `Exception`은 모든 예외의 기본 클래스입니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        try
        {
            int number = int.Parse("abc");
        }
        catch (Exception ex)
```

```
        {
            Console.WriteLine($"General Error: {ex.Message}");
        }
    }
}
```

5. `throw` 구문으로 직접 예외 발생시키기

설명: `throw`를 사용하여 특정 조건에서 예외를 발생시킬 수 있습니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        try
        {
            int age = -5;
            if (age < 0)
            {
                throw new ArgumentException("Age cannot be negative");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Exception: {ex.Message}");
        }
    }
}
```

컬렉션 사용하기

1. 배열과 컬렉션

설명: 배열은 고정된 크기의 데이터를 저장하는 데 사용됩니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int[] numbers = { 1, 2, 3, 4, 5 };
        foreach (var num in numbers)
        {
            Console.WriteLine(num);
        }
    }
}
```

2. 리스트 출력 구문

설명: `List<T>`는 가변 크기의 데이터를 저장하는 데 사용됩니다.

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        List<string> names = new List<string> { "Alice", "Bob", "Charlie" };
        names.Add("Dave");
        names.Remove("Bob");

        foreach (var name in names)
        {
            Console.WriteLine(name);
        }
    }
}
```

5. `Stack` 클래스

설명: `Stack`은 LIFO(Last-In, First-Out) 구조입니다.

```
using System;
using System.Collections;

class Program
{
    static void Main(string[] args)
    {
        Stack stack = new Stack();
        stack.Push(1);
        stack.Push(2);
        stack.Push(3);

        while (stack.Count > 0)
        {
            Console.WriteLine(stack.Pop());
        }
    }
}
```

---

6. `Queue` 클래스

설명: `Queue`는 FIFO(First-In, First-Out) 구조입니다.

```
using System;
using System.Collections;

class Program
{
    static void Main(string[] args)
    {
        Queue queue = new Queue();
        queue.Enqueue(1);
        queue.Enqueue(2);
        queue.Enqueue(3);

        while (queue.Count > 0)
```

```
        {
            Console.WriteLine(queue.Dequeue());
        }
    }
}
```

7. ArrayList 클래스

설명: ArrayList는 크기가 동적으로 조정되는 배열로, 다양한 형식의 데이터를 저장할 수 있습니다.

```
using System;
using System.Collections;

class Program
{
    static void Main(string[] args)
    {
        // ArrayList 생성
        ArrayList arrayList = new ArrayList();

        // 요소 추가
        arrayList.Add(1);          // 정수
        arrayList.Add("Hello");    // 문자열
        arrayList.Add(3.14);       // 실수

        // 요소 접근
        Console.WriteLine("ArrayList 요소:");
        foreach (var item in arrayList)
        {
            Console.WriteLine(item);
        }

        // 요소 제거
        arrayList.Remove(1); // 값이 1인 요소 제거

        Console.WriteLine("\nArrayList 요소 제거 후:");
        foreach (var item in arrayList)
```

```
        {
            Console.WriteLine(item);
        }
    }
}
```

출력:

```
ArrayList 요소:
1
Hello
3.14

ArrayList 요소 제거 후:
Hello
3.14
```

---

8. `Hashtable` 클래스

설명: `Hashtable`은 키-값 쌍을 저장하는 컬렉션입니다. 키를 사용해 값을 빠르게 검색할 수 있습니다.

```
using System;
using System.Collections;

class Program
{
    static void Main(string[] args)
    {
        // Hashtable 생성
        Hashtable hashtable = new Hashtable();

        // 키-값 쌍 추가
        hashtable["Alice"] = 25;
        hashtable["Bob"] = 30;
        hashtable["Charlie"] = 35;

        // 값 접근
```

```
        Console.WriteLine("Hashtable 요소:");
        foreach (DictionaryEntry entry in hashtable)
        {
            Console.WriteLine($"Key: {entry.Key}, Value: {entry.Value}");
        }

        // 특정 키의 값 가져오기
        Console.WriteLine($"\nAlice의 나이: {hashtable["Alice"]}");

        // 요소 제거
        hashtable.Remove("Bob");

        Console.WriteLine("\n요소 제거 후 Hashtable:");
        foreach (DictionaryEntry entry in hashtable)
        {
            Console.WriteLine($"Key: {entry.Key}, Value: {entry.Value}");
        }
    }
}
```

출력:

```
Hashtable 요소:
Key: Charlie, Value: 35
Key: Bob, Value: 30
Key: Alice, Value: 25

Alice의 나이: 25

요소 제거 후 Hashtable:
Key: Charlie, Value: 35
Key: Alice, Value: 25
```

요약

- `ArrayList`: 크기가 가변적이고, 다양한 데이터 형식을 저장할 수 있는 배열과 유사한 컬렉션.
- `Hashtable`: 키-값 쌍을 사용해 데이터를 저장하며, 키를 이용한 빠른 검색이 가능.

제네릭 사용하기(Generics)

1. Cup of T

설명: 제네릭 클래스를 사용하면 특정 타입에 종속되지 않는 범용 클래스를 만들 수 있습니다.

```
using System;

class Cup<T>
{
    public T Content { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Cup<string> cupOfString = new Cup<string> { Content = "Coffee" };
        Cup<int> cupOfInt = new Cup<int> { Content = 42 };

        Console.WriteLine($"CupOfString: {cupOfString.Content}");
        Console.WriteLine($"CupOfInt: {cupOfInt.Content}");
    }
}
```

2. Stack 제네릭 클래스 사용하기

설명: `Stack<T>`를 사용해 LIFO(후입선출) 구조를 구현합니다.

```
using System;
using System.Collections.Generic;

class Program
```

```
{
    static void Main(string[] args)
    {
        Stack<int> stack = new Stack<int>();
        stack.Push(10);
        stack.Push(20);
        stack.Push(30);

        while (stack.Count > 0)
        {
            Console.WriteLine(stack.Pop());
        }
    }
}
```

3. List〈T〉 제네릭 클래스 사용하기

설명: `List<T>`는 동적으로 크기가 늘어나는 배열 형태의 컬렉션입니다.

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        List<string> names = new List<string> { "Alice", "Bob", "Charlie" };
        names.Add("Dave");

        foreach (var name in names)
        {
            Console.WriteLine(name);
        }
    }
}
```

4. Enumerable 클래스로 컬렉션 만들기

설명: `IEnumerable<T>`를 구현하여 커스텀 컬렉션을 만들 수 있습니다.

```
using System;
using System.Collections;
using System.Collections.Generic;

class SimpleCollection : IEnumerable<int>
{
    private int[] data = { 1, 2, 3, 4, 5 };
    public IEnumerator<int> GetEnumerator()
    {
        foreach (var item in data)
        {
            yield return item;
        }
    }
    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}

class Program
{
    static void Main(string[] args)
    {
        var collection = new SimpleCollection();
        foreach (var i in collection)
        {
            Console.WriteLine(i);
        }
    }
}
```

5. Dictionary〈T, T〉 제네릭 클래스 사용하기

설명: `Dictionary<TKey, TValue>`는 키-값 쌍을 효율적으로 관리하는 컬렉션입니다.

```
using System;
using System.Collections.Generic;
```

```
class Program
{
    static void Main(string[] args)
    {
        Dictionary<string, int> ages = new Dictionary<string, int>
        {
            { "Alice", 25 },
            { "Bob", 30 }
        };

        ages["Charlie"] = 35;

        foreach (var pair in ages)
        {
            Console.WriteLine($"{pair.Key}: {pair.Value}");
        }
    }
}
```

널(null) 다루기

1. null 값

설명: 참조 형식은 null을 가질 수 있으며, 값 형식은 기본적으로 null을 가질 수 없습니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        string str = null;
        if (str == null)
        {
            Console.WriteLine("str is null");
        }
```

```
    }
}
```

2. null 가능 형식: Nullable〈T〉 형식

설명: `int?`와 같은 형식으로 값 형식에 null을 허용할 수 있습니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int? nullableInt = null;
        Console.WriteLine(nullableInt.HasValue ? nullableInt.Value.ToString() :
"No value");

        nullableInt = 10;
        Console.WriteLine(nullableInt.HasValue ? nullableInt.Value.ToString() :
"No value");
    }
}
```

3. null 값을 다루는 연산자 소개하기 (??, ?. 연산자)

설명: `??` 연산자를 사용해 null인 경우 대체값을 제공하고, `?.`은 null 안전 접근을 합니다.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        string str = null;
        Console.WriteLine(str ?? "Default Value"); // str이 null이면 "Default Valu
e"

        str = "Hello";
```

```
        Console.WriteLine(str?.Length); // str이 null이 아니므로 길이 출력
    }
}
```

LINQ

1. LINQ 개요

설명: LINQ(Language Integrated Query)를 사용해 컬렉션을 쿼리할 수 있습니다.

2. 확장 메서드 사용하기

설명: LINQ는 확장 메서드 형태로 제공됩니다.

```
using System;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        int[] numbers = { 1, 2, 3, 4, 5 };

        var evenNumbers = numbers.Where(n => n % 2 == 0);
        foreach (var num in evenNumbers)
        {
            Console.WriteLine(num);
        }
    }
}
```

3. 화살표 연산자와 람다 식으로 조건 처리

설명: `=>`를 사용한 람다 식으로 간결하게 조건을 표현할 수 있습니다. (위 예제 참조)

4. 데이터 정렬과 검색

설명: `OrderBy`, `OrderByDescending`, `First`, `Single` 등의 메서드로 정렬 및 검색이 가능합니다.

```
using System;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        string[] names = { "Charlie", "Alice", "Bob" };
        var sortedNames = names.OrderBy(n ⇒ n);

        foreach (var name in sortedNames)
        {
            Console.WriteLine(name);
        }

        var firstName = names.First(n ⇒ n.StartsWith("A"));
        Console.WriteLine($"First name starting with A: {firstName}");
    }
}
```

5. 메서드 구문과 쿼리 구문

설명: LINQ는 메서드 구문과 쿼리 구문 두 가지 방식으로 사용할 수 있습니다.

```
using System;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        int[] nums = { 5, 3, 8, 1 };

        // 메서드 구문
        var sortedMethod = nums.OrderBy(n ⇒ n);

        // 쿼리 구문
```

```
        var sortedQuery = from n in nums
                          orderby n
                          select n;

        Console.WriteLine("Method syntax:");
        foreach (var n in sortedMethod) Console.WriteLine(n);

        Console.WriteLine("Query syntax:");
        foreach (var n in sortedQuery) Console.WriteLine(n);
    }
}
```

6. Select( ) 확장 메서드를 사용하여 새로운 형태로 가공하기

설명: `Select`를 사용하여 각 요소를 새로운 형태로 변환합니다.

```
using System;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        string[] words = { "apple", "banana", "cherry" };

        var lengths = words.Select(w => w.Length);

        foreach (var length in lengths)
        {
            Console.WriteLine(length);
        }
    }
}
```

7. ForEach( ) 메서드로 반복 출력하기

설명: LINQ 자체에 ForEach는 없지만, `List<T>`에서는 `ForEach` 메서드를 사용할 수 있습니다.

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        List<int> numbers = Enumerable.Range(1, 5).ToList();
        numbers.ForEach(n ⇒ Console.WriteLine(n));
    }
}
```

알고리즘과 절차 지향 프로그래밍

아래는 알고리즘 개념을 간단한 코드로 구현한 예시입니다. (실전에서는 더 복잡한 로직이 가능)

1. 알고리즘

설명: 문제를 해결하기 위한 단계적 절차.

2. 합계 구하기: SUM 알고리즘

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int[] data = { 1, 2, 3, 4, 5 };
        int sum = 0;
        foreach (var d in data) sum += d;
        Console.WriteLine($"Sum: {sum}");
    }
}
```

---

3. 개수 구하기: COUNT 알고리즘

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int[] data = { 2, 4, 6, 8, 10 };
        int count = data.Length; // 개수
        Console.WriteLine($"Count: {count}");
    }
}
```

4. 평균 구하기: AVERAGE 알고리즘

```
using System;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        int[] data = { 1, 2, 3, 4, 5 };
        double avg = data.Average();
        Console.WriteLine($"Average: {avg}");
    }
}
```

5. 최댓값 구하기: MAX 알고리즘

```
using System;
using System.Linq;

class Program
{
    static void Main(string[] args)
```

```
    {
        int[] data = { 10, 3, 5, 2, 8 };
        int max = data.Max();
        Console.WriteLine($"Max: {max}");
    }
}
```

6. 최솟값 구하기: MIN 알고리즘

```
using System;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        int[] data = { 10, 3, 5, 2, 8 };
        int min = data.Min();
        Console.WriteLine($"Min: {min}");
    }
}
```

7. 근삿값 구하기: NEAR 알고리즘

설명: 배열에서 특정 값에 가장 가까운 값을 찾는 예제

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int[] data = { 10, 12, 20, 25, 30 };
        int target = 22;
        int nearest = data[0];

        foreach (var d in data)
```

```
        {
            if (Math.Abs(d - target) < Math.Abs(nearest - target))
                nearest = d;
        }

        Console.WriteLine($"Nearest to {target}: {nearest}");
    }
}
```

8. 순위 구하기: RANK 알고리즘

설명: 각 요소가 몇 번째로 큰지 순위를 매기는 예제

```
using System;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        int[] scores = { 90, 70, 50, 70, 60 };

        for(int i = 0; i < scores.Length; i++)
        {
            int rank = 1;
            for(int j = 0; j < scores.Length; j++)
            {
                if(scores[j] > scores[i]) rank++;
            }
            Console.WriteLine($"Score: {scores[i]}, Rank: {rank}");
        }
    }
}
```

9. 순서대로 나열하기: SORT 알고리즘

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int[] data = { 5, 2, 8, 1, 9 };
        Array.Sort(data);

        foreach (var d in data) Console.WriteLine(d);
    }
}
```

10. 특정 값 검색하기: SEARCH 알고리즘 (선형 검색)

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int[] data = { 5, 2, 8, 1, 9 };
        int target = 8;
        int index = -1;

        for (int i = 0; i < data.Length; i++)
        {
            if (data[i] == target)
            {
                index = i;
                break;
            }
        }
        Console.WriteLine(index >= 0 ? $"Found at index {index}" : "Not foun
d");
```

```
    }
}
```

11. 배열을 하나로 합치기: MERGE 알고리즘

```
using System;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        int[] arr1 = { 1, 3, 5 };
        int[] arr2 = { 2, 4, 6 };

        int[] merged = arr1.Concat(arr2).ToArray();
        foreach (var d in merged) Console.WriteLine(d);
    }
}
```

12. 최빈값 구하기: MODE 알고리즘

설명: 가장 자주 등장하는 값 찾기

```
using System;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        int[] data = { 1, 2, 2, 3, 3, 3, 4 };
        var grouped = data.GroupBy(x ⇒ x)
                          .OrderByDescending(g ⇒ g.Count())
                          .First();

        Console.WriteLine($"Mode: {grouped.Key}, Count: {grouped.Count
```

```
()}");
    }
}
```

13. 그룹화하기: GROUP 알고리즘

설명: 데이터를 특정 기준으로 그룹화하기

```
using System;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        string[] fruits = { "apple", "banana", "blueberry", "cherry", "apricot" };

        var groups = fruits.GroupBy(f ⇒ f[0]); // 첫 글자로 그룹화

        foreach (var group in groups)
        {
            Console.WriteLine($"Key: {group.Key}");
            foreach (var item in group)
            {
                Console.WriteLine($"  {item}");
            }
        }
    }
}
```

개체 만들기

1. 클래스와 개체

설명: 클래스로부터 개체(인스턴스)를 생성하여 사용할 수 있습니다.

```
using System;

class Person
{
    public string Name;
    public void SayHello()
    {
        Console.WriteLine($"Hello, my name is {Name}");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person p = new Person();
        p.Name = "Alice";
        p.SayHello(); // Hello, my name is Alice
    }
}
```

---

2. 개체와 인스턴스

설명: 클래스는 설계도이고, 개체(인스턴스)는 그 클래스로부터 만들어진 실체입니다.

```
using System;

class Car
{
    public string Model;
}

class Program
```

```
{
    static void Main(string[] args)
    {
        Car car = new Car(); // car는 Car 클래스의 인스턴스
        car.Model = "Tesla";
        Console.WriteLine(car.Model); // Tesla
    }
}
```

---

3. 인스턴스 메서드

설명: 인스턴스 메서드는 개체를 통해 호출하는 메서드입니다.

```
using System;

class Calculator
{
    public int Add(int a, int b) ⇒ a + b;
}

class Program
{
    static void Main(string[] args)
    {
        Calculator calc = new Calculator();
        Console.WriteLine(calc.Add(3, 5)); // 8
    }
}
```

---

4. 익명 형식

설명: 익명 형식은 타입 이름 없이 프로퍼티만 가지는 개체를 생성할 수 있습니다.

```
using System;

class Program
{
    static void Main(string[] args)
```

```
    {
        var anonymous = new { Name = "Alice", Age = 30 };
        Console.WriteLine($"{anonymous.Name}, {anonymous.Age}");
    }
}
```

5. 정적 멤버와 인스턴스 멤버

설명: 정적 멤버는 클래스명으로 접근하고, 인스턴스 멤버는 개체를 생성한 뒤 접근합니다.

```
using System;

class MathUtil
{
    public static double Pi = 3.14;  // 정적 필드
    public int Value;                // 인스턴스 필드
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(MathUtil.Pi);
        MathUtil mu = new MathUtil();
        mu.Value = 10;
        Console.WriteLine(mu.Value);
    }
}
```

6. 프로젝트에 클래스를 여러 개 사용하기

설명: 하나의 프로젝트 내에 여러 클래스 파일을 만들어 사용 가능합니다. (아래 예제는 한 파일에 모두 쓰지만, 실제로는 분리 가능)

```
using System;

class Animal
{
```

```
    public string Name;
}

class Dog : Animal
{
    public void Bark() => Console.WriteLine("Woof!");
}

class Program
{
    static void Main(string[] args)
    {
        Dog d = new Dog { Name = "Buddy" };
        Console.WriteLine(d.Name);
        d.Bark();
    }
}
```

7. ToString( ) 메서드 오버라이드

설명: `ToString()`을 오버라이드하여 개체 정보를 문자열로 표현할 수 있습니다.

```
using System;

class Person
{
    public string Name;
    public override string ToString() => $"Person: {Name}";
}

class Program
{
    static void Main(string[] args)
    {
        Person p = new Person { Name = "Alice" };
        Console.WriteLine(p.ToString()); // Person: Alice
```

```
    }
}
```

8. 클래스 배열

설명: 클래스 타입으로 배열을 만들 수 있습니다.

```
using System;

class Person
{
    public string Name;
}

class Program
{
    static void Main(string[] args)
    {
        Person[] people = new Person[]
        {
            new Person { Name = "Alice" },
            new Person { Name = "Bob" }
        };

        foreach (var person in people)
        {
            Console.WriteLine(person.Name);
        }
    }
}
```

9. var 키워드를 사용하여 클래스의 인스턴스 생성하기

설명: `var` 키워드는 컴파일러가 타입을 추론하도록 합니다.

```
using System;

class Person
```

```
{
    public string Name;
}

class Program
{
    static void Main(string[] args)
    {
        var p = new Person { Name = "Charlie" };
        Console.WriteLine(p.Name);
    }
}
```

네임스페이스

1. 네임스페이스

설명: 네임스페이스는 코드를 논리적으로 그룹화합니다.

```
using System;

namespace MyNamespace
{
    class MyClass
    {
        public void SayHi() ⇒ Console.WriteLine("Hi from MyNamespace!");
    }
}

class Program
{
    static void Main(string[] args)
    {
        MyNamespace.MyClass mc = new MyNamespace.MyClass();
        mc.SayHi();
    }
}
```

2. 네임스페이스 만들기

설명: 새로운 네임스페이스를 정의하고, 내부에 클래스를 정의할 수 있습니다.

```
using System;

namespace YourNamespace
{
    class YourClass
    {
        public void SayHello() => Console.WriteLine("Hello from YourNamespa
ce!");
    }
}

class Program
{
    static void Main(string[] args)
    {
        YourNamespace.YourClass yc = new YourNamespace.YourClass();
        yc.SayHello();
    }
}
```

---

3. using 지시문

설명: using을 사용하면 네임스페이스 이름을 매번 명시하지 않아도 됩니다.

```
using System;
using YourNamespace;

namespace YourNamespace
{
    class MyClass
    {
        public void Greet() => Console.WriteLine("Greetings!");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        MyClass mc = new MyClass(); // using YourNamespace 때문에 바로 사
용 가능
        mc.Greet();
    }
}
```

필드 만들기

1. 필드

설명: 필드는 클래스의 데이터를 저장하는 멤버입니다.

```
using System;

class Person
{
    public string name; // 필드
}

class Program
{
    static void Main(string[] args)
    {
        Person p = new Person();
        p.name = "Alice";
        Console.WriteLine(p.name);
    }
}
```

2. 액세스 한정자

설명: public, private 등의 접근 제한자로 필드 접근을 제어합니다.

```
using System;

class Person
{
    private string name; // 외부 접근 불가
    public void SetName(string n) ⇒ name = n;
    public string GetName() ⇒ name;
}

class Program
{
    static void Main(string[] args)
    {
        Person p = new Person();
        p.SetName("Bob");
        Console.WriteLine(p.GetName());
    }
}
```

---

3. 여러 가지 형태의 필드 선언, 초기화, 참조 구현하기

설명: 필드를 선언하면서 초기화할 수도 있습니다.

```
using System;

class Person
{
    private int age = 20; // 초기화
    public void PrintAge() ⇒ Console.WriteLine(age);
}

class Program
{
    static void Main(string[] args)
    {
        Person p = new Person();
        p.PrintAge(); // 20
```

```
    }
}
```

생성자

1. 생성자

설명: 생성자는 클래스 인스턴스가 만들어질 때 호출되는 특수한 메서드입니다.

```
using System;

class Person
{
    public string Name;
    public Person() // 생성자
    {
        Name = "Unknown";
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person p = new Person();
        Console.WriteLine(p.Name); // Unknown
    }
}
```

2. 매개변수가 있는 생성자 만들기

```
using System;

class Person
{
    public string Name;
    public Person(string name)
```

```
    {
        Name = name;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person p = new Person("Alice");
        Console.WriteLine(p.Name); // Alice
    }
}
```

3. 클래스에 생성자 여러 개 만들기

```
using System;

class Person
{
    public string Name;
    public int Age;

    public Person() { Name = "Unknown"; Age = 0; }
    public Person(string name) { Name = name; Age = 0; }
    public Person(string name, int age) { Name = name; Age = age; }
}

class Program
{
    static void Main(string[] args)
    {
        Person p1 = new Person();
        Person p2 = new Person("Bob");
        Person p3 = new Person("Charlie", 30);
        Console.WriteLine(p1.Name + ", " + p1.Age);
        Console.WriteLine(p2.Name + ", " + p2.Age);
```

```
        Console.WriteLine(p3.Name + ", " + p3.Age);
    }
}
```

4. 정적 생성자와 인스턴스 생성자

설명: 정적 생성자는 클래스 로드 시 한 번 호출되며, 인스턴스 생성자는 개체 생성 시마다 호출됩니다.

```
using System;

class MyClass
{
    public static int Count;
    static MyClass()
    {
        Count = 100;
        Console.WriteLine("Static constructor called");
    }

    public MyClass()
    {
        Count++;
        Console.WriteLine("Instance constructor called");
    }
}

class Program
{
    static void Main(string[] args)
    {
        MyClass m1 = new MyClass();
        MyClass m2 = new MyClass();
        Console.WriteLine($"Count: {MyClass.Count}");
    }
}
```

5. this( ) 생성자로 다른 생성자 호출하기

```
using System;

class Person
{
    public string Name;
    public int Age;

    public Person() : this("Unknown", 0) { }

    public Person(string name) : this(name, 0) { }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person p = new Person();
        Console.WriteLine($"{p.Name}, {p.Age}"); // Unknown, 0
    }
}
```

6. 생성자를 사용하여 읽기 전용 필드 초기화

```
using System;

class Person
{
    public readonly string Name;
    public Person(string name)
```

```
    {
        Name = name; // 생성자에서만 초기화 가능
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person p = new Person("Alice");
        Console.WriteLine(p.Name);
    }
}
```

7. 식 본문 생성자

```
using System;

class Person
{
    public string Name;
    public Person(string name) => Name = name; // 식 본문 생성자
}

class Program
{
    static void Main(string[] args)
    {
        var p = new Person("Bob");
        Console.WriteLine(p.Name);
    }
}
```

소멸자

1. 종료자

설명: 소멸자는 가비지 컬렉션으로 객체가 제거될 때 호출됩니다(직접 호출 불가).

```
using System;

class MyResource
{
    ~MyResource()
    {
        Console.WriteLine("Destructor called");
    }
}

class Program
{
    static void Main(string[] args)
    {
        MyResource r = new MyResource();
        // GC에 의해 나중에 소멸자 호출
    }
}
```

---

2. 가비지 수집기

설명: 가비지 컬렉션은 더 이상 참조되지 않는 객체를 정리합니다.
(별도 코드 없이 개념 설명)

---

3. 생성자, 메서드, 소멸자 실행 시점 살펴보기

설명: 생성자는 객체 생성 시, 메서드는 호출 시, 소멸자는 GC 시점에 호출.

---

4. 소멸자를 사용한 클래스 역할 마무리하기

설명: 소멸자에서 리소스 해제 가능 (파일 닫기 등)

---

5. 생성자, 메서드, 소멸자 함께 사용하기

설명: 하나의 클래스 안에 생성자(객체 생성), 메서드(동작), 소멸자(정리)가 공존.

메서드와 매개변수

1. 메서드

설명: 메서드는 클래스의 동작을 정의합니다.

```
using System;

class Calculator
{
    public int Add(int a, int b) ⇒ a + b;
}

class Program
{
    static void Main(string[] args)
    {
        var calc = new Calculator();
        Console.WriteLine(calc.Add(3, 4));
    }
}
```

---

2. 메서드의 매개변수 전달 방식 (ref, out)

```
using System;

class Program
{
    static void Increase(ref int x) { x++; }

    static void GetValues(out int a, out int b)
    {
        a = 10;
        b = 20;
    }

    static void Main(string[] args)
    {
        int value = 5;
        Increase(ref value);
```

```
        Console.WriteLine(value); // 6

        int x, y;
        GetValues(out x, out y);
        Console.WriteLine($"{x}, {y}");
    }
}
```

3. 가변 길이 매개변수 (params)

```
using System;

class Program
{
    static int Sum(params int[] numbers)
    {
        int sum = 0;
        foreach(var n in numbers) sum += n;
        return sum;
    }

    static void Main(string[] args)
    {
        Console.WriteLine(Sum(1,2,3,4,5));
    }
}
```

4. 메서드 본문을 줄여 표현하기 (식 본문 메서드)

```
using System;

class Program
{
    static int Square(int x) ⇒ x * x;
    static void Main(string[] args)
    {
        Console.WriteLine(Square(5));
```

```
    }
}
```

5. 선택적 매개변수

```
using System;

class Program
{
    static void PrintMessage(string message = "Hello")
    {
        Console.WriteLine(message);
    }
    static void Main(string[] args)
    {
        PrintMessage(); // Hello
        PrintMessage("Hi"); // Hi
    }
}
```

속성 사용하기

1. 속성

```
using System;

class Person
{
    private string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        var p = new Person();
        p.Name = "Alice";
        Console.WriteLine(p.Name);
    }
}
```

2. 접근자와 전체 속성

설명: 위 예제와 동일한 개념.

3. 자동으로 구현된 속성

```
using System;

class Person
{
    public string Name { get; set; } // 자동 구현 속성
}

class Program
{
    static void Main(string[] args)
    {
        var p = new Person { Name = "Bob" };
        Console.WriteLine(p.Name);
    }
}
```

4. 자동 속성 이니셜라이저

```
using System;

class Person
```

```
{
    public string Name { get; set; } = "Unknown";
}

class Program
{
    static void Main(string[] args)
    {
        var p = new Person();
        Console.WriteLine(p.Name); // Unknown
    }
}
```

5. 읽기 전용 속성과 쓰기 전용 속성

```
using System;

class Person
{
    public string Name { get; } = "Alice"; // 읽기 전용
}

class Program
{
    static void Main(string[] args)
    {
        var p = new Person();
        // p.Name = "Bob"; // 불가능
        Console.WriteLine(p.Name);
    }
}
```

6 ~ 16. 속성 관련 다양한 기능들은 앞서 다룬 개념(?., ??, 생성자에서 검증, 익명 형식 등)과 결합하여 사용할 수 있음.

예: ?.와 ?? 사용하기

```
using System;

class Person
{
    public string Name { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Person p = null;
        Console.WriteLine(p?.Name ?? "No Name");
    }
}
```

인덱서와 반복기

1. 인덱서

```
using System;

class MyCollection
{
    private string[] data = { "Apple", "Banana", "Cherry" };
    public string this[int index]
    {
        get { return data[index]; }
        set { data[index] = value; }
    }
}

class Program
{
    static void Main(string[] args)
    {
        MyCollection c = new MyCollection();
```

```
        Console.WriteLine(c[0]); // Apple
        c[1] = "Blueberry";
        Console.WriteLine(c[1]); // Blueberry
    }
}
```

4. 반복기와 yield 키워드

```
using System;
using System.Collections.Generic;

class Program
{
    static IEnumerable<int> GetNumbers()
    {
        yield return 1;
        yield return 2;
        yield return 3;
    }

    static void Main(string[] args)
    {
        foreach (var n in GetNumbers())
        {
            Console.WriteLine(n);
        }
    }
}
```

대리자(Delegate)와 이벤트(Event)

1. 대리자(위임/델리게이트)

```
using System;

delegate int MyDelegate(int x, int y);
```

```
class Program
{
    static int Add(int a, int b) ⇒ a + b;
    static void Main(string[] args)
    {
        MyDelegate del = Add;
        Console.WriteLine(del(3,4)); // 7
    }
}
```

2. 대리자를 사용하여 메서드 대신 호출하기

```
using System;

delegate void PrintDelegate(string message);

class Printer
{
    public void Print(string msg) ⇒ Console.WriteLine(msg);
}

class Program
{
    static void Main(string[] args)
    {
        Printer printer = new Printer();
        PrintDelegate del = printer.Print;
        del("Hello via delegate");
    }
}
```

3. 대리자를 사용하여 메서드 여러 개를 다중 호출하기 (멀티캐스트 대리자)

```
using System;
```

```
delegate void MultiDel();

class Program
{
    static void Hello() => Console.WriteLine("Hello");
    static void World() => Console.WriteLine("World");

    static void Main(string[] args)
    {
        MultiDel del = Hello;
        del += World;
        del(); // Hello\nWorld
    }
}
```

4. 무명 메서드

```
using System;

delegate void PrintDel(string msg);

class Program
{
    static void Main(string[] args)
    {
        PrintDel del = delegate (string m) { Console.WriteLine(m); };
        del("Anonymous method");
    }
}
```

5. 메서드의 매개변수에 대리자 형식 사용하기

```
using System;

delegate bool Compare(int x);

class Program
```

```
{
    static void PrintIf(int[] data, Compare comp)
    {
        foreach(var d in data)
        {
            if(comp(d)) Console.WriteLine(d);
        }
    }

    static void Main(string[] args)
    {
        int[] numbers = {1,2,3,4,5};
        PrintIf(numbers, x ⇒ x > 3);
    }
}
```

6. Action, Func, Predicate 대리자

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Action<string> printAction = msg ⇒ Console.WriteLine(msg);
        printAction("Hello Action");

        Func<int,int,int> addFunc = (a,b) ⇒ a+b;
        Console.WriteLine(addFunc(3,4));

        Predicate<int> isEven = x ⇒ x%2==0;
        Console.WriteLine(isEven(4));
    }
}
```

7. 메서드의 매개변수로 메서드 전달하기

설명: 이미 위 예제들에서 람다식을 활용해 메서드 전달.

이벤트

1. 이벤트

```
using System;

delegate void MyEventHandler(string msg);

class Publisher
{
    public event MyEventHandler MyEvent;
    public void RaiseEvent(string msg)
    {
        MyEvent?.Invoke(msg);
    }
}

class Subscriber
{
    public void Handler(string msg) ⇒ Console.WriteLine($"Received: {ms
g}");
}

class Program
{
    static void Main(string[] args)
    {
        Publisher pub = new Publisher();
        Subscriber sub = new Subscriber();

        pub.MyEvent += sub.Handler;
        pub.RaiseEvent("Hello Event");
    }
}
```

클래스 기타

1. 부분 클래스 (partial class)

```
// File1.cs
public partial class MyPartialClass
{
    public void Method1() ⇒ System.Console.WriteLine("Method1");
}

// File2.cs
public partial class MyPartialClass
{
    public void Method2() ⇒ System.Console.WriteLine("Method2");
}

// Program.cs
using System;

class Program
{
    static void Main(string[] args)
    {
        MyPartialClass pc = new MyPartialClass();
        pc.Method1();
        pc.Method2();
    }
}
```

(동일 네임스페이스에서 partial class를 여러 파일에 나눌 수 있음)

---

2. 정적 클래스

```
using System;

static class Utility
{
    public static void Print(string msg) ⇒ Console.WriteLine(msg);
```

```
}

class Program
{
    static void Main(string[] args)
    {
        Utility.Print("Static Class Call");
    }
}
```

3. 필드에 public을 붙여 외부 클래스에 공개하기

```
using System;

class MyClass
{
    public int data = 10;
}

class Program
{
    static void Main(string[] args)
    {
        MyClass mc = new MyClass();
        Console.WriteLine(mc.data);
    }
}
```

4. 함수형 프로그래밍 스타일: 메서드 체이닝

```
using System;

class Chain
{
    public Chain Step1() { Console.WriteLine("Step1"); return this; }
    public Chain Step2() { Console.WriteLine("Step2"); return this; }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        new Chain().Step1().Step2();
    }
}
```

5. 불변 형식

설명: 한 번 생성 후 내부 상태가 변하지 않는 클래스를 의미(프로퍼티에 set이 없음).

```
using System;

class Immutable
{
    public int Value { get; }
    public Immutable(int value)
    {
        Value = value;
    }
}

class Program
{
    static void Main(string[] args)
    {
        var im = new Immutable(10);
        Console.WriteLine(im.Value);
    }
}
```

상속으로 클래스 확장하기

1. 클래스 상속하기

```
using System;

class Parent
{
    public void Hello() => Console.WriteLine("Hello from Parent");
}

class Child : Parent
{
    public void Hi() => Console.WriteLine("Hi from Child");
}

class Program
{
    static void Main(string[] args)
    {
        Child c = new Child();
        c.Hello();
        c.Hi();
    }
}
```

2. 부모 클래스와 자식 클래스

설명: 위 예제와 동일 개념.

3. Base 클래스와 Sub 클래스

설명: Base 클래스 = 부모 클래스, Sub 클래스 = 자식 클래스

4. Object 클래스 상속

설명: 모든 클래스는 System.Object를 상속합니다. (별도 코드 생략)

5. 부모 클래스 형식 변수에 자식 클래스의 개체 할당하기

```
using System;
```

```
class Parent {}
class Child : Parent {}

class Program
{
    static void Main(string[] args)
    {
        Parent p = new Child(); // 업캐스팅
    }
}
```

6. is a 관계

설명: Child is a Parent.

7. this와 this(), base와 base()

설명: this: 자신의 멤버, base: 부모 멤버에 접근

```
using System;

class Parent
{
    public string Name = "Parent";
}

class Child : Parent
{
    public string Name = "Child";
    public void PrintNames()
    {
        Console.WriteLine(this.Name);  // Child
        Console.WriteLine(base.Name);  // Parent
    }
}

class Program
{
    static void Main(string[] args)
```

```
    {
        Child c = new Child();
        c.PrintNames();
    }
}
```

8. 봉인 클래스 (sealed class)

```
using System;

sealed class FinalClass
{
    public void Print() => Console.WriteLine("I'm sealed");
}

// class SubClass : FinalClass {} // 오류! sealed 클래스는 상속 불가.

class Program
{
    static void Main(string[] args)
    {
        FinalClass fc = new FinalClass();
        fc.Print();
    }
}
```

9. 추상 클래스 (abstract)

```
using System;

abstract class Animal
{
    public abstract void Speak();
}

class Dog : Animal
{
```

```
    public override void Speak() => Console.WriteLine("Woof!");
}

class Program
{
    static void Main(string[] args)
    {
        Animal a = new Dog();
        a.Speak();
    }
}
```

10. 자식 클래스에만 멤버 상속하기

설명: protected 멤버는 자식 클래스에게만 공개됩니다.

```
using System;

class Parent
{
    protected int secret = 42;
}

class Child : Parent
{
    public void Reveal() => Console.WriteLine(secret);
}

class Program
{
    static void Main(string[] args)
    {
        Child c = new Child();
        c.Reveal(); // 42
    }
}
```

11. 기본 클래스의 멤버 숨기기

```
using System;

class Parent
{
    public void Show() ⇒ Console.WriteLine("Parent Show");
}

class Child : Parent
{
    public new void Show() ⇒ Console.WriteLine("Child Show");
}

class Program
{
    static void Main(string[] args)
    {
        Child c = new Child();
        c.Show(); // Child Show

        Parent p = c;
        p.Show(); // Parent Show
    }
}
```

이 예제들을 통해 클래스와 개체, 필드, 속성, 생성자, 소멸자, 메서드, 인덱서, 대리자와 이벤트, 그리고 상속의 기초를 익힐 수 있습니다. 각 개념을 필요에 따라 확장하고 실제 프로젝트에서 적용해보면 더욱 깊은 이해를 얻을 수 있습니다.

메서드 오버라이드(Method Override)

1. 메서드 오버라이드: 재정의

설명: 부모 클래스에 정의된 메서드를 자식 클래스에서 재정의(override)하여 다른 동작을 구현할 수 있습니다.

```
using System;

class Parent
{
    public virtual void ShowMessage()
    {
        Console.WriteLine("Message from Parent");
    }
}

class Child : Parent
{
    public override void ShowMessage()
    {
        Console.WriteLine("Message from Child");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Parent p = new Parent();
        p.ShowMessage(); // Parent 메서드

        Child c = new Child();
        c.ShowMessage(); // Child 메서드(오버라이드된 메서드)
    }
}
```

---

2. 상속 관계에서 메서드 오버라이드

설명: 오버라이드는 상속 관계에서 부모의 메서드를 자식이 재정의하는 상황에서 사용합니다.

```
using System;

class Animal
{
    public virtual void Speak()
    {
        Console.WriteLine("Animal sound");
    }
}

class Dog : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Woof!");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Animal a = new Animal();
        a.Speak();

        Animal d = new Dog(); // 업캐스팅
        d.Speak(); // Dog의 오버라이드된 Speak 메서드
    }
}
```

---

3. 메서드 오버로드와 오버라이드

설명: 오버로드(Overload)는 같은 이름 다른 매개변수로 여러 메서드를 정의, 오버라이드(Override)는 부모의 메서드를 재정의.

```
using System;
```

```
class Calculator
{
    // 오버로드: 매개변수 유형, 개수로 구분
    public int Add(int a, int b) ⇒ a + b;
    public int Add(int a, int b, int c) ⇒ a + b + c;
}

class BaseClass
{
    public virtual void Print() ⇒ Console.WriteLine("Base Print");
}

class SubClass : BaseClass
{
    // 오버라이드: 상속받은 메서드를 재정의
    public override void Print() ⇒ Console.WriteLine("Sub Print");
}

class Program
{
    static void Main(string[] args)
    {
        Calculator calc = new Calculator();
        Console.WriteLine(calc.Add(1,2));
        Console.WriteLine(calc.Add(1,2,3));

        BaseClass b = new BaseClass();
        b.Print();
        SubClass s = new SubClass();
        s.Print();
    }
}
```

4. 메서드 오버라이드 봉인(sealed)

설명: `sealed` 키워드를 사용하여 더 이상 자식 클래스에서 재정의할 수 없게 합니다.

```
using System;

class Parent
{
    public virtual void Show() ⇒ Console.WriteLine("Parent Show");
}

class Child : Parent
{
    public sealed override void Show() ⇒ Console.WriteLine("Child Show");
}

class GrandChild : Child
{
    // public override void Show() ⇒ Console.WriteLine("GrandChild Show");
// 오류! 봉인됐기 때문에 오버라이드 불가
}

class Program
{
    static void Main(string[] args)
    {
        Child c = new Child();
        c.Show();
    }
}
```

5. ToString( ) 메서드 오버라이드

설명: 모든 클래스는 Object로부터 상속받는 ToString() 메서드를 오버라이드 가능.

```
using System;

class Person
{
    public string Name;
    public override string ToString() ⇒ $"Person: {Name}";
```

```
}

class Program
{
    static void Main(string[] args)
    {
        Person p = new Person { Name = "Alice" };
        Console.WriteLine(p.ToString()); // Person: Alice
    }
}
```

6. 메서드 오버라이드로 메서드 재사용하기

설명: 부모 메서드를 호출하면서 추가 기능을 구현할 수 있습니다.

```
using System;

class BaseLogger
{
    public virtual void Log(string msg)
    {
        Console.WriteLine("Base Log: " + msg);
    }
}

class FileLogger : BaseLogger
{
    public override void Log(string msg)
    {
        base.Log(msg); // 부모 메서드 호출
        Console.WriteLine("Additional File Log: " + msg);
    }
}

class Program
{
    static void Main(string[] args)
    {
```

```
        FileLogger logger = new FileLogger();
        logger.Log("Hello");
    }
}
```

인터페이스(Interface)

1. 인터페이스

설명: 인터페이스는 구현해야 할 메서드, 속성, 이벤트 등의 청사진을 정의합니다.

```
using System;

interface IAnimal
{
    void Speak();
}

class Dog : IAnimal
{
    public void Speak() => Console.WriteLine("Woof!");
}

class Program
{
    static void Main(string[] args)
    {
        Dog d = new Dog();
        d.Speak();
    }
}
```

---

2. 인터페이스 형식 개체에 인스턴스 담기

설명: 인터페이스 타입 변수에 인터페이스 구현한 클래스의 인스턴스를 할당할 수 있습니다.

```
using System;
```

```
interface IAnimal
{
    void Speak();
}

class Cat : IAnimal
{
    public void Speak() => Console.WriteLine("Meow!");
}

class Program
{
    static void Main(string[] args)
    {
        IAnimal animal = new Cat();
        animal.Speak();
    }
}
```

---

3. 생성자의 매개변수에 인터페이스 사용하기

설명: 의존성 주입(DI) 패턴에서 인터페이스 사용으로 유연한 코드 구현 가능.

```
using System;

interface ILogger
{
    void Log(string msg);
}

class ConsoleLogger : ILogger
{
    public void Log(string msg) => Console.WriteLine(msg);
}

class Application
{
    private ILogger logger;
```

```
    public Application(ILogger logger)
    {
        this.logger = logger;
    }
    public void Run()
    {
        logger.Log("App is running...");
    }
}

class Program
{
    static void Main(string[] args)
    {
        ILogger logger = new ConsoleLogger();
        Application app = new Application(logger);
        app.Run();
    }
}
```

4. 인터페이스를 사용한 다중 상속 구현하기

설명: C#은 클래스 다중 상속 불가하지만, 인터페이스 다중 구현은 가능.

```
using System;

interface IFly
{
    void Fly();
}

interface ISwim
{
    void Swim();
}

class Duck : IFly, ISwim
{
```

```
    public void Fly() => Console.WriteLine("Duck flying");
    public void Swim() => Console.WriteLine("Duck swimming");
}

class Program
{
    static void Main(string[] args)
    {
        Duck d = new Duck();
        d.Fly();
        d.Swim();
    }
}
```

5. 명시적인 인터페이스 구현하기

설명: 같은 이름의 멤버가 인터페이스별로 충돌할 때 명시적으로 구현 가능합니다.

```
using System;

interface IPrinter
{
    void Print();
}

interface IScanner
{
    void Print();
}

class MultiDevice : IPrinter, IScanner
{
    void IPrinter.Print() => Console.WriteLine("Printing as Printer");
    void IScanner.Print() => Console.WriteLine("Printing as Scanner");
}

class Program
{
```

```
    static void Main(string[] args)
    {
        MultiDevice md = new MultiDevice();
        ((IPrinter)md).Print();
        ((IScanner)md).Print();
    }
}
```

6. 인터페이스와 추상 클래스 비교하기

설명: 인터페이스는 구현 없는 멤버 정의만, 추상클래스는 구현 가능.
(개념 비교, 별도 코드 생략 가능)

7. IEnumerator 인터페이스 사용하기

설명: `IEnumerator`, `IEnumerable`로 커스텀 컬렉션 순회.

```
using System;
using System.Collections;

class MyCollection : IEnumerable
{
    private string[] data = {"A","B","C"};
    public IEnumerator GetEnumerator()
    {
        foreach (var item in data)
            yield return item;
    }
}

class Program
{
    static void Main(string[] args)
    {
        MyCollection mc = new MyCollection();
        foreach(var item in mc)
        {
            Console.WriteLine(item);
        }
```

```
    }
}
```

8. IDisposable 인터페이스 사용하기

설명: IDisposable 구현해 `using` 문으로 리소스 정리.

```
using System;

class MyResource : IDisposable
{
    public void Dispose()
    {
        Console.WriteLine("Resource disposed");
    }
}

class Program
{
    static void Main(string[] args)
    {
        using(MyResource r = new MyResource())
        {
            Console.WriteLine("Using resource");
        }
    }
}
```

9. 인터페이스를 사용하여 멤버 이름 강제로 적용하기

설명: 인터페이스를 구현하면 해당 인터페이스의 멤버를 강제로 구현해야 함.

특성과 리플렉션(Attribute & Reflection)

1. 특성(Attribute)

설명: 클래스나 메서드에 메타데이터를 제공하는 데 사용.

```
using System;

[Obsolete("This class is obsolete")]
class OldClass {}

class Program
{
    static void Main(string[] args)
    {
        OldClass o = new OldClass(); // 경고 발생
    }
}
```

2. Obsolete 특성 사용하기

설명: 특정 메서드나 클래스 사용 시 경고 또는 오류.
(위 예제 참조)

3. 특성의 매개변수

설명: 특성은 생성자를 통해 매개변수를 전달할 수 있음.

```
using System;

[Obsolete("Use NewClass instead", true)] // true이면 컴파일 오류
class OldClass {}

class Program
{
    static void Main(string[] args)
    {
        // OldClass o = new OldClass(); // 컴파일 오류
    }
}
```

4. [Conditional] 특성 사용하기

설명: 조건부 컴파일. 특정 심볼 정의 시에만 메서드 호출 포함.

```
#define DEBUG
using System;
using System.Diagnostics;

class Program
{
    [Conditional("DEBUG")]
    static void DebugOnly() => Console.WriteLine("Debug mode");

    static void Main(string[] args)
    {
        DebugOnly(); // DEBUG 심볼이 정의돼 있어야 호출됨
    }
}
```

---

5. 특성을 사용하여 메서드 호출 정보 얻기

설명: 리플렉션으로 특성 정보를 읽어올 수 있음.

```
using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.Method)]
class InfoAttribute : Attribute
{
    public string Message { get; }
    public InfoAttribute(string msg) { Message = msg; }
}

class Test
{
    [Info("This is a test method")]
    public void MyMethod(){}
}

class Program
{
```

```
    static void Main(string[] args)
    {
        MethodInfo mi = typeof(Test).GetMethod("MyMethod");
        var attr = (InfoAttribute)mi.GetCustomAttribute(typeof(InfoAttribute));
        Console.WriteLine(attr.Message);
    }
}
```

6. 사용자 지정 특성 만들기

설명: 위 예제에서 InfoAttribute가 사용자 지정 특성.

7. 리플렉션

설명: 런타임에 어셈블리, 타입 정보 등을 조사.
(위 예제 사용, 개념 설명)

8. Type과 Assembly 클래스

```
using System;
using System.Reflection;

class Program
{
    static void Main(string[] args)
    {
        Assembly asm = Assembly.GetExecutingAssembly();
        Console.WriteLine(asm.FullName);

        Type t = typeof(string);
        Console.WriteLine(t.Namespace);
    }
}
```

9. 특정 클래스의 메서드와 속성을 동적으로 호출하기

```
using System;
using System.Reflection;

class Person
{
    public string Name { get; set; }
    public void Greet() => Console.WriteLine($"Hello, {Name}");
}

class Program
{
    static void Main(string[] args)
    {
        Type t = typeof(Person);
        object p = Activator.CreateInstance(t);
        PropertyInfo pi = t.GetProperty("Name");
        pi.SetValue(p, "Alice");

        MethodInfo mi = t.GetMethod("Greet");
        mi.Invoke(p, null);
    }
}
```

---

10. Type 클래스로 클래스의 멤버 호출하기

설명: 위 예제와 동일한 개념.

---

11. 특정 속성에 적용된 특성 읽어 오기

(위 InfoAttribute 예제와 유사)

---

12. Type과 Activator 클래스로 개체의 인스턴스를 동적 생성하기

(위 예제에서 `Activator.CreateInstance` 사용)

개체와 개체 지향 프로그래밍(OOP)

개체 지향 프로그래밍 소개하기

설명: OOP는 캡슐화, 상속, 다형성, 추상화를 기본 개념으로 함.

2. 현실 세계의 자동차 설계도 및 자동차 개체 흉내 내기

```
using System;

class Car
{
    public string Model { get; set; }
    public void Drive() ⇒ Console.WriteLine($"{Model} driving...");
}

class Program
{
    static void Main(string[] args)
    {
        Car c = new Car { Model = "Tesla" };
        c.Drive();
    }
}
```

---

3. 개체 지향 프로그래밍의 네 가지 큰 개념

- 캡슐화(Encapsulation)
- 상속(Inheritance)
- 다형성(Polymorphism)
- 추상화(Abstraction)

4. 캡슐화를 사용하여 좀 더 세련된 프로그램 만들기

```
using System;

class BankAccount
{
    private decimal balance;
    public void Deposit(decimal amount) ⇒ balance += amount;
    public void Withdraw(decimal amount)
```

```
    {
        if (balance >= amount) balance -= amount;
        else Console.WriteLine("Insufficient funds");
    }
    public decimal GetBalance() => balance;
}

class Program
{
    static void Main(string[] args)
    {
        BankAccount acc = new BankAccount();
        acc.Deposit(100);
        acc.Withdraw(30);
        Console.WriteLine(acc.GetBalance()); // 70
    }
}
```

5. 다형성 기법을 사용하여 프로그램 융통성 높이기

설명: 다형성을 통해 같은 메서드 호출이 다른 클래스에서 다르게 동작.
(앞서 Animal, Dog 예제 참조)

6. 클래스의 멤버 종합 연습: 자동차 클래스 구현하기

```
using System;

class Car
{
    private string model;
    private int speed;

    public Car(string model)
    {
        this.model = model;
        this.speed = 0;
    }
```

```
    public void Accelerate(int amount) ⇒ speed += amount;
    public void Brake(int amount) ⇒ speed = (speed - amount < 0) ? 0 : spee
d - amount;

    public override string ToString() ⇒ $"{model} is going {speed} km/h";
}

class Program
{
    static void Main(string[] args)
    {
        Car c = new Car("BMW");
        c.Accelerate(50);
        Console.WriteLine(c);
        c.Brake(20);
        Console.WriteLine(c);
    }
}
```

이 예제들을 통해 메서드 오버라이드, 인터페이스, 특성(애트리뷰트), 리플렉션, 개체 지향 프로그래밍(OOP) 개념을 종합적으로 살펴볼 수 있습니다. 강의 시 각 예제를 실행해보고, 코드 변경을 통해 다양한 상황을 실습하면 이해가 더욱 깊어질 것입니다.

예제를 통한 실습으로 다양하게 클래스 이해하기