Actions and Reducers 101

We must only allow state to change in a controlled way

- When done right, state will only change when we dispatch an action that is handled in a reducer.
- But what do those words mean?

Actions

An action is an object which fully describes the change to be made to state

- It always has a "type" property which is a string
- It usually has a payload ... a series of additional properties that may be needed.
  - "Move forward? How many spaces?"
  - "Increase balance? By how much?"
  - "Change the name? To what?"

You'll begin by listing out all the valid ways that data can change

- Ask yourself ... "How can my data change?"
- And list all the ways out. List them ALL out.
- Make this a chart of every action and the payloads they are likely to carry.

The Reducer

A reducer is a pure function that receives in the old state and an action and returns a different state object

It MUST have this shape:

```
(oldState, action) => newState
```

All reducers do something like this:

```
if (action.type === "foo") {
  const newState = getCopyOfOldState(oldState);
  newState.prop1 = action.newValue1;
  newState.prop2 = action.newValue2;
  return newState;
}
```

Most folks use a switch statement

```
const reducer = (state, action) => {
  if (!action) return state;
  switch (action.type) {
    case "SET_FIRST":
      return {...state, first:action.first};
    case "SET_ZIP":
      return {...state, zip:action.zipcode};
    default:
      return state;
  }
}
```

How to avoid the worst Redux mistakes

Some people have to learn the hard way. :-(

Rookie mistake 1: Reference assigning

```
const newState = oldState
newState.prop1 = action.newValue1
```

- State will change here, but change detection fails because Redux does essentially this behind the scenes:

```
if (newState !== oldState)
  runAllTheListeners()
```

- When you change newState, you're also changing oldState
- Redux will only recognize that state has changed when a deep comparison of oldState is different from newState.

Rookie mistake 2: Not returning state

- If you don't return a state, the dispatch causes state to be undefined!
- Always return the old state if ...

1. There is no action
2. Action has an unknown type

Rookie mistake 3: Changing state directly

- You can totally do this...

```
const state = myStore.getState();
state.prop1 = "Some new value";
```

And it absolutely works! State will change. But there's never a good reason to do this

- Lots of reasons why NOT to ...

1. Subscriptions won't fire
2. Other devs can't find where state is changing
3. Impossible to debug