Follow Along

- Code on slides is Tough
- https://github.com/tjtaill/tdd-presentatio
- https://tdd-presentation.herokuapp.com/
- The presentation is a spectacle js React App with CD to Heroku

Making Local API Harder to Misuse

> **APIs should be easy to use and "hard to misuse". It should be easy to do simple things. possible to do complex things. "impossible, or at least difficult, to do wrong things"**

- Joshua Bloch

Type Systems

- Are a key tool to making API hard to misuse.
- They act as a constraint language greatly reducing the state space of legal programs.
- These techniques fall under the label Type Driven Development
- Making Illegal States Unrepresentable

How Many Possible Javascript Literal Assignment?

```
let value = ?;
```

How Many Possible Typescript Literal Assignment?

```
let value: boolean = ?;
```

Type	Literals
boolean	2
number	2^{54}
string	$(2^{53} - 1) * 95$
array	$(2^{32} - 1) * \xi$
object	$< (2^{32} - 1) * \xi$

Java Phantom Typing Money

```
interface Currency {};
interface Usd extends Currency {};
interface Cdn extends Currency {};

class Money<C extends Currency> {
    private final long cents;

    Money(long cents) { this.cents = cents; }
    long value() { return this.cents; }
    Money<C> add(Money<C> other) {
        return new Money<>(this.value() + other.value());
    }
}

class PhantomMoney {
    public static void main(String[] args) {
        var m1 = new Money<Usd>(500);
        var m2 = new Money<Cdn>(200);
        var m3 = new Money<Cdn>(1000);
        var m4 = m3.add(m2); // works java
        // var m5 = m1.add(m2); // won't compile
    }
}
```

Showing Relevance

- $\binom{n}{k} = \frac{n\,!}{k\,!\,(n-k)\,!}$
- \mathchoice$\left(((\binom{n+k-1}{k}\right.$\mathchoice$\left.\right)))) = \frac{n+k-1!}{k!(n-1)!}$
- If we have 180 currencies in the world
- \mathchoice$\left(((\binom{180+2-1}{2}\right.$\mathchoice$\left.\right)))) = \frac{181\,!}{2\,!\,(179)\,!} = 2$

Typing Stragtegies

Typing	Description	Languages
Nominal	Typing based on name not data shape, better error messages	Java
Structural	Typing based on shape of data, more powerful and flexible, bad error messages	Typescript
Duck	Flexible like Structural typing but no compile time safety doesn't suffer from bad error messages	Javascript

Typescript Phantom Typing Class Branding

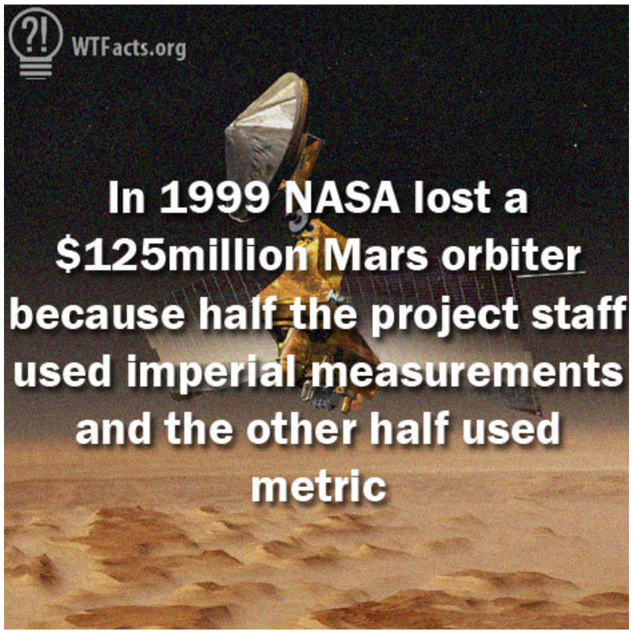
```
export interface Unit { _unitBrand: any };
export interface Meters extends Unit { _metersBrand: any };
export interface Feet extends Unit { _feetBrand: any };

export class Quantity<U extends Unit> {
    private _quantityBrand: U;
    value : number;
    constructor(value : number) {
        this.value = value;
        this._quantityBrand = <U>{};
    }

    add(quantity : Quantity<U>) {
        return new Quantity<U>(this.value + quantity.value);
    }
}

let m1 = new Quantity<Meters>(1);
const m2 = new Quantity<Meters>(5);
const f1 = new Quantity<Feet>(42);
const m3 = m1.add(m2);
const m4 = m1.add(f1); // doesn't compile
```

More Motivation


WTFacts.org

In 1999 NASA lost a $125million Mars orbiter because half the project staff used imperial measurements and the other half used metric

Typescript Phantom Typing Intersection Branding

```
type Currency<T> = number & { _currencyBrand: T };

function sum<T>( a1 : Currency<T>, a2: Currency<T>) : Currency<T> {
    return (a1 + a2) as Currency<T>
}

type USD = Currency<"USD">;
type CDN = Currency<"CDN">;
type GBP = Currency<"GBP">;

const usd = 10 as USD;
const cdn = 10 as CDN;
const gbp = 10 as GBP;

const n : USD = cdn; // this rightfully doesn't compile

sum(usd, usd); // ok
sum(cdn, usd); // compiles but you don't want it to

function add(a1: USD, a2: USD) : USD {
    return (a1 + a2) as USD;
}

add(usd, usd); // ok
add(cdn, usd); // doesn't type check
```

Unit Libraries

Language	Liraries
C++	boost units
Java	jscience, units of measure
Typescript	safe-units

Java Phantom Callbacks

```java
interface ResponseEndCalled {}

interface Response {
    default ResponseEndCalled end(){ return new ResponseEndCalled() {}; }
}

interface Handler {
    public ResponseEndCalled handle(Response resp);
}

interface Route {
    default void handler(Handler handler) {};
}

interface Router {
    default Route route(String path) { return new Route(){}; }
}

class PhantomCallbacks {
    public static void main(String[] args) {
        Router router = new Router(){};
        router.route("/").handler((Response response) -> {
            // will not compile without this
            return response.end();
        });
    }
}
```

Typescript Phantom Callbacks

```
type CallbackCalled = { _brand: string };

interface Callback {
    (err?: Error | null): CallbackCalled;
}

function cb() : CallbackCalled {
    return <CallbackCalled>{};
}

function foo(callback : Callback) : CallbackCalled {
    // return callback();
}
```

Typescript Phantom State

```typescript
type Data<K, T> = K & { _dataBrand: T };

type Received<K> = Data<K, "RECIEVED">;
type Validated<K> = Data<K, "VALIDATED">;

function receive<K>(data : K) : Received<K> {
    return data as Received<K>;
}

function validate<K>(data : Received<K>) : Validated<K> {
    return <Validated<K>>{};
}

const num : number = 5;
const r = receive(n);
const v = validate(r);
validate(num); // this rightly doesn't type check
```

Java Phantom State

```java
interface Received {}
interface Validated {}
interface Sanitized {}

class Data<K, S> {
    private final K data;
    Data(K data) { this.data = data; }
    K value() { return data; }
}

class PhantomState {
    public static <K> Data<K, Received> received(K data) {
        return new Data<K, Received>(data);
    }

    public static <K> Data<K, Validated> validate(Data<K, Received> data) {
        return new Data<K, Validated>(data.value());
    }

    public static <K> Data<K, Sanitized> sanitized(Data<K, Validated> data) {
        return new Data<K, Sanitized>(data.value());
    }

    public static void main(String[] args) {
        var r = received(5);
        var v = validate(r);
        var s = sanitized(r); // won't compile
    }
}
```

Composition Types Sum vs Product

Type	Description	Examples
Product	Composition types whose enumerated possible values are the cartesian product of all composed members	• Typescript : Object, Tuple, Intersections, Interface, Class • Java: Class, Map
Sum	Composition type whose enumerated possible values is the sume of all composed members	• Typescript : Unions, enum • Java: Optional, Enum, Try(vavr)

Typescript Sum Type Option

```
type Some<T> = { tag: "some", value: T };
type None = { tag: "none" };
type Option<T> = Some<T> | None;

const opt : Option<string> = { tag: "some", value: "Foo" };

// with strictNullChecks enabled and wraped in function
// with a return type that doesn't include undefined
// you can make a swith case be exhaustively checked
function map<A,B>(opt: Option<A>, f: (a: A) => B) : Option<B> {
    switch(opt.tag) {
        case "some":
            return { tag: "some", value: f(opt.value)};
        case "none":
            return { tag: "none" };
    }
}

function flatMap<A,B>(opt: Option<A>, f: (a: A) => Option<B>) : Option<B> {
    switch(opt.tag) {
        case "some":
            return f(opt.value);
        case "none":
            return { tag: "none" };
    }
}

const optBar: Option<string> = map({ tag: "some", value: "Foo" }, (a: string) => "Bar");
```

Java Optional Sum Type

```
import java.util.Optional;

class Optionals {
    public static void main(String[] args) {
        var foo = Optional.of("Foo");
        String nil = null;
        var none = Optional.ofNullable(nil);
        var bar = foo.map((a) -> "Bar");
        var alsoNone = none.map((a) -> "Bar");
        System.out.println(alsoNone.isEmpty());
    }
}
```

Typescript Sum Type Result

```
interface Ok<V> {
    tag: "ok"
    value: V
}

interface Failure<F> {
    tag: "failure"
    failure: F
}

type Result<V,F> = Ok<V> | Failure<F>

function map<A,B,C>(r: Result<A,B>, f: (a: A) => C) : Result<C,B> {
    switch(r.tag) {
        case "ok":
            return { tag: "ok", value: f(r.value) };
        case "failure":
            return { tag: "failure", failure: r.failure };
    }
}

const foo: Result<string, Error> = { tag: "ok", value: "Foo"};
const bar = map(foo, (a) => "Bar");
const failure: Result<string, Error> = { tag: "failure", failure: new Error("Failed!") };
const alsoFailure = map(failure, (a) => "Bar");
if (alsoFailure.tag == "failure") {
    console.log(alsoFailure.failure.message);
}
```

Java Vavr Try Sum Type

```
import io.vavr.control.Try;

class Result {
    public static void main(String[] args) {
        var foo = Try.of(() -> "Foo");
        var bar = foo.map((a) -> "Bar");
        var err = Try.of(() -> { throw new RuntimeException("failure"); });
        err
            .map((a) -> "Bar")
            .orElseRun(System.out::println);
    }
}
```

Java 14 Switch Expressions

```java
import java.util.function.Function;

enum ResultState {
    OK,
    ERROR
}

class Result<V> {
    final V value;
    final Throwable error;
    final ResultState state;

    Result(V value, Throwable error, ResultState state) {
        this.value = value;
        this.error = error;
        this.state = state;
    }

    static <V> Result<V> of(V value) {
        return new Result<>(value, null, ResultState.OK);
    }

    // new switch expression standard in java 14
    // is exhaustive unlike regular switch
    <W> Result<W> map(Function<V,W> func) {
        return switch(this.state) {
                case OK -> Result.of(func.apply(this.value));
                case ERROR -> new Result<>(null, this.error, ResultState.ERROR);
            };
    }
}

class SwitchExpression {
    public static void main(String[] args) {
        var foo = Result.of("Foo");
        var bar = foo.map((a) -> "Bar");
        var err = new Result<String>(null, new RuntimeException("error"), ResultState.ERROR);
        var alsoErr = err.map((a) -> "Bar");
    }
}
```

Value Objects

- One of the building blocks of Domain Driven Design
- Small objects representing values without identity
- Immutable
- Support standard interface for serialization, equality and hashing
- Maintain their State Invariants during Runtime
- Most portable and simplest technique shown can be used with dynamically typed languages

Typescript Value Objects

```
import * as EmailValidator from 'email-validator';

class EmailAddress {
    readonly value: string;

    constructor(value: string) {
        if (!EmailValidator.validate(value)){
            throw new Error("Invalid email address");
        };
        this.value = value;
    }
}

// here the question should I do defensive programming
// and validate them email address a.k.a fail slowly error
// will be reported further from where it happened.
function sendEmail1(emailAddress: string, email: string) {
}

// Here it is known that the email address is valid, could use
// PhantomTypes too. This is fali fast error is reported closer to
// where it actually happened
function sendEmail2(emailAddress: EmailAddress, email: string) {
}
```

Java Value Objects

```java
import java.time.ZonedDateTime;

class DateRange {
    final ZonedDateTime start;
    final ZonedDateTime end;

    DateRange(String start, String end) {
        var s = ZonedDateTime.parse(start);
        var e = ZonedDateTime.parse(end);
        if (!e.isAfter(s)) {
            throw new IllegalArgumentException("end must be after start");
        }
        this.start = s;
        this.end = e;
    }
}

class ValueObjects {
    public static void main(String[] args) {
    }
}
```

Value Object Libraries

Languages	Libraries
Java	Lombok, Immutables
Javascript, Typescript	tiny-types, dataclass

Making illegal states unrepresentable is all about statically proving that all runtime values (without exception) correspond to valid objects in the business domain. The effect of this technique on eliminating meaningless runtime states is astounding and cannot be overstated.

- John A De Goes (@jdegoes) January 28, 2019

The End

- Phantom Types, Sum Types and Value Objects are awesome
- Please send feedback
- Consider giving a tech talk yourself to foster a culture of learning