Pure Functions

Redux and React require you to write pure functions

This protects the data in your “state”

Key questions:

1) What is a pure function?
2) What is an impure function?
3) Why does Redux require your reducer to be pure?
4) How can I push to an array without mutating the array?
5) How can I change an array element without mutating the array?
6) How can I remove an array element without mutating the array?
7) How can I change an object’s keys without mutating it?

Pure functions calculate an output solely based on input parameters

No outside data allowed! (look up “deterministic” functions)

PURE

```
function squareNumber(x) {
    return x * x;
}
```

IMPURE

```
var interest  = Database.getInterest();
function principleTimesInterest(x) {
        return x * interest;
}
```

Pure functions don’t mutate inputs or variables outside their scope

No side effects!

PURE

```
function markTodoCompleted(todo) {
    return {
        todoText: todo.todoText,
        isCompleted: true
    }
}
```

IMPURE

```
function markTodoCompleted(todo) {
    todo.isCompleted = true;
    return todo;
}
```

Everything about pure functions screams “easy”!

Because pure functions always return the same result for the same inputs (deterministic), and the output doesn’t rely on outside data (no side-effects), they are ridiculously easy to read, understand, and debug

Redux needs you to write pure reducers

It’s because they’re afraid an impure reducer will mess with the STATE in an unpredictable (non-deterministic) way

Redux needs you to write pure reducers

```
function myTodoReducer(state, action) {
    if  (action.type == “ADD_TODO”) {
        return state.todos.push(action.text);
    } else {
        return state;
    }
}
```

This line is problematic! It changes the STATE input, which messes with Redux’s state management code

Arrays are easy to accidentally mutate

```
myArray.push( “hi” );

myArray.pop(“hi”);

myArray.splice(4, 1, “hi”);

myArray[1] = “hi”;
```

MUTATION!

```
var newArray = myArray.concat(“hi”);

var newArray = myArray.slice();

newArray[0] = “hi”;
```

No mutation!

Objects are easy to accidentally mutate

```
myObject.foo =  “hi”;

var newObject = myObject;

newObject.foo = “hi”

delete myObject.hi;
```

MUTATION!

```
var newObject = Object.assign({}, myObject);

newObject.foo = “hi”
```

No mutation!

The “spread” operator (...) is a great way to avoid mutations!

Using ... to update a “shallow” object

```
function updateObject(object) {
    var newObject = {
        ...object,      // Copy all the key/values from object
        foo: “bar”      // Overwrite the “foo” key from object
    };
}
```

Using … to update a “deep” object

```
function updateObject(object) {
    var newObject = {
        ...object,      // Copy all the key/values from object
        nestedObject: {
            ...object.nestedObject // The nested object needs to be spread too
        }
    };
}
```

Using ... to copy an array

```
function copyArray(array) {
    var newArray = [
        ...array     // Copy all the items from array
    ];
}
```

Using ... to update an array item

```
function updateArray(array, someIndex) {
    var newArray = [
        ...array.slice(0, someIndex),
        100,      // Changing the value at someIndex
        ...array.slice(someIndex+1)
    ];
}
```

.map(), .filter(), and .concat() are safe!

Use them as needed when messing with arrays in your redux state