Assignment 7
Due on **March 27th, at the end of the day**. Please follow the submission instructions in the "notes for all labs" on Moodle. This lab may overlap some reading assignments that may require short responses, but won't overlap other coding projects.

- Read the "Notes for All Labs" document on Moodle. All assignments must be submitted as specified there.
- This Assignment
- Your code should be in at least three files, called
    - your_id_330_lab7.hs, where your_id is your Earlham user ID (dmbarbe16, or whatever); this should contain the Coding Assignments
    - your_id_war.hs; this should contain the first Project
    - your_id_jack.hs; this should contain the second Project
- Any additional files you use should use a similar naming convention. For example, you might decide that some code that will be used by both of the projects should go in a fourth file. Such a file might be called your_id_partners_id_cards.hs
- You only need to submit **one** set of code and **one** design for your pair. If you both submit something, I will look at whichever one I see first. (Both people should submit something for the final written question separately.)
- All code files and associated material you submit must have both partners' first and last names in it. (Probably in a header at the top.)
- Explain anything about your code that is not obvious in comments.
- You should endeavor to work on the assignment together as much as possible. Do not simply divide up the assignment into pieces. Both people are responsible for all of the design and for all of the implementation. This is a pair-programming assignment, not a divide-and-conquer assignment.

Assignment Overview 7.0 (0 Points)
Please read the entire assignment carefully before you start.

This assignment involves a set of programming tasks for which functional programming is *not* a particularly good fit. We are taking on these challenges within a functional programming context in order to learn how to work with things that functional programming does not handle quite as simply as some other languages - randomness, input, and output. You should read **Chapter 8** before you start. The Real World Haskell supplemental text offers another window into how to work with these things, and Learn You a Haskell does as well.

You may need to use or want to use things that we have not covered in class, or will not cover in class until part of the way through this project.

Your grade on the projects will not be based merely on whether or not they work (although it is critical that they do work), but on their overall design. Your code should be as clean, clear, and functional as possible. The outputs displayed by your code should be clean and easy to understand.

This assignment will be available for substantially longer than most; I expect it to take a longer period of time.

Coding Assignment 7.1: (1 Point)
Write a “Hello, world” function. Your function should be called `helloWorld`. It should output (not return) the message “Hello, world”

Coding Assignment 7.2: (1 Point)
Write a “Hello, [name]” function. Your function should be called `helloName`. It should ask the user for their name, then output (not return) the message “Hello, [name]”

Coding Assignment 7.3: (5 Points)
Write a function called `eightBall` that simulates a Magic Eight-Ball toy. The toy is an opaque plastic globe with a circular window. The ball contains a dark-colored fluid in which a 20-sided solid floats. The user asks a yes-or-no question of the toy, shakes it, and rotates it so that the window is facing up. This causes the 20-sided solid to float up against the window, revealing the message on one of its sides. Some of the sides have a variation of “yes” on them, such as “It is certain” or “You may rely on it.” Some of the sides have a variation of “no” on them, such as “Outlook not so good.” Some of the sides have an ambiguous or unhelpful answer on them, such as “Reply hazy try again.” (The toy’s answer is totally random, and it has no way of knowing what question was asked; it’s just for fun.)

You can see an animated character using the toy here.

Your function should allow the user to ask a question, and then randomly output a response. (The user’s question is totally ignored and doesn’t affect the answer in any way.) Your version doesn’t need to have twenty possible responses, but it should have at least one response from each category.

Project Assignment 7.4: (20 Points)
For this project, you will build a system that plays the card game “War” against the user. If you are unfamiliar with the game, a basic description of it appears here:
https://www.wikihow.com/Play-War-(Card-Game)
I recognize that this game may not be familiar to everybody. If you are unfamiliar with the rules, please get in touch with me early in the project and we will walk through them. Your implementation only needs to handle the basic rules outlined in the first five steps.

You might notice that the game War (which is usually played by children) does not involve any decisions being made on the part of the players. You should build two functions (plus, most likely, several supporting functions.)

`autoWar` should play though a game of War without any user input. When the function is run, it will deal the cards out to the players, then play against itself automatically, displaying the results as it goes.
`interactiveWar` should play though a game of War with user input. Because the player only has one action they can take (playing their next card), your interface will likely be extremely simple - they just need to hit a key to play their next card. (Seriously, don't overthink this.)

Your main should call `interactiveWar`.

Project Assignment 7.5: (40 Points)

For this project, you will build a system that plays the card game "Blackjack" against the user. While the game has many advanced rules, the very basics are these:

- One person acts as the *dealer*. (In a casino, this is an employee of the casino.) Everybody else is a *player*. In your implementation for this lab, there will be one player (the user), and the computer will take the role of the dealer.
- Each person is initially dealt two cards. The players are dealt both cards face down (but they can look at them). The dealer is dealt one card face down, and one card face up. Everybody can see the dealer's face-up card and use that information to make their decisions.
- Each card is worth the number of points that matches its number. Jacks, Queens, and Kings are all worth 10 points. An Ace can be worth 1 point or 11, whichever is better for the person who has it. In this game, card suits (Hearts, Clubs, Spades, Diamonds) do not matter at all, although you should still display the suits when you show the player the cards. Jokers are not used in normal Blackjack.
- The goal is to have as high as score as possible without going over 21. At the end of the round, if the player has a higher score than the dealer (without going over 21), they win that round. If the player or dealer goes over 21 at any point, they *bust*, meaning that they immediately lose that round.
- If a player's hand or the dealer's initial two-card hand has an Ace and any card worth 10 points, that player has *blackjack*, and they win right away. Nobody needs to take any other actions. If both the player and the dealer have blackjack, it is a tie.
- After a player is dealt their cards, they can choose to *hit* or *stand*. If they choose to *hit*, they are dealt another card, and they add its points to the current value of their hand. A player can hit as many times as they like, but if they go over 21 points, they immediately lose the round. Once a player is satisfied with their hand, they can choose to *stand*. After a player stands, they can no longer hit - their hand remains how it is for the remainder of the round.
- After the player has chosen to stand, the dealer takes their turn. Unlike the player, who can make choices about what to do based on their own cards and on the dealer's face-up card, the dealer *must* follow the rule that if their hand is worth 17 points or more (counting Aces as 11), they stand. If their hand is worth 16 points or less, they hit, and continue to hit until their hand is worth at least 17 points. Your computer dealer should

follow this rule. Just as with the player, if the dealer's hand is ever worth more than 21 points (counting aces as whichever is better for them), they bust and instantly lose that round.
    - The fact that the dealer goes last is what allows the casino to make money on the game, because sometimes the player will bust before the dealer has to go.
- After the dealer is finished, if neither busted, the player's hand is compared to the dealer's. If the player's hand is worth more points, the player wins. If they are the same, it is a tie. Otherwise, the dealer wins.
- To protect against cheating, many casinos will use multiple decks piled together. Your implementation should use exactly one deck, and all of the cards should be shuffled back in each round.

I recognize that this game may not be familiar to everybody. If you are unfamiliar with the rules, please get in touch with me early in the project and we will walk through them.

Your primary function should be called `blackjack`. You will almost certainly want other functions to support it. Your main should call this function.

You do **not** have to incorporate any betting elements into your implementation of your game, although you can do so if you wish. You also do not have to incorporate splits, insurance, doubling down, or any other blackjack rules, although you may choose to do so if you wish. Both your readme and your interface should make it clear how to use any of these that you choose to implement.

Because Blackjack is normally played as a gambling game, I understand if this is a project that not everybody may be able to pursue. If this is true for you, you should contact me as soon as possible so that we can design an alternate project.

Written Assignment 7.6: (20 Points)

Describe, **in a few paragraphs**, your design and implementation process for the project.