중앙대학교 ZeroPage Devils Camp

Rust 게임 프로그래밍

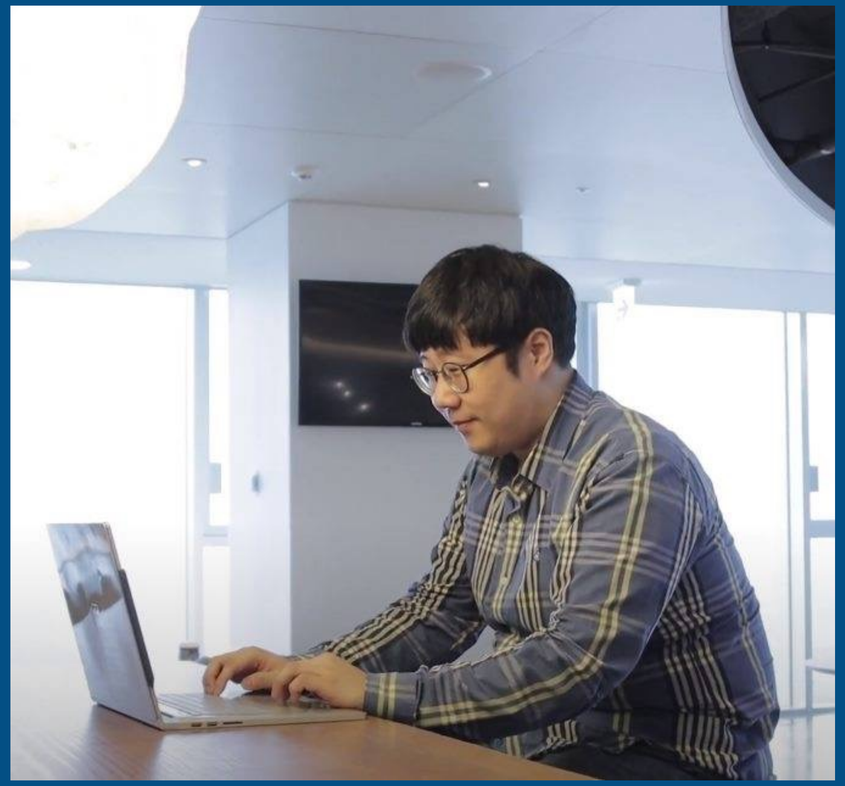
옥찬호

utilForever@gmail.com

발표자 소개

utilForever@gmail.com
utilForever

- 옥찬호 (Chris Ohk)
  - (현) Momenti Engine Engineer
  - (전) Nexon Korea Game Programmer
  - Microsoft Developer Technologies MVP
  - C++ Korea Founder & Administrator
  - Reinforcement Learning KR Administrator
  - IT 전문서 집필 및 번역 다수
    - 게임샐러드로 코드 한 줄 없이 게임 만들기 (2013)
    - 유니티 Shader와 Effect 제작 (2014)
    - 2D 게임 프로그래밍 (2014), 러스트 핵심 노트 (2017)
    - 모던 C++ 입문 (2017), C++ 최적화 (2019)

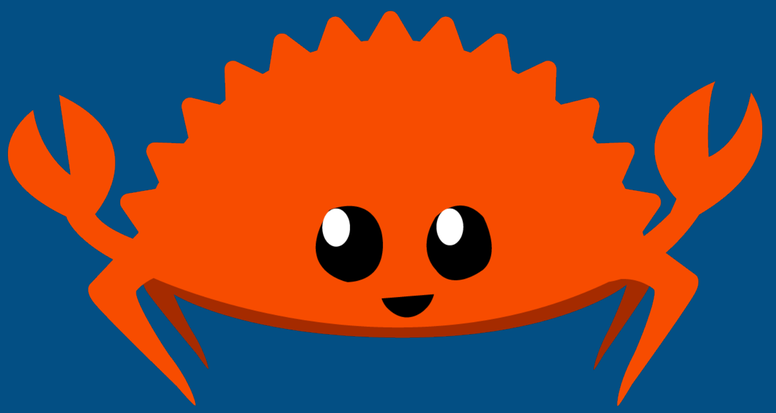


시작하기 전에

- Rust 문법을 간단하게 살펴봅니다. (1시간 내외)
- ECS 프로그래밍을 간단하게 살펴봅니다. (30분 내외)
- 시간이 남으면 간단한 게임을 같이 만들어봅니다. (30분 내외)
- Rust 문법을 자세하게 공부하고 싶다면 Rust 공식 문서를 참고하세요.
- 발표 자료와 예제 코드는 다음 저장소에서 확인 가능합니다.
  https://github.com/utilForever/2022-CAU-Rust-GameProgramming
- ECS를 사용한 코드에 관심이 있다면 다음 프로젝트를 참고하세요.
  - 타워 디펜스 : https://github.com/utilForever/CubbyTower

Rust란?

- https://www.rust-lang.org/
- 모질라 재단에서 2010년 7월 7일 처음 발표
- 현재는 러스트 재단으로 독립해서 개발되고 있다.
- Rust 언어의 특징
  - 안전한 메모리 관리
  - 철저한 예외나 에러 관리
  - 특이한 enum 시스템
  - 트레이트
  - 하이지닉 매크로
  - 비동기 프로그래밍
  - 제네릭



Rust 설치 방법

- Windows
  - 32bit : https://static.rust-lang.org/rustup/dist/i686-pc-windows-msvc/rustup-init.exe
  - 64bit : https://static.rust-lang.org/rustup/dist/x86_64-pc-windows-msvc/rustup-init.exe
- Windows Subsystem for Linux
  - curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
- Linux and MacOS
  - curl https://sh.rustup.rs -sSf | sh -s -- --help

변수

- let 키워드를 사용
- 변수의 자료형을 대부분 유추할 수 있다.
- 변수 숨김(Variable Shadowing)을 지원
- 변수의 이름은 언제나 snake_case 형태로 짓는다.
- Rust에서 변수는 기본적으로 변경 불가(Immutable) 타입이다.
- 변경 가능(Mutable)한 값을 원한다면 mut 키워드로 표시해줘야 한다.

```rust
fn main() {
    let x = 13;
    println!("{}", x);

    let x: f64 = 3.14159;
    println!("{}", x);

    let x;
    x = 0;
    println!("{}", x);
}
```

기본 자료형

- 부울 값 - 참/거짓 값을 나타내는 `bool`
- 부호가 없는 정수형 - 양의 정수를 나타내는 `u8`, `u16`, `u32`, `u64`, `u128`
- 부호가 있는 정수형 - 양/음의 정수를 나타내는 `i8`, `i16`, `i32`, `i64`, `i128`
- 포인터 사이즈 정수 - 메모리에 있는 값들의 인덱스와 크기를 나타내는 `usize`, `isize`
- 부동 소수점 - `f32`, `f64`
- 튜플(tuple) - stack에 있는 값들의 고정된 순서를 전달하기 위한 `(값, 값, ...)`
- 배열(array) - 컴파일 타임에 정해진 길이를 갖는 유사한 원소들의 모음(collection)인 `[값, 값, ...]`
- 슬라이스(slice) - 런타임에 길이가 정해지는 유사한 원소들의 collection
- str(문자열 slice) - 런타임에 길이가 정해지는 텍스트

상수

- 상수는 변수와 달리 반드시 명시적으로 자료형을 지정해야 한다.
- 상수의 이름은 언제나 SCREAMING_SNAKE_CASE 형태로 짓는다.

```rust
const PI: f32 = 3.14159;

fn main() {
    println!(
        PI
    );
}
```

배열

- 고정된 길이로 된 모든 같은 자료형의 자료를 갖는 Collection
- `[T; N]`으로 표현한다.
  - `T`는 원소의 자료형
  - `N`은 컴파일 타임에 주어지는 고정된 길이
- 각각의 원소는 `[x]` 연산자로 가져올 수 있다.

```rust
fn main() {
    let nums: [i32; 3] = [1, 2, 3];
    println!("{:?}", nums);
    println!("{}", nums[1]);
}
```

함수

- 함수는 0개 또는 그 이상의 인자를 가진다.
- 함수의 이름은 언제나 snake_case 형태로 짓는다.
- 함수에서 튜플(Tuple)을 리턴하면 여러개의 값을 리턴할 수 있다.

```rust
fn add(x: i32, y: i32) -> i32 {
    return x + y;
}

fn main() {
    println!("{}", add(42, 13));
}
```

```rust
fn swap(x: i32, y: i32) -> (i32, i32) {
    return (y, x);
}

fn main() {
    let result = swap(123, 321);
    println!("{} {}", result.0, result.1);

    let (a, b) = swap(result.0, result.1);
    println!("{} {}", a, b);
}
```

조건문

- 괄호가 없다.

```rust
fn main() {
    let x = 42;
    if x < 42 {
        println!("Less than 42");
    } else if x == 42 {
        println!("Equal 42");
    } else {
        println!("Greater than 42");
    }
}
```

loop

- 무한 반복문이 필요할 때 사용한다.

```rust
fn main() {
    let mut x = 0;
    loop {
        x += 1;
        if x == 42 {
            break;
        }
    }
    println!("{}", x);
}
```

for

- `..` 연산자는 시작 숫자에서 끝 숫자 전까지의 숫자들을 생성하는 반복자를 만든다.
- `..=` 연산자는 시작 숫자에서 끝 숫자까지의 숫자들을 생성하는 반복자를 만든다.
- `.rev()` 함수를 사용해 값을 감소시키는 반복자로 바꿀 수 있다.
- `.step_by(n)` 함수를 사용해 값을 n만큼 증가시키는 반복자로 바꿀 수 있다.

```rust
fn main() {
    for x in 0..5 {
        println!("{}", x);
    }

    for x in 0..=5 {
        println!("{}", x);
    }
}
```

```rust
fn main() {
    for x in (0..5).rev() {
      println!("{}", x);
    }

    for x in (0..5).step_by(2) {
      println!("{}", x);
    }
}
```

match

- switch를 대체하는 구문
- 모든 케이스를 빠짐 없이 처리해야 한다.

```rust
fn main() {
    let x = 42;

    match x {
        0 => {
            println!("Found 0");
        }
        1 | 2 => {
            println!("Found 1 or 2!");
        }
        3..=9 => {
            println!("Found between 3 and 9!");
        }
        matched_num @ 10..=100 => {
            println!("Found {} between 10 and 100!", matched_num);
        }
        _ => {
            println!("Found something else!");
        }
    }
}
```

구조체

- 필드(Field)들의 Collection
- 메모리 상에 필드들을 어떻게 배치할 지에 대한 컴파일러의 청사진
- 스태틱 메소드(Static Methods)
  - 자료형 그 자체에 속하는 메소드
  - `::` 연산자를 이용해 호출
- 인스턴스 메소드(Instance Methods)
  - 자료형의 인스턴스에 속하는 메소드
  - `.` 연산자를 이용해 호출

```rust
struct SeaCreature {
    animal_type: String,
    name: String,
    arms: i32,
    legs: i32,
    weapon: String,
}
```

```rust
fn main() {
    let s = String::from("Hello world!");
    println!("The length of {} is {}.", s, s.len());
}
```

열거형

- enum 키워드를 통해 몇 가지 태그된 원소의 값을 갖는 새로운 자료형을 생성할 수 있다.
- match와 함께 사용하면 품질 좋은 코드를 만들 수 있다.

```rust
enum Species {
    Crab,
    Octopus,
    Fish,
    Clam,
}

struct SeaCreature {
    species: Species,
    name: String,
}

fn main() {
    let ferris = SeaCreature {
        species: Species::Crab,
        name: String::from("Ferris"),
    };

    match ferris.species {
        Species::Crab => println!("{} is Crab", ferris.name),
        Species::Octopus => println!("{} is Octopus", ferris.name),
        Species::Fish => println!("{} is Fish", ferris.name),
        Species::Clam => println!("{} is Clam", ferris.name),
    }
}
```

Option

- null을 쓰지 않고도 Nullable한 값을 표현할 수 있는 내장된 Generic 열거체

```rust
enum Option<T> {
    None,
    Some(T),
}
```

```rust
struct BagOfHolding<T> {
    item: Option<T>,
}

fn main() {
    let i32_bag = BagOfHolding::<i32> { item: None };
    if i32_bag.item.is_none() {
        println!("Nothing!")
    } else {
        println!("Found Something!")
    }

    let i32_bag = BagOfHolding::<i32> { item: Some(42) };
    if i32_bag.item.is_some() {
        println!("Found Something!")
    } else {
        println!("Nothing!")
    }

    match i32_bag.item {
        Some(v) => println!("Found {}!", v),
        None => println!("Nothing"),
    }
}
```

Result

- 실패할 가능성이 있는 값을 리턴할 수 있도록 해주는 내장된 Generic 열거체

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

```rust
fn do_something_that_might_fail(i: i32) -> Result<f32, String> {
    if i == 42 {
        Ok(13.0)
    } else {
        Err(String::from("Not match!"))
    }
}

fn main() {
    let result = do_something_that_might_fail(12);

    match result {
        Ok(v) => println!("Found {}", v),
        Err(e) => println!("Error: {}", e),
    }
}
```

벡터

- Vec 구조체로 표현하는 가변 크기의 리스트
- vec!매크로를 통해 손쉽게 생성할 수 있다.
- iter() 메소드를 통해 반복자를 생성할 수 있다.

```rust
fn main() {
    let mut float_vec = Vec::new();
    float_vec.push(1.3);
    float_vec.push(2.3);
    float_vec.push(3.4);

    let string_vec = vec![String::from("Hello"), String::from("World")];

    for word in string_vec.iter() {
        println!("{}", word);
    }
}
```

소유권 및 범위 기반 리소스 관리

- 자료형을 인스턴스화해 변수명에 할당(Binding)하면,
  Rust 컴파일러가 전체 생명 주기(Lifetime) 동안 검증할 메모리 리소스를 생성한다.
- 할당된 변수는 리소스의 소유자(Owner)라고 한다.
- Rust는 범위(Scope)가 끝나는 곳에서 리소스를 소멸하고 할당 해제한다.
  이 소멸과 할당 해제를 의미하는 용어로 drop을 사용한다.
  (C++에서는 Resource Acquisition Is Initialization(RAII)라고 부른다).
- 구조체가 Drop될 때 구조체 자신이 제일 먼저 Drop되고, 이후 그 자식들이 각각 Drop된다.

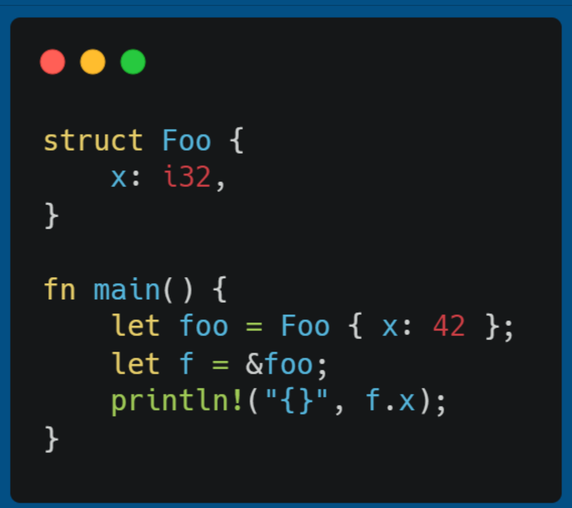
소유권 이전

- 소유자가 함수의 인자로 전달되면, 소유권은 그 함수의 매개 변수로 이동(Move)된다.
- 이동된 이후에는 원래 함수에 있던 변수는 더 이상 사용할 수 없다.

```rust
struct Foo {
    x: i32,
}

fn do_something(f: Foo) {
    println!("{}", f.x);
}

fn main() {
    let foo = Foo { x: 42 };
    do_something(foo);
}
```

참조로 소유권 대여하기

- & 연산자를 통해 참조로 리소스에 대한 접근 권한을 대여할 수 있다.
- &mut 연산자를 통해 리소스에 대해 변경 가능한 접근 권한도 대여할 수 있다.
- 참조도 다른 리소스와 마찬가지로 Drop된다.
- 리소스의 소유자는 변경 가능하게 대여된 상태에서 이동되거나 변경될 수 없다.



```rust
struct Foo {
    x: i32,
}

fn main() {
    let foo = Foo { x: 42 };
    let f = &foo;
    println!("{}", f.x);
}
```

대여한 데이터 전달하기

- Rust의 참조 규칙
  - 단 하나의 변경 가능한 참조 또는 여러개의 변경 불가능한 참조만 허용하며, 둘 다는 안된다.
  - 참조는 그 소유자보다 더 오래 살 수 없다.
- 보통 함수로 참조를 넘겨줄 때에는 문제가 되지 않는다.

```rust
struct Foo {
    x: i32,
}

fn do_something(f: &mut Foo) {
    f.x += 1;
}

fn main() {
    let mut foo = Foo { x: 42 };
    do_something(&mut foo);
    do_something(&mut foo);
}
```

명시적인 생명 주기

- Rust의 컴파일러는 모든 변수의 생명 주기를 이해하며
  참조가 절대로 그 소유자보다 더 오래 존재하지 못하도록 검증을 시도한다.
- 함수에서는 어떤 매개 변수와 리턴 값이 서로 같은 생명 주기를 공유하는지
  식별할 수 있도록 심볼로 표시해 명시적으로 생명 주기를 지정할 수 있다.
- 생명 주기 지정자는 언제나 '로 시작한다.
  (예 : 'a, 'b, 'c)

```
struct Foo {
    x: i32,
}

fn do_something<'a>(foo: &'a Foo) -> &'a i32
{   return &foo.x;
}

fn main() {
    let mut foo = Foo { x: 42 };
    let x = &mut foo.x;
    *x = 13;
    let y = do_something(&foo);
    println!("{}", y);
}
```

여러 개의 생명 주기

- 생명 주기 지정자는 컴파일러가 스스로 함수 매개 변수들의 생명 주기를 판별하지 못하는 경우, 이를 명시적으로 지정할 수 있게 도와준다.

```rust
struct Foo {
    x: i32,
}

fn do_something<'a, 'b>(foo_a: &'a Foo, foo_b: &'b Foo) -> &'b i32
{   println!("{}", foo_a.x);
    println!("{}", foo_b.x);
    return &foo_b.x;
}

fn main() {
    let foo_a = Foo { x: 42 };
    let foo_b = Foo { x: 12 };
    let x = do_something(&foo_a, &foo_b);
    println!("{}", x);
}
```

정적인 생명 주기

- static 변수는 컴파일 타임에 생성되어 프로그램의 시작부터 끝까지 존재하는 메모리 리소스다. 이들은 명시적으로 자료형을 지정해 주어야 한다.
- static 생명 주기는 프로그램이 끝날 때까지 무한정 유지되는 메모리 리소스다. 따라서 'static이라는 특별한 생명 주기 지정자를 갖는다.
- 'static한 리소스는 절대 Drop되지 않는다.
- 만약 static 생명 주기를 갖는 리소스가 참조를 포함하는 경우, 그들도 모두 'static이어야 한다. (그 이하의 것들은 충분히 오래 살아남지 못한다.)

정적인 생명 주기


```rust
static PI: f64 = 3.1415;

fn main() {
    static mut SECRET: &'static str = "swordfish";

    let msg: &'static str = "Hello World!";
    let p: &'static f64 = &PI;
    println!("{} {}", msg, p);

    unsafe {
        SECRET = "abracadabra";
        println!("{}", SECRET);
    }
}
```

문자열

- 문자열의 자료형은 &’static str이다.
  - &은 메모리 내의 장소를 참조하고 있다는 의미 (mut가 없으므로 값의 변경을 허용하지 않음)
  - ’static은 문자열 데이터가 프로그램이 끝날 때까지 유효하다는 의미 (절대로 Drop되지 않음)
  - str은 언제나 유효한 UTF-8 바이트 열을 가리키고 있다는 의미

```rust
fn main() {
    let a: &'static str = "hi 🦀";
    println!("{} {}", a, a.len());
}
```

문자열 슬라이스

- 메모리 상의 바이트 열에 대한 참조이며, 언제나 유효한 UTF-8이어야 한다.
- &str에서 자주 사용하는 메소드는 다음과 같다.
  - len은 문자열의 바이트 길이를 가져온다. (글자수 아님)
  - starts_with와 ends_with는 기본적인 비교에 쓰인다.
  - is_empty는 길이가 0일 경우 true를 리턴한다.
  - find는 주어진 텍스트가 처음 등장하는 위치인 Option<usize> 값을 리턴한다.

```rust
fn main() {
    let a = "hi 🦀";
    println!("{}", a.len());
    let first_word = &a[0..2];
    let second_word = &a[3..7];
    // let half_crab = &a[3..5]; FAILS
    println!("{} {}", first_word, second_word);
}
```

문자

- Rust에서는 UTF-8 바이트 열을 char 타입의 벡터로 돌려주는 기능을 제공한다.
- char 하나는 4바이트다.

```rust
fn main() {
    let chars = "hi 🦀".chars().collect::<Vec<char>>();
    println!("{}", chars.len());
    println!("{}", chars[3] as u32);
}
```

스트링

- UTF-8 바이트 열을 힙 메모리에 소유하는 구조체
- 문자열과는 달리 변경하거나 기타 등등을 할 수 있다. (메모리가 힙에 있기 때문)
- 자주 사용하는 메소드는 다음과 같다.
  - push_str은 스트링의 맨 뒤에 UTF-8 바이트들을 더 붙일 때 사용한다.
  - replace는 UTF-8 바이트 열을 다른 것으로 교체할 때 사용한다.
  - to_lowercase와 to_uppercase는 대소문자를 비교할 때 사용한다.
  - trim은 공백을 제거할 때 사용한다.

```rust
fn main() {
    let mut helloworld = String::from("hello");
    helloworld.push_str(" world");
    helloworld = helloworld + "!";
    println!("{}", helloworld);
}
```

객체 지향 프로그래밍

- 객체 지향 프로그래밍은 다음과 같은 상징적 특징을 갖는 프로그래밍 언어를 뜻한다.
  - 캡슐화(Encapsulation) : 객체라 불리는 단일 타입의 개념적 단위에 데이터와 함수를 연결지음
  - 추상화(Abstraction) : 데이터와 함수를 숨겨 객체의 상세 구현 사항을 알기 어렵게 함
  - 다형성(Polymorphism) : 다른 기능적 관점에서 객체와 상호 작용하는 능력
  - 상속(Inheritance) : 다른 객체로부터 데이터와 동작을 상속받는 능력

Rust는 OOP가 아니다

- Rust에서는 어떠한 방법으로도 데이터와 동작의 상속이 불가능하다.
  - 구조체는 부모 구조체로부터 필드를 상속받을 수 없다.
  - 구조체는 부모 구조체로부터 함수를 상속받을 수 없다.

메소드 캡슐화하기

- Rust는 메소드가 연결된 구조체인 객체라는 개념을 지원한다.
- 모든 메소드의 첫번째 매개변수는 메소드 호출과 연관된 인스턴스에 대한 참조여야 한다.
  - `&self` : 인스턴스에 대한 변경 불가능한 참조
  - `&mut self` : 인스턴스에 대한 변경 가능한 참조
- 메소드는 `impl` 키워드를 쓰는 구현 블록 안에 정의한다.

```rust
struct SeaCreature {
    noise: String,
}

impl SeaCreature {
    fn get_sound(&self) -> &str {
        &self.noise
    }
}

fn main() {
    let creature = SeaCreature {
        noise: String::from("blub"),
    };
    println!("{}", creature.get_sound());
}
```

선택적 노출을 통한 추상화

- Rust는 객체의 내부 동작을 숨길 수 있다.
- 기본적으로, 필드와 메소드들은 그들이 속한 모듈에서만 접근 가능하다.
- pub 키워드는 구조체의 필드와 메소드를 모듈 밖으로 노출시킨다.

```rust
struct SeaCreature {
    pub name: String,
    noise: String,
}

impl SeaCreature {
    pub fn get_sound(&self) -> &str {
        &self.noise
    }
}

fn main() {
    let creature = SeaCreature {
        name: String::from("Ferris"),
        noise: String::from("blub"),
    };
    println!("{}", creature.get_sound());
}
```

다형성과 트레잇

- Rust는 트레잇으로 다형성을 지원한다.
  트레잇은 메소드의 집합을 구조체 데이터 타입에 연결할 수 있게 해준다.
- 먼저 트레잇 안에 메소드 원형을 정의한다. 구조체가 트레잇을 구현할 때,
  실제 데이터 타입이 무엇인지 알지 못하더라도 트레잇 데이터 타입을 통해
  간접적으로 구조체와 상호 작용할 수 있도록 협약을 맺게 된다.
- 구조체의 구현된 트레잇 메소드들은 구현 블록 안에 정의된다.

다형성과 트레잇


```rust
struct SeaCreature {
    pub name: String,
    noise: String,
}

impl SeaCreature {
    pub fn get_sound(&self) -> &str {
        &self.noise
    }
}

trait NoiseMaker {
    fn make_noise(&self);
}

impl NoiseMaker for SeaCreature {
    fn make_noise(&self) {
        println!("{}", &self.get_sound());
    }
}

fn main() {
    let creature = SeaCreature {
        name: String::from("Ferris"),
        noise: String::from("blub"),
    };
    creature.make_noise();
}
```

트레잇에 구현된 메소드

- 트레잇에 메소드를 구현해 넣을 수 있다.
- 함수가 구조체 내부의 필드에 직접 접근할 수는 없지만,
  트레잇 구현체들 사이에서 동작을 공유할 때 유용하게 쓰인다.

```rust
struct SeaCreature {
    pub name: String,
    noise: String,
}

impl SeaCreature {
    pub fn get_sound(&self) -> &str {
        &self.noise
    }
}

trait NoiseMaker {
    fn make_noise(&self);
    fn make_alot_of_noise(&self) {
        self.make_noise();
        self.make_noise();
        self.make_noise();
    }
}
```

```rust
impl NoiseMaker for SeaCreature {
    fn make_noise(&self) {
        println!("{}", &self.get_sound());
    }
}

fn main() {
    let creature = SeaCreature {
        name: String::from("Ferris"),
        noise: String::from("blub"),
    };
    creature.make_alot_of_noise();
}
```

트레잇 상속

- 트레잇은 다른 트레잇의 메소드들을 상속 받을 수 있다.

```rust
struct SeaCreature {
    pub name: String,
    noise: String,
}

impl SeaCreature {
    pub fn get_sound(&self) -> &str {
        &self.noise
    }
}

trait NoiseMaker {
    fn make_noise(&self);
}
```

```rust
trait LoudNoiseMaker: NoiseMaker {
    fn make_alot_of_noise(&self) {
        self.make_noise();
        self.make_noise();
        self.make_noise();
    }
}

impl NoiseMaker for SeaCreature {
    fn make_noise(&self) {
        println!("{}", &self.get_sound());
    }
}

impl LoudNoiseMaker for SeaCreature {}

fn main() {
    let creature = SeaCreature {
        name: String::from("Ferris"),
        noise: String::from("blub"),
    };
    creature.make_alot_of_noise();
}
```

Rust 게임 프로그래밍

- Rust로 게임 프로그래밍하기는 쉽지 않다.
  - 게임에서는 변수를 사용할 때가 많은데, 기본적으로 불변이다. (mut를 자주 사용해야 함)
  - 기본적으로 소유권이 이동하므로 수명과 함께 신경써야 한다.
  - Rust는 OOP를 지원하지 않기 때문에 C++과 같은 방식으로 접근할 수 없다.
- Rust로 쉽게 게임 프로그래밍하는 방법 → 엔진 또는 프레임워크 사용하기

Rust 게임 엔진

- bracket-lib
  - https://github.com/amethyst/bracket-lib
  - 로그라이크 게임을 만드는데 도움을 주는 라이브러리
- Bevy
  - https://github.com/bevyengine/bevy
  - 데이터 기반 게임 엔진, ECS (Entity Component System) 사용
- Fyrox
  - https://github.com/FyroxEngine/Fyrox
  - 2D/3D 게임 엔진, 클래식 OOP + Composition over Inheritance 사용

ECS

- 게임 개발에서 주로 사용하는 소프트웨어 아키텍처 패턴
  “Composition over Inheritance”의 원칙을 따름
- 상태와 행동(기능)을 분리
- 쿼리 기반 (캡슐화 없음, 메시지 전달 없음)

Entity **C**omponent **S**ystem

Entity

+

Component

+

System

Entity

- 컴포넌트들을 담아두는 곳
- 행동(기능)이나 데이터를 갖고 있지 않다.
- 어떤 데이터가 있는지 식별하는 역할을 한다.

Component

- 실제 데이터를 저장하는 곳
- 단순 태그일 수도 있고, 값을 저장하는 컨테이너일 수도 있고,
  무언가를 참조하는 컨테이너일 수도 있다.

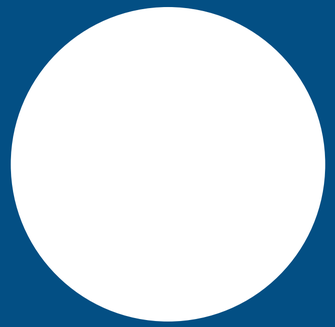
System

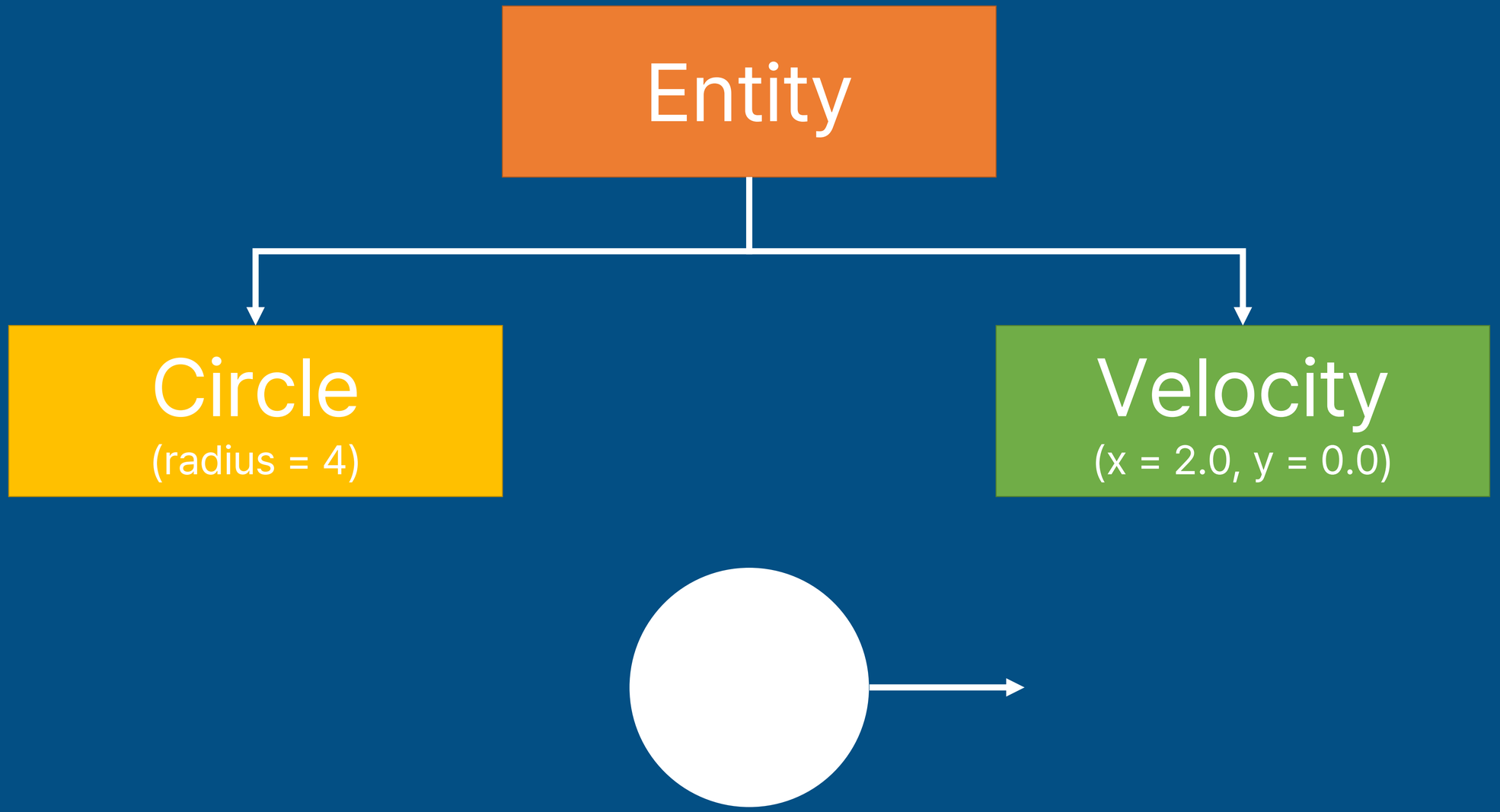
- 컴포넌트의 데이터를 현재 상태에서 다음 상태로 변환하는 논리
- 전체 엔티티 중 특정 컴포넌트를 가리키는 엔티티만 필터링한 다음, 컴포넌트와 관련해 필요한 동작을 수행한다.
  - 데이터를 변경하거나
  - 새로운 컴포넌트를 추가하거나
  - 기존 컨테이너를 삭제하거나

Entity

Entity

Circle
(radius = 4)



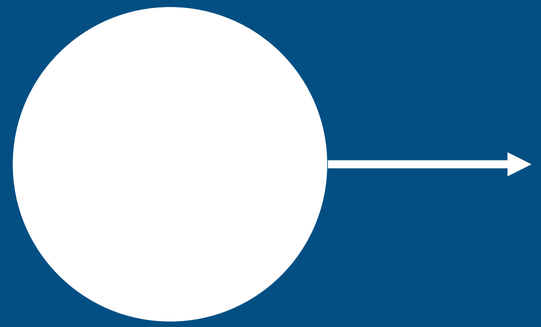

Entity
Circle
(radius = 4)
Velocity
(x = 2.0, y = 0.0)

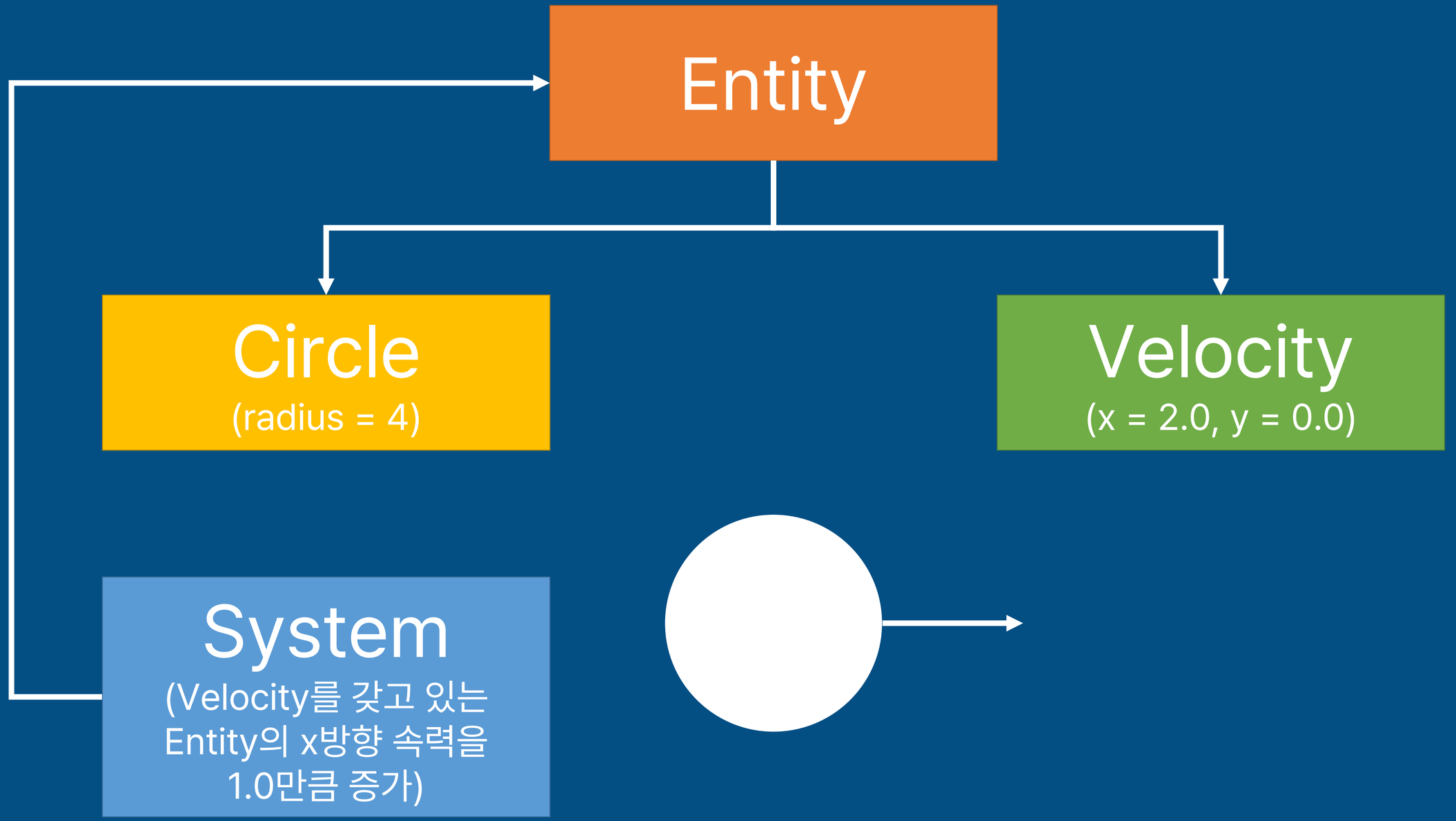
Entity

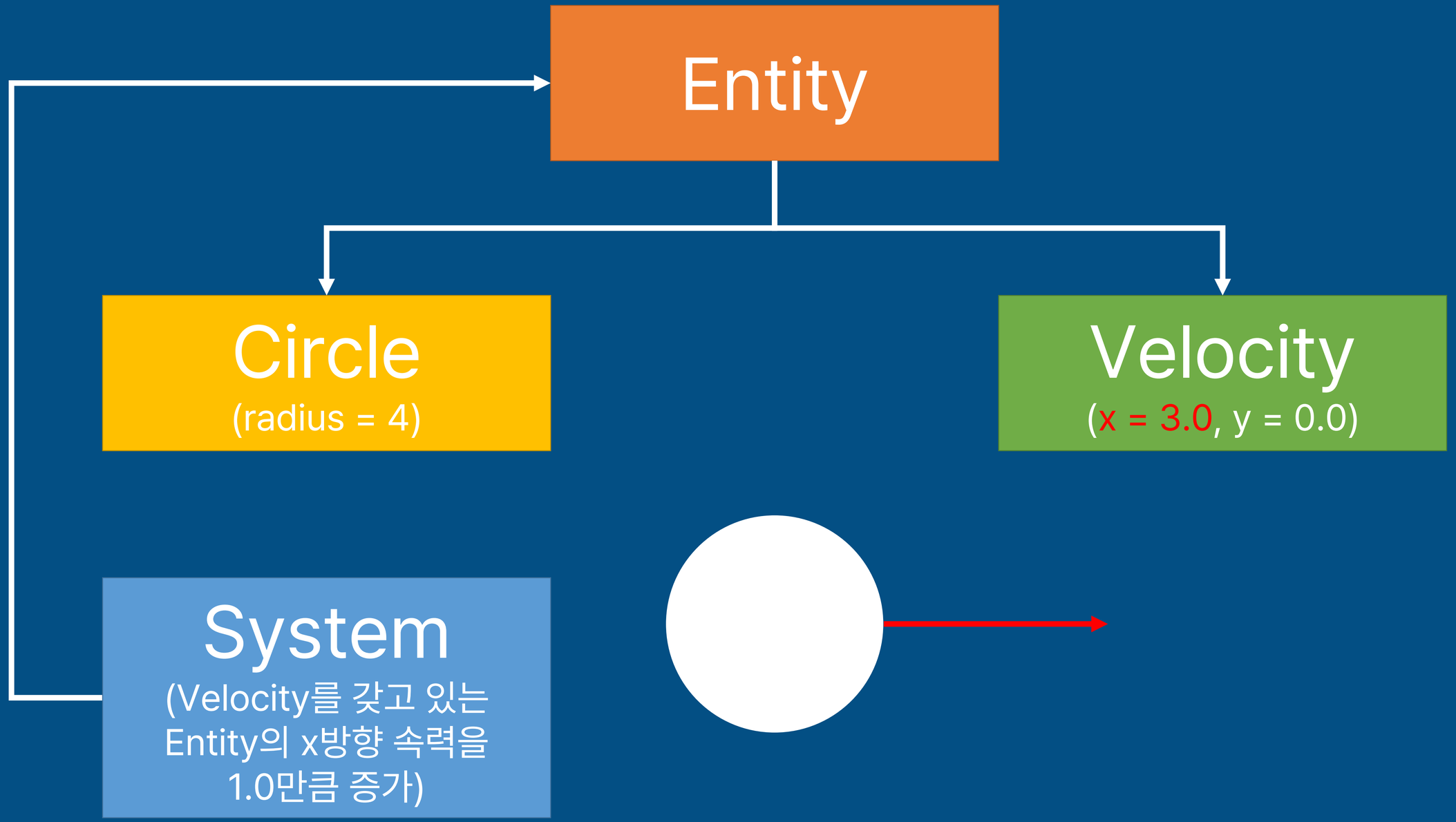
Circle
(radius = 4)

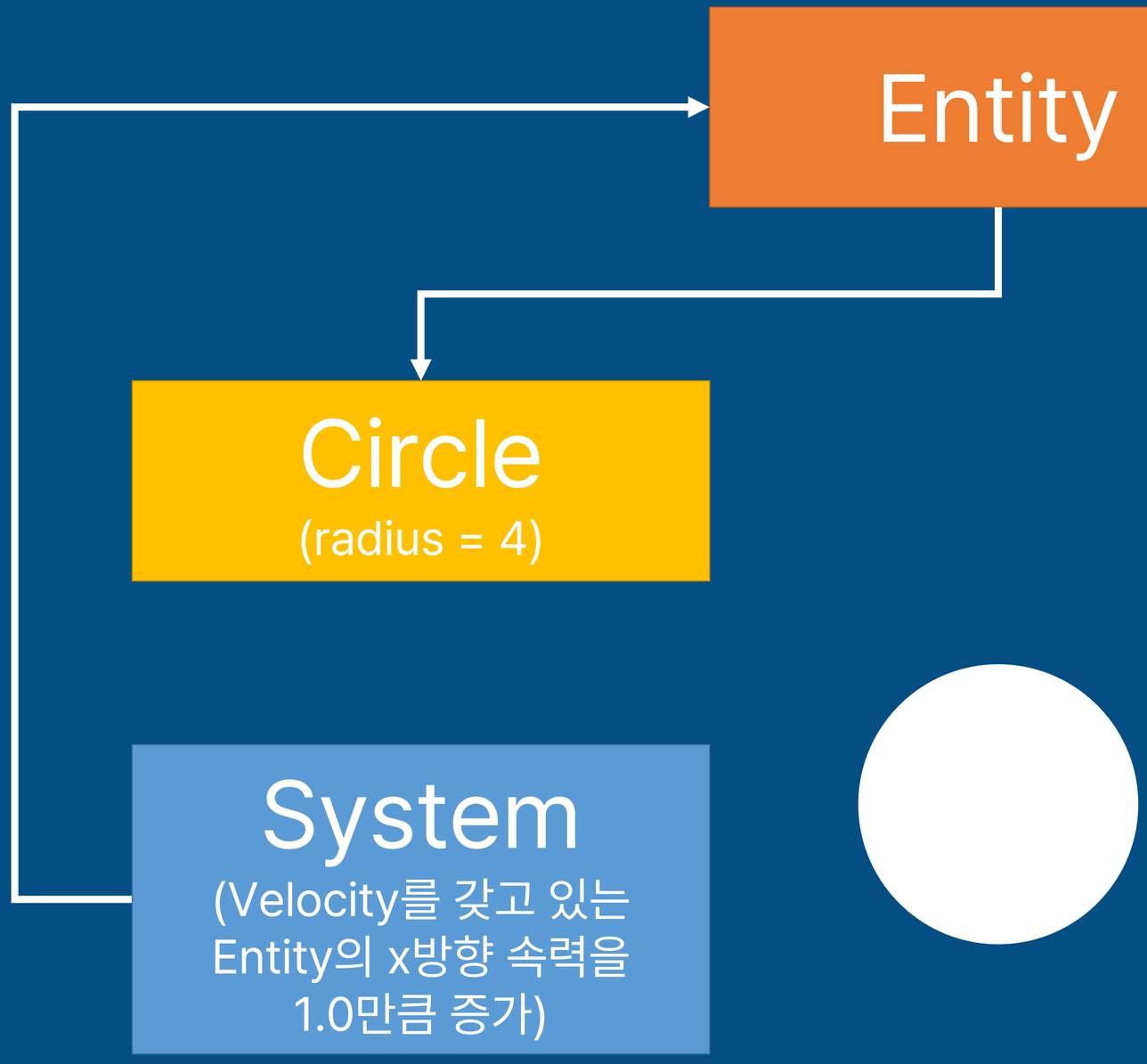
Velocity
(x = 2.0, y = 0.0)

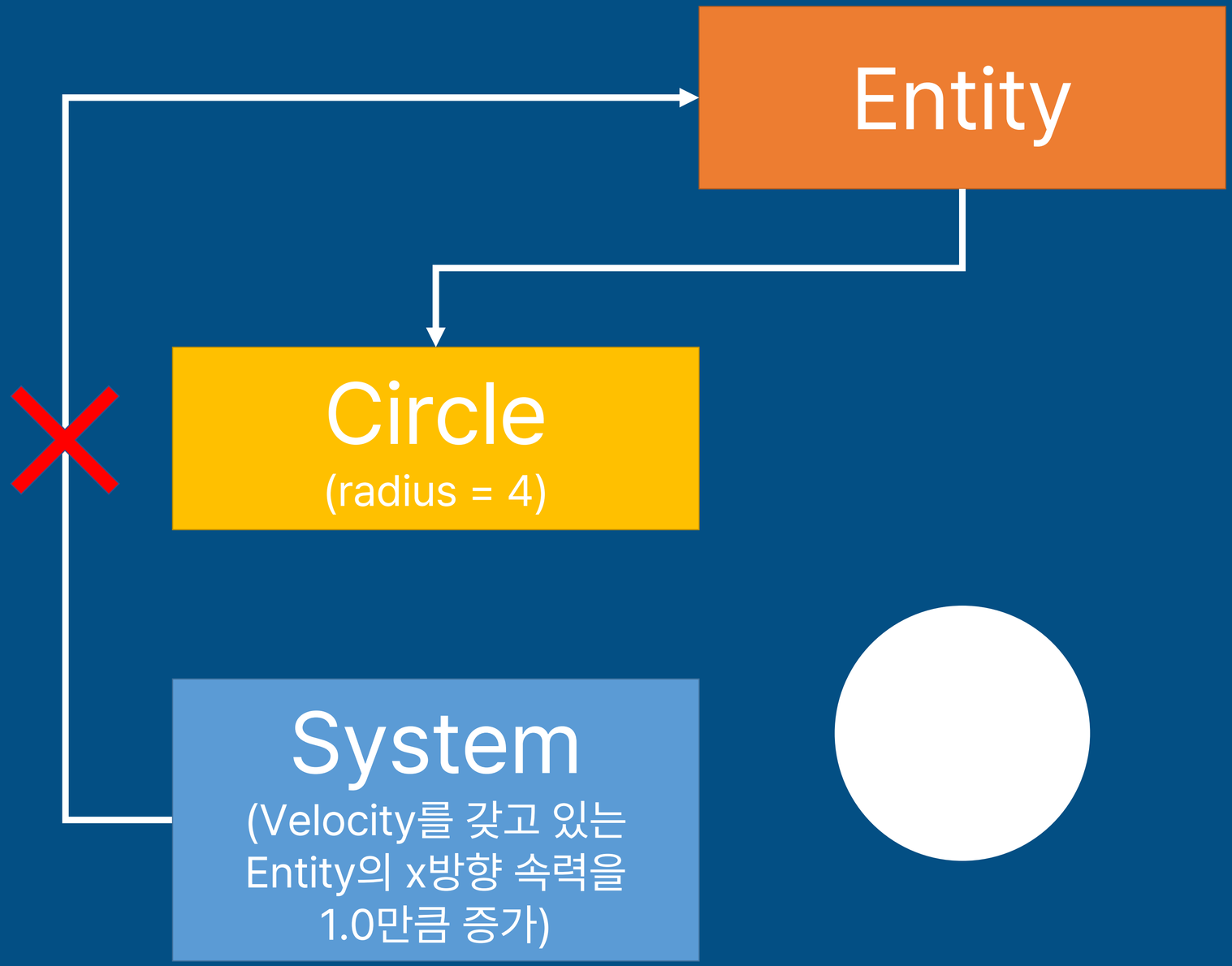
System
(Velocity를 갖고 있는
Entity의 x방향 속력을
1.0만큼 증가)

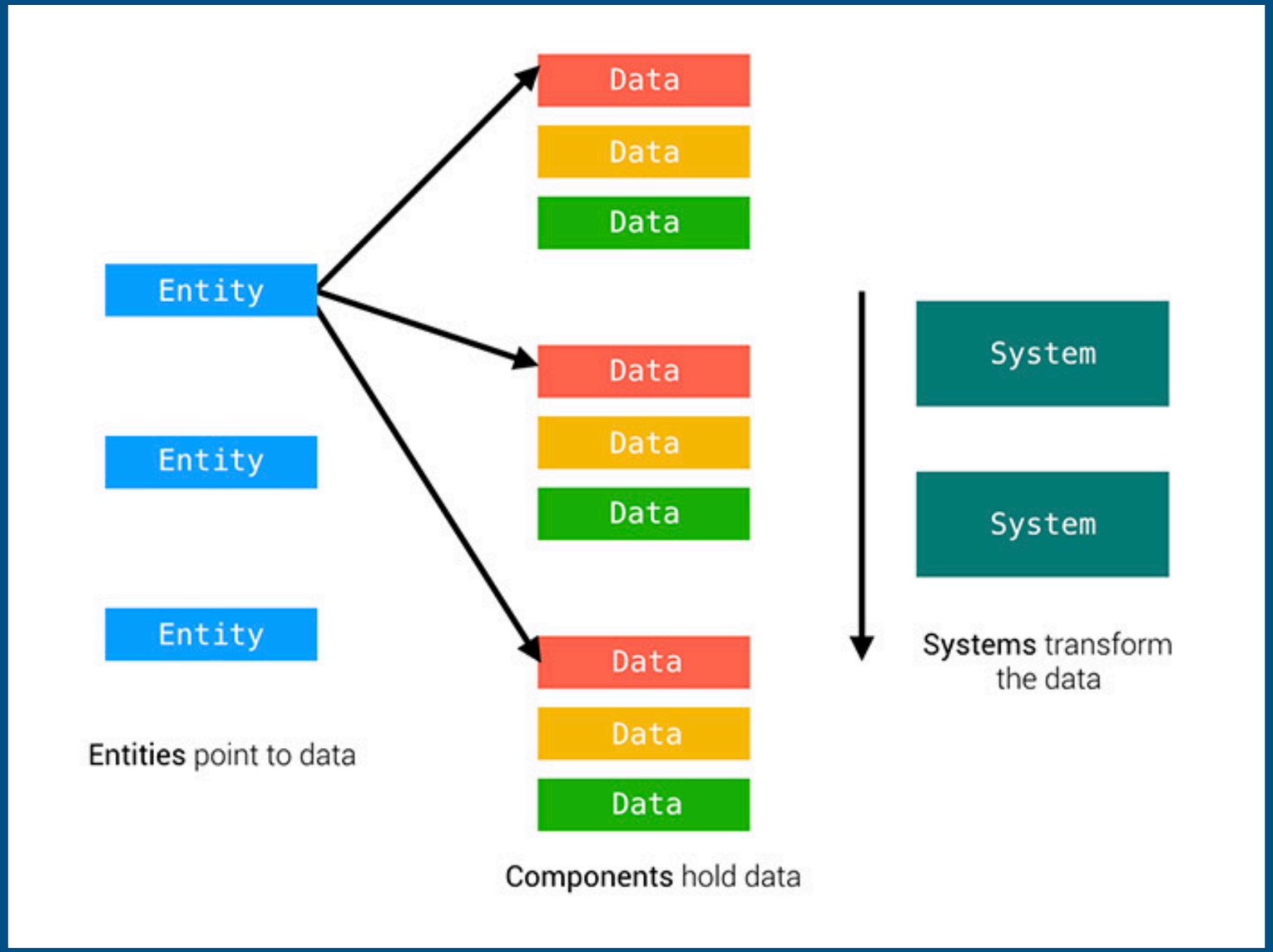



Entity
Circle
(radius = 4)
Velocity
(x = 2.0, y = 0.0)
System
(Velocity를 갖고 있는
Entity의 x방향 속력을
1.0만큼 증가)


Entity
Circle
(radius = 4)
Velocity
(x = 3.0, y = 0.0)
System
(Velocity를 갖고 있는
Entity의 x방향 속력을
1.0만큼 증가)


Entity
Circle
(radius = 4)
System
(Velocity를 갖고 있는
Entity의 x방향 속력을
1.0만큼 증가)


Entity
Circle
(radius = 4)
System
(Velocity를 갖고 있는
Entity의 x방향 속력을
1.0만큼 증가)



https://www.raywenderlich.com/7630142-entity-component-system-for-unity-getting-started

ECS를 쓰는 이유

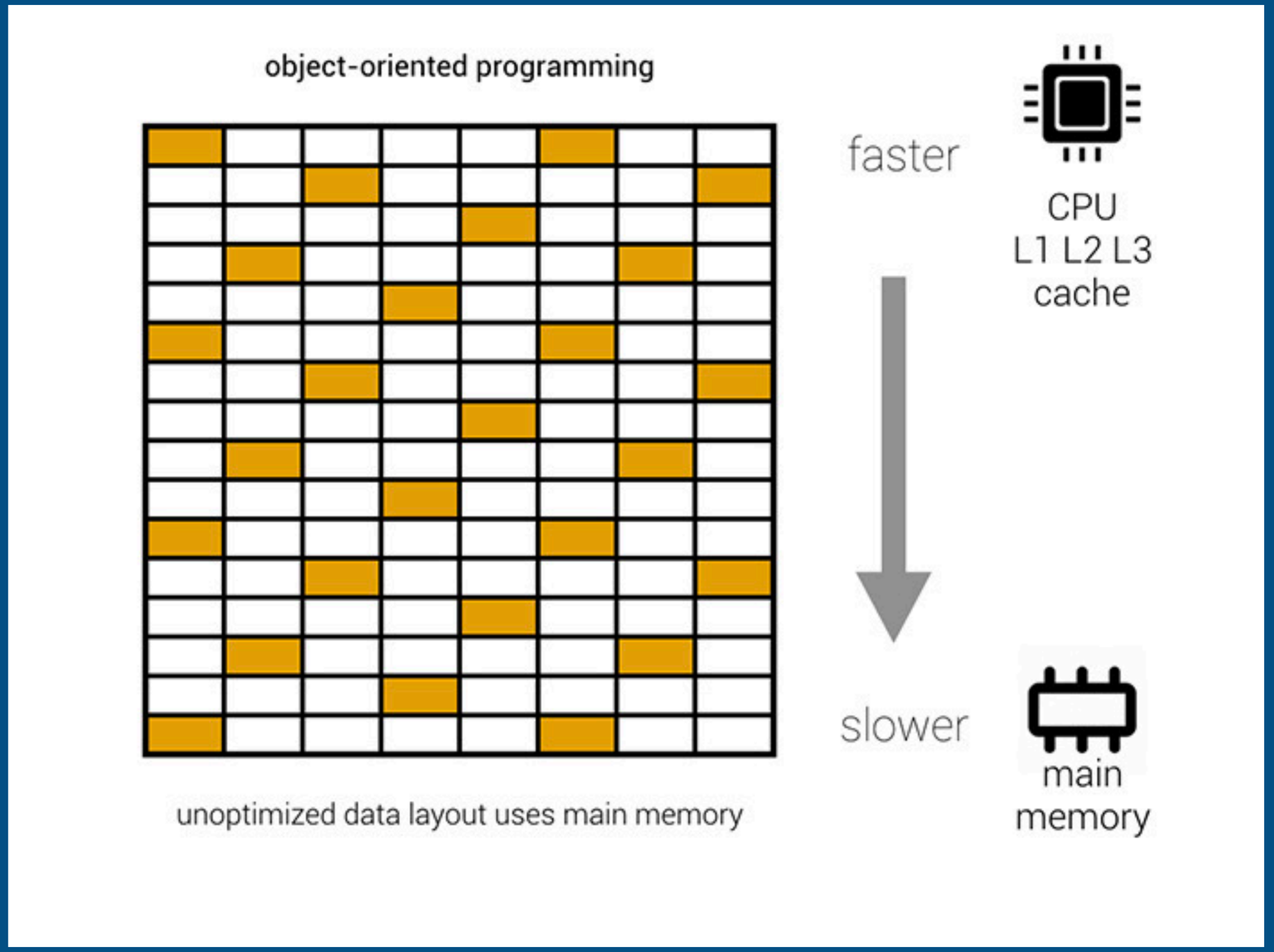
- 객체 지향 프로그래밍에 비해 성능이 뛰어나다.
- 그 이유는 데이터 지향 디자인(Data Oriented Design) 때문이다.
- 지역성(Locality) 개념을 알고 있다면 이해하기 쉽다.

지역성

- 캐시가 효율적으로 동작하려면, 캐시에 저장할 데이터가 지역성을 가져야 한다.
- 지역성이란 데이터 접근이 시간적, 혹은 공간적으로 가깝게 일어나는 것을 의미한다.
- 공간적 지역성
  - 특정 데이터와 가까운 주소가 순서대로 접근되었을 경우를 공간적 지역성이라고 한다.
    CPU 캐시나 디스크 캐시의 경우 한 메모리 주소에 접근할 때 그 주소뿐 아니라
    해당 블록을 전부 캐시에 가져오게 된다.
    이때 메모리 주소를 오름차순이나 내림차순으로 접근한다면, 캐시에 이미 저장된
    같은 블록의 데이터를 접근하게 되므로 캐시의 효율성이 크게 향상된다.

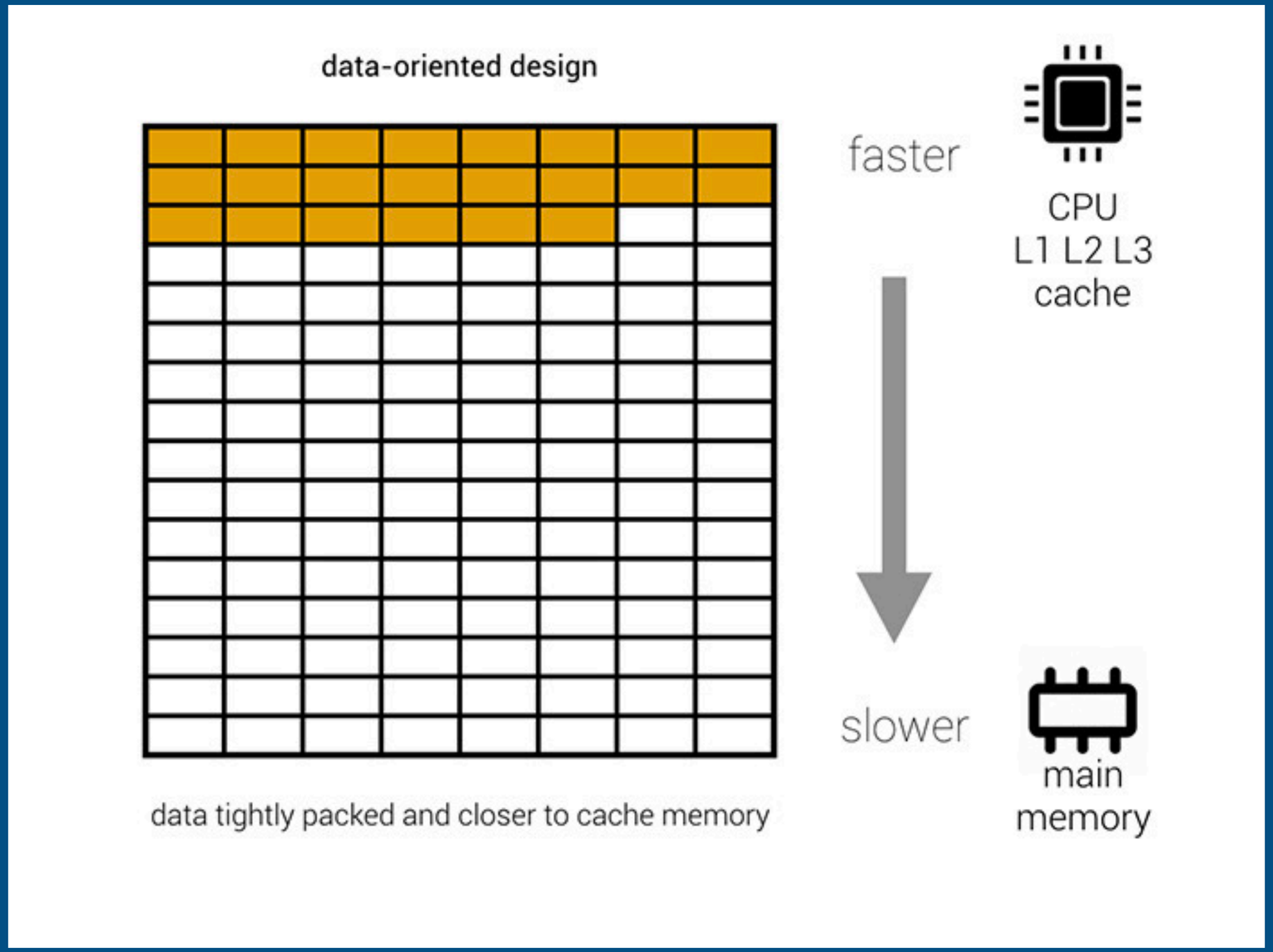
```cpp
class Circle
{
public:
    Circle() = default;
    Circle(std::string name, float radius) : m_name(name), m_radius(radius)
    {
        m_position = std::make_pair(0.0f, 0.0f);
        m_velocity = std::make_pair(0.0f, 0.0f);
    }

    void IncreaseSpeed(float xVel, float yVel)
    {
        m_velocity.first += xVel;
        m_velocity.second += yVel;
    }

private:
    std::string m_name;
    float m_radius;
    std::pair<float, float> m_position;
    std::pair<float, float> m_velocity;
};
```

```cpp
std::vector<Circle> circles;
circles.reserve(1000);

for (int i = 0; i < 1000; ++i)
{
    circles.emplace_back(Circle("Circle", 4.0f))
}

for (auto& circle : circles)
{
    circle.IncreaseSpeed(1.0f, 1.0f);
}
```



https://www.raywenderlich.com/7630142-entity-component-system-for-unity-getting-started

```cpp
std::vector<std::string> names;
std::vector<std::float> radii;
std::vector<std::pair<float, float>> positions;
std::vector<std::pair<float, float>> velocities;

...
velocities.reserve(1000);

for (int i = 0; i < 1000; ++i)
{
    ...
    velocities.emplace_back(std::make_pair(0.0f, 0.0f));
}

for (auto& velocity : velocities)
{
    velocity.first += 1.0f;
    velocity.second += 1.0f;
}
```



https://www.raywenderlich.com/7630142-entity-component-system-for-unity-getting-started

Demo – flappybird-rs

- https://github.com/utilForever/flappybird-rs

정리

- Rust는 메모리 안전성과 성능 및 편의성에 중점을 둔 프로그래밍 언어다.
- Rust 게임 프로그래밍은 쉽지 않지만, 엔진 등을 통해 도움을 받을 수 있다.
- ECS는 데이터 지향 디자인을 통해 성능에 초점을 맞춘 패턴이다.
- Entity는 데이터를 가리키며, Component는 데이터를 저장하며, System은 데이터를 변환한다.
- OOP로 작성된 코드를 ECS로 바꾸려면 생각의 전환이 필요하다.

참고 자료

- https://en.wikipedia.org/wiki/Entity_component_system
- https://www.raywenderlich.com/7630142-entity-component-system-for-unity-getting-started
- https://mrbinggrae.tistory.com/223

감사합니다.

utilForever@gmail.com

https://github.com/utilForever

Facebook, Twitter: @utilForever