Лекція 7

Traits: Поліморфізм у Rust

Спільна поведінка для різних типів

trait • impl • derive • dyn • bounds

🤖 Приклади: trait Agent, поліморфні команди, абстракції сенсорів

Частина 1: Основи traits та стандартні traits

О. С. Бичков • 2025

План лекції (Частина 1)

1. Що таке поліморфізм?
2. Traits в Rust
3. Оголошення trait
4. Реалізація trait для типу
5. Default методи
6. Стандартні traits (Display, Debug)
7. Clone та Copy
8. PartialEq та Eq
9. PartialOrd та Ord
10. Default trait
11. derive — автоматична реалізація
12. 🤖 Trait для агентів
13. 🤖 Display для дронів
14. 🤖 Порівняння пріоритетів
15. Orphan rule

Частина 2: Generics, trait bounds, trait objects, власні абстракції

Що таке поліморфізм?

Поліморфізм — можливість використовувати єдиний інтерфейс для різних типів

"Один інтерфейс, багато реалізацій"

OOP (Java, C++)

- Наслідування класів
- Інтерфейси
- Віртуальні методи

FP (Haskell)

- Type classes
- Ad-hoc polymorphism

Rust

- Traits
- Схоже на інтерфейси
- + type classes
- Без наслідування!

Rust traits: визначають спільну поведінку БЕЗ наслідування даних

```rust
// Приклад з реального життя
trait Flyable {
    fn take_off(&mut self);
    fn fly_to(&mut self, destination: Position);
    fn land(&mut self);
    fn altitude(&self) -> f64;
}

// Різні типи можуть "літати"
impl Flyable for Drone { /* ... */ }
impl Flyable for Helicopter { /* ... */ }
impl Flyable for Plane { /* ... */ }
```

```rust
// Синтаксис оголошення trait
trait TraitName {
    // Обов'язкові методи (без тіла)
    fn required_method(&self) -> ReturnType;

    // Методи з default реалізацією
    fn optional_method(&self) {
        // Реалізація за замовчуванням
    }

    // Асоційовані константи
    const MAX_VALUE: u32;

    // Асоційовані типи
    type Item;
}

// Приклад
trait Summary {
    fn summarize(&self) -> String;  // Обов'язковий

    fn preview(&self) -> String {   // Default
        format!("Читати більше: {}", self.summarize())
    }
}
```

```
struct Article {
    title: String,
    author: String,
    content: String,
}

// impl TraitName for TypeName
impl Summary for Article {
    fn summarize(&self) -> String {
        format!("{} by {}", self.title, self.author)
    }

    // preview() використовує default реалізацію
}

struct Tweet {
    username: String,
    text: String,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("@{}: {}", self.username, self.text)
    }

    // Можемо перевизначити default
    fn preview(&self) -> String {
        self.text.chars().take(50).collect()
    }
}
```

```rust
let article = Article {
    title: String::from("Rust Traits"),
    author: String::from("Author"),
    content: String::from("Content..."),
};

let tweet = Tweet {
    username: String::from("rustlang"),
    text: String::from("Hello, Rust!"),
};

// Виклик методів trait
println!("{}", article.summarize());  // "Rust Traits by Author"
println!("{}", tweet.summarize());    // "@rustlang: Hello, Rust!"

// Default методи
println!("{}", article.preview());    // "Читати більше: Rust Traits by Author"
println!("{}", tweet.preview());      // "Hello, Rust!" (перевизначено)
```

Trait має бути в scope для виклику його методів (use)!

```
trait Sensor {
    // Обов'язковий — кожен сенсор має свою реалізацію
    fn read_raw(&self) -> f64;

    // Default — типова обробка
    fn read(&self) -> f64 {
        let raw = self.read_raw();
        self.calibrate(raw)
    }

    fn calibrate(&self, value: f64) -> f64 {
        value  // За замовчуванням без калібрування
    }

    fn is_valid(&self, value: f64) -> bool {
        value.is_finite()  // За замовчуванням перевірка на NaN/Inf
    }
}

// GPS сенсор — перевизначає calibrate
impl Sensor for GpsSensor {
    fn read_raw(&self) -> f64 { /* ... */ }

    fn calibrate(&self, value: f64) -> f64 {
        value + self.offset  // Своя калібрація
    }
}
```

Стандартні traits: огляд

Rust має багато вбудованих traits у стандартній бібліотеці:

Форматування:
• Display — для користувача (println!("{}"))
• Debug — для розробника (println!("{:?}"))

Порівняння:
• PartialEq, Eq — рівність (==, !=)
• PartialOrd, Ord — порівняння (<, >, <=, >=)

Клонування:
• Clone — явне копіювання (.clone())
• Copy — неявне копіювання (побітове)

Інші:
• Default — значення за замовчуванням
• Hash — обчислення хешу

```rust
use std::fmt;

struct Point {
    x: f64,
    y: f64,
}

// Display — для println!("{}", value)
impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}

let p = Point { x: 3.0, y: 4.0 };
println!("{}", p);      // (3, 4)
println!("Point: {}", p); // Point: (3, 4)

// format! теж працює
let s = format!("Координати: {}", p);  // "Координати: (3, 4)"

// Display автоматично дає ToString
let string: String = p.to_string();  // "(3, 4)"
```

```rust
use std::fmt;

// Ручна реалізація
impl fmt::Debug for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        f.debug_struct("Point")
            .field("x", &self.x)
            .field("y", &self.y)
            .finish()
    }
}

// АБО автоматично через derive (найчастіше)
#[derive(Debug)]
struct Point {
    x: f64,
    y: f64,
}

let p = Point { x: 3.0, y: 4.0 };
println!("{:?}", p);   // Point { x: 3.0, y: 4.0 }
println!("{:#?}", p);  // Гарне форматування з відступами

// dbg! макрос — виводить значення + повертає його
let x = dbg!(2 + 2);  // [src/main.rs:10] 2 + 2 = 4
```

```rust
// Clone — явне глибоке копіювання
trait Clone {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) { /* default */ }
}

#[derive(Clone)]  // Автоматична реалізація
struct Drone {
    id: u32,
    name: String,  // String реалізує Clone
    position: Position,
}

let drone1 = Drone { /* ... */ };
let drone2 = drone1.clone();  // Повна копія

// drone1 все ще доступний!
println!("{:?}", drone1);
println!("{:?}", drone2);

// Vec::clone() — клонує всі елементи
let v1 = vec![1, 2, 3];
let v2 = v1.clone();  // Новий вектор з копіями
```

```rust
// Copy — marker trait для побітового копіювання
trait Copy: Clone {}  // Вимагає Clone

// Copy типи копіюються автоматично при присвоєнні
let x = 5;       // i32 реалізує Copy
let y = x;       // x копіюється, не move
println!("{}", x);  // x все ще доступний!

// НЕ Copy типи — move
let s1 = String::from("hello");  // String НЕ Copy
let s2 = s1;     // s1 moved!
// println!("{}", s1);  // ❌ Error: s1 moved

// Copy можна derive тільки якщо всі поля Copy
#[derive(Copy, Clone)]  // Clone обов'язковий
struct Point {
    x: f64,  // f64 is Copy
    y: f64,  // f64 is Copy
}

// ❌ Не можна — String не Copy
// #[derive(Copy, Clone)]
// struct Drone { name: String }
```

```rust
// PartialEq — часткова рівність (==, !=)
trait PartialEq {
    fn eq(&self, other: &Self) -> bool;
    fn ne(&self, other: &Self) -> bool { !self.eq(other) }
}

// Eq — повна рівність (marker trait)
// Додає: a == a завжди true (рефлексивність)
trait Eq: PartialEq {}

#[derive(PartialEq, Eq)]  // Порівнює всі поля
struct DroneId {
    group: u32,
    number: u32,
}

let id1 = DroneId { group: 1, number: 5 };
let id2 = DroneId { group: 1, number: 5 };
let id3 = DroneId { group: 2, number: 5 };

id1 == id2  // true
id1 == id3  // false
id1 != id3  // true

// f64 реалізує PartialEq, але НЕ Eq (через NaN)
// NaN != NaN — порушує рефлексивність
```

```rust
use std::cmp::Ordering;

// PartialOrd — часткове впорядкування (<, >, <=, >=)
trait PartialOrd: PartialEq {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering>;
}

// Ord — повне впорядкування
trait Ord: Eq + PartialOrd {
    fn cmp(&self, other: &Self) -> Ordering;  // Less, Equal, Greater
}

#[derive(PartialEq, Eq, PartialOrd, Ord)]
struct Priority(u8);  // 0 = найнижчий

let low = Priority(1);
let high = Priority(10);

low < high   // true
high > low   // true
low.cmp(&high)  // Ordering::Less

// Для сортування
let mut priorities = vec![Priority(5), Priority(1), Priority(10)];
priorities.sort();  // [Priority(1), Priority(5), Priority(10)]
```

Default

```
// Default — створення значення за замовчуванням
trait Default {
    fn default() -> Self;
}

// Стандартні типи
i32::default()      // 0
String::default()   // ""
Vec::<i32>::default()  // []
Option::<i32>::default() // None
bool::default()     // false

// derive — всі поля мають Default
#[derive(Default)]
struct DroneConfig {
    max_speed: f64,      // 0.0
    max_altitude: f64,   // 0.0
    battery_warn: u8,    // 0
}

let config = DroneConfig::default();

// Частковий override
let config = DroneConfig {
    max_speed: 50.0,
    ..Default::default()  // Решта за замовчуванням
};
```

```rust
// derive генерує реалізацію автоматично
#[derive(Debug, Clone, PartialEq, Eq, Hash, Default)]
struct Drone {
    id: u32,
    name: String,
    battery: u8,
}

// Еквівалентно ручній реалізації всіх traits!
// Працює тільки якщо всі поля реалізують відповідні traits

// Типові derive комбінації:

// Для структур даних
#[derive(Debug, Clone, PartialEq)]
struct Data { /* ... */ }

// Для ключів HashMap
#[derive(Debug, Clone, PartialEq, Eq, Hash)]
struct Key { /* ... */ }

// Для сортування
#[derive(Debug, Clone, PartialEq, Eq, PartialOrd, Ord)]
struct Sortable { /* ... */ }

// Для конфігурації
#[derive(Debug, Clone, Default)]
struct Config { /* ... */ }
```

```rust
/// Базовий trait для всіх агентів у системі
trait Agent {
    /// Унікальний ідентифікатор
    fn id(&self) -> u32;

    /// Поточна позиція
    fn position(&self) -> Position;

    /// Цикл сприйняття-рішення-дія
    fn perceive(&mut self, world: &World);
    fn decide(&mut self) -> Action;
    fn act(&mut self, action: Action);

    /// Tick — один крок симуляції (default)
    fn tick(&mut self, world: &World) {
        self.perceive(world);
        let action = self.decide();
        self.act(action);
    }

    /// Чи активний агент
    fn is_active(&self) -> bool {
        true  // За замовчуванням активний
    }
}
```

```rust
struct Drone {
    id: u32,
    position: Position,
    battery: u8,
    state: DroneState,
    current_mission: Option<Mission>,
}

impl Agent for Drone {
    fn id(&self) -> u32 { self.id }
    fn position(&self) -> Position { self.position }

    fn perceive(&mut self, world: &World) {
        // Оновити інформацію про навколишній світ
        self.update_nearby_objects(world);
        self.check_obstacles(world);
    }

    fn decide(&mut self) -> Action {
        match &self.current_mission {
            Some(mission) => self.plan_next_step(mission),
            None => Action::Idle,
        }
    }

    fn act(&mut self, action: Action) {
        self.execute_action(action);
        self.battery -= 1;  // Витрата енергії
    }
```

MAC Display Debugging

```
use std::fmt;

impl fmt::Display for Drone {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "Drone #{} [{}%] at ({:.1}, {:.1})",
            self.id,
            self.battery,
            self.position.x,
            self.position.y
        )
    }
}

impl fmt::Debug for Drone {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        f.debug_struct("Drone")
            .field("id", &self.id)
            .field("position", &self.position)
            .field("battery", &self.battery)
            .field("state", &self.state)
            .field("mission", &self.current_mission)
            .finish()
    }
}

let drone = Drone { /* ... */ };
println!("{}", drone);   // Drone #42 [85%] at (100.0, 50.0)
println!("{:?}", drone); // Drone { id: 42, position: ..., ... }
```

```rust
#[derive(Debug, Clone, PartialEq, Eq)]
struct Mission {
    id: u32,
    priority: u8,      // 1 = низький, 10 = критичний
    deadline: u64,
    mission_type: MissionType,
}

// Сортування: вищий пріоритет першим, при рівному — раніший deadline
impl Ord for Mission {
    fn cmp(&self, other: &Self) -> std::cmp::Ordering {
        // Порівняти пріоритет (зворотній порядок — вищий перший)
        other.priority.cmp(&self.priority)
            // При рівному пріоритеті — за deadline
            .then_with(|| self.deadline.cmp(&other.deadline))
    }
}

impl PartialOrd for Mission {
    fn partial_cmp(&self, other: &Self) -> Option<std::cmp::Ordering> {
        Some(self.cmp(other))
    }
}

// Тепер можна сортувати!
let mut missions = vec![mission1, mission2, mission3];
missions.sort();  // Від найважливішої до найменш важливої
```

```rust
#[derive(Debug, Clone)]
struct AgentConfig {
    max_speed: f64,
    sensor_range: f64,
    communication_range: f64,
    battery_capacity: u32,
    low_battery_threshold: u8,
    behavior_mode: BehaviorMode,
}

impl Default for AgentConfig {
    fn default() -> Self {
        AgentConfig {
            max_speed: 10.0,
            sensor_range: 100.0,
            communication_range: 500.0,
            battery_capacity: 3600,
            low_battery_threshold: 20,
            behavior_mode: BehaviorMode::Normal,
        }
    }
}

// Використання
let default_config = AgentConfig::default();
let custom_config = AgentConfig {
    max_speed: 20.0,
    sensor_range: 200.0,
    ..Default::default()  // Решта за замовчуванням
};
```

Orphan Rule — правило сироти

```
// Ваш trait для чужого типу
impl MyTrait for Vec<i32> {}

// Чужий trait для вашого типу
impl Display for MyStruct {}
```

❌ Заборонено

Причина: уникнення конфліктів коли два crate реалізують те саме

```
// Чужий trait для чужого типу
impl Display for Vec<i32> {}
// ❌ Error!
```

```rust
// Хочемо Display для Vec<Drone>, але не можемо (orphan rule)

// Рішення: обгортка (newtype)
struct DroneList(Vec<Drone>);

impl std::fmt::Display for DroneList {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        writeln!(f, "Drones ({}):", self.0.len())?;
        for drone in &self.0 {
            writeln!(f, "  - {}", drone)?;
        }
        Ok(())
    }
}

// Deref для прозорого доступу до внутрішнього Vec
impl std::ops::Deref for DroneList {
    type Target = Vec<Drone>;
    fn deref(&self) -> &Self::Target { &self.0 }
}

let list = DroneList(vec![drone1, drone2]);
println!("{}", list);     // Використовує наш Display
println!("Count: {}", list.len());  // Працює завдяки Deref
```

Підсумок: Частина 1

Traits — механізм поліморфізму в Rust:
• Визначають спільну поведінку для типів
• Схожі на інтерфейси, але потужніші
• Default методи з реалізацією

Стандартні traits:
• Display / Debug — форматування
• Clone / Copy — копіювання
• PartialEq / Eq — рівність
• PartialOrd / Ord — порівняння
• Default — значення за замовчуванням

derive — автоматична реалізація

🤖 MAC: trait Agent для уніфікації агентів
→ Частина 2: Generics, trait bounds, trait objects

Лекція 7 (продовження)

Traits: Generics та Trait Objects

Статичний та динамічний поліморфізм

bounds • impl Trait • dyn Trait • Box<dyn>

🤖 Приклади: поліморфні агенти, сенсори, команди

Частина 2: Trait bounds та trait objects

О. С. Бичков • 2025

План лекції (Частина 2)

1. Trait як параметр функції
2. Trait bounds
3. where клаузи
4. Множинні bounds
5. impl Trait — повернення
6. Trait objects (dyn Trait)
7. Box<dyn Trait>
8. Статичний vs динамічний dispatch
9. Object safety
10. Supertraits
11. Associated types
12. 🤖 Поліморфні агенти
13. 🤖 Абстракція сенсорів
14. 🤖 Система команд
15. 🤖 Плагіни поведінки
16. Практичні поради

```rust
trait Summary {
    fn summarize(&self) -> String;
}

// Синтаксис 1: impl Trait (найпростіший)
fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}

// Синтаксис 2: Trait bound (більш явний)
fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}

// Обидва варіанти еквівалентні!
// Приймає будь-який тип, що реалізує Summary

let article = Article { /* ... */ };
let tweet = Tweet { /* ... */ };

notify(&article);  // ✓ Article реалізує Summary
notify(&tweet);    // ✓ Tweet реалізує Summary
```

```rust
// Trait bound — обмеження на тип T
fn process<T: Clone + Debug>(item: T) {
    let copy = item.clone();
    println!("{:?}", copy);
}

// Різні bounds для різних параметрів
fn compare<T: PartialOrd, U: Display>(a: T, b: T, label: U) {
    if a < b {
        println!("{}: a < b", label);
    }
}

// Bounds на асоційовані типи
fn sum<I: Iterator<Item = i32>>(iter: I) -> i32 {
    iter.sum()
}

// Bounds в impl блоках
impl<T: Display + Clone> MyStruct<T> {
    fn show(&self) {
        println!("{}", self.value.clone());
    }
}
```

```rust
// Без where — важко читати
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {
    // ...
}

// З where — набагато краще!
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
{
    // ...
}

// where необхідний для складних bounds
fn complex<T, U>(t: T, u: U)
where
    T: Iterator,
    T::Item: Display,  // Bound на асоційований тип
    U: Fn(T::Item) -> bool,  // Bound з closure
{
    for item in t {
        if u(item) {
            println!("{}", item);
        }
    }
}
```

```rust
// + для комбінації traits
fn print_clone<T: Display + Clone>(item: &T) {
    let copy = item.clone();
    println!("{}", copy);
}

// Багато bounds
fn complex<T>(item: T)
where
    T: Display + Debug + Clone + Send + Sync,
{
    // T має реалізувати ВСІ ці traits
}

// Bounds з lifetimes
fn longest<'a, T>(x: &'a T, y: &'a T) -> &'a T
where
    T: PartialOrd,
{
    if x > y { x } else { y }
}

// Практичний приклад
fn serialize_and_log<T>(item: &T)
where
    T: Serialize + Debug,
{
    log::debug!("Serializing: {:?}", item);
    let json = serde_json::to_string(item).unwrap();
}
```

```rust
// impl Trait у позиції повернення
// "Повертаю якийсь тип, що реалізує цей trait"

fn make_iterator() -> impl Iterator<Item = i32> {
    vec![1, 2, 3].into_iter()  // Конкретний тип прихований
}

// Корисно для складних типів
fn filtered_even() -> impl Iterator<Item = i32> {
    (0..100).filter(|x| x % 2 == 0)
    // Без impl Trait: -> std::iter::Filter<std::ops::Range<i32>, fn(&i32) -> bool>
}

// Для closures (вони мають унікальний тип)
fn make_adder(n: i32) -> impl Fn(i32) -> i32 {
    move |x| x + n
}
```

⚠️ impl Trait завжди один конкретний тип — не можна повертати різні типи!

```rust
let add_5 = make_adder(5);
println!("{}", add_5(10));  // 15
```

```rust
trait Animal {
    fn speak(&self) -> String;
}

struct Dog;
struct Cat;

impl Animal for Dog {
    fn speak(&self) -> String { "Woof!".to_string() }
}

impl Animal for Cat {
    fn speak(&self) -> String { "Meow!".to_string() }
}

// dyn Trait — динамічний поліморфізм
// Різні типи в одній колекції!
let animals: Vec<Box<dyn Animal>> = vec![
    Box::new(Dog),
    Box::new(Cat),
    Box::new(Dog),
];

for animal in &animals {
    println!("{}", animal.speak());  // Woof!, Meow!, Woof!
}

// dyn завжди через посилання або Box (розмір невідомий)
```

```rust
// Чому Box<dyn Trait>?
// dyn Trait — "unsized" тип, розмір невідомий на компіляції

// ❌ Не можна
// let animal: dyn Animal;  // Error: size not known

// ✓ Через посилання
let animal: &dyn Animal = &Dog;

// ✓ Через Box (heap allocation)
let animal: Box<dyn Animal> = Box::new(Dog);

// Функція приймає trait object
fn make_sound(animal: &dyn Animal) {
    println!("{}", animal.speak());
}

make_sound(&Dog);  // Woof!
make_sound(&Cat);  // Meow!

// Функція повертає trait object
fn random_animal() -> Box<dyn Animal> {
    if rand::random() {
        Box::new(Dog)
    } else {
        Box::new(Cat)
    }
}
```

Статичний vs Динамічний dispatch

Статичний (generics)

```
fn process<T: Trait>(x: T)

✓ Monomorphization — окрема
   копія для кожного типу
✓ Інлайн можливий
✓ Нульовий runtime overhead
✗ Більший бінарний розмір
✗ Тип відомий на компіляції
```

Динамічний (dyn)

```
fn process(x: &dyn Trait)

✓ Один екземпляр функції
✓ Менший бінарний розмір
✓ Тип визначається в runtime
✗ Vtable lookup кожен виклик
✗ Немає інлайну
✗ Невеликий overhead
```

Правило: generics за замовчуванням, dyn коли потрібна гетерогенна колекція

Object Safety — обмеження dyn

```
trait Draw {
    fn draw(&self);
}
// Можна: &dyn Draw
```

❌ НЕ object-safe

Clone не object-safe через Self у поверненні — компілятор повідомить!

```
trait Clone {
    fn clone(&self) -> Self;
}
```

```rust
// Supertrait — trait, що вимагає іншого trait
trait Animal {
    fn name(&self) -> &str;
}

// Pet вимагає Animal — "Pet: Animal"
trait Pet: Animal {
    fn owner(&self) -> &str;
}

// Для реалізації Pet потрібно реалізувати і Animal
struct Dog {
    name: String,
    owner: String,
}

impl Animal for Dog {
    fn name(&self) -> &str { &self.name }
}

impl Pet for Dog {
    fn owner(&self) -> &str { &self.owner }
}

// Можна викликати методи обох traits
let dog = Dog { name: "Rex".into(), owner: "Alice".into() };
println!("{} belongs to {}", dog.name(), dog.owner());
```

```rust
// Associated type — тип, визначений у trait
trait Iterator {
    type Item;  // Асоційований тип

    fn next(&mut self) -> Option<Self::Item>;
}

// Реалізація визначає конкретний тип
struct Counter {
    count: u32,
}

impl Iterator for Counter {
    type Item = u32;  // Item = u32 для Counter

    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;
        if self.count < 6 {
            Some(self.count)
        } else {
            None
        }
    }
}

// Чому не generics? trait Iterator<T> { fn next(&mut self) -> Option<T> }
// Бо тоді можна impl Iterator<u32> for X та impl Iterator<String> for X
// Associated type — один Item на тип
```

```rust
trait Agent {
    fn id(&self) -> u32;
    fn tick(&mut self, world: &World);
    fn position(&self) -> Position;
}

struct Drone { /* ... */ }
struct GroundRobot { /* ... */ }
struct Satellite { /* ... */ }

impl Agent for Drone { /* ... */ }
impl Agent for GroundRobot { /* ... */ }
impl Agent for Satellite { /* ... */ }

// Гетерогенна колекція агентів
struct Swarm {
    agents: Vec<Box<dyn Agent>>,
}

impl Swarm {
    fn tick_all(&mut self, world: &World) {
        for agent in &mut self.agents {
            agent.tick(world);  // Динамічний dispatch
        }
    }

    fn add_agent(&mut self, agent: Box<dyn Agent>) {
        self.agents.push(agent);
    }
}
```

```rust
trait Sensor {
    type Reading;  // Асоційований тип

    fn read(&self) -> Result<Self::Reading, SensorError>;
    fn calibrate(&mut self);
    fn is_operational(&self) -> bool;
}

struct GpsSensor;
struct RadarSensor;
struct CameraSensor;

impl Sensor for GpsSensor {
    type Reading = GpsCoordinates;
    fn read(&self) -> Result<GpsCoordinates, SensorError> { /* ... */ }
    fn calibrate(&mut self) { /* ... */ }
    fn is_operational(&self) -> bool { true }
}

impl Sensor for RadarSensor {
    type Reading = RadarScan;
    fn read(&self) -> Result<RadarScan, SensorError> { /* ... */ }
    fn calibrate(&mut self) { /* ... */ }
    fn is_operational(&self) -> bool { true }
}

// Функція працює з будь-яким сенсором
fn read_sensor<S: Sensor>(sensor: &S) -> Result<S::Reading, SensorError> {
    sensor.read()
}
```

```
trait Command {
    fn execute(&self, drone: &mut Drone) -> Result<(), CommandError>;
    fn can_execute(&self, drone: &Drone) -> bool;
    fn priority(&self) -> u8 { 5 }  // Default priority
}

struct TakeOffCommand;
struct FlyToCommand { destination: Position }
struct LandCommand;
struct ScanAreaCommand { area: Area }

impl Command for TakeOffCommand {
    fn execute(&self, drone: &mut Drone) -> Result<(), CommandError> {
        drone.set_state(DroneState::Flying);
        Ok(())
    }
    fn can_execute(&self, drone: &Drone) -> bool {
        drone.state == DroneState::Landed && drone.battery > 20
    }
}

// Черга команд — гетерогенна
struct CommandQueue {
    commands: Vec<Box<dyn Command>>,
}
```

```
impl CommandQueue {
    fn new() -> Self {
        CommandQueue { commands: Vec::new() }
    }

    fn enqueue(&mut self, cmd: Box<dyn Command>) {
        self.commands.push(cmd);
        // Сортуємо за пріоритетом (вищий — перший)
        self.commands.sort_by(|a, b| b.priority().cmp(&a.priority()));
    }

    fn execute_next(&mut self, drone: &mut Drone) -> Option<Result<(), CommandError>> {
        // Знайти першу команду, що може виконатись
        let idx = self.commands.iter()
            .position(|cmd| cmd.can_execute(drone))?;

        let cmd = self.commands.remove(idx);
        Some(cmd.execute(drone))
    }

    fn execute_all(&mut self, drone: &mut Drone) {
        while let Some(result) = self.execute_next(drone) {
            if let Err(e) = result {
                log::error!("Command failed: {:?}", e);
            }
        }
    }
}
```

```
trait Behavior {
    fn name(&self) -> &str;
    fn evaluate(&self, agent: &Drone, world: &World) -> Option<Action>;
    fn priority(&self) -> u8;
}

struct AvoidObstacles;
struct ReturnToBase;
struct FollowPath;
struct AttackTarget;

impl Behavior for AvoidObstacles {
    fn name(&self) -> &str { "AvoidObstacles" }
    fn priority(&self) -> u8 { 10 }  // Найвищий пріоритет

    fn evaluate(&self, agent: &Drone, world: &World) -> Option<Action> {
        if let Some(obstacle) = world.nearest_obstacle(agent.position()) {
            if obstacle.distance < 10.0 {
                return Some(Action::Evade(obstacle.direction.opposite()));
            }
        }
        None  // Не активується
    }
}

// Subsumption architecture — поведінки за пріоритетом
struct BehaviorManager {
    behaviors: Vec<Box<dyn Behavior>>,  // Відсортовані за priority
}
```

```
impl BehaviorManager {
    fn new() -> Self {
        BehaviorManager { behaviors: Vec::new() }
    }

    fn add_behavior(&mut self, behavior: Box<dyn Behavior>) {
        self.behaviors.push(behavior);
        self.behaviors.sort_by(|a, b| b.priority().cmp(&a.priority()));
    }

    /// Subsumption: повертає дію від найпріоритетнішої активної поведінки
    fn select_action(&self, agent: &Drone, world: &World) -> Action {
        for behavior in &self.behaviors {
            if let Some(action) = behavior.evaluate(agent, world) {
                log::debug!("Behavior '{}' activated", behavior.name());
                return action;
            }
        }
        Action::Idle  // Жодна поведінка не активувалась
    }
}

// Використання
let mut manager = BehaviorManager::new();
manager.add_behavior(Box::new(AvoidObstacles));
manager.add_behavior(Box::new(ReturnToBase));
manager.add_behavior(Box::new(FollowPath));

let action = manager.select_action(&drone, &world);
```

```rust
trait CommunicationProtocol {
    fn send(&self, target: AgentId, message: &Message) -> Result<(), CommError>;
    fn receive(&self) -> Option<(AgentId, Message)>;
    fn broadcast(&self, message: &Message) -> Result<(), CommError>;
    fn range(&self) -> f64;
}

struct RadioProtocol { frequency: f64, power: f64 }
struct LaserProtocol { bandwidth: u32 }
struct AcousticProtocol { /* for underwater */ }

impl CommunicationProtocol for RadioProtocol {
    fn send(&self, target: AgentId, msg: &Message) -> Result<(), CommError> { /* ... */ }
    fn receive(&self) -> Option<(AgentId, Message)> { /* ... */ }
    fn broadcast(&self, msg: &Message) -> Result<(), CommError> { /* ... */ }
    fn range(&self) -> f64 { self.power * 100.0 }
}

// Дрон може використовувати будь-який протокол
struct Drone {
    comm: Box<dyn CommunicationProtocol>,
    // ...
}
```

Практичні поради

✓ Використовуйте generics за замовчуванням (zero-cost)
✓ dyn Trait — для гетерогенних колекцій
✓ derive де можливо
✓ Маленькі, фокусовані traits
✓ Default реалізації для зручності

❌ Не створюйте "God traits" з десятками методів
❌ Не використовуйте dyn без потреби
❌ Не забувайте про object safety

🤖 Для MAC:
• trait Agent — базова абстракція
• trait Sensor — уніфікація сенсорів
• trait Command — поліморфні команди
• trait Behavior — плагіни поведінки

Коли що використовувати?

Ситуація	Підхід	Приклад
Один тип	Конкретний тип	fn process(d: Drone)
Один trait, відомий тип	impl Trait	fn f(x: impl Display)
Гнучкість + performance	Generics	fn f<T: Trait>(x: T)
Гетерогенна колекція	dyn Trait	Vec<Box<dyn Agent>>
Повернення різних типів	Box<dyn Trait>	-> Box<dyn Animal>
Приховати тип	impl Trait return	-> impl Iterator

Generics = compile-time, dyn = runtime

Основні стандартні traits

Trait	Методи	derive
Display	fmt(&self, f)	❌
Debug	fmt(&self, f)	✓
Clone	clone(&self) -> Self	✓
Copy	(marker)	✓
PartialEq	eq(&self, other) -> bool	✓
Eq	(marker, requires PartialEq)	✓
PartialOrd	partial_cmp() -> Option<Ordering>	✓
Ord	cmp() -> Ordering	✓
Default	default() -> Self	✓
Hash	hash(&self, state)	✓

Підсумок лекції

Traits — поліморфізм у Rust:
• Спільна поведінка без наслідування
• derive для автоматичної реалізації

Trait bounds:
• <T: Trait> — обмеження на generics
• where клаузи для читабельності
• impl Trait для спрощення

Trait objects (dyn):
• Динамічний поліморфізм
• Гетерогенні колекції
• Object safety обмеження

🤖 MAC: Agent, Sensor, Command, Behavior traits
→ Наступна лекція: Generics — параметричний поліморфізм

Завдання для самостійної роботи

1. Базове: Створіть trait Shape з методами area() та perimeter().
   Реалізуйте для Circle, Rectangle, Triangle.

2. Display/Debug: Реалізуйте Display та Debug для структури
   DroneStatus { id, position, battery, state }.

3. Поліморфізм: Створіть систему сенсорів:
   - trait Sensor з read(), calibrate()
   - Різні типи сенсорів (GPS, Radar, Camera)
   - Колекція Box<dyn Sensor>

4. MAC: Реалізуйте trait Agent та BehaviorManager:
   - Мінімум 3 різні поведінки
   - Subsumption architecture
   - Симуляція з кількома агентами

Дякую за увагу!

Traits • Generics • dyn Trait • Bounds

Питання?