The `getState()` Method

```
const initialState = 0;
const countUpReducer = (
  state = initialState,
  action
) => {
  switch (action.type) {
    case 'increment':
      return state += 1;
    default:
      return state;
}};

const store = createStore(countUpReducer);

console.log(store.getState());
// Output: 0
```

The `getState()` method of a Redux `store` returns the current state tree of your application. It is equal to the last value returned by the `store`'s reducer.

- In the one-way data flow model (store → view → action → store), `getState` is the only way for the view to access the store's state.
- The state value returned by `getState()` should not be modified directly.

The `subscribe()` Method

```
const printCurrentState = () => {
  const state = store.getState()
  console.log(`state: ${state}`);
}

store.subscribe(printCurrentState);
```

The `subscribe(listener)` method of a Redux `store` adds a callback function to a list of callbacks maintained by the `store`. When the `store`'s state changes, all of the *listener* callbacks are executed. A function that unsubscribes the provided callback is returned from `subscribe(listener)`.

Often, `store.getState()` is called inside the subscribed callback to read the current state tree.

Slices

```
/*
This state has two slices:
1) state.todos
2) state.filter
*/
const state = {
  todos: [
    {
      text: 'Learn React',
      completed: true
    },
    {
      text: 'Learn Redux',
      completed: false
    },
  ],
  filter: 'SHOW_COMPLETED'
}

/*
This slice reducer handles only
the state.todos slice of state.
*/
const initialTodosState = [];
const todosReducers = (
  state=initialTodosState,
  action
) => {
  switch (action.type) {
    case 'todos/clearTodos':
      return [];
    case 'todos/addTodo':
      return [...state, action.payload];
    default:
      return state;
}};
```

A *slice* is the portion of Redux code that relates to a specific set of data and actions within the `store`'s state.

A *slice reducer* is the reducer responsible for handling actions and updating the data for a given slice. This allows for smaller reducer functions that focus on a slice of state.

Often, the actions and reducers that correspond to the same slice of the state are grouped together into a single file.

Installing Redux

```
npm install redux
```

The `redux` package is added to a project by first installing it with `npm`.

Some of the resources imported from `redux` are:

- `createStore`
- `combineReducers`

The `dispatch()` Method

```
const initialState = 0;
const countUpReducer = (
  state = initialState,
  action
) => {
  switch (action.type) {
    case 'increment':
      return state += 1;
    case 'incrementBy':
      return state += action.payload;
    default:
      return state;
}};

const store = createStore(countUpReducer);

store.dispatch({ type: 'increment' });
// state is now 1.

store.dispatch({ type: 'incrementBy'
                 payload: 3 });
// state is now 4.
```

The `dispatch(action)` method of a Redux `store` is the only way to trigger a state change. It accepts a single argument, `action`, which must be an object with a `type` property describing the change to be made. The `action` object may also contain additional data to pass to the reducer, conventionally stored in a property called `payload`.

Upon receiving the `action` object via `dispatch()`, the store's reducer function will be called with the current value of `getState()` and the `action` object.

Create the Redux Store

```
const initialState = 0;
const countUpReducer = (
  state = initialState,
  action
) => {
  switch (action.type) {
    case 'increment':
      return state += 1;
    default:
      return state;
}};

const store = createStore(countUpReducer);
```

The `createStore()` helper function creates and returns a Redux `store` object that holds and manages the complete state tree of your app. The only required argument is a reducer function, which is called every time an action is dispatched.

The `store` object returned has three key methods that ensure that all interactions with the application state are executed through the `store`:

- `store.getState()`
- `store.dispatch(action)`
- `store.subscribe(listener)`

The `combineReducers()` Function

```
const rootReducer = combineReducers({
  todos: todosReducer,
  filter: filterReducer
})
```

The `combineReducers()` helper function accepts an object of slice reducers and returns a single "root" reducer. The keys of the input object become the names of the slices of the `state` and the values are the associated slice reducers.

The returned root reducer can be used to create the `store` and, when executed, delegates actions and the appropriate slices of state to the slice reducers and then recombines their results into the next `state` object.

Action Creators

```
// Creates an action with no payload.
const clearTodos = () => {
  return { type: 'clearTodos' };
}
store.dispatch(clearTodos());

// Creates an action with a payload.
const addTodo = todo => {
  return {
    type: 'addTodo',
    payload: {
      text: todo
      completed: false
    }
  }
};
store.dispatch(addTodo('Sleep'));
```

An *action creator* is a function that returns an action, an object with a `type` property and an optional `payload` property. They help ensure consistency and readability when supplying an action object to `store.dispatch()`, particularly when a `payload` is included.