Records and Tuples

a.k.a. Immutable Data Structures

for Stage 1

Robin Ricard & Richard Button
Bloomberg

Advisor: Daniel Ehrenberg
Igalia

A proposal by example

```
const city = #{
      city: "New York",
      state: "NY",
      country: "USA",
};
```

```
const fib = #[1, 1, 2, 3, 5, 8];
```

```
const cities = #[
    {   city: "New York",
        state: "NY",
        country: "USA" },
    {   city: "London",
        region: "Greater London",
        country: "UK" }
];
```

```
const nyOffice = #{
    isHQ: true,
    city: {
        city: "New York",
        state: "NY",
        country: "USA" },
};
```

```
const london = {
  city: "London",
  region: "Greater London",
  country: "UK"
};


const office = #{
  isHQ: false,
  city: london // TypeError
};
```

```
const albany = #{ ...nyc, city: "Albany" };
```

```
let fib = #[1, 1];
for (let i = 0; i < 100; ++i) {
    fib = fib.push(fib[i] + fib[i+1]);
}
// fib = #[1, 1, 2, 3, 5, 8, ...];
```

```
let food = #["pasta", "tomato sauce",
             "concrete"];
const materials = #[food[2]];
food = food.pop();
// food =  #["pasta", "tomato sauce"];
// materials =  #["concrete"];
```

```
let fib = #[0, 1, 2, 3, 0, 8];
fib = fib.with(0, 1).with(4, 5);
// fib = #[1, 1, 2, 3, 5, 8];
```

```
assert(nyOffice === #{
  isHQ: true,
  city: {
      city: "New York",
      state: "NY",
      country: "USA" },
});
```

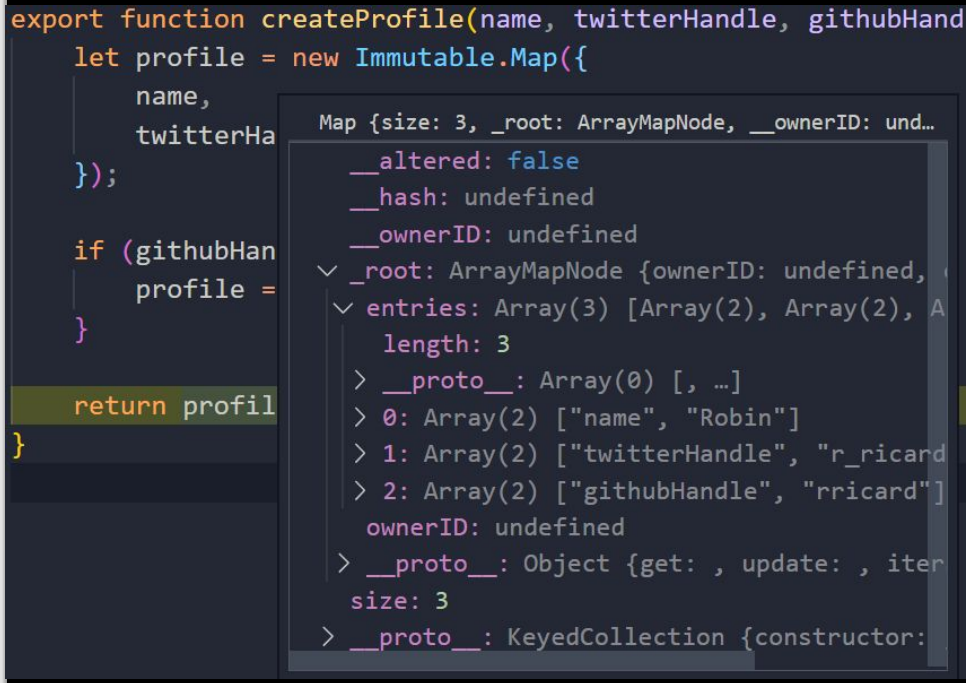
```
assert(nyOffice !== #{
  isHQ: false,
  city: {
    city: "London",
    region: "GreaterLondon",
    country: "UK" },
});
```

Motivations

Can it be done in userland?

- Several approaches exist in the community
  - Immutable - similar semantics, could be a transpilation target
  - Immer - more of a pattern, could be used with Records and Tuples
- These libraries are fantastic! But they have usability drawbacks

Internal implementation is “magical”, makes debugging more difficult.



Not adopted by the rest of the ecosystem (libraries) because they need to be handled as special cases.

Same issues within a large project.

```
const ProfileRecord = Immutable.Record({
    name: "Anonymous User",
    githubHandle: null,
});

function getGithubUrl(profile) {
    if (Immutable.Record.isRecord(profile)) {
        return `https://github.com/${
            profile.get("githubHandle")
        }`;
    }
    return `https://github.com/${
        profile.githubHandle
    }`;
}
```

Mixing immutables and non-immutables can be a foot-gun.

Leads to bugs in large codebases.

```
const user = { name: "Robin" };

const commit = CommitRecord({
   hash: "5a8945",
   user,
});

Immutable.Record.isRecord(commit);
// => true

Immutable.Record.isRecord(
    commit.get("user"));
// => false
```

Interoperability with the rest of the ecosystem requires lots of costly conversions to and from standard js objects.

```
const jobResult = Immutable.fromJS(
    ExternalLib.processJob(
        jobDescription.toJS()
    )
);
```

This can be mitigated by the proposal since we can use the same access idioms as objects

If it works with an object it will likely work with a record or tuple.

```
function getGithubUrl(profile) {
    if (Immutable.Record.isRecord(profile)) {
        return `https://github.com/${
            profile.get("githubHandle")
        }`;
    }
    return `https://github.com/${
        profile.githubHandle
    }`;
}
```

```
function getGithubUrl(profile) {
    return `https://github.com/${
        profile.githubHandle
    }`;
}

getGithubUrl(#{ githubHandle: “rricard” })
// => https://github.com/rricard

getGithubUrl({ githubHandle: “rickbutton” })

// => https://github.com/rickbutton
```

Using deeply frozen objects?

- You can write your own deep freezing, deep equality and deep clone!
- Immer actually results in cloned + deeply frozen objects

From Immer’s docs: the frozen state stops out of simple objects and arrays.

```
const state = {
    set: new Set()
}
const nextState = produce(state, draft => {
    // Don't use any Set methods,
    // as that mutates the instance!
    draft.set.add("foo") // ❌

    // 1. Instead, clone the set (just once)
    const newSet = new Set(draft.set) // ✔️
    // 2. Mutate the clone
    // (just in this producer)
    newSet.add("foo")
    // 3. Update the draft with the new set
    draft.set = newSet
})
```

It also prompts the question of what is deep equality and deep cloning in the general case.

In a large project or across library boundaries, the meaning of “deep” can change.

This is a possible source of bugs.

This feature defines equality semantics between value types and throws when an incomparable value is introduced in the structure.

```
const london = {
  city: "London",
  region: "Greater London",
  country: "UK"
};


const office = #{
  isHQ: false,
  city: london // TypeError
};
```

Builds on existing value types

- string, number, … have “special” semantics today compared to objects
- Records and Tuples are a generalization of those semantics
- We can imagine further proposals that expand in that domain

Interactions with the rest of the language

```
assert(#{ a:  1 } === #{ a: 1 });
assert(#[1] === #[1]);
assert(#{ a: -0 } !== #{ a: +0 });
assert(#[-0] !== #[+0]);
assert(#{ a: NaN } === #{ a: NaN });
assert(#[NaN] === #[NaN]);
```

```
assert(#{ a: 1 } == #{ a: 1 });
assert(#[1] == #[1]);

assert(Object.is(#{ a: 1 }, #{ a: 1 }));
assert(Object.is(#[1], #[1]));
```

```
assert(new Map().set(#{a:1}, true).get(#{a:1}));
assert(new Map().set(#[1], true).get(#[1]));
assert(new Set([#{a:1}]).has(#{a:1}));
assert(new Set([#[1]]).has(#[1]));
```

```
assert(Object(#{a:1}) instanceof Record);
assert(Object(#[1]) instanceof Tuple);
assert(typeof #{a:1} === "record");
assert(typeof #[1] === "tuple");
```

Boxing objects prototypes

- `Record.prototype` and `Tuple.prototype` are neither `Record` or `Tuple` instances. They have `Object.prototype` as prototype.
- Value types follow `GetValue` semantics (implicit boxing + follow prototype)
- `Record.prototype` is empty
- `Tuple.prototype` ~= `Array.prototypes` with a few changes
  - .with() added
  - .shift() / .unshift() / .pop() / .push() all return a new tuple

Iteration

- Records are not iterable
- Tuples are iterable
  - Any iterable consumer will be able to go through a Tuple’s values
    - for(const v of tuple)
  - Additionally the Tuple boxing object has a non-writable, non-enumerable, non-configurable length property that reflects the number of elements in the tuple
    - for(let i = 0; i < tuple.length; ++i)

Discussion!

What does the committee think about immutable data structures in JavaScript?

What does the committee think about this proposal as a starting point for this space?