# Creating a simplified Monopoly

## Lab Overview

In this lab session, you will model a simplified form of the popular board game Monopoly, using a combination of the tuples, records, discriminated unions and options.

## Prerequisites

* Visual Studio 2015

## Time Estimate

* 30 minutes

## Table of Contents

[Creating a simplified Monopoly 1](#_Toc433802957)

[Lab Overview 1](#_Toc433802958)

[Prerequisites 1](#_Toc433802959)

[Time Estimate 1](#_Toc433802960)

[Table of Contents 1](#_Toc433802961)

[Exercise 1: Exploring the solution 2](#_Toc433802962)

[Exercise 2: Calculating Outcomes when landing on Properties 2](#_Toc433802963)

[Exercise 3: Creating a stateful Player 3](#_Toc433802965)

[Bonus Exercise: Automatic Play 3](#_Toc433802969)

## Exercise 1: Exploring the solution

In this exercise, you will familiarize yourself with a stub solution which has been provided for you to start with.

1. Open the SimpleMonopoly solution. This solution contains a number of files: -
   1. **Types.fs**. Contains the different types of Positions on the Monopoly board, modelled as a discriminated union. Notice how some positions have properties attached to them as required; others do not.
   2. **Data.fs**. Contains the data of the Monopoly board. Each item in the list is a different Position e.g. Property or Station; the type of the Board is a Position list.
   3. **Controller.fs**. This module contains functionality that we will use to interact with the board. It currently has a single function that calculates a move on the board, given a dice roll and a current position and a stub method. We’ll add more functions to this module throughout this lab.

The [a; b; c; …] syntax represents an F# List. Don’t worry too much about this for now – just think of it as similar to an Array except F# Lists are immutable.

1. Open **Script.fsx**. This contains some code which we’ll use to guide the development of our domain. Start by testing out the calculateMove function in the Controller module and explore the implementation of the function. Notice: -

* how the dice are modelled as a simple tuple of int
* use of pattern matching when trying to find the index of the supplied position

## Exercise 2: Calculating Outcomes when landing on Properties

In this exercise, we’ll enhance the application so that we can calculate the earnings (or cost) of a player landing on a specific location given some arbitrary logic that we’ll use to “simulate” the game state. Given the position of a player, we want to calculate an “Outcome” that is one of three cases: -

* Pay out some money
* Earn some money
* Go bankrupt

Secondly, we want to create function that, given a position on the board, will calculate the Outcome based on the following rules: -

* If the position is Euston Road, Earn 150
* If the position is Marylebone Road, Earn 150
* If the position is Jail, go Bankrupt
* If the position is Go, Earn 500
* If the position is Free Parking, Earn 250
* If the position is any other Property, Pay 100
* If any other position, there is no outcome at all

1. Model the Outcome in the Types.fs file as a three-case Discriminated Union. You’ll need to attach a property to the Pay Out and Earn cases to record how much money is being paid out / earned.
2. Create a function in the Controller module called calculateOutcome which will implement the logic defined above by taking in a Position and trying to return an Outcome. Its signature should be calculateOutcome : Position -> Outcome option. You’ll want to use pattern matching to match over the Position that is provided in order to calculate the appropriate Outcome. You should also use the Option type on order to model the “no outcome at all”.
3. Create a function in the controller module call move. This function will take in a dice and a position. It should first call calculateNextMove, and with the result of that, then call calculateOutcome. It should return both the new position and the financial outcome for that move (if any). The function definition should look as follows:

let move (dice: int \* int) (position : Position) : (Position \* Outcome option)

## Exercise 3: Creating a stateful Player

We’ll now create a Player record which we will use to manage the state of a player across multiple turns. A player should have three properties: -

* Their name
* The Position that they are currently on
* The money that they have

1. Update the move function to take in a player rather than a position. The move function should also return an updated Player object with the new position. The signature should now look as follows: -

let move (dice: int \* int) (player : Player) : (Player \* Outcome option)

Remember that to perform a copy-and-update of a record we use the “with” syntax e.g

let updatedPerson = { person with Age = 45 }

You may prefer to create a small helper function to perform the update to the player Position (that takes in a player and returns the newly updated player), or simply perform the update inline.

1. Create a new function, updatePlayerEarnings which will update the Player’s earnings based on a given optional Outcome e.g. if a player has Money of 500 and we call this function with an outcome of Some(Earn 150), a newly updated player should be returned with Money of 650. The function signature should be Outcome option -> Player -> Player. If no outcome is provided to the function, you can simply return back out the player that was passed in.

These functions should be defined in partially applied form (rather than as tupled). All function arguments should be supplied with spaces e.g. let function arg1 arg2 = () rather than let function (arg1, arg2)

1. Update the move function so that it now calls the updatePlayerEarnings function as well, and simply returns the updated Player back out. The signature should now look as follows: -

let move (dice: int \* int) (player : Player) : Player

1. Run through the test cases supplied to ensure that your code still works (you may need to update the test code in case the sample definition of Player does not match yours exactly).

## Bonus Exercise: Automatic Play

See if you can write some code that will randomly generate a dice roll, using a System.Random object. With this code you can create a loop that will run a number of iterations, generating a random dice roll every time and supplying this to the move function. You’ll have to create a mutable field to store the player object across iterations – we’ll see how to avoid doing this in another module.

Ensure that you create a static Random object in order to generate random dice rolls, rather than creating a new one on each call to your generator function!