# 3-1: Encapsulation and Generics (Theory)

Artem Pavlov, TII, Abu Dhabi, 16.04.2025

# Method syntax

- It’s common to have functions associated with a certain type, e.g. rectangle_area
- To improve ergonomics of using such function they can be tied to the type by becoming a method
- Methods can take self, &self or &mut self and called using “dot” syntax, i.e. rectangle.area()

# Method syntax: example

```rust
#[derive(Debug)]
pub struct Rectangle {
    pub width: u32,
    pub height: u32,
}

impl Rectangle {
    pub fn new(width: u32, height: u32) -> Self {
        Self { width, height }
    }

    pub fn get_shape_name() -> &'static str {
        "rectangle"
    }

    pub fn area(&self) -> u32 {
        self.width * self.height
    }

    pub fn make_bigger(&mut self, times: u32) {
        self.width *= times;
        self.height *= times;
    }

    pub fn consume(self) {
        println!("consumed: {self:?}");
    }
}
```

```rust
let rec_name = Rectangle::get_shape_name();
let mut rec = Rectangle::new(42, 13);

let area1 = rec.area();
println!("{rec_name} area: {area1}");

rec.make_bigger(3);
let area2 = rec.area();
println!("{rec_name} area: {area2}");

rec.consume();
```

# Associated constants

```rust
impl Rectangle {
    pub const DIMENSIONS: u32 = 2;
    pub const MIN: Self = Self {
        width: 0,
        height: 0,
    };
    pub const MAX: Self = Self {
        width: u32::MAX,
        height: u32::MAX,
    };
}
```

```rust
println!(
    "Rectangle has {} dimensions",
    Rectangle::DIMENSIONS,
);
println!(
    "Rectangle range: min {:?}, max {:?}",
    Rectangle::MIN,
    Rectangle::MAX,
);
```

# Setters and getters

```rust
#[derive(Debug)]
pub struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    pub fn get_width(&self) -> u32 {
        self.width
    }

    pub fn set_width(&mut self, width: u32) {
        self.width = width
    }
}
```

```rust
let mut rec = Rectangle::new(42, 13);
let width1 = rec.get_width();
println!("rectangle width: {width1}");

rec.set_width(50);
let width2 = rec.get_width();
println!("rectangle width: {width2}");
```

# impl blocks

- You can have multiple impl blocks, spread across different modules
- You can use Self in impl blocks instead of typing full type name
- Self can be used in arguments, return types, method bodies, and associated constants

# Generics

- Different types often exhibit similar APIs
- For example, area method can be implemented for Rectangle, Circle, Triangle, etc.
- To reduce code duplication we often want to write code “generic” over different types which have same APIs

# Area trait

```rust
pub trait Area {
    fn area(&self) -> u32;
}

#[derive(Debug)]
pub struct Rectangle {
    width: u32,
    height: u32,
}

impl Area for Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    pub fn new(width: u32, height: u32) -> Self {
        Self { width, height }
    }
}
```

```rust
use shapes::{Area, Rectangle};

let rec = Rectangle::new(42, 13);
let area = rec.area();
println!("rectangle area: {area}");
```

# Orphan rules

- You can implement trait for a type only if either the type or the trait is defined in the current crate
- Inherent methods can be implemented only for types defined in the current crate
- The rules are necessary to prevent potential conflicts between crates
- https://doc.rust-lang.org/reference/items/implementations.html

# Generic functions

```rust
use shapes::Area;

fn compute_total_area<T: Area>(shapes: &[T]) -> u32 {
    let mut total_area = 0;
    for shape in shapes {
        total_area += shape.area();
    }
    total_area
}
```

# Const generics

```rust
pub fn sum_array<const N: usize>(val: [u32; N]) -> u32 {
    val.iter().sum()
}

pub fn duplicate_val<const N: usize>(val: u32) -> [u32; N] {
    [val; N]
}
```

# Const generics limitations

```rust
enum Algorithms {
    Alg1,
    Alg2,
}

pub fn use_alg<const ALG: Algorithms>(data: &mut [u8]) {
    // ..
}

pub fn hex_encode<const N: usize>(val: [u8; N]) -> [u8; 2 * N] {
    // ..
}
```

# where clauses

```rust
use std::fmt::{Debug, Display};

pub fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {
    // ..
}

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

# impl Trait syntax

```rust
// `impl Area` is a "universal" type, i.e. "caller chooses"
pub fn compute_total_area(shapes: &[impl Area]) -> u32 {
    let mut total_area = 0;
    for shape in shapes {
        total_area += shape.area();
    }
    total_area
}

// `impl Area` is an "existential" type, i.e. "callee chooses"
pub fn get_an_area_shape() -> impl Area {
    Rectangle::new(42, 10)
}
```

impl Trait can be used to return only one type,
i.e. you can't return Rectangle or Circle from the same function

# Generic types

```rust
#[derive(Debug)]
pub struct ShapePair<T: Area> {
    first: T,
    second: T,
}

impl<T: Area> ShapePair<T> {
    fn get_first(&self) -> &T {
        &self.first
    }

    fn get_second(&self) -> &T {
        &self.second
    }
}
```

```rust
impl<T: Area + Debug> ShapePair<T> {
    pub fn print(&self) {
        println!(
            "first: {:?} second: {:?}",
            self.first,
            self.second,
        );
    }
}

impl ShapePair<Rectangle> {
    fn rect_overlap(&self) -> bool {
        self.first.overlaps(&self.second)
    }
}

impl<T: Area> Area for ShapePair<T> {
    fn area(&self) -> u32 {
        self.first.area() + self.second.area()
    }
}
```

# Default implementations

```rust
pub trait Area {
    fn area(&self) -> u32;

    fn is_area_bigger_than<S: Area>(&self, shape: S) -> bool {
        self.area() > shape.area()
    }
}

pub struct Point;

impl Area for Point {
    fn area(&self) -> u32 {
        0
    }

    fn is_area_bigger_than<S: Area>(&self, _: S) -> bool {
        false
    }
}
```

# Associated types

```rust
use num_traits::Num;

pub trait Area {
    type Area: Num;

    fn area(&self) -> Self::Area;
}
```

```rust
#[derive(Debug)]
pub struct Rectangle<T: Num + Clone> {
    width: T,
    height: T,
}

impl<T: Num + Clone> Rectangle<T> {
    pub fn new(width: T, height: T) -> Self {
        Self { width, height }
    }
}

impl<T: Num + Clone> Area for Rectangle<T> {
    type Area = T;
    fn area(&self) -> T {
        self.width.clone() * self.height.clone()
    }
}
```

# Associated types 2

```
pub struct Rectangle32 {
    width: u32,
    height: u32,
}

impl Area for Rectangle32 {
    type Area = u64;
    fn area(&self) -> u64 {
        (self.width as u64) * (self.height as u64)
    }
}
```

# Associated types 3

```rust
pub fn compute_total_area32(shapes: &[impl Area<Area = u32>]) -> u32 {
    let mut total_area = 0;
    for shape in shapes {
        total_area += shape.area();
    }
    total_area
}

pub fn compute_total_area<T: Area>(shapes: &[T]) -> T::Area {
    use num_traits::Zero;
    let mut total_area = T::Area::zero();
    for shape in shapes {
        total_area = total_area + shape.area();
    }
    total_area
}
```

# Generic traits

```rust
// Defined in `std`
pub trait From<T>: Sized {
    fn from(value: T) -> Self;
}

pub struct MyUint(pub u64);

impl From<u32> for MyUint {
    fn from(value: u32) -> Self {
        Self(value as u64)
    }
}

impl From<u64> for MyUint {
    fn from(value: u64) -> Self {
        Self(value)
    }
}
```

# Associated constants

```rust
pub trait Area {
    type Area: Num;
    const NAME: &'static str;

    fn area(&self) -> Self::Area;
}
```

```rust
impl Area for Rectangle32 {
    type Area = u64;
    const NAME: &'static str = "32 -bit rectangle";
    fn area(&self) -> u64 {
        (self.width as u64) * (self.height as u64)
    }
}
```

# Const generics restrictions

```
pub trait CryptoHash {
    const OUTPUT_SIZE: usize;

    fn hash_data(data: &[u8]) -> [u8; Self::OUTPUT_SIZE];
}
```

```
generic parameters may not be used in const operations
type parameters may not be used in const expressions
add `#![feature(generic_const_exprs)]` to allow generic const
expressions rustc(Click for full compiler diagnostic)
View Problem (Alt+F8)    No quick fixes available
```

# Supertraits

- Supertraits are traits that are required to be implemented for a type to implement a specific trait.
- The trait with a supertrait is called a subtrait of its supertrait.
- Subtrait has access to the associated items of its supertraits (e.g. when used in generic bounds).

# Supertraits

```rust
pub trait Area {
    fn area(&self) -> f32;
}

pub trait Perimiter {
    fn perimeter(&self) -> f32;
}

pub trait Shape: Area + Perimiter {
    fn bounding_box(&self) -> ([f32; 2], [f32; 2]);

    fn area_to_perimeter(&self) -> f32 {
        self.area() / self.perimeter()
    }
}

pub fn print_shape_properties(s: &impl Shape) {
    let area = s.area();
    let perimeter = s.perimeter();
    println!("area: {area} perimeter: {perimeter}");
}
```

# Object safety

- All supertraits must also be object safe.
- Sized must not be a supertrait. In other words, it must not require Self: Sized.
- It must not have any associated constants.
- It must not have any associated types with generics.
- All associated functions must either be dispatchable from a trait object or be explicitly non-dispatchable.

# Dynamic dispatch

```rust
pub fn use_dyn_area(s: &dyn Area) {
    println!("area: {}", s.area());
}

pub fn get_dyn_area(triangle: bool) -> Box<dyn Area> {
    if triangle {
        Box::new(Rectangle::new(42, 13))
    } else {
        Box::new(Circle::new(42))
    }
}

fn main() {
    let rec = Rectangle::new(42, 13);
    use_dyn_area(&rec);

    let dyn_rec = &rec as &dyn Area;
    use_dyn_area(dyn_rec);

    let boxed_rec = Box::new(rec);
    let boxed_dyn_rec = boxed_rec as Box<dyn Area>;
    use_dyn_area(&*boxed_dyn_rec);
}
```

Questions?