For any given trait `Trait` there may be a related *type* called the *trait object type* which is typically written as `dyn Trait`. In earlier editions of Rust, trait object types were written as plain `Trait` (just the name of the trait, written in type positions) but this was a bit too confusing, so we now write `dyn Trait`.

Some traits are not allowed to be used as trait object types. The traits that are allowed to be used as trait object types are called "object-safe" traits. Attempting to use a trait object type for a trait that is not object-safe will trigger error E0038.

Two general aspects of trait object types give rise to the restrictions:

1. Trait object types are dynamically sized types (DSTs), and trait objects of these types can only be accessed through pointers, such as `&dyn Trait` or `Box<dyn Trait>`. The size of such a pointer is known, but the size of the `dyn Trait` object pointed-to by the pointer is *opaque* to code working with it, and different trait objects with the same trait object type may have different sizes.
2. The pointer used to access a trait object is paired with an extra pointer to a "virtual method table" or "vtable", which is used to implement dynamic dispatch to the object's implementations of the trait's methods. There is a single such vtable for each trait implementation, but different trait objects with the same trait object type may point to vtables from different implementations.

The specific conditions that violate object-safety follow, most of which relate to missing size information and vtable polymorphism arising from these aspects.

The trait requires `Self: Sized`

Traits that are declared as `Trait: Sized` or which otherwise inherit a constraint of `Self:Sized` are not object-safe.

The reasoning behind this is somewhat subtle. It derives from the fact that Rust requires (and defines) that every trait object type `dyn Trait` automatically implements `Trait`. Rust does this to simplify error reporting and ease interoperation between static and dynamic polymorphism. For example, this code works:

```
trait Trait {
}

fn static_foo<T:Trait + ?Sized>(b: &T) {
}

fn dynamic_bar(a: &dyn Trait) {
    static_foo(a)
}
```

This code works because `dyn Trait`, if it exists, always implements `Trait`.

However as we know, any `dyn Trait` is also unsized, and so it can never implement a sized trait like `Trait:Sized`. So, rather than allow an exception to the rule that `dyn Trait` always implements `Trait`, Rust chooses to prohibit such a `dyn Trait` from existing at all.

Only unsized traits are considered object-safe.

Generally, `Self: Sized` is used to indicate that the trait should not be used as a trait object. If the trait comes from your own crate, consider removing this restriction.

Method references the `Self` type in its parameters or return type

This happens when a trait has a method like the following:

```
trait Trait {
    fn foo(&self) -> Self;
}

impl Trait for String {
    fn foo(&self) -> Self {
        "hi".to_owned()
    }
}

impl Trait for u8 {
    fn foo(&self) -> Self {
        1
    }
}
```

(Note that `&self` and `&mut self` are okay, it's additional `Self` types which cause this problem.)

In such a case, the compiler cannot predict the return type of `foo()` in a situation like the following:

```
trait Trait {
    fn foo(&self) -> Self;
}

fn call_foo(x: Box<dyn Trait>) {
    let y = x.foo(); // What type is y?
    // ...
}
```

If only some methods aren't object-safe, you can add a `where Self: Sized` bound on them to mark them as explicitly unavailable to trait objects. The

functionality will still be available to all other implementers, including `Box<dyn Trait>` which is itself sized (assuming you `impl Trait for Box<dyn Trait>`).

```
trait Trait {
    fn foo(&self) -> Self where Self: Sized;
    // more functions
}
```

Now, `foo()` can no longer be called on a trait object, but you will now be allowed to make a trait object, and that will be able to call any object-safe methods. With such a bound, one can still call `foo()` on types implementing that trait that aren't behind trait objects.

Method has generic type parameters

As mentioned before, trait objects contain pointers to method tables. So, if we have:

```
trait Trait {
    fn foo(&self);
}

impl Trait for String {
    fn foo(&self) {
        // implementation 1
    }
}

impl Trait for u8 {
    fn foo(&self) {
        // implementation 2
    }
}
// ...
```

At compile time each implementation of `Trait` will produce a table containing the various methods (and other items) related to the implementation, which will be used as the virtual method table for a `dyn Trait` object derived from that implementation.

This works fine, but when the method gains generic parameters, we can have a problem.

Usually, generic parameters get *monomorphized*. For example, if I have

```
fn foo<T>(x: T) {
    // ...
}
```

The machine code for `foo::<u8>()`, `foo::<bool>()`, `foo::<String>()`, or any other type substitution is different. Hence the compiler generates the implementation on-demand. If you call `foo()` with a `bool` parameter, the compiler will only generate code for `foo::<bool>()`. When we have additional type parameters, the number of monomorphized implementations the compiler generates does not grow drastically, since the compiler will only generate an implementation if the function is called with unparameterized substitutions (i.e., substitutions where none of the substituted types are themselves parameterized).

However, with trait objects we have to make a table containing *every* object that implements the trait. Now, if it has type parameters, we need to add implementations for every type that implements the trait, and there could theoretically be an infinite number of types.

For example, with:

```
trait Trait {
    fn foo<T>(&self, on: T);
    // more methods
}

impl Trait for String {
    fn foo<T>(&self, on: T) {
        // implementation 1
    }
}

impl Trait for u8 {
    fn foo<T>(&self, on: T) {
        // implementation 2
    }
}

// 8 more implementations
```

Now, if we have the following code:

```
# trait Trait { fn foo<T>(&self, on: T); }
# impl Trait for String { fn foo<T>(&self, on: T) {} }
# impl Trait for u8 { fn foo<T>(&self, on: T) {} }
# impl Trait for bool { fn foo<T>(&self, on: T) {} }
# // etc.
fn call_foo(thing: Box<dyn Trait>) {
    thing.foo(true); // this could be any one of the 8 types above
    thing.foo(1);
    thing.foo("hello");
}
```

We don't just need to create a table of all implementations of all methods of

`Trait`, we need to create such a table, for each different type fed to `foo()`. In this case this turns out to be (10 types implementing `Trait`)*(3 types being fed to `foo()`) = 30 implementations!

With real world traits these numbers can grow drastically.

To fix this, it is suggested to use a `where Self: Sized` bound similar to the fix for the sub-error above if you do not intend to call the method with type parameters:

```
trait Trait {
    fn foo<T>(&self, on: T) where Self: Sized;
    // more methods
}
```

If this is not an option, consider replacing the type parameter with another trait object (e.g., if `T: OtherTrait`, use `on: Box<dyn OtherTrait>`). If the number of types you intend to feed to this method is limited, consider manually listing out the methods of different types.

Method has no receiver

Methods that do not take a `self` parameter can't be called since there won't be a way to get a pointer to the method table for them.

```
trait Foo {
    fn foo() -> u8;
}
```

This could be called as `<Foo as Foo>::foo()`, which would not be able to pick an implementation.

Adding a `Self: Sized` bound to these methods will generally make this compile.

```
trait Foo {
    fn foo() -> u8 where Self: Sized;
}
```

Trait contains associated constants

Just like static functions, associated constants aren't stored on the method table. If the trait or any subtrait contain an associated constant, they cannot be made into an object.

```
trait Foo {
    const X: i32;
}

impl Foo {}
```

A simple workaround is to use a helper method instead:

```
trait Foo {
    fn x(&self) -> i32;
}
```

Trait uses `Self` as a type parameter in the supertrait listing

This is similar to the second sub-error, but subtler. It happens in situations like the following:

```
trait Super<A: ?Sized> {}

trait Trait: Super<Self> {
}

struct Foo;

impl Super<Foo> for Foo{}

impl Trait for Foo {}

fn main() {
    let x: Box<dyn Trait>;
}
```

Here, the supertrait might have methods as follows:

```
trait Super<A: ?Sized> {
    fn get_a(&self) -> &A; // note that this is object safe!
}
```

If the trait `Trait` was deriving from something like `Super<String>` or `Super<T>` (where `Foo` itself is `Foo<T>`), this is okay, because given a type `get_a()` will definitely return an object of that type.

However, if it derives from `Super<Self>`, even though `Super` is object safe, the method `get_a()` would return an object of unknown type when called on the function. `Self` type parameters let us make object safe traits no longer safe, so they are forbidden when specifying supertraits.

There's no easy fix for this. Generally, code will need to be refactored so that you no longer need to derive from `Super<Self>`.