CoE 164

Computing Platforms

03b: Generics and Traits

POLYMORPHISM

Polymorphism is one of the four "pillars" of object oriented programming - defined as *something that occurs in several forms*.

Rust enables programmers to write polymorphic data types and functions.

EXAMPLE: STRING REPRESENTATION

Any data type in Rust can have its own string representation. Types that have a representation can be converted by using the `to_string()` method.

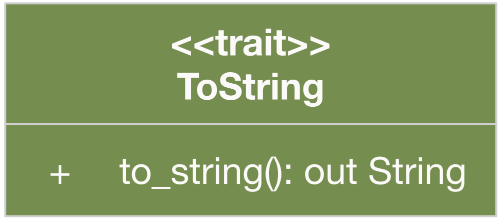
```rust
struct SampleStruct;

let ss = SampleStruct;

println!("Number 3: {}", 3.to_string());
println!("bool: {}", true.to_string());
println!("struct: {}", ss.to_string());
```

Example

INTERFACES

An **interface** is a collection of properties and function *signatures* that a class must implement. It can be implemented even by totally unrelated classes!

Rust **traits** are a similar implementation of interfaces.



TRAITS

Traits are declared using the `trait` keyword. Method signatures are placed inside the block with the first argument (`&self`) always pointing to the instance of the `struct` or `enum` that implements it.

```
pub trait ToString {
    fn to_string(&self) -> String;
}
```

Example

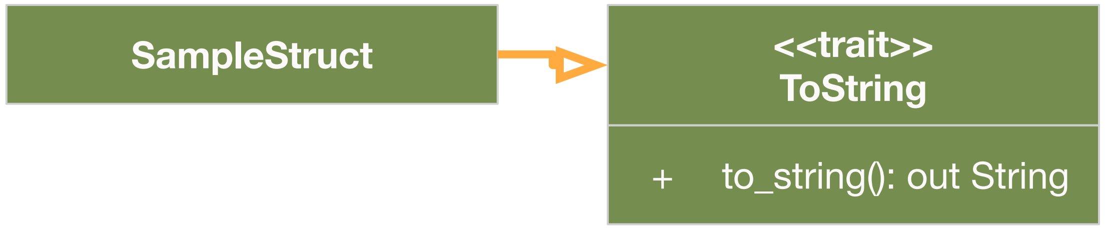
TRAITS: IMPLEMENTERS

Data types can implement traits by using the `for` keyword in an `impl` block for the trait. These data types should have an implementation for *every* method in the trait block.

Example

```
struct SampleStruct;

impl ToString for SampleStruct {
    fn to_string(&self) -> String {
        return "Sample class!";
    }
}

let ss = SampleStruct;
println!("struct: {}",
ss.to_string());
```

TRAITS: INHERITANCE TREE



TRAITS: DEFAULT IMPLEMENTATIONS

It is also possible to insert a default implementation of a trait method. In this case, data types that will have the trait are *not required* to implement the method, but they can override it by writing a new implementation.

```rust
pub trait ToString {
    fn to_string(&self) -> String {
        "str()".to_string()
    }
}
```

Example

TRAITS: FUNCTIONS

Traits can be treated as data types - that is, functions and methods can return and accept data types of a certain trait.

```
fn ps_display(obj: &impl ToString) -> String {
    format!("PS> {}", obj.to_string())
}

println!("{}", ps_display(ss));
```

Example

TRAITS: MULTIPLE TRAITS

Functions and methods can accept data types that have multiple traits by listing using "+". Arguments should have *all* of the traits specified.

Note that Rust only supports returning of data types that have *all* of the traits.

```
use core::fmt::{Debug, Display};

fn disp_both(obj: &(impl Debug + Display)) {
    println!("Dsp> {}", obj);
    println!("Dbg> {:?}", obj);
}
```

Example

TRAITS: DERIVE

Some traits that have their own default implementation that may be exposed from a library. However, they are not automatically added to any struct or enum type by default.

```
struct SampleStruct;

// Compile error!
let ss = SampleStruct;
println!("{:?}", ss);
```

Example

TRAITS: DERIVE

The `derive` attribute can be added before a struct or enum definition to give it the default implementation of a trait.

```
#[derive(Debug)]
struct SampleStruct;

// NO compile error
let ss = SampleStruct;
println!("{:?}", ss); // "SampleStruct"
```

Example

GENERIC DATA TYPES

Generic data types enable code that is duplicated among different data types be reused and consolidated into a single reference.

Rust usually indicates `T` or any single letter as some placeholder for some (random) data type.

GENERICS: FUNCTIONS

Functions can have generic types in both of their parameters and return statements. A list of generic types should be listed first between angled brackets after the function name.

```
fn largest <T>(list: &[T]) -> &T {
    // code here
}
```

Example

GENERICS: STRUCTS

Generic types should be enumerated after the struct name. Note that multiple types can be included, and any name can be substituted for the placeholders.

```
struct UvWrapper <U, V> {
    u: U,
    v: V,
}
```

Example

GENERICS: ENUMS

Generic types should be enumerated after the enum name. The types can also appear in enum entries that hold data.

```
enum NullErrOk <E, S> {
    Null,
    Err(E),
    Ok(S),
}
```

Example

GENERICS: METHODS

Generic types should be enumerated after the `impl` keyword. The name of the struct or enum to which the method is delegated to should be declared with its respective generic types list. The type names can be different from the ones used in the struct or enum declaration.

```rust
impl <E, S> NullErrOk <E, S> {
    fn unwrap_err(&self) -> &E {
        if let NullErrOk::Err(x) = self {
            x
        }
        else {
            panic!("Not an error!");
        }
    }
}
```

Example

GENERICS: METHODS

It is possible to write methods for a specific set of data types by writing the types inside the places where there would have been a generic type. In this case, the method is used only for that specific data types.

```
impl UvWrapper <f32, f32> {
    fn sum(&self) -> f32 {
        self.u + self.v
    }
}
```

Example

GENERICS: METHODS

Generics from struct or enum declarations can be mixed up with generics from the methods themselves.

```
impl <U, V> UvWrapper <U, V> {
    fn dot <W, X> (&self, other: &UvWrapper <W, X>) -> f32 {
        self.u * other.u + self.v * other.v
    }
}
```

Example

GENERICS: TRAIT BOUNDS

The "expanded" way of writing function signatures that accept or return data types of a certain trait is through *generic trait bounds*. The generic types act as placeholders for the data types that should have a certain trait.

```
fn ps_display(obj: &impl ToString) -> String {
    format!("PS> {}", obj.to_string())
}

fn ps_display_v2 <T: ToString>(obj: &T) -> String {
    format!("PS> {}", obj.to_string())
}
```

Example

GENERICS: TRAIT BOUNDS

The trait bound syntax is useful when multiple parameters with the same trait is needed or when the function signature gets longer.

```rust
fn concat <T: Debug>(a: &T, b: &T) -> String {
    // Concatenate debug messages here
}
```

Example

GENERICS: MULTIPLE TRAIT BOUNDS

Multiple traits can be matched using the trait bound syntax using the + operator.

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

fn disp_both <D: Debug + Display>(obj: &D) {
    println!("Dsp> {}", obj);
    println!("Dbg> {:?}", obj);
}
```

Example

GENERICS: "WHERE" TRAIT BOUNDS

Alternatively, the trait bounds can be written in a `where` clause in case it becomes too long.

```
fn disp_both <D, V>(obj: &D, obj2: &V) -> bool
where
    D: Debug + Display,
    V: Debug + Display + Clone,
{
    // Do something
}
```

Example

GENERICS: METHODS

It is possible to write methods for a specific set of traits by writing the traits inside the places where there would have been a generic type using the trait bound syntax. *Blanket implementations* can also be written where a trait has a method applicable on any data type that has another trait.

```rust
impl <T: PartialOrd> UVWrapper <T, T> {
    // some implementation here
}

impl <T: Display> ToString for T {
    // some implementation here
}
```

Example

RESOURCES

- The Rust Book

CoE 164

Computing Platforms

03b: Generics and Traits