DISPATCH

```
// define an example struct, make it printable
#[derive(Debug)]
struct Foo;

// an example trait
trait Bar {
    fn baz(&self);
}

// implement the trait for Foo
impl Bar for Foo {
    fn baz(&self) { println!("{:?}", self) }
}

// This is a generic function that takes any T that implements trait Bar.
// It must resolve to a specific concrete T at compile time.
// The compiler creates a different version of this function
// for each concrete type used to call it so &T here is NOT
// a trait object (as T will represent a known, sized type
// after compilation)
fn static_dispatch<T>(t: &T) where T:Bar {
    t.baz(); // we can do this because t implements Bar
}

// This function takes a pointer to a something that implements trait Bar
// (it'll know what it is only at runtime). &dyn Bar is a trait object.
// There's only one version of this function at runtime, so this
// reduces the size of the compiled program if the function
// is called with several different types vs using static_dispatch.
// However performance is slightly lower, as the &dyn Bar that
// dynamic_dispatch receives is a pointer to the object +
// a vtable with all the Bar methods that the object implements.
// Calling baz() on t means having to look it up in this vtable.
fn dynamic_dispatch(t: &dyn Bar) {
    // ----------------^
    // this is the trait object! It would also work with Box<dyn Bar> or
    // Rc<dyn Bar> or Arc<dyn Bar>
    //
    t.baz(); // we can do this because t implements Bar
}

fn main() {
    let foo = Foo;
    static_dispatch(&foo);
    dynamic_dispatch(&foo);
}
```

For further reference, there is a good [Trait Objects chapter of the Rust book]