BinaryTraits.jl

This package provides a traits framework for defining formal interfaces with respect to data types. The system is designed to be easy to use/learn/understand. Because it uses Holy Traits pattern as the underlying mechanism, it works out of the box today!

I want to...

- Specify formal interfaces in code
- Verify correctness of interface implementation
- Dispatch using traits rather than abstract types

The old way...

1. **Document the interface** and hope everyone reads the manual
2. **Manually define an abstract type** and respective concrete types for traits
3. **Develop a dispatcher** function to forward calls to implementations

BinaryTraits!

1. Define trait: @trait
2. Define interface: @implement
3. Assign types: @assign
4. Verify interface: @check
5. Trait-aware functions: @traitfn

These macros provide a clean syntax when using traits.

Tom Kwong (GitHub: tk3369, Twitter: tomkwong)

Defining traits and interface contracts


```julia
# Use package and import desired positive/negative trait type aliases
using BinaryTraits
using BinaryTraits.Prefix: Can

# Define a trait and its interface contracts
@trait Fly
@implement Can{Fly} by fly(_, destination::Location, speed::Float64)

# Define your data type and implementation
struct Bird end
fly(::Bird, destination::Location, speed::Float64) = "Wohoo! Arrived! 🎉"

# Assign your data type to a trait
@assign Bird with Can{Fly}

# Verify that your implementation is correct
@check(Bird)
```

Tom Kwong (GitHub: tk3369, Twitter: tomkwong)

Dispatching on traits easily...

How to work with multiple types having the same trait?

```
struct Duck end
struct Bird end

@assign Duck with Can{Fly}
@assign Bird with Can{Fly}

@traitfn function flap(x::Can{Fly}, frequency::Float64)
    description = frequency > 30 ? "fast" : "slow"
    @info "Flapping wings $description"
end

flap(Bird(), 50.0)
flap(Duck(), 20.0)
```

The **@traitfn** macro defines a dispatcher function automatically.

Tom Kwong (GitHub: tk3369, Twitter: tomkwong)