



# Object Oriented Programming with Rust

방석현



# Rust 팔아요

## Integrating Rust Into the Android Open Source Project


May 11, 2021






Posted by Ivan Lozano, Android Team








The Android team has been working on introducing the Rust programming language into the Android Open Source Project (AOSP) since 2019 as a memory-safe alternative for platform native code development. As with any large project, introducing a new







# Rust 팔아요

 **torvalds** / **linux** Public


<> **Code**  Pull requests 307  Actions  Projects  Security  Insights








 master ▾  1 branch  806 tags Go to file Code ▾

 **torvalds** Merge tag 'nfs-for-6.6-2' of git://git.linux-nfs.org/projects/anna/... 2cf0f71 2 days ago  1,215,405 commits


 rust	Merge tag 'docs-6.6' of git://git.lwn.net/linux	3 weeks ago
 samples	Merge tag 'vfio-v6.6-rc1' of <a href="https://github.com/awilliam/linux-vfio">https://github.com/awilliam/linux-vfio</a>	3 weeks ago
 scripts	Merge tag 'kbuild-fixes-v6.6' of git://git.kernel.org/pub/scm/linux/k...	3 days ago
 security	selinux: fix handling of empty opts in selinux_fs_context_submount()	last week
 sound	Merge tag 'sound-fix-6.6-rc1' of git://git.kernel.org/pub/scm/linux/k...	2 weeks ago
 tools	Merge tag 'objtool-urgent-2023-09-17' of git://git.kernel.org/pub/sc...	3 days ago

### Languages



-  C 98.2%
-  Assembly 0.8%
-  Shell 0.4%
-  Makefile 0.2%
-  Python 0.2%
-  Perl 0.1%
-  Other 0.1%

# Rust 팔아요



## RustRover Preview

A brand new JetBrains IDE for Rust Developers

Try a new feature-rich Rust IDE with timely support, regular updates,  
and an out-of-the-box experience. Enjoy coding with Rust and focus on what matters.

[Download Preview](#)

# Object Oriented Programming

- 객체 지향적 프로그래밍
- 쉽게 설명하자면...
  - 자료구조(데이터) + 함수(데이터 변형)을 하나의 객체로 생각
  - 굳이? C로 짜도 되는데?

# Object Oriented Programming

- Data Abstraction
- Encapsulation
- Object Composition / Inheritance (has-a / is-a)
- Polymorphism
- 근데 어쨌피 디자인 패턴때문에 써야됩니다 ㅋㅋ;

# C Style Solution

- 오리를 만들어봅시다, 근데 소리를 내는

```
struct Duck {  
    char* quack;  
};  
  
void quack(Duck duck){  
    printf( format: "%s\n", duck.quack);  
}
```

- 이제 이 오리가 어떤 소리를 낼지 예상해보세요
- 이 오리를 여러마리 만들면?

# C Style Solution

```
struct Bird {  
    char* quack;  
};  
  
void quack(Bird bird){  
    printf( format: "%s\n", bird.quack);  
}  
  
int main() {  
    Bird duck = { .quack: "quack" };  
    Bird duck2 = { .quack: "quack2" };  
    Bird parrot = { .quack: "squawk" };  
  
    Bird* birds[] = { [0]: &duck, [1]: &duck2, [2]: &parrot };  
    for(int i = 0; i < 3; i++){  
        quack( bird: *birds[i]);  
    }  
}
```



# C Style Solution

- 과연 내가 만든 새는 오리인가 아닌가 그것이 문제로다
- 애초에 오리 한마리 만들때마다 적는게... 너무 힘들지 않나
- 새가 한마리만 필요할까?
  - 오... 이걸 새 한 마리마다 반복해야 돼?

# Java Style Solution

- 그럼 새를 만들고... 새 > 오리니까 오리를...

```
public abstract class Bird {  
    void Quack(){  
        System.out.println("Quack!");  
    }  
    void fly(){  
        System.out.println("I'm flying!");  
    }  
}
```

```
public class DuckAbstract extends Bird{  
}
```

```
public class ParrotAbstract extends Bird{  
    @Override  
    void Quack(){  
        System.out.println("squawk!");  
    }  
}
```

# Java Style Solution

- 이세계 최강자 새의 등장
- ...애 날아?



# Strategy Pattern

- 행동을 Interface화 하여 (Generic한 Function으로 빼서)
  - 각 행동을 따로 구현
  - Object의 재활용성이 높아짐
  - Single Responsibility Principle

# Strategy Pattern

- 행동을 Interface화 하여 (Generic한 Function으로 빼서)

```
public interface Quack {  
    void quack();  
}
```

```
public class Duck implements Quack{  
    public void quack() {  
        System.out.println("Quack!");  
    }  
}
```

```
public class RandomBird implements Quack{  
    private boolean isParrot;  
  
    public RandomBird(boolean isParrot) { this.isParrot = isParrot; }  
  
    public void quack() {  
        if(isParrot){  
            System.out.println("squawk");  
        } else {  
            System.out.println("quack!");  
        }  
    }  
}
```

# 다시 Java로...

- Class는 답이 없다

I once attended a Java user group meeting where James Gosling (Java's inventor) was the featured speaker. During the memorable Q&A session, someone asked him: "If you could do Java over again, what would you change?" "I'd leave out classes," he replied. After the laughter died down, he explained that the real problem wasn't classes per se, but rather implementation inheritance (the **extends** relationship). Interface inheritance (the **implements** relationship) is preferable. You should avoid implementation inheritance whenever possible.

# Rust의... 어...

- Class = 데이터 + 함수
- 근데 거 데이터랑 함수랑 맨날 잘 들어맞는거 맞긴 하쇼?
- Has – A Relationship은 항상 성립하는가?
- 그럴꺼면...

# Trait

- 데이터 타입간의 관계를 정의
- 정적 / 동적으로 사용 가능
  - 이거 컴파일에서 정의됨 (Rust는 컴파일 언어입니다)
  - 동적으로 사용 → Heap에 Allocate → Size 정보 필요



# Trait

```
trait Quack {  
    3 implementations new *  
    fn quack(&self);  
}
```

```
struct Duck {}  
  
new *  
impl Quack for Duck {  
    new *  
    fn quack(&self) { println!("quack!"); }  
}
```

```
struct RandomBird {  
    is_a_parrot: bool,  
}  
  
new *  
impl Quack for RandomBird {  
    new *  
    fn quack(&self) {  
        if !self.is_a_parrot {  
            println!("quack!");  
        } else {  
            println!("squawk!");  
        }  
    }  
}
```

```
impl Quack for i32 {  
    new *  
    fn quack(&self) {  
        for i:i32 in 0..*self {  
            print!("quack {} ", i);  
        }  
        println!();  
    }  
}
```

```
fn main() {  
    let duck1 = Duck {};  
    let duck2 = RandomBird { is_a_parrot: false };  
    let parrot = RandomBird { is_a_parrot: true };  
  
    let ducks: Vec<&dyn Quack> = vec! [&duck1, &duck2, &parrot, &3];  
  
    for d :&&dyn Quack in &ducks {  
        d.quack();  
    }  
}
```

# Generic Type In Rust

- 완성된 Struct를 사용할 수도 있지만...
  - 보통 인터페이스를 Generic Type으로 선언한다
  - Rust도 마찬가지로 Trait를 Generic Type으로 선언할 수 있다
- 단, 이렇게 선언된 Trait는 Constraint로써 작용한다
  - 이거 아니면 못들어가요~의 선언

# Generic Type In Rust

```
fn please_quack<Q>(q: &Q) where Q: Quack + ?Sized{  
    q.quack();  
}
```

```
fn main() {  
    let duck1 = Duck {};  
    let duck2 = RandomBird { is_a_parrot: false };  
    let parrot = RandomBird { is_a_parrot: true };  
  
    let ducks: Vec<&dyn Quack> = vec! [&duck1, &duck2, &parrot, &3];  
  
    for d: &&dyn Quack in &ducks {  
        d.quack();  
        please_quack( q: *d);  
    }  
}
```