Generic Types

Generics (aka, Parameterized Types) Examples

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &Self
{
    &self.x
}

impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

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

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

Monomorphization

- **Monomorphization** is a compile-time process where polymorphic functions are transformed to many monomorphic functions for each unique instantiation.

```rust
fn id<T>(x: T) -> T {
    x
}


fn main() {
    let int: i32 = id(10);
    let string = id("some text");

    println!("{int}, {string}");
}
```

```rust
fn id_i32(x: i32) -> i32 {
    x
}
```

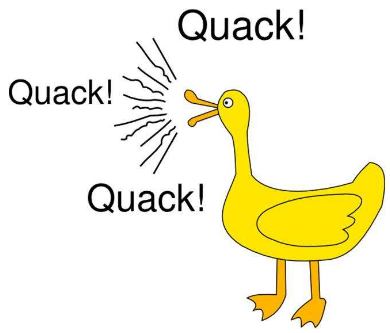
```rust
fn id_str(x: &str) -> &str {
    x
}
```

```rust
fn main() {
    let int = id_i32(10);
    let string = id_str("some text");
    println!("{int}, {string}");
}
```


Traits

What is a Trait?

- Interface?
- Collection of methods?
- Shared Behavior between types?
- Type Class?
  - A family of types
  - eg., Haskell and Scala



The Expression Problem

"The desire to extend modules without modifying source code while retaining type safety"

	OO languages	FP languages
Add New Types	O	X
Add New Operations	X	O

Philip Wadler, 12 November 1998

Answers to Expression Problem

- Haskell -> Typeclass
- Scala -> Type Class Pattern
- Rust -> ???

Haskell

```
main = do
    print(area (Circle 3.0))
    print(perimeter (Rectangle 5.0 7.0))

data Shape = Circle Float
           | Rectangle Float Float


area :: Shape -> Float
area (Circle r) = pi * r ^ 2
area (Rectangle w h) = w * h


perimeter :: Shape -> Float
perimeter (Circle r) = 2 * pi * r
perimeter (Rectangle w h) = 2 * (w + h)
```

Easy to add new `perimeter` function.

What if to add a new shape like Square?

Haskell

```haskell
main = do
    print(area (Circle 3.0))
    print(perimeter (Rectangle 5.0 7.0))

data Shape = Circle Float
           | Rectangle Float Float
           | Square Float
```

```haskell
area :: Shape -> Float
area (Circle r) = pi * r ^ 2
area (Rectangle w h) = w * h
area (Square w) = w * w
```

```haskell
perimeter :: Shape -> Float
perimeter (Circle r) = 2 * pi * r
perimeter (Rectangle w h) = 2 * (w + h)
perimeter (Square w) = 4 * w
```

Typeclass in Haskell

```haskell
data Circle = Circle Float
data Rectangle = Rectangle Float Float

-- typeclasses
class Area a where
    area:: a -> Float

class Perimeter a where
    perimeter:: a -> Float

-- instances for Area
instance Area Circle where
    area (Circle r) = pi * r ^ 2

instance Area Rectangle where
    area (Rectangle w h) = w * h
```

```haskell
-- instances for Perimeter
instance Perimeter Circle where
    perimeter (Circle r) = 2 * pi * r

instance Perimeter Rectangle where
    perimeter (Rectangle w h) = 2 * (w + h)

circle = Circle 5.0
rectangle = Rectangle 3.0 5.0

main = do
    print(area circle)
    print(perimeter circle)
    print(area rectangle)
    print(perimeter rectangle)
```

Typeclass in Haskell

```haskell
data Circle = Circle Float
data Rectangle = Rectangle Float Float

-- typeclasses
class Area a where
    area:: a -> Float

class Perimeter a where
    perimeter:: a -> Float

-- instances for Area
instance Area Circle where
    area (Circle r) = pi * r ^ 2

instance Area Rectangle where
    area (Rectangle w h) = w * h
```

```haskell
-- instances for Perimeter
instance Perimeter Circle where
    perimeter (Circle r) = 2 * pi * r

instance Perimeter Rectangle where
    perimeter (Rectangle w h) = 2 * (w + h)

circle = Circle 5.0
rectangle = Rectangle 3.0 5.0

main = do
    print(area circle)
    print(perimeter circle)
    print(area rectangle)
    print(perimeter rectangle)
```


Can we do it in Rust?

- Is it feasible to add **toJson** or **toXML** methods to any type?

```rust
#[derive(Copy, Clone)]
struct Address { street: String, city: String }

struct Person { name: String, address: Address }


let address = Address {
    street: "123 Main St".to_string(),
    city: "Anytown".to_string(),
};

let john = Person {
    name: "John".to_string(),
    address: address.clone()
};

println!("{}", address.to_json()); // What if Address doesn't have to_json()?

println!("{}", john.to_json());    // What if Person doesn't have to_json()?
```


Define a Trait

```rust
trait ToJson {
    // abstract method
    fn to_json(&self) -> String;

    // default method
    fn indentation(level: usize) -> (String, String) {
        (" ".repeat(level), " ".repeat(level * 2))
    }
}
```

Traits are implemented from outside the type itself (attached to it)

Self is an implicit type parameter that refers to "the type that is implementing this interface".

```
impl ToJson for Address {
    fn to_json(&self) -> String {
        let (outdent, indent) = Self::indentation(1);
        format!("{{{}\"street\": {},{}\"city\": {}{}}}",
            outdent, self.street, indent, self.city, outdent
        )
    }
}

// Now, it is possible to call to_json()
println!("{}", address.to_json());
// { "street": 123 Main St,  "city": Anytown }
```


Typeclasses

(= decoupled *Ad hoc* Polymorphism)

- The **type class (pattern)** is a very powerful technique that **allows to add new behavior to existing types**, *without using inheritance* and *without altering the original source code*.
- The type class (pattern) is a mechanism of ensuring one type conforms to some abstract interface.

Polymorphisms

In programming languages and type theory, a polymorphism is a provision of a single interface to entities of different types.

The same operation working on different types of values.

- **Subtype polymorphism** (aka, Pure Polymorpshim)
- **Parametric Polymorphism** (e.g., Java Generics, C++ templates)
- **Ad hoc Polymorphism**
  - Based on mixin the behaviors using traits

Trait

A trait describes an *abstract interface* that types can implement.

This interface consists of **associated items**:

- **functions**
- **types**
- **constants**

```
trait Example {
    const CONST_NO_DEFAULT: i32;
    const CONST_WITH_DEFAULT: i32 = 99;
    type TypeNoDefault;
    fn method_without_default(&self);
    fn method_with_default(&self) {}
}
```

Generic Traits

- Type parameters can be specified for a trait to make it generic.

```
trait Seq<T> {
    fn len(&self) -> u32;
    fn elt_at(&self, n: u32) -> T;
    fn iter<F>(&self, f: F) where F: Fn(T);
}
```


Supertraits and Subtraits

- **Supertraits** are traits that are *required to be implemented* for a type to implement a specific trait.

```
trait Shape { fn area(&self) -> f64; }

trait Circle: Shape {
    fn radius(&self) -> f64;
}

// same as above
trait Circle where Self: Shape {
    fn radius(&self) -> f64;
}
```

- A trait can have multiple supertraits.

```
trait Circle: Shape + std::fmt::Display {
    fn radius(&self) -> f64;
}
```

A subtraits has access to the associated items of its supertraits

```
trait Shape {
    fn area(&self) -> f64;
}

trait Circle: Shape {
    fn radius(&self) -> f64 {
        // A = pi * r^2
        // so algebraically,
        // r = sqrt(A / pi)
        (self.area() / std::f64::consts::PI).sqrt()
    }
}
```


Traits can be implemented on any type

```
struct Dog;

impl Dog {
    fn bark(&self) {
        println!("Woof!");
    }
}
```

trait you defined

```
trait Animal {
    fn make_sound(&self);
}
```

for types you defined

```
impl Animal for Dog {
    fn make_sound(&self) {
        self.bark();
    }
}
```

for primitive types

```
impl Animal for i32 {
    fn make_sound(&self) {
        println!("i32");
    }
}
```

for types you didn't define

```
impl Animal for ThirdParty {
    fn make_sound(&self) {
        self.third_party_method();
    }
}
```

Traits can even be implemented on Generic Types

```
trait Identity {
    fn id(self) -> Self;
}

impl<T> Identity for T {
    fn id(self) -> T {
        self
    }
}
```

What will happen if … ?

```
impl Identity for i32 {
    fn id(self) -> i32 {
        self
    }
}
```


Blanket Implementation

- It is an implementation of a trait either for all types, or for all types that match some condition. For example, the standard library has this `impl`:

```
impl<T> ToString for T
  where
    T: Display + ?Sized,
{ ... }
```

It is a blanket `impl` that implements `ToString` for all types that implement the `Display` trait.

Trait Coherence and Orphan Rule

Trait coherence (or simply "coherence") is the property that **there is at most one implementation of a trait for any given type**.

Orphan Rule

you can implement only

- Local traits on any type
- Foreign traits on local types

This rule prevents ambiguous implementation

Calling Trait Functions

```
trait Animal {
    fn name(&self) -> String;
    fn die() {
        println!("Oh no! I'm dead!");
    }
}

struct Dog;

impl Animal for Dog {
    fn name(&self) -> String {
        “Jindol".to_string()
    }
}
```

```
let dog = Dog;
let name = dog.name();
let name = Animal::name(&dog);
let name = <Dog as Animal>::name(&dog);

// must call via impl
Animal::die(); // oops!
<Dog as Animal>::die();
```

Two Ways to Use Traits

- As trait bounds for generics to define constraints

or

- Defining trait objects for *dynamic dispatching*

Trait Bounds

- When working with generics, the type parameters often use traits as bounds to stipulate what functionality a type implements.

```
// Define a function `printer` that takes a generic type `T`
// `T` must implement trait `Display`.
fn printer<T: Display>(t: T) {
    println!("{t}");
}
```

- Bounding restricts the generic instances to types that conform the bounds.

```
struct Foo<T: Display>(T);

// Error! `Vec<T>` does not implement `Display`.
let foo = Foo(vec![1]);
```

Trait Bounds (con'd)

- Generic instances are allowed to access the methods of traits specified in the bounds.

```
trait HasArea { fn area(&self) -> f64; }

impl HasArea for Rectangle {
    fn area(&self) -> f64 { self.length * self.height }
}

struct Rectangle { length: f64, height: f64 }
struct Triangle { length: f64, height: f64 }

fn area<T: HasArea>(t: &T) -> f64 { t.area() } // `T` must implement `HasArea`

fn main() {
  let rectangle = Rectangle { length: 3.0, height: 4.0 };
  let _triangle = Triangle { length: 3.0, height: 4.0 };

  println!("Area: {}", area(&rectangle));
  println!("Area: {}", area(&_triangle)); // Error: does not implement`HasArea`
}
```


impl Trait

- `impl Trait` provides ways to specify *unnamed but concrete types that implement a specific trait*.
- It can appear in two places:
  1. argument position (as an **anonymous type parameter**)
  2. return position (as an **abstract return type**)

```
trait Trait {}

// argument position: anonymous type parameter
fn foo(arg: impl Trait) {
}

// return position: abstract return type
fn bar() -> impl Trait {
}
```

Anonymous type parameters
("impl Trait in argument position")

- Functions can use `impl` followed by a set of trait bounds to declare a parameter as having an anonymous type.
- These two forms are *almost* equivalent:

```
trait Trait {}

// generic type parameter
fn foo<T: Trait>(arg: T) {
}

// impl Trait in argument position
fn foo(arg: impl Trait) { // just syntactic sugar
}
```


Which one is wrong?

```
let dog = Dog {
    age: 7
};

let cat = Cat {
    age: 5
};

shout1(&dog, &cat);
shout2(&dog, &cat);
shout3(&dog, &cat);
```

```
fn shout1(a: &impl Animal, b: &impl Animal) {
    a.make_sound(); // Woof!
    b.make_sound(); // Meow!
}

fn shout2<T: Animal, R: Animal>(a: &T, b: &R) {
    a.make_sound(); b.make_sound();
}

fn shout3<T: Animal>(a: &T, b: &T) {
    a.make_sound(); b.make_sound();
}
```

Abstract return types
("impl Trait in return position")

- Functions can use `impl Trait` to return an abstract return type.
  - This is particularly useful with closures and iterators.

```
fn returns_closure() -> impl Fn(i32) -> i32 {
   |x| x + 1
}
```

```
// Compare to using trait objects
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
     Box::new(|x| x + 1)
}
```

incur performance penalties from heap allocation and dynamic dispatch

Caveat:
Multiple arm's concrete return type must be the same

- Note that there can only be one concrete type in return positions.
- The following is an error even though both types implement `Animal`:

```
fn adopt_pet(kind: bool) -> impl Animal {
    match kind {
         true  => Dog { age: 7 },
         false => Cat { age: 5 },
    }
}
```

Limitation

- `impl Trait` can only appear as a parameter or return type of a free or inherent method function.

- It cannot appear inside implementations of traits, nor can it be the type of a `let` binding or appear inside a type alias.

Trait Objects

- Trait objects, like `&dyn Foo` or `Box<dyn Foo>`, are normal values that store a value of *any* type that implements the `Foo` trait
  - where the precise type can only be known at runtime.

- Trait objects use a special record of function pointers, called **vtable** for *dynamic dispatching* (*aka* dynamic polymorphism).

- In Rust, traits are "unsized " types, which means that they are always passed by pointer like `Box` (which points onto the heap) or `&` (which can point anywhere).

Obtaining a Trait Object

- A trait object can be obtained from a pointer to a concrete type that implements the trait by *casting* it (e.g. `&x as &dyn Foo`) or *coercing* it (e.g. using `&x` as an argument to a function that takes `&dyn Foo`).

```
let animal: &dyn Animal = &dog; // casting

shout(&dog); // coercion

fn shout(animal: &dyn Animal) {
    animal.make_sound();
}
```

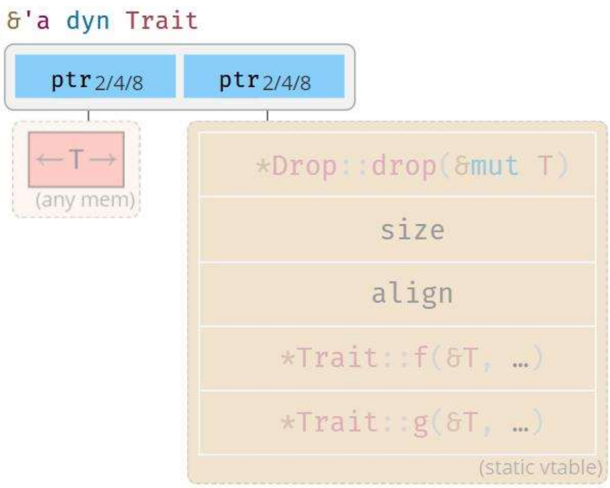
`&mut x` to `&mut dyn Foo` also works.

- This operation can be seen as "erasing" the compiler's knowledge about the specific type of the pointer, and hence trait objects are sometimes referred to "**type erasure**".

Representation (just for demo purpose)

```
// use std::raw::TraitObject
pub struct TraitObject {
    pub data: *mut (),    // data pointer
    pub vtable: *mut (), // vtable pointer
}
```

- The `data` pointer addresses the data (of some unknown type `T`) that the trait object is storing.
- The `vtable` pointer points to the **vtable** ("virtual method table") corresponding to the implementation of a *Trait* for `T`.



Meta points to vtable, where `*Drop::drop()`, `*Trait::f()`, ... are pointers to their respective `impl` for `T`.

Trait Object

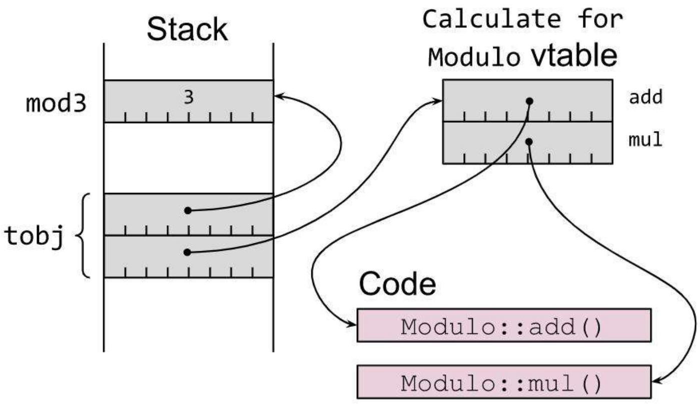
```
trait Calculate {
  fn add(&self, l: u64, r: u64) -> u64;
  fn mul(&self, l: u64, r: u64) -> u64;
}

struct Modulo(pub u64);

impl Calculate for Modulo {
  fn add(&self, l: u64, r: u64) -> u64 {
    (l + r) % self.0
  }
  fn mul(&self, l: u64, r: u64) -> u64 {
    (l * r) % self.0
  }
}

let mod3 = Modulo(3);
```

```
let tobj: &dyn Calculate = &mod3;
```



Dynamically Sized Types (DSTs)

- There's two classes of examples in current Rust:
  - `[T]` and *`Trait`*
- Unsized values must always appear behind a pointer at runtime, like `&[T]` or `Box<dyn Trait>`.

Note: The `str` is usually considered a slice, since it is just a `[u8]` with the guarantee that the bytes are valid UTF-8.

?Sized

```
fn foo<T: Sized>() {} // can only be used with sized T

fn bar<T: ?Sized>() {} // can be used with both sized and unsized T
```

- `Sized` is a default bound for type parameters.
- `?Sized` is a way to opt-in to a parameter not necessarily being sized.

Trait Objects as Parameters

```
fn play_sound(a: &dyn Animal, b: &dyn Animal) {
    a.make_sound();
    b.make_sound();
}

let dog = Dog {
    name: String::from("Spot"),
};

let cat = Cat {
    name: String::from("Felix"),
};

play_sound(&dog, &cat);
play_sound(&dog, &dog);
play_sound(&cat, &dog);
play_sound(&cat, &cat);
```

Trait Objects as Return Values

```
fn adopt_a_pet(kind: bool) -> Box<dyn Animal> {
    // `match` arms have different compatible types
    match kind {
        true => Box::new(Dog { name: String::from("Spot") }),
        false => Box::new(Cat { name: String::from("Felix") }),
    }
}

let pet = adopt_a_pet(false);
pet.make_sound();
```


Object Safety

A trait object can only be constructed out of traits that satisfy certain restrictions, which are collectively called "*object safety*".

1. **Method's return type must not use `Self`.**
   - `&self`, `Box[Self]` **etc. are okay.**
2. **Methods must not be generic.**
3. Must not require `Sized`.
4. **Must not have associated constants.**
5. Supertraits must be object safe.

A good intuition is "except in special circumstances, if your trait's method uses Self, it is not object-safe."

Choosing impl Trait or dyn Trait

	Pros	Cons
dyn Trait	• a single variable, argument, or return value can take values of multiple different types.	• virtual dispatch means slower method calls • objects must always be passed by pointer • requires object safety
Impl Trait	• fine-grained control of properties of types using **where** clauses • can have multiple trait bounds (e.g., impl (Foo + Qux) is allowed, but dyn (Foo + Qux) is not),	• monomorphisation causes increased code size.

Associated Types

- Improves the overall readability of code by moving inner types locally into a trait as output types.

associated type

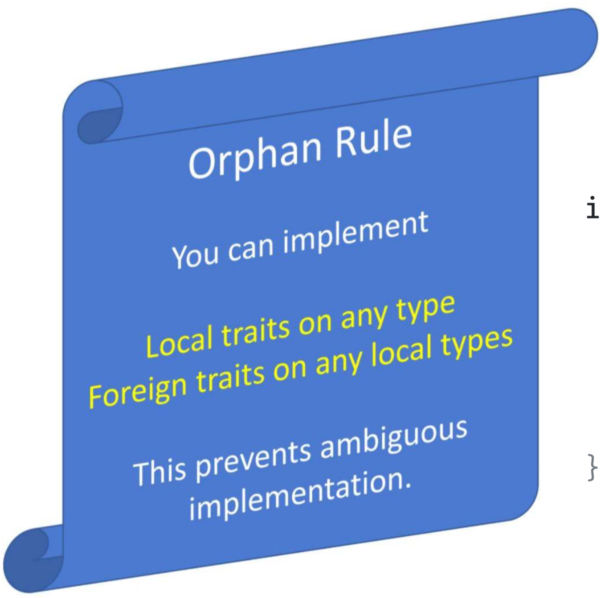
```
trait Iterator {
    type Item;

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

Generic Parameters vs. Associated Types

- Generic parameters are like trait's "*input types*" - when a method is being called, it's the trait's user who gets to state them.
  - You can implement the same Trait for the same struct multiple times with different generic types respectively, because `Trait<i32>` is a different type than `Trait<bool>`.
  - In short, use generics when you want to type `A` to **be able to implement a trait any number of times for different type parameters**, such as in the case of the `From` trait.
- Associated types are like trait's "*output types*" - when a method is being called, it's the trait's implementer who gets to state them.
  - You can implement the same Trait for the same struct only once for a single associated type.
  - Use associated types if **it makes sense for a type to only implement the trait once**, such as with `Iterator` and `Deref`.

Bypassing the Orphan Rule for Traits




```rust
impl fmt::Display for rand::rngs::StdRng {
    fn fmt(&self,
           f: &mut fmt::Formatter<'_>)
    -> Result<(), fmt::Error> {
        write!(f, "<StdRng instance>")
    }
}
```

Bypassing the Orphan Rule for Traits (cont'd)

```
// Use Newtype Pattern
struct MyRng(rand::rngs::StdRng);

impl fmt::Display for MyRng {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> Result<(), fmt::Error> {
        write!(f, "<Rng instance>")
    }
}
```


Standard Traits

- `Clone`: Items of this type can make a copy of themselves when asked.
- `Copy`: If the compiler makes a bit-for-bit copy of this item's memory representation, the result is a valid new item.
- `Default`: It's possible to make new instance of this type with sensible default values.
- `PartialEq`: There's a partial equivalence relation for items of this type – any two items can be definitively compared, but it's not always true that x==x.
- `Eq`: There's an equivalence relation for items of this type: any two items can be definitively compared.
- `PartialOrd`: Some items of this type can be compared and ordered.
- `Ord`: All items of this type can be compared and ordered.
- `Hash`: Items of this type can produce a stable hash of their contents when asked.
- `Debug`: Items of this type can be displayed to programmers.
- `Display`: Items of this type can be displayed to users.

Non-derive able Standard Traits

- `Fn`, `FnOnce` and `FnMut`: Items of this type represent closures that can be invoked.
- `Error`: Items of this type represent error information that can be displayed to users or programmers, and which may hold nested sub-error information.
- `Drop`: Items of this type perform processing when they are destroyed, which is essential for RAII patterns.
- `From` and `TryFrom`: Items of this type can be automatically created from items of some other type, but with a possibility of failure in the latter case.
- `Deref` and `DerefMut`: Items of this type are pointer-like objects that can be dereferenced to get access to an inner item.
- `Iterator` and friends: Items of this type can be iterated over.
- `Send` and `Sync`: Items of this type are safe to transfer between, or be referenced by, multiple threads.

Marker Traits

- There are no methods in marker traits.
- Marker traits are used to indicate some constraint on a type that's not directly expressed in the type system.
- Examples:
  - `Send`
  - `Sync`
  - `Copy`
  - `Sized`
  - `Eq`

  etc.

```
pub trait Copy: Clone { }
```

Auto Traits

- Auto traits gets implemented automatically by the compiler.
- All auto traits are marker traits, but not vice versa.
- Examples
  - `Send`
  - `Sync`
  - `Unpin`

  etc.

```
pub unsafe auto trait Send {
    // empty.
}
```


Copy trait

```
pub trait Copy: Clone { }
```

- The meaning of `Copy` marker is that not only can values be cloned, but also be duplicated as a bit-wise copy.
- Effectively, this trait is a marker that says that a type is a "*plain old data*" (POD) type.
- It shifts the compiler *from move semantics to copy semantics*.

Copy trait (Cont'd)

- Method 1

```
#[derive(Copy, Clone)]
struct MyStruct;
```

The derive strategy will also place a Copy bound on type parameters

- Method 2

```
struct MyStruct;

impl Copy for MyStruct { }

impl Clone for MyStruct {
    fn clone(&self) -> MyStruct {
        *self
    }
}
```

If a type is Copy then its Clone implementation only needs to return *self

Copy trait (Cont'd)

- A type can implement `Copy` if all of its components implement `Copy`.

```
#[derive(Copy, Clone)]
struct Point {
    x: i32,
    y: i32,
}
```

```
#[derive(Copy, Clone)]
struct PointList {
    points: Vec<Point>, // error since Vec
                        // is not a Copy
}
```

- Shared references (`&T`) are also `Copy`, so a type can be `Copy`, even when it holds shared references of types `T` that are not `Copy`.

- `&mut T` or any type implementing `Drop` (i.e., managing resources) can't be `Copy`.

Default trait

- The `Default` trait defines a default constructor, via a `default()`.
- The most useful aspect of the `Default` trait is its combination with **struct update syntax**.

```
#[derive(Default)]
struct Color {
    red: u8,
    green: u8,
    blue: u8,
}
```

```
let c = Color {
    red: 128,
    ..Default::default()
};
```

- Another useful usage is to apply `std::mem::take(&mut T)`.

PartialEq and Eq traits

- The `PartialEq` and `Eq` traits allow you to define equality for user-defined types.
  – The compiler will automatically use them for equality (`==`) checks.
- `eq` can also be written `==`, and `ne` can be written `!=`.

```
pub trait PartialEq<Rhs = Self>
Where Rhs: ?Sized {
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool {…}
}

pub trait Eq: PartialEq<Self> { }
```

PartialEq and Eq traits (Cont'd)

```
enum BookFormat { Paperback, Hardback, Ebook }

struct Book {
    isbn: i32,
    format: BookFormat,
}

impl PartialEq for Book {
    fn eq(&self, other: &Self) -> bool {
        self.isbn == other.isbn
    }
}

impl Eq for Book {}
```

Note that the derive strategy #[derive(Eq)] requires all fields are Eq, which isn't always desired.

PartialOrd and Ord

- The ordering traits `PartialOrd` and `Ord` allow comparisons between two items.
  - The compiler will automatically use them for `<`, `>`, `<=`, `>=`.

```
pub enum Ordering {
    Less,
    Equal,
    Greater,
}
```

```
pub trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
where Rhs: ?Sized {
  fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;

  fn lt(&self, other: &Rhs) …
  fn le(&self, other: &Rhs) …
  fn gt(&self, other: &Rhs) …
  fn ge(&self, other: &Rhs) …
}
```

```
pub trait Ord: Eq + PartialOrd<Self> {
  fn cmp(&self, other: &Self) -> Ordering;

  fn max(self, other: Self) …
  fn min(self, other: Self) …
  fn clamp(self, min: Self, max: Self) -> Self
  where Self: Sized + PartialOrd<Self> {…}
}
```

Operator Overloading

- You can make your own types support arithmetic and other operators, too, just by implementing a few built-in traits.

```
trait std::ops::Add<RHS=Self> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}
```

```
struct Point2d {
    x: i32,
    y: i32,
}

impl std::ops::Add for Point2d {
    type Output = Point2d;

    fn add(self, rhs: Point2d) -> Point2d {
        Point2d {
            x: self.x + rhs.x,
            y: self.y + rhs.y,
        }
    }
}
```

```
let x = Point2d { x: 1, y: 2 };
let y = Point2d { x: 3, y: 4 };

let z = x + y;
```


Many uses of Traits

- Conditional APIs (with Trait Bounds)
- Extension methods
- Overloading
- Closures
- Operators
- Markers

```
// Implement `Default()` only for type `T` that also
// implements `Default`.
impl<T: Default> Default for Foo<T> {
    fn default() -> Self {
        Self::new(T::default())
    }
}

struct Pair<A, B> { first: A, second: B }
impl<A: Hash, B: Hash> Hash for Pair<A, B> {
    fn hash(&self) -> u64 {
        self.first.hash() ^ self.second.hash()
    }
}
```