게임엔진프로그래밍응용

3. C# 프로그래밍

청강문화산업대학교 게임콘텐츠스쿨

반 경 진

코로나 유의사항

코로나 유의 사항

- 2023학년도 1학기 방역 및 학사운영 방안
  https://www.ck.ac.kr/archives/193175
- 2023학년도 1학기 국가공휴일 및 대학 행사 수업 대체 일정 공지
  https://www.ck.ac.kr/archives/193109

온라인 수업 저작권 유의 사항

온라인 수업 저작권 유의 사항

온라인수업 저작권 유의사항 안내

강의 저작물을 다운로드, 캡처하여
교외로 유출하는 행위는
불 법 입 니 다

저작권자의 허락 없이 저작물을 복제, 공중송신 또는 배포하는 것은
저작권 침해에 해당하며 저작권법에 처벌받을 수 있습니다.

강의 동영상과 자료 일체는 교수 및 학교의 저작물로서 저작권이 보호됩니다.
수업자료를 무단 복제 또는 배포, 전송 시 민형사상 책임을 질 수 있습니다.

Index

C# Introduction & Basic Syntax

C# Introduction

- 객체 지향적(Object-oriented), 컴포넌트 지향적(component-oriented)
- C 언어 계열에서 파생
- 정방향 선언이 거의 필요 없음
- 패키지 형태의 라이브러리
- Garbage collection - 객체가 점유하는 메모리를 자동 회수
- IL코드로 컴파일 후 닷넷 공통 언어 런타임(CLR, Common Language Runtime) 통해 실행

C# Basic Syntax

- 네임스페이스

  1. 대규모 코드 프로젝트를 구성합니다.
  2. ‘.’ 연산자를 사용하여 구분됩니다.
  3. using 지시문은 사용된 후 네임스페이스 이름을 지정할 필요가 없습니다.
  4. global 네임스페이스는 “root” 네임스페이스입니다. global::System은 항상 .NET System 네임스페이스를 가리킵니다.

C# Basic Syntax

- 변수(Variable) - 객체
- pass-by-value, pass-by-reference
- 모든 객체는 object class 상속

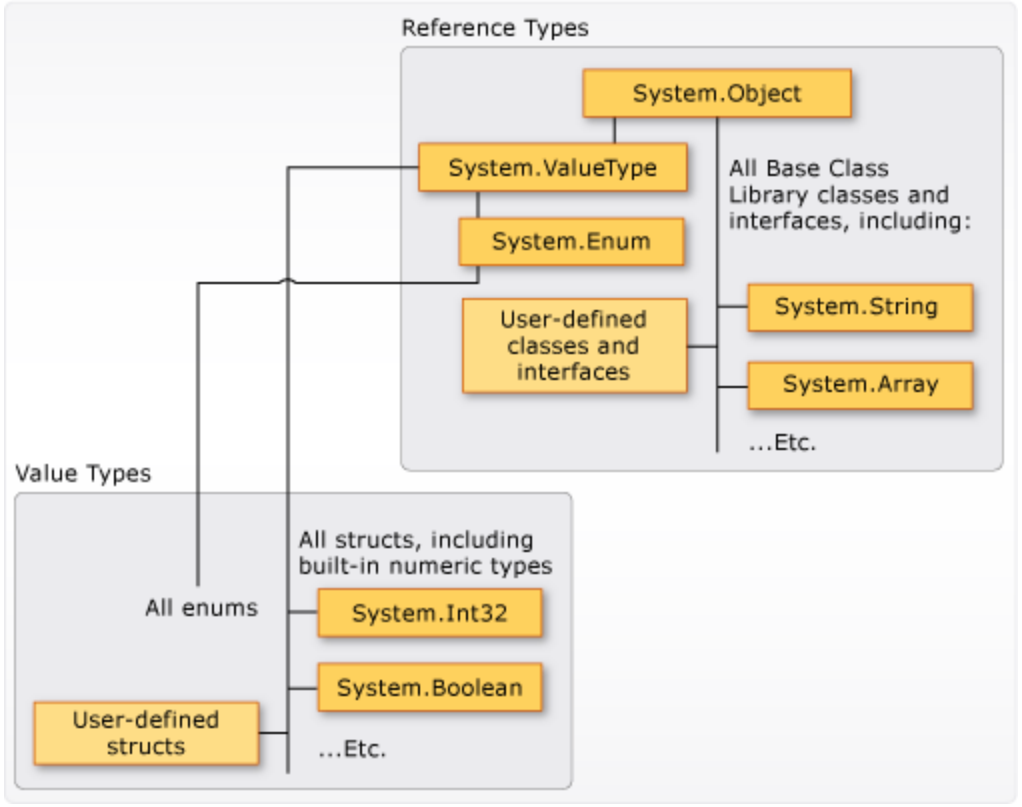
C# Basic Syntax

범주		설명
Value Type	Simple types	부호 있는 정수 : sbyte, short, int, long
		부호 없는 정수: byte, ushort, uint, ulong
		유니코드 문자: char
		부동 소수점: float, double
		고정밀 10진수: decimal
		불린언: bool
	Enum types	User-defined enum
	Struct types	User-defined struct
	Nullable types	다른 모든 값 형식을 null 값까지 표시하게 확장
Reference types	Class types	다른 모든 형식의 최종 기본 클래스: object
		유니코드 문자열: string
		User-defined class
	Interface types	User-defined interface
	Array Types	일차원 및 다차원 배열 (ex: int[], int[,])
	Delegate types	User-defined delegate

C# Basic Syntax

- 형식에 저장된 정보에는 다음 항목이 포함

1. 형식 변수에 필요한 스토리지 공간.
2. 형식이 나타낼 수 있는 최대값 및 최소값.
3. 형식에 포함되는 멤버(메서드, 필드, 이벤트 등).
4. 형식이 상속하는 기본 형식.
5. 구현하는 인터페이스.
6. 허용되는 작업 유형.



C# Basic Syntax

범주	비트	형식	범위/정밀도
부호 있는 정수	8	sbyte	-128…127
	16	short	-32,768…32,767
	32	int	-2,147,483,648…2.147.483.647
	64	long	-9,223,372,036,854,775,808… 9,223,372,036,854,775,807
부호 없는 정수	8	byte	0…255
	16	ushort	0…65,535
	32	uint	0…4,294,967,295
	64	ulong	0…18,446,744,073,709,551,615
부동소수점	32	float	$1.5 \times 10^{-45} \cdots 3.4 \times 10^{38}$, 7자리 정밀도
	64	double	$5.0 \times 10^{-324} \cdots 1.7 \times 10^{308}$, 15자리 정밀도
10진수	128	decimal	$1.0 \times 10^{-28} \cdots 7.9 \times 10^{28}$, 28자리 정밀도

C# Basic Syntax

- 클래스

  데이터 멤버(필드)와 함수 멤버(메소드와 속성 등)
  단일 상속, 다형성, 상속된 클래스가 기본 클래스를 확장

- 구조체

  데이터 멤버(필드)와 함수 멤버(메소드와 속성 등) 그러나 값 형식(value type)
  힙 할당을 필요로 하지 않음

C# Basic Syntax

- 인터페이스(interface)

  공용 함수 멤버의 명명된 집합
  상속 받은 클래스나 구조체는 인터페이스의 함수 멤버의 구현을 제공해야 함
  다수의 인터페이스를 상속 받을 수 있다.

- 대리자(delegate)

  특정 매개 변수 목록과 반환 형식을 가진 메소드에 대한 참조
  함수 포인터 개념과 유사

C# Basic Syntax

- 열거형(enum)

  명명된 상수를 가지는 고유 형식

- 배열(Array)

  int [] - 1차원 배열
  Int [,] - 2차원 배열
  Int [][] - 일차원 배열의 일차원 배열

C# Basic Syntax

- 다차원 배열과 배열의 배열(가변 배열) 차이점

```
// 3x3x3배열 선언과 사용
int[, ,] array546 = new int[3, 3, 3];
array546[0, 0, 0] = 3;

// 3x2x3배열 선언과 동시에 초기화
int[, ,] array322 = new int[, ,]
{
    {//1,n,m
        { 111, 112 },
        { 121, 122 }
    },
    {//2,n,m
        { 211, 212 },
        { 221, 222 }
    },
    {//3,n,m
        { 311, 312 },
        { 321, 322 }
    },
};
    }
}
```

```
// 3x3 가변 배열의 선언
int[][] jaggedArray = new int[3][];

jaggedArray[0] = new int[3];
jaggedArray[1] = new int[3];
jaggedArray[2] = new int[3];
```

```
// 가변 배열의 특성
int[][] jaggedArray = new int[3][];
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[9];

// 위 배열의 그림
// [] [] [] [] []
// [] [] [] []
// [] [] [] [] [] [] [] [] []
```

```
// 가변배열 초기화
int[][] jaggedArray1 = new int[3][]
jaggedArray[0] = new int[] { 11, 12, 13 };
jaggedArray[1] = new int[] { 21, 22, 23, 24 };
jaggedArray[2] = new int[] { 31, 32 };

// 가변배열 선언과 동시에 초기화
int[][] jaggedArray2 = new int[][]
{
    new int[] { 11, 12, 13 },
    new int[] { 21, 22, 23, 24 },
    new int[] { 31, 32 }
};
```

```
int[][,] jaggedArray4 = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};    int j = (int)o; // unboxing
    }
}
```

C# Basic Syntax

- Nullable

기본값 형식의 값과 추가 null 값(예: bool? true, false, null)
(C#8.0 부터는 nullable 참조 형식)

System.Nullable<T> 구조체의 인스턴스

Nullable<T> or T?

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

C# Basic Syntax

- 박싱(boxing), 언박싱(unboxing)

값 형식의 값을 object 형식으로 변환할 때 그 값을 유지하기 위해 '박스'라고 불리는 객체 인스턴스가 할당되고, 해당 값은 그 박스로 복사된다. 역으로 object 참조가 값 형식으로 형 변환이 일어날 때 그 참조된 객체가 올바른 값 형식의 박스인지 검사가 이루어지고, 복사가 성공하면 그 박스의 값이 전부 복사 된다.

```
using System;

calss Test
{
    static void Main()
    {
        int i = 123;
        object o = i; // boxing
        int j = (int)o; // unboxing
    }
}
```

C# Basic Syntax

- 표현식(Expressions)

  연산자(operators) - +, -, *, /, new 등

  피연산자(operands) - 리터럴, 필드, 지역 변수, 표현식

C# Basic Syntax

- 연산자

범주	표현식	설명
기본(primary)	x.m	멤버 접근
	x(…)	메소드와 대리자 호출
	x[…]	배열과 인덱서 접근
	x++	후위 증가
	x--	후위 감소
	new T(…)	객체와 대리자 생성
	new T(…) {…}	초기화로 객체 생성
	new {…}	익명 객체 초기화
	new T[…]	배열 생성
	typeof(T)	T에 대한 System.Type 객체 얻기
	checked(x)	checked - 컨텍스트의 표현식 평가
	unchecked(x)	uchecked - 컨텍스트의 표현식 평가

C# Basic Syntax

```
int a = 1000000;
int b = 1000000;

int c = checked(a * b); // 이 식만 점검

// 블록 안의 모든 식을 점검
checked
{
    c = a * b;
}
```

식에 checked 연산자를 지정하면 실행 시점에서 해당 형식의 산술 한계를 넘는 연산이 일어났을 때 OverflowException 예외를 던지게 할 수 있습니다.

* checked 연산자는 double, float, decimal 형식에는 아무런 영향을 미치지 않습니다.

```
처리되지 않은 예외: System.OverflowException: 산술 연산으로 인해 오버플로가 발생했습니다.
```

C# Basic Syntax

특정 식에 대해 점검을 끄고 싶으면 unchecked 연산자를 사용하면 됩니다.

```
int x = int.MaxValue;
int y = unchecked(x + 1);
unchecked
{
    int z = x + 1;
}
```

컴파일 시점에 평가되는 식은 checked 연산자와 무관하게 항상 넘침 점검이 일어납니다.
이때 unchecked 연산자를 지정하면 점검이 생략됩니다.

```
int x = int.MaxValue + 1;             // 컴파일 시점 오류
int y = unchecked(int.MaxValue + 1); // 오류 X
```

C# Basic Syntax

- 연산자

범주	표현식	설명
기본(primary)	default(T)	T형식 기본값 얻기
	delegate {…}	
단항(Unary)	+x	더하기
	-x	빼기
	!x	논리 부정
	~x	비트 부정 연산
	++x	전위 증가
	--x	전위 감소
	(T)x	명시적으로 x를 T형식으로 변환
곱셈(Multiplicative)	x * y	곱셈
	x / y	나눗셈
	x % y	나머지

C# Basic Syntax

- 연산자

범주	표현식	설명
가산(Additive)	x + y	더하기
	x - y	빼기
자리 옮김(Shift)	x << y	왼쪽 자리 옮김
	x >> y	오른쪽 자리 옮김
비교 및 형식 테스트 (Relational and type testing)	x < y	y보다 x가 작은
	x > y	y보다 x가 큰
	x <= y	y보다 x가 적거나 같은
	x >= y	y보다 x가 크거나 같은
	x is T	x가 T이면 참, 아니면 거짓
	x as T	T로 형식화 된 x를 반환 하거나,x가 T가 아니면 null
등식(Equality)	x == y	같음
	x != y	같지 않음

C# Basic Syntax

- 연산자

범주	표현식	설명
논리(Logical)	x & y	논리 AND
	x ^ y	논리 XOR
	x | y	논리 OR
	x && y	조건문 AND
	x || y	조건문 OR
널 병합	x ?? y	x가 null이면 y를 반환하고 그렇지 않으면 x를 반환
조건	x ? y : z	x가 참이면 y를 반환하고 아니면 z를 반환
대입 이나 익명 함수	x = y	대입
	x op= y	복합 대임 연산자 *=, /=, %=, +=. -=. <<=, >>=, &=, ^=, |=
	(T x) => y	익명 함수(람다식)

C# Basic Syntax

- 함수(Function, Method)
  1. Method signatures
     [access level] [return type] [method name](method parameters)
  2. Method access
     public, private, abstract, sealed, override, static
  3. Method parameters vs. arguments
  4. Passing by reference vs. passing by value
  5. Return values
     c# 7.0 ref keyword

```
public ref double GetEstimatedDistance()
{
    return ref estDistance;
}
```

  6. Async methods

C# Basic Syntax

- Method parameters - params keyword

```
public static void UseParams(params int[] list)
{
    for (int i = 0; i < list.Length; i++)
    {
        Console.Write(list[i] + " ");
    }
    Console.WriteLine();
}

public static void UseParams2(params object[] list)
{
    for (int i = 0; i < list.Length; i++)
    {
        Console.Write(list[i] + " ");
    }
    Console.WriteLine();
}
```

C# Basic Syntax

- 반목문 - for

```
for (int i = 0; i < 3; i++)
{
    Console.Write(i);
}
// Output:
// 012
```

C# Basic Syntax

- 반목문 - foreach

```
var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
foreach (int element in fibNumbers)
{
    Console.Write($"{element} ");
}
// Output:
// 0 1 1 2 3 5 8 13
```

C# Basic Syntax

- 반목문 - foreach

C# 7.3부터는 열거자의 Current 속성이 Current(ref T 여기서 T는 컬렉션 요소의 형식임)을 반환하는 경우 다음 예제와 같이 ref 또는 ref readonly 한정자를 사용하여 반복 변수를 선언

```
Span<int> storage = stackalloc int[10];
int num = 0;
foreach (ref int item in storage)
{
    item = num++;
}
foreach (ref readonly var item in storage)
{
    Console.Write($"{item} ");
}
// Output:
// 0 1 2 3 4 5 6 7 8 9
```

C# Basic Syntax

- 반목문 - await foreach

C# 8.0부터는 await foreach 문을 사용하여 비동기 데이터 스트림

```
await foreach (var item in GenerateSequenceAsync())
{
    Console.WriteLine(item);
}
```

C# Basic Syntax

- 반목문 - do

```
int n = 0;
do
{
    Console.Write(n);
    n++;
} while (n < 5);
// Output:
// 01234
```

C# Basic Syntax

- 반목문 - while

```
int n = 0;
while (n < 5)
{
    Console.Write(n);
    n++;
}
// Output:
// 01234
```

C# Basic Syntax

- 선택문 - if

```
DisplayWeatherReport(15.0);  // Output: Cold.
DisplayWeatherReport(24.0);  // Output: Perfect!

void DisplayWeatherReport(double tempInCelsius)
{
    if (tempInCelsius < 20.0)
    {
        Console.WriteLine("Cold.");
    }
    else
    {
        Console.WriteLine("Perfect!");
    }
}
```

```
DisplayMeasurement(45);
// Output: The measurement value is 45
DisplayMeasurement(-3);
// Output: Warning: not acceptable value!
// The measurement value is -3

void DisplayMeasurement(double value)
{
    if (value < 0 || value > 100)
    {
        Console.Write("Warning: not acceptable value! ");
    }

    Console.WriteLine($"The measurement value is {value}");
}
```

C# Basic Syntax

- 선택문 - if

```
DisplayCharacter('f');  // Output: A lowercase letter: f
DisplayCharacter('R');  // Output: An uppercase letter: R
DisplayCharacter('8');  // Output: A digit: 8
DisplayCharacter(',');  // Output: Not alphanumeric character: ,

void DisplayCharacter(char ch)
{
    if (char.IsUpper(ch))
    {
        Console.WriteLine($"An uppercase letter: {ch}");
    }
    else if (char.IsLower(ch))
    {
        Console.WriteLine($"A lowercase letter: {ch}");
    }
    else if (char.IsDigit(ch))
    {
        Console.WriteLine($"A digit: {ch}");
    }
    else
    {
        Console.WriteLine($"Not alphanumeric character: {ch}");
    }
}
```

C# Basic Syntax

- 선택문 - switch

  식 결과를 상수와 비교하는 관계형 패턴
  (C# 9.0 이상에서 사용 가능)

  식 결과가 상수와 같은 지 테스트하기 위한 상수 패턴(C# 7.0 이상에서 사용 가능)

```
DisplayMeasurement(-4);  // Output: Measured value is -4; too low.
DisplayMeasurement(5);  // Output: Measured value is 5.
DisplayMeasurement(30);  // Output: Measured value is 30; too high.
DisplayMeasurement(double.NaN);  // Output: Failed measurement.

void DisplayMeasurement(double measurement)
{
    switch (measurement)
    {
        case < 0.0:
            Console.WriteLine($"Measured value is {measurement}; too low.");
            break;

        case > 15.0:
            Console.WriteLine($"Measured value is {measurement}; too high.");
            break;

        case double.NaN:
            Console.WriteLine("Failed measurement.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}
```

C# Basic Syntax

- 선택문 - switch

케이스 가드

1. 일치하는 패턴과 함께 충족되어야 하는 추가 조건

2. 케이스 가드는 부울 식

```
DisplayMeasurements(3, 4);  // Output: First measurement is 3, second measurement is 4.
DisplayMeasurements(5, 5);  // Output: Both measurements are valid and equal to 5.

void DisplayMeasurements(int a, int b)
{
    switch ((a, b))
    {
        case (> 0, > 0) when a == b:
            Console.WriteLine($"Both measurements are valid and equal to {a}.");
            break;

        case (> 0, > 0):
            Console.WriteLine($"First measurement is {a}, second measurement is {b}.");
            break;

        default:
            Console.WriteLine("One or both measurements are not valid.");
            break;
    }
}
```

C# Basic Syntax

- 점프문 - break

break 문은 가장 가까운 break(즉, for, foreach, while 또는 do 루프) 또는 for을 종료

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
foreach (int number in numbers)
{
    if (number == 3)
    {
        break;
    }

    Console.Write($"{number} ");
}
Console.WriteLine();
Console.WriteLine("End of the example.");
// Output:
// 0 1 2
// End of the example.
```

C# Basic Syntax

- 점프문 - continue

continue 문은 가장 가까운 바깥쪽 continue(즉, for, foreach, while 또는 do 루프)의 새 반복을 시작

```
for (int i = 0; i < 5; i++)
{
    Console.Write($"Iteration {i}: ");

    if (i < 3)
    {
        Console.WriteLine("skip");
        continue;
    }

    Console.WriteLine("done");
}
// Output:
// Iteration 0: skip
// Iteration 1: skip
// Iteration 2: skip
// Iteration 3: done
// Iteration 4: done
```

C# Basic Syntax

- 점프문 - return

return 문은 해당 문이 나타나는 함수의 실행을 종료하고 컨트롤과 함수의 결과(있는 경우)를 호출자로 반환

```
double surfaceArea = CalculateCylinderSurfaceArea(1, 1);
Console.WriteLine($"{surfaceArea:F2}"); // output: 12.57

double CalculateCylinderSurfaceArea(double baseRadius, double height)
{
    double baseArea = Math.PI * baseRadius * baseRadius;
    double sideArea = 2 * Math.PI * baseRadius * height;
    return 2 * baseArea + sideArea;
}
```

C# Basic Syntax

- lock

lock 문은 지정된 개체에 대한 상호 배제 잠금을 획득하여 명령문 블록을 실행한 다음, 잠금을 해제

```
lock (x)
{
    // Your code...
}
```

```
object __lockObj = x;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

C# Basic Syntax

- $ - 문자열 보간

```
string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for a reply :-{{");
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
// Expected output is:
// He asked, "Is your name Horace?", but didn't wait for a reply :-{
// Horace is 34 years old.
```

C# Basic Syntax

- @ - 약어 식별자

@ 특수 문자는 축자 식별자로 사용

1. C# 키워드를 식별자로 사용하도록 설정
2. 문자열 리터럴이 축자로 해석
3. 이름이 서로 충돌하는 경우 특성 간에 구분

C# Basic Syntax

- @ - 약어 식별자

  C# 키워드를 식별자로 사용하도록 설정

```
string[] @for = { "John", "James", "Joan", "Jamie" };
for (int ctr = 0; ctr < @for.Length; ctr++)
{
   Console.WriteLine($"Here is your gift, {@for[ctr]}!");
}
// The example displays the following output:
//      Here is your gift, John!
//      Here is your gift, James!
//      Here is your gift, Joan!
//      Here is your gift, Jamie!
```

C# Basic Syntax

- @ - 약어 식별자

  문자열 리터럴이 축자로 해석

```
string filename1 = @"c:\documents\files\u0066.txt";
string filename2 = "c:\\documents\\files\\u0066.txt";

Console.WriteLine(filename1);
Console.WriteLine(filename2);
// The example displays the following output:
//     c:\documents\files\u0066.txt
//     c:\documents\files\u0066.txt

string s1 = "He said, \"This is the last \u0063hance\x0021\"";
string s2 = @"He said, ""This is the last \u0063hance\x0021""";

Console.WriteLine(s1);
Console.WriteLine(s2);
// The example displays the following output:
//     He said, "This is the last chance!"
//     He said, "This is the last \u0063hance\x0021"
```

C# Basic Syntax

- @ - 약어 식별자

  이름이 서로 충돌하는 경우 특성 간에 구분

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class Info : Attribute
{
   private string information;

   public Info(string info)
   {
      information = info;
   }
}

[AttributeUsage(AttributeTargets.Method)]
public class InfoAttribute : Attribute
{
   private string information;

   public InfoAttribute(string info)
   {
      information = info;
   }
}

[Info("A simple executable.")]
// Generates compiler error CS1614. Ambiguous Info and InfoAttribute.
// Prepend '@' to select 'Info' ([@Info("A simple executable.")]). Specify the full name 'InfoAttribute' to select it.
public class Example
{
   [InfoAttribute("The entry point.")]
   public static void Main()
   {
   }
}
```

C# Generic, Linq, Lambda

C# Generic

제네릭은 .NET에 형식 매개 변수의 개념을 도입하여 클래스 또는 메서드가 클라이언트 코드에 의해 선언되고 인스턴스화될 때까지 하나 이상의 형식의 사양을 연기하는 클래스 및 메서드를 디자인할 수 있도록 합니다.
예를 들어 제네릭 형식 매개 변수 T를 사용하여 여기에 표시된 것처럼, 다른 클라이언트 코드에서 런타임 캐스팅 또는 boxing 작업에 대한 비용이나 위험을 발생하지 않고 사용할 수 있는 단일 클래스를 작성할 수 있습니다.

제네릭 클래스 및 메서드는 제네릭이 아닌 클래스 및 메서드에서는 결합할 수 없는 방식으로 재사용성, 형식 안전성 및 효율성을 결합합니다. 제네릭은 컬렉션 및 해당 컬렉션에서 작동하는 메서드에서 가장 자주 사용됩니다.
System.Collections.Generic 네임스페이스에는 몇 가지 제네릭 기반 컬렉션 클래스가 있습니다.
ArrayList와 같은 제네릭이 아닌 컬렉션은 권장되지 않으며 호환성을 위해 유지 관리됩니다.

C# Generic

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

C# Generic

```
// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node? next;
        public Node? Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }
```

```
    private Node? head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node? current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

```
class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

C# Generic

- 제네릭 형식을 사용하여 코드 재사용, 형식 안전성 및 성능을 최대화합니다.
- 가장 일반적으로 제네릭은 컬렉션 클래스를 만드는 데 사용됩니다.
- .NET 클래스 라이브러리에는 System.Collections.Generic 네임스페이스의 여러 제네릭 컬렉션 클래스가 포함됩니다.
- 제네릭 컬렉션은 네임스페이스 같은 ArrayListSystem.Collections 클래스 대신 가능하면 언제든지 사용해야 합니다.
- 사용자 고유의 제네릭 인터페이스, 클래스, 메서드, 이벤트 및 대리자를 만들 수 있습니다.
- 제네릭 클래스는 특정 데이터 형식의 메서드에 액세스할 수 있도록 제한될 수 있습니다.
- 제네릭 데이터 형식에 사용되는 형식에 대한 정보는 리플렉션을 사용하여 런타임 시 얻을 수 있습니다.

C# Generic

- C++ 템플릿과 C# 제네릭의 차이점

1. C# 제네릭은 C++ 템플릿과 동일한 수준의 유연성을 제공하지 않습니다. 예를 들어 C# 제네릭 클래스에서 산술 연산자는 호출할 수 없지만 사용자 정의 연산자는 호출할 수 있습니다.
2. C#에서는 template C<int i> {} 같은 비형식 템플릿 매개 변수를 허용하지 않습니다.
3. C#은 명시적 특수화 즉, 특정 형식에 대한 템플릿의 사용자 지정 구현을 지원하지 않습니다.
4. C#은 부분 특수화 즉, 형식 인수의 하위 집합에 대한 사용자 지정 구현을 지원하지 않습니다.
5. C#에서는 형식 매개 변수를 제네릭 형식에 대한 기본 클래스로 사용할 수 없습니다.
6. C#에서는 형식 매개 변수가 기본 형식을 사용할 수 없습니다.
7. C#에서 제네릭 형식 매개 변수 자체는 제네릭이 될 수 없지만 생성된 형식은 제네릭으로 사용할 수 있습니다. C++에서는 템플릿 매개 변수를 허용합니다.

C# Generic

8. C++에서는 템플릿의 일부 형식 매개 변수에 적합하지 않아 형식 매개 변수로 사용되는 특정 형식을 확인하는 코드를 허용합니다. C#에서는 제약 조건을 충족하는 모든 형식에서 작동하는 방식으로 작성할 코드가 클래스에 필요합니다. 예를 들어 C++에서는 산술 연산자 + 및 -를 사용하는 함수를 형식 매개 변수의 개체에서 작성하여 이러한 연산자를 지원하지 않는 형식으로 템플릿을 인스턴스화할 때 오류를 생성할 수 있습니다. C#에서는 이를 허용하지 않습니다. 허용되는 유일한 언어 구문은 제약 조건에서 추론할 수 있는 구문입니다.

C# Linq

- **LINQ(Language-Integrated Query)**

  C# 언어에 직접 쿼리 기능을 통합하는 방식을 기반으로 하는 기술 집합

```
// Specify the data source.
int[] scores = { 97, 92, 81, 60 };

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}

// Output: 97 92 81
```

C# Linq

```
var queryGroupMax =
    students
        .GroupBy(student => student.Year)
        .Select(studentGroup => new
        {
            Level = studentGroup.Key,
            HighestScore = studentGroup.Select(student2 => student2.ExamScores.Average()).Max()
        });

int count = queryGroupMax.Count();
Console.WriteLine($"Number of groups = {count}");

foreach (var item in queryGroupMax)
{
    Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
}
```

C# Lambda

- **Lambda**

  '람다 식'을 사용하여 익명 함수를 만듭니다. 람다 선언 연산자를 사용하여 본문에서 람다의 매개 변수 목록을 구분합니다. 람다 식은 다음과 같은 두 가지 형식 중 하나일 수 있습니다.

  식 람다 - (input-parameters) => expression

  문 람다 - (input-parameters) => { <sequence-of-statements> }

  람다 식을 만들려면 람다 연산자 왼쪽에 입력 매개 변수를 지정하고(있는 경우) 다른 쪽에 식이나 문 블록을 지정합니다.

C# Lambda

- **Lambda**

  람다 식은 대리자 형식으로 변환할 수 있습니다.

  람다 식을 변환할 수 있는 대리자 형식은 해당 매개 변수 및 반환 값의 형식에 따라 정의됩니다.

  람다 식에서 값을 반환하지 않는 경우 Action 대리자 형식 중 하나로 변환할 수 있습니다.

  값을 반환하는 경우 Func 대리자 형식으로 변환할 수 있습니다.
  예를 들어 매개 변수는 두 개지만 값을 반환하지 않는 람다 식은 Action<T1,T2> 대리자로 변환할 수 있습니다.

  매개 변수가 하나이고 값을 반환하는 람다 식은 Func<T,TResult> 대리자로 변환할 수 있습니다.

C# Lambda

- **Lambda**

  람다 식은 대리자 형식으로 변환할 수 있습니다.

  람다 식을 변환할 수 있는 대리자 형식은 해당 매개 변수 및 반환 값의 형식에 따라 정의됩니다.

  람다 식에서 값을 반환하지 않는 경우 Action 대리자 형식 중 하나로 변환할 수 있습니다.

  값을 반환하는 경우 Func 대리자 형식으로 변환할 수 있습니다.
  예를 들어 매개 변수는 두 개지만 값을 반환하지 않는 람다 식은 Action<T1,T2> 대리자로 변환할 수 있습니다.

  매개 변수가 하나이고 값을 반환하는 람다 식은 Func<T,TResult> 대리자로 변환할 수 있습니다.

C# Lambda

- **Lambda**

  람다 식은 대리자 형식으로 변환할 수 있습니다.

  람다 식을 변환할 수 있는 대리자 형식은 해당 매개 변수 및 반환 값의 형식에 따라 정의됩니다.

  람다 식에서 값을 반환하지 않는 경우 Action 대리자 형식 중 하나로 변환할 수 있습니다.

  값을 반환하는 경우 Func 대리자 형식으로 변환할 수 있습니다.
  예를 들어 매개 변수는 두 개지만 값을 반환하지 않는 람다 식은 Action<T1,T2> 대리자로 변환할 수 있습니다.

  매개 변수가 하나이고 값을 반환하는 람다 식은 Func<T,TResult> 대리자로 변환할 수 있습니다.

C# Lambda

식 람다

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25

System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x;
Console.WriteLine(e);
// Output:
// x => (x * x)

int[] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);
Console.WriteLine(string.Join(" ", squaredNumbers));
// Output:
// 4 9 16 25
```

C# Lambda

문 람다

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

C# Lambda

비동기 람다

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += button1_Click;
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

C# Lambda

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event handler.\n";
        };
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

Unity C# Pattern

Unity C# Pattern

- Singleton Pattern

  단 하나의 클래스 인스턴스만을 갖도록 보장

  전역적인 접근점을 제공

  유니티에 적용을 해보자면 게임을 관리하는 매니저(Manager) 계열의 클래스에 적합

Unity C# Pattern

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

/// <summary>
/// Inherit from this base class to create a singleton.
/// e.g. public class MyClassName : Singleton<MyClassName> {}
/// </summary>
public class MySingleton2<T> : MonoBehaviour where T : MonoBehaviour
{
    // Check to see if we're about to be destroyed
    private static bool m_ShuttingDown = false;
    private static object m_Lock = new object();
    private static T m_Instance;

    /// <summary>
    /// Access singleton instance through this propriety.
    /// </summary>
    public static T Instance
    {
```

Unity C# Pattern

```
get
{
    if (m_ShuttingDown)
    {
        Debug.LogWarning("[Singleton] Instance '" + typeof(T) +
            "' already destroyed. Returning null.");
        return null;
    }

    lock (m_Lock)
    {
        if (m_Instance == null)
        {
            // Search for existing instance.
            m_Instance = (T)FindObjectOfType(typeof(T));

            // Create new instance if one doesn't already exist.
            if (m_Instance == null)
            {
                // Need to create a new GameObject to attach the singleton to.
                var singletonObject = new GameObject();
                m_Instance = singletonObject.AddComponent<T>();
                singletonObject.name = typeof(T).ToString() + " (Singleton)";
```

Unity C# Pattern

```
                    // Make instance persistent.
                    DontDestroyOnLoad(singletonObject);
                }
            }

            return m_Instance;
        }
    }

    private void OnApplicationQuit()
    {
        m_ShuttingDown = true;
    }


    private void OnDestroy()
    {
        m_ShuttingDown = true;
    }
}
```

Unity C# Pattern

- Dispatcher Pattern

  디스패처 패턴은 작업자 스레드에서 메인 스레드로 실행할 코드를 스케줄링(또는 디스패치)하는 방법.

  예) 네트워크 쓰레드에서 입력 받은 패킷 메인 스레드로 전달

  작업자 스레드에서 전역 버퍼에 객체 입력

  메인 스레드의 Update()에서 전역 버퍼의 입력된 객체 처리

Unity C# Pattern

```
public interface IDispatcher
{
    void Invoke(Action fn);
}
```

Unity C# Pattern

```
public class Dispatcher : IDispatcher
{
    private static Dispatcher instance;

    public static Dispatcher Instance
    {
        get
        {
            if (instance == null)
            {
                // Instance singleton on first use.
                instance = new Dispatcher();
            }

            return instance;
        }
    }
```

Unity C# Pattern

```
public List<Action> pending = new List<Action>();

public void Invoke(Action fn)
{
    lock (pending)
    {
        pending.Add(fn);
    }
}

public void InvokePending()
{
    lock (pending)
    {
        foreach (var action in pending)
        {
            action();
        }
```

Unity C# Pattern

```
            pending.Clear();
        }
    }
}
```

```
Dispatcher.Instance.Invoke(
    () => SomethingThatMustRunInTheMainThread()
);
```

```
public class DispatcherUpdate : MonoBehaviour
{
    void Update()
    {
        Dispatcher.Instance.InvokePending();
    }
}
```

Unity C# Pattern

```
public class Dispatcher : MonoBehaviour
{
    private static Dispatcher main = null;
    private readonly Queue<Action> queued = new Queue<Action>();

    private void Awake()
    {
        Init();
    }

    private void Update()
    {
        lock (queued)
        {
            while (queued.Count > 0)
            {
                queued.Dequeue().Invoke();
            }
        }
    }
```

Unity C# Pattern

```
public static void Init()
{
    if (main == null)
    {
        main = FindObjectOfType<Dispatcher>();
        if (main == null)
        {
            main = new GameObject("[Dispatcher]").AddComponent<Dispatcher>();
        }
    }

    DontDestroyOnLoad(main.gameObject);
}

public static void Register(Action action)
{
    main.queued.Enqueue(action);
}
```

Unity C# Pattern

```
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.SubsystemRegistration)]
    private static void ResetDomain()
    {
        main = null;
    }
}
```

Coroutine VS. Task

Coroutine

- Coroutine

```
public class CoroutineTest : MonoBehaviour
{
    public int count = 1000;

    private void Awake()
    {
        StartCoroutine(Coroutine());
    }

    IEnumerator Coroutine()
    {
        double value = 0d;
        while (true)
        {
            for (int i = 0; i < count; i++)
            {
                for (int j = 0; j < count; j++)
                {
                    value += Random.Range(-1.0f, 1.0f);
                }
            }

            yield return null;
        }
    }
}
```

유니티에서의 코루틴은 근본적으로 동기처리

1회의 코루틴 루프에서 아주 무거운 작업을 처리해야한다면 렌더링 프레임 수치에 직접적인 영향력을 행사

코루틴에서도 허용된 시간을 초과하면 yield return null으로 프레임을 넘겨 주는 식으로 나머지 작업은 다음 프레임에 처리하도록 로직을 작성. (권장하는 방법은 아님)

코루틴에 작업을 태울 때에는 해당 프로세스의 복잡도가 얼마나 높은지, 최대로 걸릴 수 있는 시간이 어느정도인지 충분히 인지하고 로직을 작성

Task

- Task - Async

  서버와 소켓 통신을 하거나, 방대한 양의 데이터를 처리해야 할 때 사용을 고려

```
class StaticDataUtil
{
    public static async void Decrypt(StaticDataContext context, Action<StaticData[]> onComplete)
    {
        int size = context.size;
        var datas = new StaticData[size];

        Task task = Task.Run(() => {
            byte[] bytes = null;
            string json = null;
            for (int i = 0; i < size; i++) {
                bytes = context.datas[i];
                json = GZipCompressor.Unzip(bytes);
                json = GZipCompressor.XOR(json, "N.EFGame.BD3");

                datas[i] = JsonConvert.DeserializeObject<StaticData>(json);
            }
        });

        await task;
        onComplete?.Invoke(datas);
    }
}
```

Unity C# Job System

Unity C# Job System

Unity C# 잡 시스템을 통해 사용자는 나머지 Unity 기능과 잘 연동하고 수정 코드 작성을 용이하게 해주는 멀티스레드 코드를 작성할 수 있습니다.

멀티스레드 코드를 작성하면 성능이 향상되는 이점을 누릴 수 있으며, 프레임 속도도 대폭 개선됩니다. 버스트 컴파일러를 C# 잡과 함께 사용하면 코드 생성 품질이 개선되며, 모바일 디바이스의 배터리 소모량도 크게 감소합니다.

C# 잡 시스템의 핵심은 Unity의 내부 기능(Unity의 네이티브 잡 시스템)과 통합된다는 점입니다. 사용자가 작성한 코드와 Unity는 동일한 워커 스레드를 공유합니다. 이러한 협력을 이용하면 CPU 코어보다 많은 스레드를 만들지 않아도 되므로 CPU 리소스에 대한 경쟁을 피할 수 있습니다.

Unity C# Job System

잡 시스템(job system)은 스레드를 대신하여 잡을 만들어 멀티스레드 코드를 관리합니다.

잡 시스템은 여러 코어에 걸쳐 워커 스레드 그룹을 관리합니다. 컨텍스트가 바뀌지 않도록 하기 위해 일반적으로 CPU 논리 코어당 하나의 워커 스레드가 있지만, 시스템이 운영체제나 기타 전용 애플리케이션에서 사용할 코어 몇 개를 예약해 둘 수 있습니다.

잡 시스템은 잡 대기열에 잡을 배치하여 실행합니다. 잡 시스템의 워커 스레드는 잡 대기열에서 항목을 가져와 실행합니다. 잡 시스템은 종속성을 관리하고 작업이 올바른 순서대로 실행되도록 합니다.

Unity C# Job System

- **NativeContainer**

  데이터 복사의 안전 시스템 프로세스에 대한 단점은 잡의 결과가 각 복사본 내에 격리된다는 것입니다. 이러한 제한을 극복하려면 결과를 NativeContainer라고 불리는 공유 메모리 타입에 저장해야 합니다.

  NativeContainer는 네이티브 메모리에 상대적으로 안전한 C# 래퍼를 제공하는 관리되는 값 타입입니다. 여기에는 관리되지 않는 할당에 대한 포인터가 들어 있습니다. Unity C# 잡 시스템과 함께 NativeContainer를 사용하면 잡이 복사본으로 작업하는 것이 아니라 메인 스레드와 공유되는 데이터에 액세스할 수 있습니다.

  Unity는 NativeArray라고 불리는 NativeContainer와 함께 제공됩니다. 또한 NativeSlice로 NativeArray를 조작하여 특정 포지션에서 NativeArray의 하위 집합을 특정 길이로 가져올 수도 있습니다.
  참고: 엔티티 컴포넌트 시스템(ECS) 패키지는 다른 타입의 NativeContainer를 포함하도록 Unity.Collections 네임스페이스를 확장합니다.

  - NativeList - 크기 변경이 가능한 NativeArray입니다.
  - NativeHashMap - 키 및 값 쌍입니다.
  - NativeMultiHashMap - 키당 여러 개의 값입니다.
  - NativeQueue - 선입선출(FIFO) 대기열입니다.

Unity C# Job System

- **잡 만들기**

  Unity에서 잡을 만들려면 IJob 인터페이스를 구현해야 합니다. IJob을 사용하면 실행 중인 다른 잡과 병렬로 실행되는 단일 잡을 예약할 수 있습니다.
  참고: '잡'은 IJob 인터페이스를 구현하는 구조체에 관한 Unity의 포괄적인 용어입니다.
  잡을 만들려면 다음을 수행해야 합니다.

  1. IJob을 구현하는 구조체를 만듭니다.
  2. 해당 잡이 사용하는 멤버 변수(blittable 타입 또는 NativeContainer 타입)를 추가합니다.
  3. 구조체에 Execute 메서드를 만들고 그 안에서 잡을 구현합니다.

  잡을 실행할 때 Execute 메서드는 단일 코어에서 실행됩니다.
  참고: 잡을 디자인할 때는 데이터 복사본에서 동작한다는 점을 기억하십시오(NativeContainer의 경우는 예외). 따라서 메인 스레드에서 잡의 데이터에 액세스하는 유일한 방법은 NativeContainer에 작성하는 것입니다.

Unity C# Job System

간단한 잡 정의 예시

```
// Job adding two floating point values together
public struct MyJob : IJob
{
    public float a;
    public float b;
    public NativeArray<float> result;

    public void Execute()
    {
        result[0] = a + b;
    }
}
```

Unity C# Job System

- **잡 예약**

  메인 스레드에서 잡을 예약하려면 다음을 수행해야 합니다.

  1. 잡을 인스턴스화합니다.
  2. 잡의 데이터를 채웁니다.
  3. Schedule 메서드를 호출합니다.

  Schedule을 호출하면 적절한 시점에 실행되도록 잡을 잡 대기열에 넣습니다. 예약된 잡은 인터럽트할 수 없습니다.
  참고: 메인 스레드에서는 Schedule만 호출할 수 있습니다.

Unity C# Job System

잡 예약 예시

```
// Create a native array of a single float to store the result. This example waits for the job to complete for illustration purposes
NativeArray<float> result = new NativeArray<float>(1, Allocator.TempJob);

// Set up the job data
MyJob jobData = new MyJob();
jobData.a = 10;
jobData.b = 10;
jobData.result = result;

// Schedule the job
JobHandle handle = jobData.Schedule();

// Wait for the job to complete
handle.Complete();

// All copies of the NativeArray point to the same memory, you can access the result in "your" copy of the NativeArray
float aPlusB = result[0];

// Free the memory allocated by the result array
result.Dispose();
```

Unity C# Job System

- JobHandle 및 종속성

  잡의 Schedule 메서드를 호출하면 JobHandle을 반환합니다.
  코드의 JobHandle을 다른 잡에 대한 종속성으로 사용할 수 있습니다.
  잡이 다른 잡의 결과에 종속되면 첫 번째 잡의 JobHandle을 파라미터로 두 번째 잡의 Schedule 메서드에 다음과 같이 전달할 수 있습니다.

```
JobHandle firstJobHandle = firstJob.Schedule();
secondJob.Schedule(firstJobHandle);
```

Unity C# Job System

- **종속성 결합**

  잡에 종속성이 많은 경우에는 JobHandle.CombineDependencies 메서드를 사용하여 결합할 수 있습니다. CombineDependencies를 이용하면 종속성을 Schedule 메서드로 전달할 수 있습니다.

```
NativeArray<JobHandle> handles = new NativeArray<JobHandle>(numJobs, Allocator.TempJob);

// Populate `handles` with `JobHandles` from multiple scheduled jobs...

JobHandle jh = JobHandle.CombineDependencies(handles);
```

Unity C# Job System

- 여러 개의 잡과 종속성 예시

Job code:

```
// Job adding two floating point values together
public struct MyJob : IJob
{
    public float a;
    public float b;
    public NativeArray<float> result;

    public void Execute()
    {
        result[0] = a + b;
    }
}

// Job adding one to a value
public struct AddOneJob : IJob
{
    public NativeArray<float> result;

    public void Execute()
    {
        result[0] = result[0] + 1;
    }
}
```

Unity C# Job System

- 여러 개의 잡과 종속성 예시

```
Main thread code:

// Create a native array of a single float to store the result in. This example waits for the job to complete
NativeArray<float> result = new NativeArray<float>(1, Allocator.TempJob);

// Setup the data for job #1
MyJob jobData = new MyJob();
jobData.a = 10;
jobData.b = 10;
jobData.result = result;

// Schedule job #1
JobHandle firstHandle = jobData.Schedule();

// Setup the data for job #2
AddOneJob incJobData = new AddOneJob();
incJobData.result = result;

// Schedule job #2
JobHandle secondHandle = incJobData.Schedule(firstHandle);

// Wait for job #2 to complete
secondHandle.Complete();

// All copies of the NativeArray point to the same memory, you can access the result in "your" copy of the NativeArray
float aPlusB = result[0];

// Free the memory allocated by the result array
result.Dispose();
```

Unity C# Job System

- **ParallelFor 잡**

  잡을 예약할 경우 하나의 잡은 하나의 작업만 수행할 수 있습니다. 게임에서는 수많은 오브젝트에 대해 동일한 작업을 수행하는 경우가 흔합니다. 따라서 이를 위해 IJobParallelFor라고 불리는 별도의 잡 타입이 제공됩니다.
  참고: 'ParallelFor'는 IJobParallelFor 인터페이스를 구현하는 구조체에 관한 Unity의 포괄적인 용어입니다.

  ParallelFor 잡은 데이터의 NativeArray를 사용하여 데이터 소스로 동작합니다. ParallelFor 잡은 여러 개의 코어에서 동작합니다. 코어당 하나의 잡이 있으며, 각각 일정량의 작업을 처리합니다. IJobParallelFor는 IJob과 비슷하게 동작하지만, 단일 Execute가 아니라 데이터 소스의 항목당 하나의 Execute 메서드를 호출합니다. Execute 메서드에는 정수 파라미터가 있습니다. 이 인덱스는 잡 구현 내에서 데이터 소스의 단일 요소에 액세스하여 동작합니다.

Unity C# Job System

- ParallelFor 잡 정의 예시

```
struct IncrementByDeltaTimeJob: IJobParallelFor
{
    public NativeArray<float> values;
    public float deltaTime;

    public void Execute (int index)
    {
        float temp = values[index];
        temp += deltaTime;
        values[index] = temp;
    }
}
```

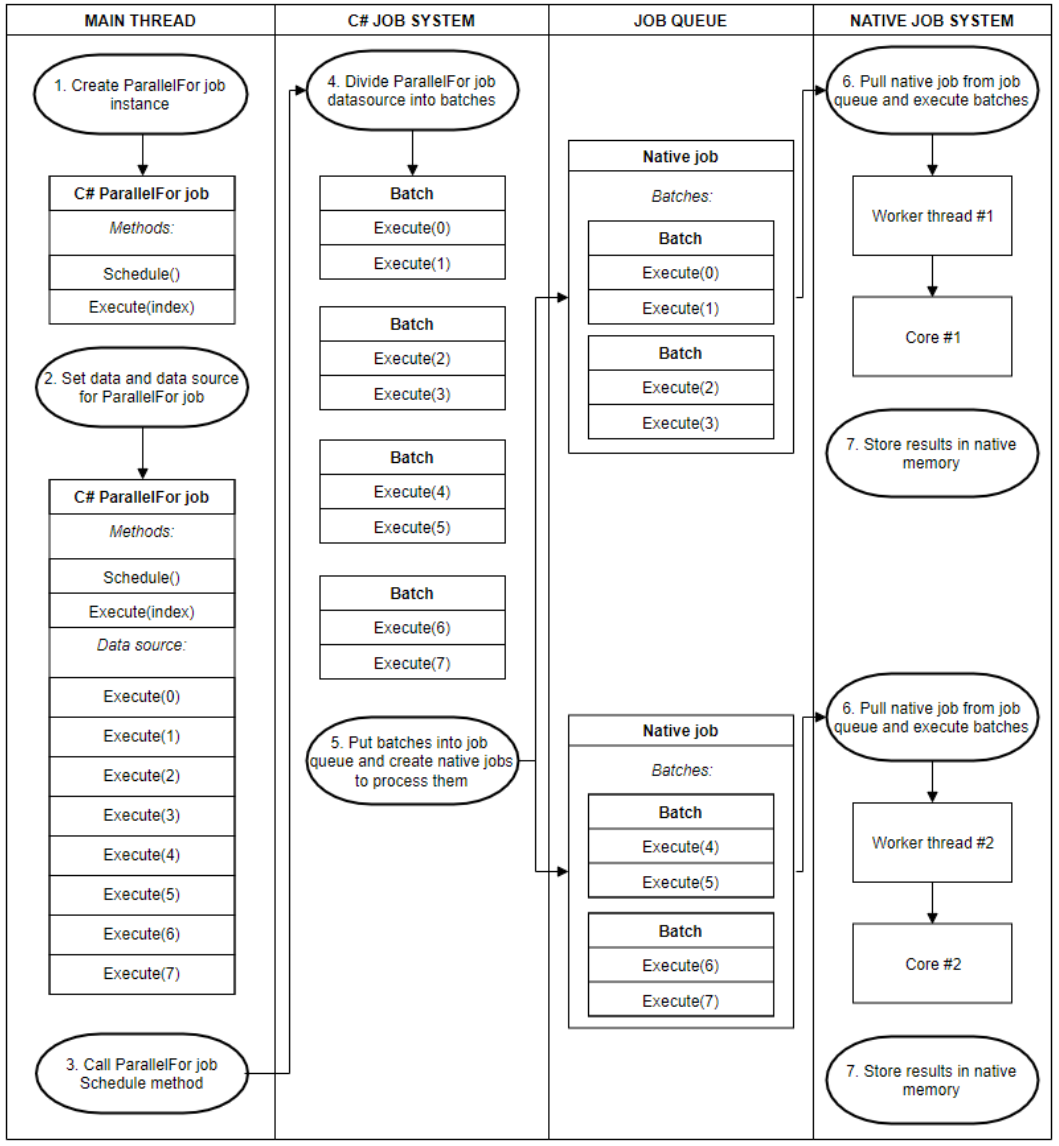
Unity C# Job System

- **ParallelFor 잡 예약**

  ParallelFor 잡을 예약할 때는 분할할 NativeArray 데이터 소스의 길이를 지정해야 합니다. Unity C# 잡 시스템은 구조체에 여러 개의 NativeArray가 있으면 사용자가 어느 것을 데이터 소스로 사용할지 알 수 없습니다. 또한 데이터 소스의 길이는 C# 잡 시스템에 예상되는 Execute 메서드 개수도 알려줍니다.
  보이지 않는 곳에서 ParallelFor 잡 예약은 더 복잡하게 동작합니다. ParallelFor 잡을 예약할 때 C# 작업 시스템은 작업을 배치로 나누어 코어 간에 배포합니다. 각 배치에는 Execute 메서드의 하위 집합이 포함됩니다. 그러면 C# 잡 시스템은 Unity의 네이티브 잡 시스템에서 CPU 코어당 하나의 잡을 예약한 후 해당 네이티브 잡에 완료할 일부 배치를 전달합니다.

  여러 코어 간에 배치를 나누는 ParallelFor 잡
  한 네이티브 잡이 다른 네이티브 잡보다 배치를 먼저 완료하면 다른 네이티브 잡의 남은 배치를 가져옵니다. 한 번에 네이티브 잡의 남은 배치의 절반만 가져오므로 캐시 집약성이 보장됩니다.
  프로세스를 최적화하려면 배치 수를 지정해야 합니다. 배치 수는 몇 개의 잡을 가져오고 스레드 간에 작업 재배포를 어떻게 세부 조정할지를 제어합니다. 작은 배치 수(예:1)를 지정하면 스레드 간에 작업을 더 균등하게 배포할 수 있습니다. 성능 소모가 더 발생하더라도 때로는 배치 수를 늘려야 할 때도 있습니다. 1부터 시작하여 성능 향상을 무시할 수 있는 수준까지 개수를 늘리는 것도 좋은 전략입니다.

Unity C# Job System



Unity C# Job System

ParallelFor 잡 예약 예시

```
Job code:

// Job adding two floating point values together
public struct MyParallelJob : IJobParallelFor
{
    [ReadOnly]
    public NativeArray<float> a;
    [ReadOnly]
    public NativeArray<float> b;
    public NativeArray<float> result;

    public void Execute(int i)
    {
        result[i] = a[i] + b[i];
    }
}
```

Unity C# Job System

ParallelFor 잡 예약 예시

```
Main thread code:

NativeArray<float> a = new NativeArray<float>(2, Allocator.TempJob);

NativeArray<float> b = new NativeArray<float>(2, Allocator.TempJob);

NativeArray<float> result = new NativeArray<float>(2, Allocator.TempJob);

a[0] = 1.1;
b[0] = 2.2;
a[1] = 3.3;
b[1] = 4.4;

MyParallelJob jobData = new MyParallelJob();
jobData.a = a;
jobData.b = b;
jobData.result = result;

// Schedule the job with one Execute per index in the results array and only 1 item per processing batch
JobHandle handle = jobData.Schedule(result.Length, 1);

// Wait for the job to complete
handle.Complete();

// Free the memory allocated by the arrays
a.Dispose();
b.Dispose();
result.Dispose();
```

Unity C# Job System

- **ParallelForTransform 잡**

  ParallelForTransform 잡은 또 다른 타입의 ParallelFor 잡으로, 트랜스폼에서 동작하도록 특별히 디자인되었습니다.
  참고: ParallelForTransform 잡은 IJobParallelForTransform 인터페이스를 구현하는 잡에 관한 Unity의 포괄적인 용어입니다.

Unity C# Job System

- **C# 잡 시스템 팁 및 문제 해결**

  Unity C# 잡 시스템을 사용할 때는 다음 사항을 준수해야 합니다.

  - 잡에서 정적 데이터에 액세스하지 않기
  - 예약된 배치를 플러시하기
  - NativeContainer 콘텐츠를 업데이트하지 않기
  - JobHandle.Complete를 호출하여 소유권 다시 얻기
  - 메인 스레드에서 예약 및 완료 사용
  - 적시에 예약 및 완료 사용
  - NativeContainer 타입을 읽기 전용으로 표시
  - 데이터 종속성 검사
    JobHandle.Complete를 찾아보면 메인 스레드를 대기하도록 만드는 데이터 종속성의 위치를 추적할 수 있습니다.
  - 잡 디버깅
    잡에는 Schedule 대신에 사용하여 메인 스레드의 잡을 즉시 실행할 수 있는 Run 함수가 있습니다. 이 함수는 디버깅 목적으로도 사용할 수 있습니다.
  - 잡에서 관리되는 메모리 할당하지 않기