



# C# From Start To Finish: Tournament Tracker

**Companion Guide**

# CONTENTS

---

**3**

LETTER FROM THE AUTHOR

**4**

INTRODUCTION

**5**

PLANNING STEPS

**28**

CODING STEPS

**44**

WRAPING UP

# LETTER FROM THE AUTHOR

*When I started my YouTube channel, I had one goal in mind: to supplement my in-class lectures at a local college with videos that my students could watch at any time. The quality was low, the topics were specific to my class and I didn't even consider the idea of others watching them.*

*Imagine my surprise when I started gaining followers. It started small at first, a few here, a few there. Pretty soon I had 1,000 subscribers and I realized people were looking for what I had. Over the past couple of years, I've talked to some of my subscribers. Over and over I've heard the same story: I was struggling to figure out how to do this, but your video made it clear.*

*I think the reason people resonate with my videos is because I struggled to learn this stuff. It wasn't easy. I patched together learning from multiple different sources before I finally understood them. When I teach, I remember how I struggled. I remember the questions I would ask so I answer them. That seems to work for my viewers.*

*The launch of this course marks a major step in the evolution of my YouTube channel. I used to help people accidentally. Then I started to help people more intentionally. Now I am working to broaden and deepen my impact in the software development community.*

*This change hasn't been cheap. Between the investment in equipment and software to improve the video quality to the hundreds of consulting hours I have passed up, my YouTube channel has cost thousands of dollars. While I am happy to do it, my budget cannot sustain that type of investment indefinitely. That is where you come in. Your purchase of these bonus resources directly impacts my ability to add more content to YouTube. So, from the bottom of my heart, thank you. I hope you find these resources valuable.*

*Tim Corey*



# INTRODUCTION

The goal of this project is to teach real-world techniques on how to build an application. The application we will build has been carefully chosen to highlight several situations that will be useful as teaching opportunities. The key here is context. Learning a topic right when you need it is so much more valuable than trying to learn it just to learn it.

Instead of droning on, let's get right into it and learn as we go.

---

## SCENARIO

Your friends come to you and ask you to create a tournament tracker. They are always playing games and want to determine who is the best. The idea is that you create a bracket tournament system where the computer will tell them who to play in a single-elimination style bracket. At the end, the winner should be identified. Their model is the NCAA Basketball tournament bracket for March Madness.

---

## STARTING POINT

Be careful. This is where you can get into big trouble. Your first step should not have anything to do with Visual Studio. Instead, you need to plan what you intend to do. In fact, I recommend you take five planning steps before you even open Visual Studio. That is why the first five videos in this course are just planning videos. In case you were one of the ones that skipped over those (you know who you are), this might be a good time to go back and watch them. Either way, we are going to go over each of the five parts here.

## Planning Step 1: Initial Planning

Here is where we look at what we have been given and figure out what are true requirements are.

This app has the following stated requirements:

1. Tracks games played and their outcomes (who won).
2. Multiple competitors play in the tournament.
3. Creates a tournament plan (who plays who in what order).
4. Schedules games.
5. A single loss eliminates a player.
6. The last player standing is the winner.

Developing this list should generate some questions about the details of our program. Here are the questions we came up with:

1. How many players will the tournament handle? Is it variable?
2. If a tournament has less than the full complement of players, how do we handle it?
3. Should the ordering of who plays each other be random or ordered by input order?
4. Should we schedule the games or are they just played whenever?
5. If the games are scheduled, how does the system know when to schedule games for?
6. If the games are played whenever, can a game from the second round be played before the first round is complete?
7. Does the system need to store a score of some kind or just who won?
8. What type of front-end should this system have (form, webpage, app, etc.)?
9. Where will the data be stored?
10. Will this system handle entry fees, prizes or other payouts?
11. What type of reporting is needed?
12. Who can fill in the results of a game?
13. Are there varying levels of access?
14. Should this system contact users about upcoming games?
15. Is each player on their own or can teams use this tournament tracker?

These questions are best asked to the person or group that requested the application in the first place. That group of people are called stakeholders. They are the ones with the knowledge of what they want. It is our job to figure out what they want and deliver it. To do that, we must ask great questions and not stop until we are sure we are on the same page as the stakeholders. The answers to our questions will fuel our second planning step.

*Great questions up front save hours and bugs down the road. If you don't have a number of questions, keep trying.*

---

## Planning Step 2: Overview Planning

In step one, we developed a list of questions. From that, we got back several clarifying answers from the stakeholders (in this case, me). Here are those answers:

**Question 1: How many players will the tournament handle? Is it variable?**

The application should be able to handle a variable number of players in a tournament.

**Question 2: If a tournament has less than the full complement of players, how do we handle it?**

A tournament with less than the perfect number (a multiple of 2, so 4, 8, 16, 32, etc.) should add in “byes”. Basically, certain people selected at random get to skip the first round and act as if they won.

**Question 3: Should the ordering of who plays each other random or ordered by input order?**

The ordering of the tournament should be random.

**Question 4: Should we schedule the games or are they just played whenever?**

The games should be played in whatever order and whenever the players want to play them.

**Question 5: If the games are scheduled, how does the system know when to schedule games for?**

They are not scheduled so we do not care.

**Question 6: If the games are played whenever, can a game from the second round be played before the first round is complete?**

No. Each round should be fully completed before the next round is displayed.

**Question 7: Does the system need to store a score of some kind or just who won?**

Storing a simple score would be nice. Just a number for each player. That way, the tracker can be flexible enough to handle a checkers tournament (the winner would have a 1 and the loser a 0) or a basketball tournament.

**Question 8: What type of front-end should this system have (form, webpage, app, etc.)?**

The system should be a desktop system for now, but down the road we might want to turn it into an app or a website.

**Question 9: Where will the data be stored?**

Ideally, the data should be stored in a Microsoft SQL database but please put in an option to store to a text file instead.

**Question 10: Will this system handle entry fees, prizes or other payouts?**

Yes. The tournament should have the option of charging an entry fee. Prizes should also be an option, where the tournament administrator chooses how much money to award a variable number of places. The total cash amount should not exceed the income from the tournament. For example, if a 16-player tournament charges a \$10 entry fee, the tournament will generate \$160. The administrator could set up that the first place gets \$100, second place gets \$50 and third place gets \$10 or the administrator could instead say that first place gets \$50 and second through tenth place get \$10. In the latter case, the extra money is “earned” by the tournament and can be done with whatever the tournament admin decides. A percentage-based system would also be nice to specify. That way if the tournament didn’t get the desired number of players, the payouts would adjust accordingly. All amounts should be rounded to the nearest whole number.

**Question 11: What type of reporting is needed?**

A simple report specifying the outcome of the games per round as well as a report that specifies who won and how much they won. These can be just displayed on a form or they can be emailed to tournament competitors and the administrator.

**Question 12: Who can fill in the results of a game?**

Anyone using the application should be able to fill in the game scores.

### Question 13: Are there varying levels of access?

No. The only method of varied access is if the competitors are not allowed into the application and instead, they do everything via email.

### Question 14: Should this system contact users about upcoming games?

Yes, the system should email users that they are due to play in a round as well as who they are scheduled to play.

### Question 15: Is each player on their own or can teams use this tournament tracker?

The tournament tracker should be able to handle the addition of other members. All members should be treated as equals in that they all get tournament emails. Teams should also be able to name their team.

Now that we have all our questions answered, it is time to adjust our plan we are starting to develop in our head. Let's start by thinking about the structure. As I see it, this application should be a Windows Forms application on the front and it should rely heavily on a class library for everything but UI-related operations. WPF is an alternative, and it is the better pure UI but it is more complicated. Since the stakeholders are already looking at possibly making this application a website or a mobile app, spending too much time on a fancy UI probably isn't the best use of our time.

As we can see, the data access code needs to handle either saving to a text file or to SQL. That complicates things a bit but we can handle it. We also can see that we are going to need to be able to send out emails to users.

## Big Picture Design

At this point, we have a good idea of what our app is going to look like (from a development perspective, not the User Interface). I like to highlight the technologies I think I'll need, as well as the three key areas of the application (structure, data, and users). Here is what those three key areas look like:

- Structure: Windows Forms application with a Class Library
- Data: Microsoft SQL and/or Text Files
- Users: One at a time on one application



Pretty straightforward. Now, let's look at the key concepts or technologies that I'll need to make sure I am ready to handle before I start coding. These concepts are:

- Email
- SQL
- Custom Events
- Error Handling
- Interfaces
- Random Ordering
- Texting

These are the big ones that I'm seeing right now. I should at least know how these will work so that I can implement them correctly in the application. I might even do a practice project or two to make sure I'm familiar enough to know how they work.

---

## Planning Step 3: Data Design

This is where we start breaking out the actual data that needs to be stored or used in this application. I tend to break this all the way down to what will become class models. Once I group them by class and list all the properties the class will have, I then assign types to each property. That just makes putting it into C# code that much easier. Here are the models we came up with:

### Team

- TeamMembers (*List<Person>*)
- TeamName (*string*)

## Person

- FirstName (*string*)
- LastName (*string*)
- EmailAddress (*string*)
- CellphoneNumber (*string*)

## Tournament

- TournamentName (*string*)
- EntryFee (*decimal*)
- EnteredTeams (*List<Team>*)
- Prizes (*List<Prize>*)
- Rounds (*List<List<Matchup>>*)

## Prize

- PlaceNumber (*int*)
- PlaceName (*string*)
- PrizeAmount (*decimal*)
- PrizePercentage (*double*)

## Matchup

- Entries (*List<MatchupEntry>*)
- Winner (*Team*)
- MatchupRound (*int*)

## MatchupEntry

- TeamCompeting (*Team*)
- Score (*double*)
- ParentMatchup (*Matchup*)

That is the basic data for this application. There are a few things that we wrestled over. For starters, the Rounds property in the Tournament model is tricky. It is a List of a List of Matchup. Why? Well, we need an unknown number of rounds and in each round, we will have one or more matchups. Thus, the nested lists.

The Matchup model was a struggle. At first glance, you could add two team entries and then two score entries and two parent entries but that is a lot of duplication and it would muddy up the code for Matchup. Instead, we broke out a new model for MatchupEntry that contained all the information for one team (who they are, what they scored and where they came from).

One thing to note is that while all this data is needed, some of the properties on the above models are derived from other properties and do not actually store their own data. For example, the Winner property in the Matchup model is just a Linq query on the Entries list for which team has the higher score.

---

## Planning Step 4: User Interface Design

This step is one I normally do on a whiteboard or legal pad. **The act of drawing strips away all the distractions of the computer and it tells your brain you are in a different mode.** You aren't worried about naming, accidentally double-clicking the items on the page, or any of a number of other issues. You simply draw. It isn't pretty (unless you are gifted that way) but it doesn't have to be. It just needs to represent a realistic form layout.

The most important benefit, though, is how easy it is to erase or tear up a lesser design. **This freedom allows you to experiment in ways you wouldn't otherwise.** Don't know how a form should be laid out? No problem! Just start putting things down. Guaranteed that simple act will spark thoughts. Maybe not huge thoughts, but you will come up with a tweak or two. Do it again and again until you get it "right". Then pretend the form is real and look at it from the perspective of the user.

So how do I do a form layout in written form? Not well. So, here's what I'm going to do. I'm going to go over the section of data I want to tackle, describe what I'm thinking and then I'll show you my form. It is still a demo, but it is in the Windows Forms layout. After, I'll explain why I implemented things the way I did and possibly what I'm going to change when I build the

real form (no, I will not be copying and pasting these forms over to the real application).

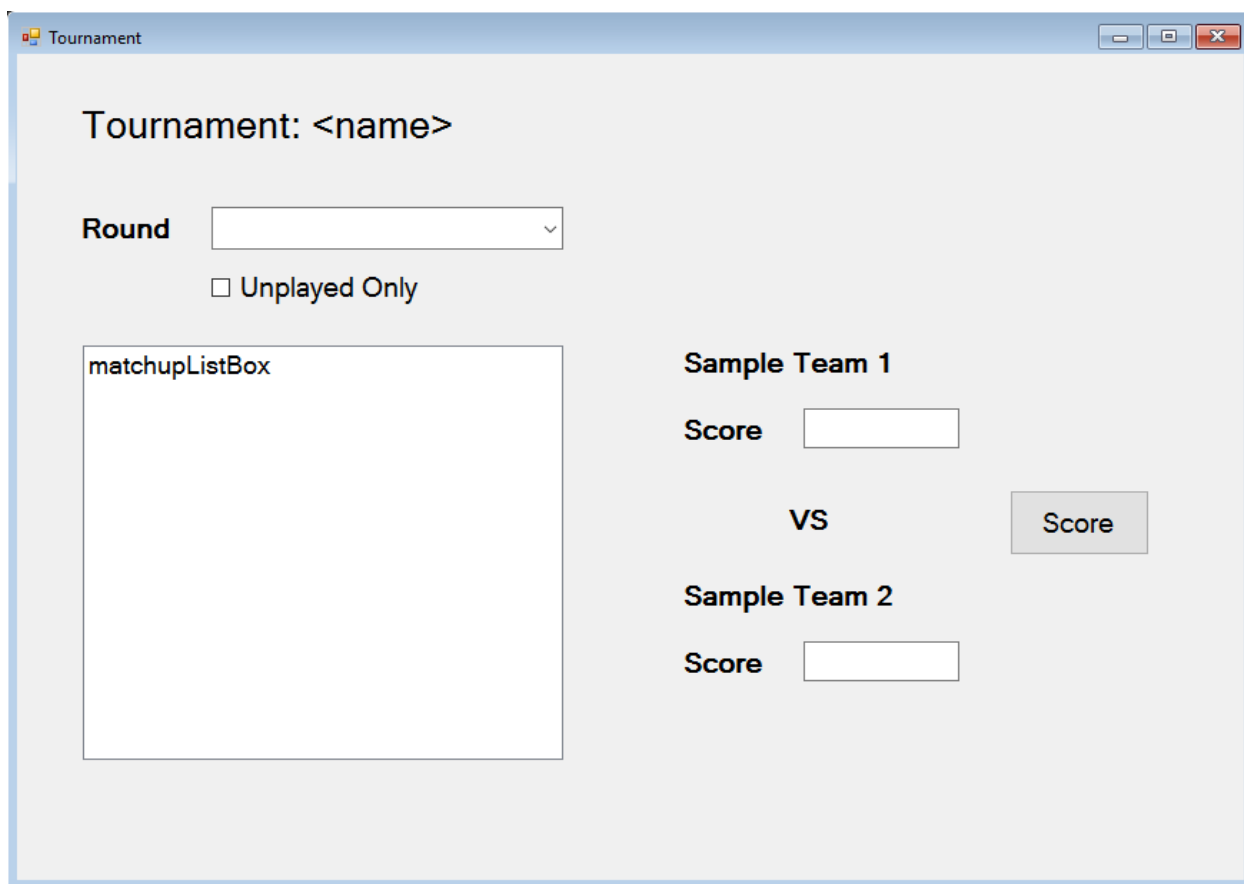
You may wonder how I know which forms to build. The way I do it is I look at the data and see what forms I need to get that data in and display that data. That, for me, dictates my form design. Think about it: almost everything to do with a user interface revolves around data in some way. So, if I have forms that handle all my data, I should be most done, if not completely done.

### Tournament Viewer Form

Which form to start with might be a question. Here's what I think: it doesn't matter. Shocking, right? Some things we sweat over. This isn't one of them. If you aren't sure where to start, look for the model that all other models revolve around. Build a form for that model. That will probably prompt you to create other forms. By the time you are done, you will have most of your forms. From there, just look for what forms are missing and wrap up the loose ends.

In this case, we are going to start with the Tournament form. This is a viewer of an existing tournament. I picked this because this is what most users will see when they open the application. You set up a tournament once but you look back at it over and over.

Here is the form that I came up with:



Note that I didn't pay attention to looks. I'm not a great design guy. I can teach you how to build a form, but look to others to help you make it pretty. That being said, this is ugly even for me. This is a concept design, not a finished product. My only focus was objects on the canvas in a rough layout that seemed to work.

OK, let's look at the actual layout. I couldn't figure out a simple way to show an actual bracket with all the games listed. Those can be complicated. Maybe a third-party control would help but even so, it seems to add a layer of unnecessary complexity. This is doubly so because one of the notes about this project is that the stakeholders are considering going to a web-based solution in the near future.

The solution I came up with for the layout was a simple one. Each round has a number of team matchups in it. I can display a round in the listbox so people can see it. The trick, then, is not overwhelming the user with tons of



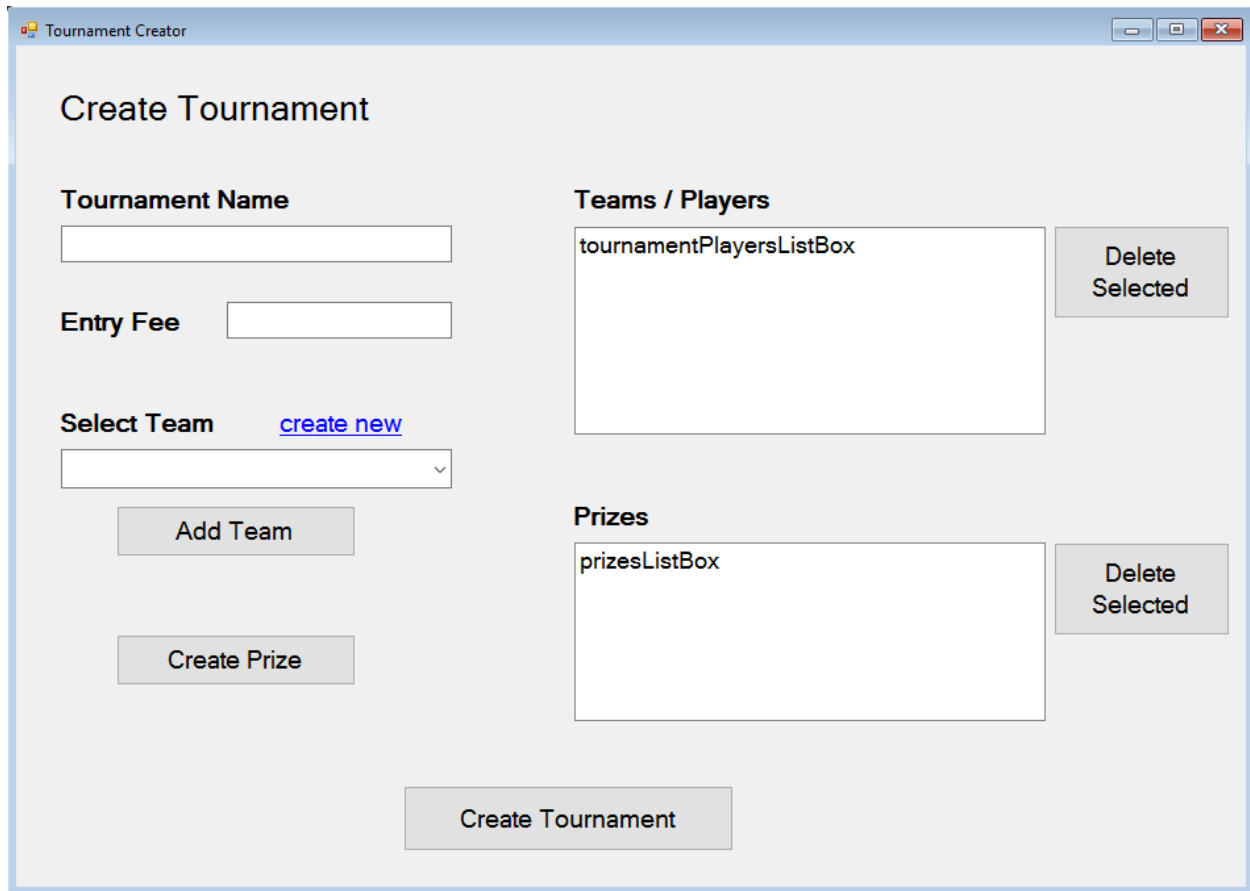
matchups in that listbox. The first way to limit that list is to only show one round. To get to the other rounds, you can choose the round with the dropdown labeled Round. The second way you can limit the list is by only displaying the unplayed games. If a round has 32 matchups (like in the NCAA tournament), that is going to be a lot. However, once games are played the list will shrink.

My next thought was that this interface was a great place to enter the results of a matchup. We already have the list. My thought is that the user selects a matchup on the left and they can enter the scores and hit the “Score” button. Note that “Sample Team 1” and “Sample Team 2” represent the team names of the two teams. These labels would be the team names to make it obvious which team’s score you are entering.

Note that this form represents one tournament. We are setting this system up to be able to run more than one tournament at a time. That means we will need a master page of some kind to show us available tournaments, allow us to create tournaments, and do other things. If you are following along, make a note of that but let’s keep moving.

## Create Tournament Form

If you have tournaments, you must create them. Thus, the Create Tournament form.

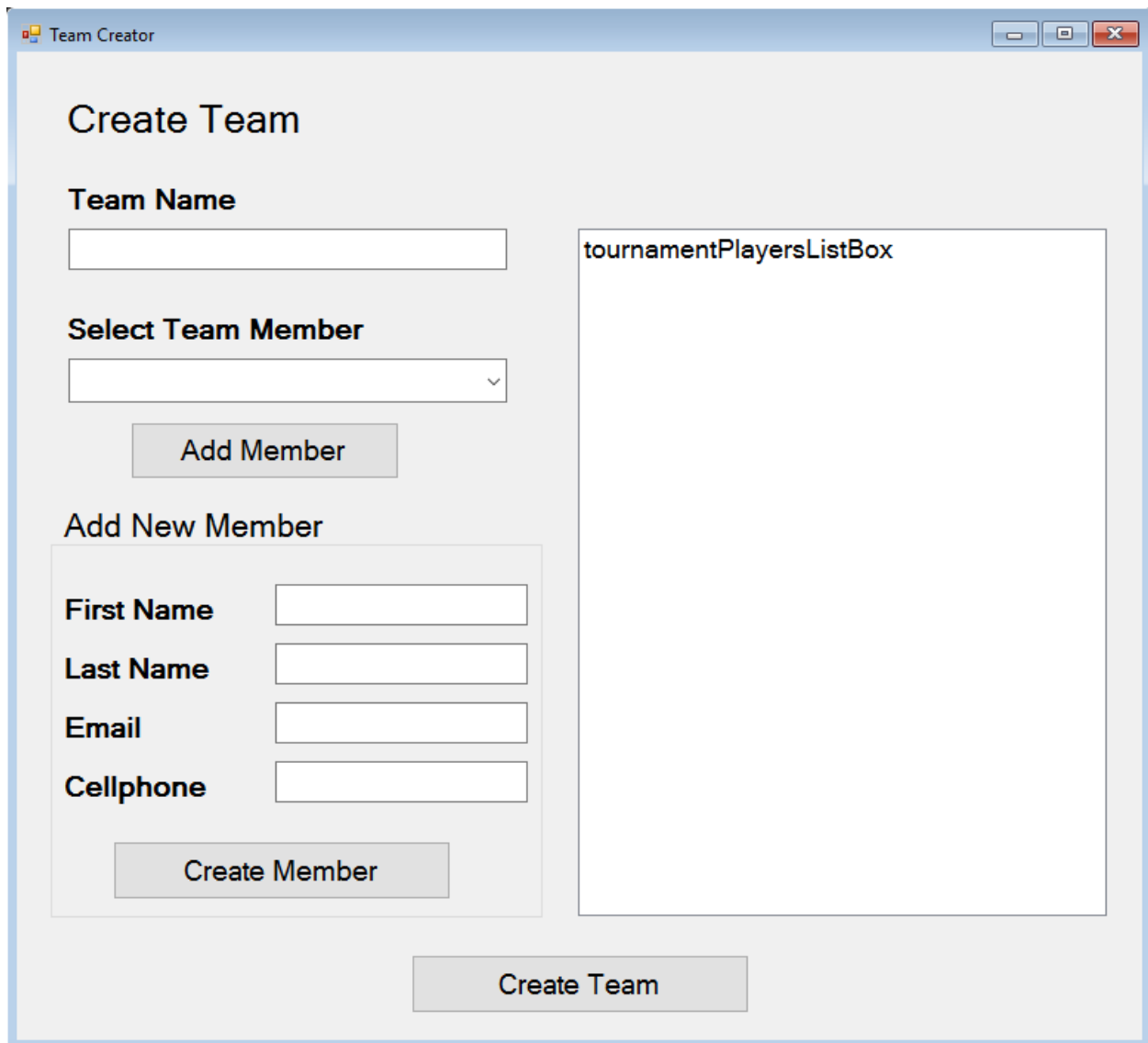


The screenshot shows a Windows-style application window titled "Tournament Creator". Inside, the "Create Tournament" form is displayed. It features several input fields and buttons. On the left, there are fields for "Tournament Name", "Entry Fee", and a "Select Team" dropdown menu with a "create new" link. Below these are "Add Team" and "Create Prize" buttons. On the right, there are two list boxes: "tournamentPlayersListBox" and "prizesListBox", each with a "Delete Selected" button. At the bottom center is a large "Create Tournament" button.

This is a more active form. The keys here are the teams and prizes. The other items are fairly simple entry boxes. My assumption is that an existing team/person could play in more than one tournament. Thus, I created a dropdown of existing players and also allowed for the creation of a new user. I try to keep everything on one form if possible. If a user is forced to go through three or more forms to complete a process, it is going to get confusing. As it is, this form is complicated. Adding prizes can't be inline without making this form do too much. Creating a new team is the same as adding a prize as far as complexity. The dropdown does reduce some complexity, though. I think it is worth it.

## Create Team Form

Since we had a link on the create tournament form for creating a new team, that prompted me to create the Create Team form.



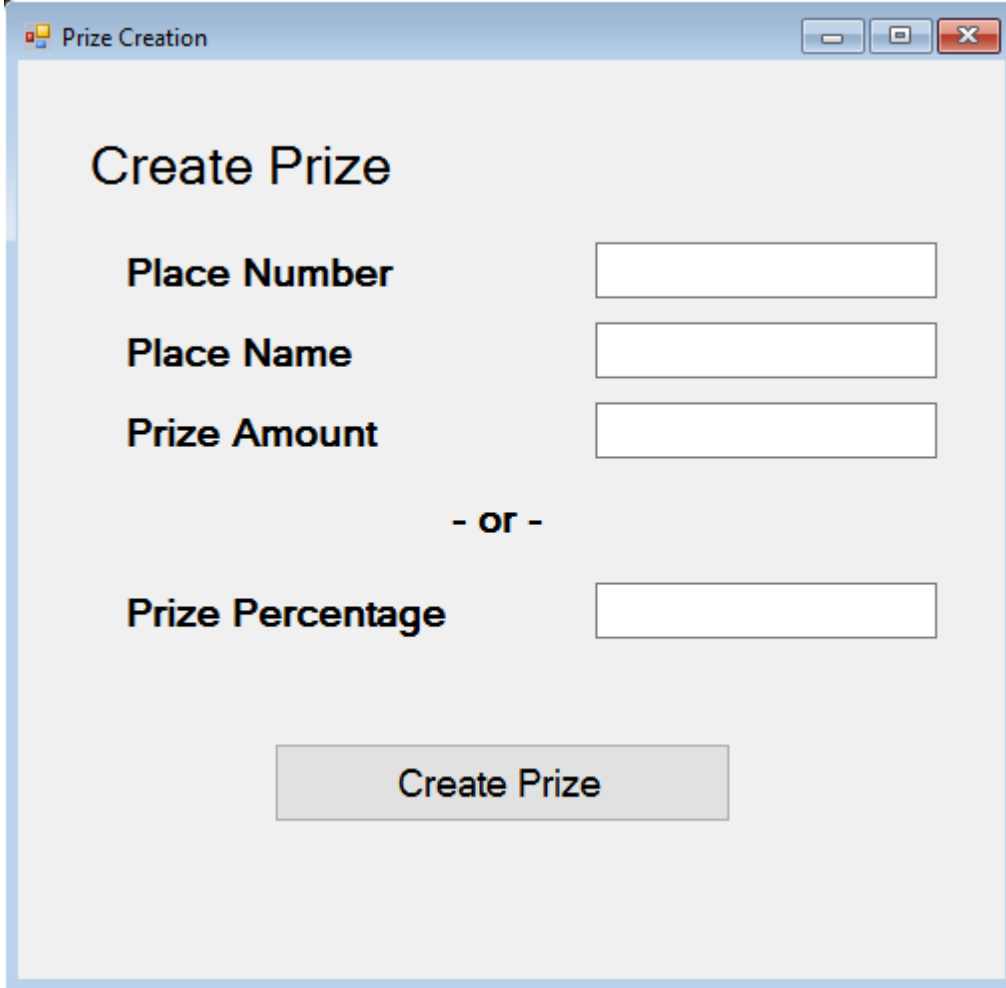
The screenshot shows a Windows-style application window titled "Team Creator". Inside the window is a form titled "Create Team". The form is divided into several sections:

- Team Name:** A single-line text input field.
- Select Team Member:** A dropdown menu with a small downward arrow on the right.
- Add Member:** A button located below the dropdown menu.
- Add New Member:** A section containing four input fields stacked vertically:
  - First Name:** Input field.
  - Last Name:** Input field.
  - Email:** Input field.
  - Cellphone:** Input field.
- Create Member:** A button located below the "Add New Member" section.
- tournamentPlayersListBox:** A large, empty rectangular box on the right side of the form, intended for displaying a list of players.
- Create Team:** A large button at the bottom center of the form.

This form has the same issue as the Create Tournament form. We can add team members from an existing list but sometimes we need to create new people. I solved the issue in the same way as I did on the Create Tournament form. You can select an existing person or you can create a new one. The wrinkle here is that I have the real-estate to add the member inline instead of via a new form.

### Create Prize Form

Just like the Create Team form, the Create Prize form comes out of a necessity made obvious by the Create Tournament form.

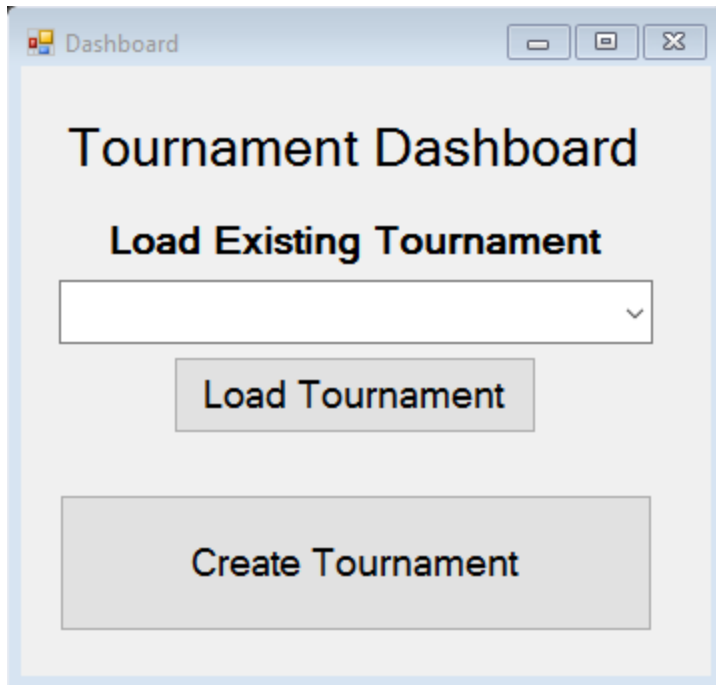


The image shows a Windows-style dialog box titled "Prize Creation". It has a light gray background and a blue border. At the top right are standard window controls: minimize, maximize, and close. The main content area is titled "Create Prize" in a large, bold, black font. Below the title, there are four input fields arranged vertically. The first three are labeled "Place Number", "Place Name", and "Prize Amount" in bold black text. The fourth is labeled "Prize Percentage" in bold black text. Between the "Prize Amount" and "Prize Percentage" fields, there is a separator consisting of a hyphen, the word "or", and another hyphen. At the bottom center of the dialog is a large, light gray button with the text "Create Prize" in bold black font.

Simple, huh?

## Tournament Dashboard Form

This one's tricky. Here is my take on what it might look like:



Dashboard

### Tournament Dashboard

#### Load Existing Tournament

Not a lot going on here. This would work, but I'm not excited about it. I might draw this four or five times and tear it up each time. I might try to mess around with the Program.cs file and start up different forms based upon if there is a tournament existing or not and try to do everything through that. The more I thought about that option, the less I liked it. In times like this, where I really don't like my options, I'll keep it but I'll keep that dissatisfaction in my mind. If I have time, I'll sleep on it and I'll kick some ideas around in the shower. Sounds weird but things like this work. Also, a walk outside where you try not to think of anything can also help. Let your mind work on the issue in the background. Just don't fill your brain with something else mentally stimulating while you wait.

In this case, I haven't found an idea I like better. I'll keep this form but I'll mark it up in my documents as something that still needs work. That way I won't just blindly create the form like I will with the others.

And that is it. I've reviewed my data design and I don't see any other items that I missed. Does that mean I've covered everything the UI needs to do? Probably not. The great thing is that I don't have to stress out about it. Try to



get everything done this step but don't panic if figure out later you didn't do so. Just return to this step, design the form, and then make sure that change doesn't impact other parts of your plan.

---

## Planning Step 5: Logic Planning

There are two parts to this step. First, there is the global part. That's what we are going to discuss here. Then, there is the ultra-specific and focused part. That happens when we figure out what the code should do for a particular piece. We are going to do the logic planning for those latter situations inline with our coding. That is perfectly fine. Just make sure that you are planning, not just writing code and then trying to fit it together in some sort of Frankensteinian way. Keep looking back at the big picture and how your code bit will fit in the overall whole.

OK, so what is the global logic we need to look at? The way I approach it is I look for what particular buttons and other action points need to do. I don't try to figure out exactly where the code should go in this step. That happens when I specifically plan that bit.

Let's start by looking at the forms again and pointing out what will happen logic-wise for each form.

## Create Tournament Form

The screenshot shows a window titled "Tournament Creator" with a "Create Tournament" form. The form is divided into several sections:

- Tournament Name:** A text input field.
- Entry Fee:** A text input field.
- Select Team:** A dropdown menu with a "create new" link next to it.
- Add Team:** A button below the "Select Team" dropdown.
- Create Prize:** A button below the "Add Team" button.
- Teams / Players:** A section containing a list box labeled "tournamentPlayersListBox" and a "Delete Selected" button.
- Prizes:** A section containing a list box labeled "prizesListBox" and a "Delete Selected" button.
- Create Tournament:** A large button at the bottom center of the form.

The first thing that pops out to me is the “create new” link button. That is going to need to open the Create Team form. When the user clicks the “Create Team” button on that form, we are going to need to get the data back to this form along with the data. Then, we should add that newly-created team to the “Teams / Players” box.

Below that is the Add Team button. That button should take the selected team from the dropdown and add it to the “Teams / Players” box. It should also hide it from the Select Team dropdown.

The Create Prize button should open the Create Prize form. When the user creates the prize on that form, the resulting object should be passed back and put into the Prizes box.

The Delete Selected buttons should delete the selected items from their respective boxes when they are clicked.

Now comes the big one. What happens when we click Create Tournament? We should validate and create the tournament object, save the tournament to the database/text file, and close the form. It should also configure the tournament. That means assigning teams to the schedule and figuring out how big the tournament is (how many rounds).

*At this point, I've got a bit of logic I want to get down. This isn't a place for code but pseudo-code is fine or just written out logic. It makes sense to record your ideas this way.*

The logic for figuring out the number of rounds for a tournament should be something like this:

*While  $i \leq \text{totalTeams}$  ( $i * 2$ ; count++)*

In case my pseudo-code isn't clear enough, let me explain. We are going to keep multiplying a variable by two until the resulting value is greater than or equal the number of teams we have. So, if we have 10 teams, we will do  $1 * 2 * 2 * 2 * 2$  to get 16. At that point, 16 is greater than 10.

The logic then is to add in "byes" or dummy teams to round out to our number (16 in this case). The people that play against a bye in the first round are marked as winners automatically. So, in our example, we would have 6 byes added. That means only two teams are actually playing each other in the first round. That is ok. In the NCAA tournament, they actually allow 68 teams in but there are four games to kick off the tournament to eliminate four teams. The other four get spots and the tournament continues in the "next" round. They call it round one but there was a pre-round before the first round. You could consider every other team as having a bye for that pre-round.

That takes care of the basic logic for figuring out the rounds and for adding the byes. Now, we should consider how to match up the teams. We are going to need to take the list of teams, randomize it (without the byes added in), and then we are going to need to assign the users to games. We will need to add in our byes until we run out when making matchups. For instance, in the case of our 10 teams for the round of 16 "teams", we would assign the teams like this (team numbers represent the order created from randomizing the team list; the two columns represent the matchup):

<i>Team 1</i>	<i>Bye</i>
<i>Team 2</i>	<i>Bye</i>
<i>Team 3</i>	<i>Bye</i>
<i>Team 4</i>	<i>Bye</i>
<i>Team 5</i>	<i>Bye</i>
<i>Team 6</i>	<i>Bye</i>
<i>Team 7</i>	<i>Team 8</i>
<i>Team 9</i>	<i>Team 10</i>

We should never have a bye against a bye because the result of every round needs to be a team. This way, we won't have a bye carry over to round two. Ever.

## Create Team Form

**Create Team**

**Team Name**

**Select Team Member**

**Add Member**

**Add New Member**

**First Name**

**Last Name**

**Email**

**Cellphone**

**Create Member**

**tournamentPlayersListBox**

**Create Team**

The Add Member button should take the selected team member and add it to the listbox on the right. That action should also remove that team member from the dropdown as an option.

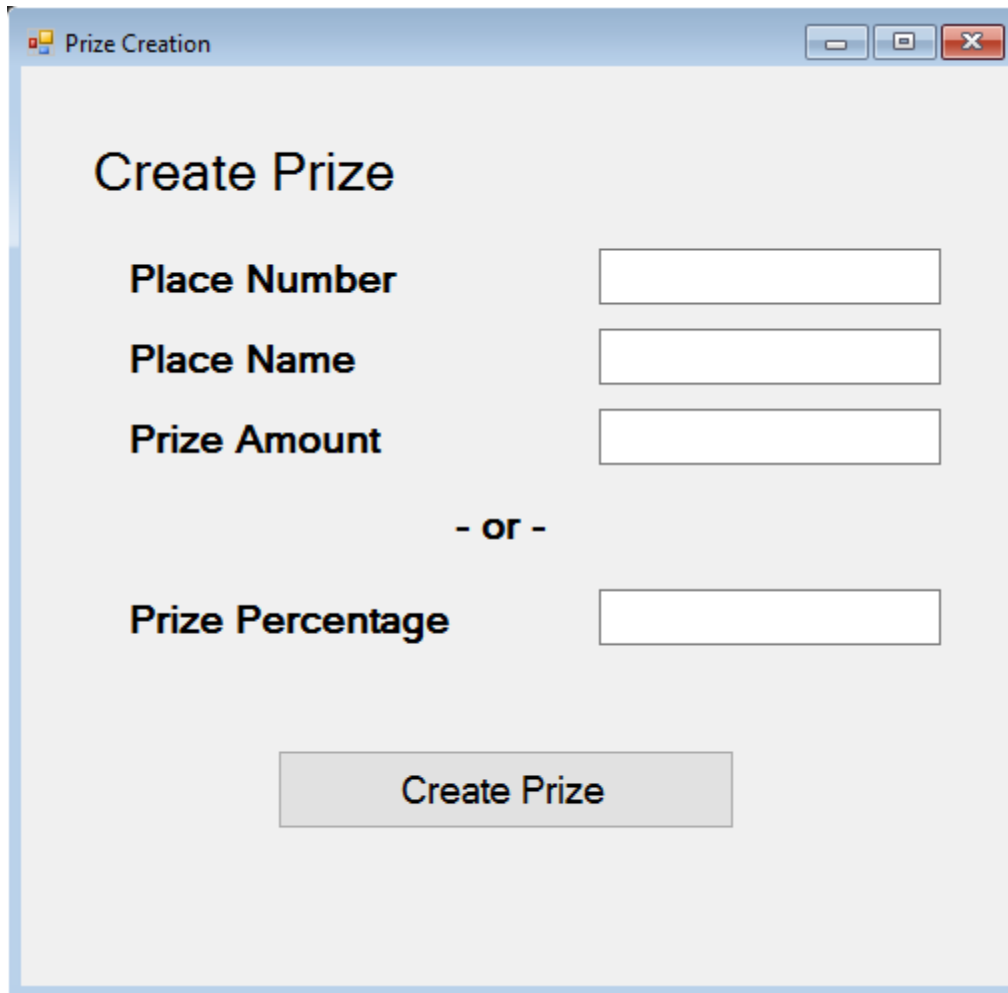
The Create Member button should take the values of the four member fields, validate them, and then add the Person object to the listbox. It should also clear out the values for those text fields and set the cursor back to the First Name field.

The Create Team button should validate that the object can be correctly created and then it should create the object and send it back to its caller before closing itself.



Note that when looking at this form, I don't see a way to remove a person from the list. I'd draw that on my form if I were doing this myself. Instead, I'll just make a note to add that and the associated logic. The idea for this button comes from doing something similar on another form. Doing things the same way every time not only helps the user know what to expect, it also helps us develop better.

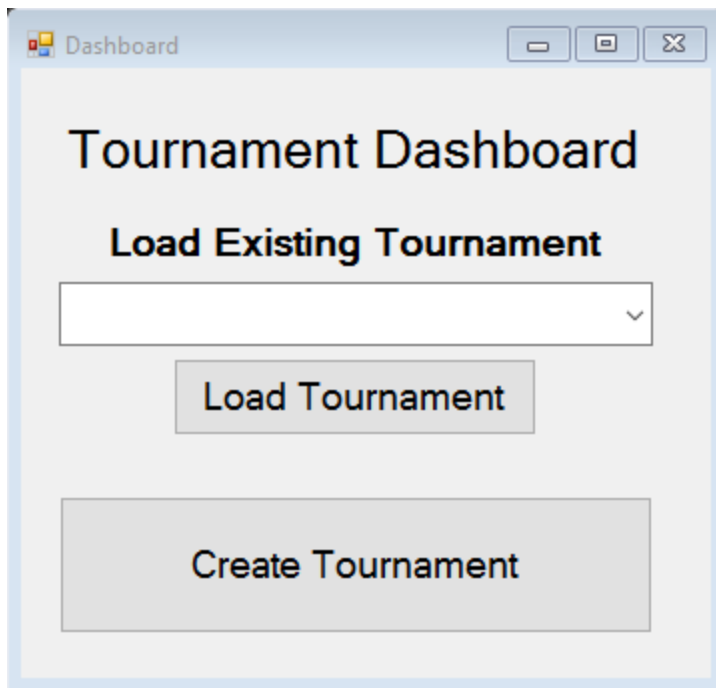
### Create Prize Form



The screenshot shows a Windows-style window titled "Prize Creation". Inside the window, the title "Create Prize" is at the top left. Below it, there are four input fields arranged vertically. The first three are labeled "Place Number", "Place Name", and "Prize Amount". The fourth is labeled "Prize Percentage". Between the "Prize Amount" and "Prize Percentage" fields, there is a text label "- or -". At the bottom center of the form is a button labeled "Create Prize".

This one is rather simple. When the Create Prize button is clicked, we will validate the form, create the Person object and send it back to the caller. We will then close the form.

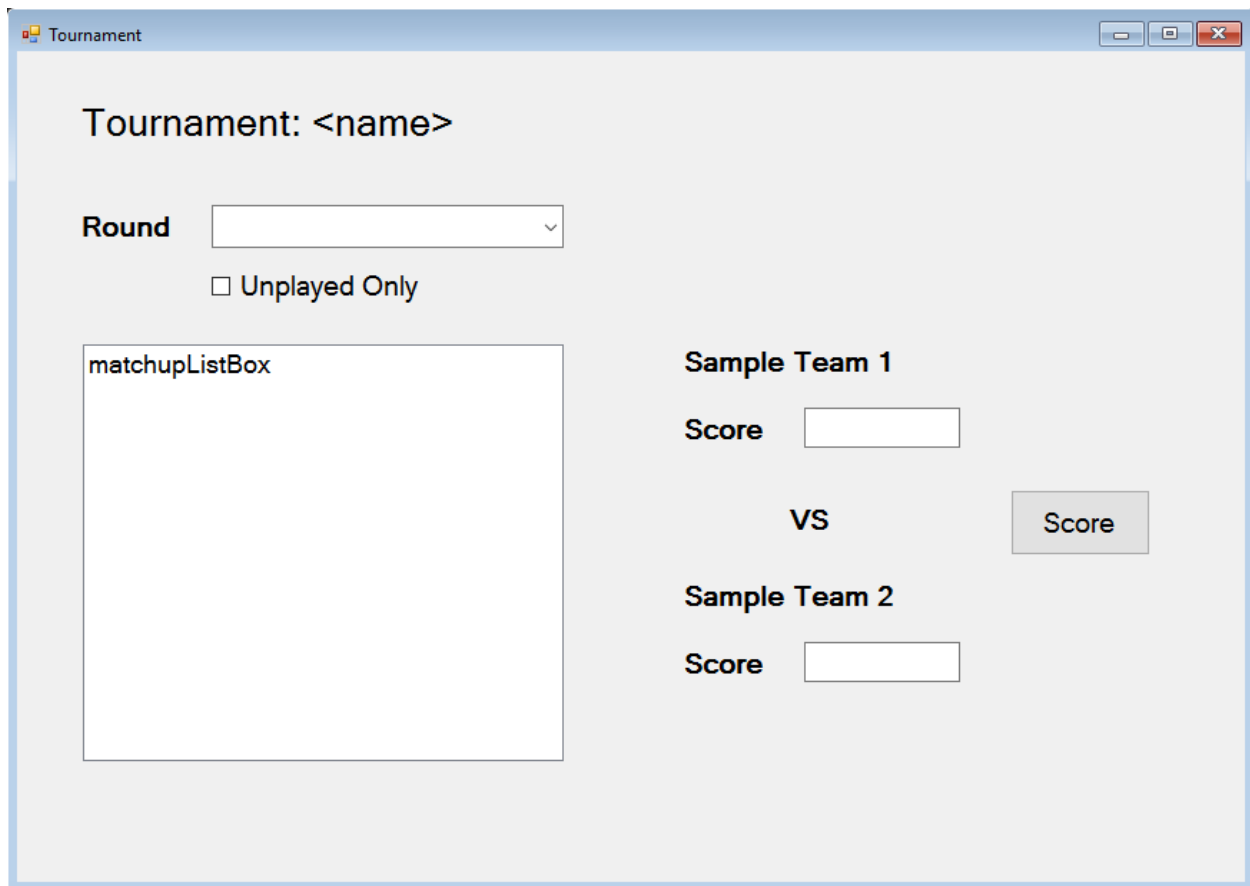
## Tournament Dashboard Form

A screenshot of a Windows Forms application window titled "Dashboard". The window has a light blue title bar with standard minimize, maximize, and close buttons. The main content area has a light gray background. At the top, the text "Tournament Dashboard" is displayed in a large, bold, black font. Below this, the text "Load Existing Tournament" is displayed in a smaller, bold, black font. Underneath, there is a white rectangular dropdown menu with a small downward arrow on the right side. Below the dropdown menu is a rectangular button with a gray background and the text "Load Tournament" in black. At the bottom of the form is a larger rectangular button with a gray background and the text "Create Tournament" in black.

Ah, the form I hate. OK, here's the deal. Windows Forms needs the opening form to stay open for the life of the app. This form can fill that need. When the user clicks the Load Tournament button, we will open the tournament form. When the user hits the Create Tournament button, we will open the Create Tournament form. Once the form is created, we will capture which new form was added and add it to the dropdown.

Note that throughout all of this, we will not close this form. It will stay open.

## Tournament Viewer Form



The screenshot shows a Windows-style application window titled "Tournament". Inside the window, the header text is "Tournament: <name>". Below this, there is a "Round" label followed by a dropdown menu. Under the dropdown is a checkbox labeled "Unplayed Only". To the left of the right-hand section is a large empty rectangular box labeled "matchupListBox". On the right side, there are two team sections. The first section is for "Sample Team 1", with a "Score" label and an input field. Below this is the text "VS" and a grey button labeled "Score". The second section is for "Sample Team 2", also with a "Score" label and an input field.

I saved this one for last because it has some of the fun, complex stuff behind it. In actual practice, where the code is that will be run based on the actions on this form doesn't matter. We just need to discuss it here because here is where we kick it off.

We will update the header label with the name of the tournament at runtime.

The Round dropdown will be a text list that has an entry for each round. We will need to generate this list based upon the passed in tournament. If the tournament has four rounds like in our example from the Create Tournament form, this dropdown would have the following entries:

- Round 1 (2)
- Round 2 (4)
- Round 3 (2)
- Round 4 (1)

The number in parenthesis will be the number of games yet to be played in that round. I set up the example as if we are just starting out. Even though Round 1 has 10 teams in it plus 6 byes, there will only be two games that are actually played. Thus, the two.

Based on the selection in the dropdown, the listbox will populate with the matchups. If the checkbox is checked, only the unplayed matchups will be shown.

When you select a matchup in the listbox, the team names will be populated on the right, as well as the scores (if any). If the user hits the Score button, the values will be put into the selected Matchup object. The winner will also be assigned based upon the score (we may need a setting for what is the winning condition: high score like in basketball or low score like in golf).

Marking the winner should trigger a few actions. First, the winning team should be assigned to their next round game. After that, an email should be sent to the winner and the loser. Each email should be custom to the outcome for that user. It should tell the user the score of the game and what happens next for them. Next, an email should be sent if this is the last game of the round. The email should recap the matches and the outcomes. Other emails should also be triggered, one per matchup, that tells the users who they will be playing next. Finally, if this was the last game of the tournament and it is now scored, an email should go out to all participants letting them know that the tournament is over and who won. Prize amounts and recipients should also be announced in that email.

Prizes should be calculated and assigned to the user. The administrator of the tournament (we don't seem to have this yet in our data or in our UI design) should also get an email to let them know who to pay what amount and what the remaining balance of the tournament is (if any).

If the tournament is complete, an indication of this should be placed on the form somewhere just to be sure the user does not attempt to change the values after the fact unless a mistake was made.

All changes to an existing, marked matchup will be marked as locked once the first game of the next round is complete. If that was not the case, a user

could go back and make a change that would change matchups. That would cause issues.

### Other Pieces

There are always going to be areas that you don't fully cover in planning. Don't stress out about getting everything perfect. Plan for what you can and then, when you have executed your plan, come back to your plan to work out the missing pieces.

In our case, we haven't really covered a few areas. For example, data access. We are going to need to figure out how to load and save data for all of these bits. We will figure that out when we get there.

Also, note that there are a couple areas that I pointed out that were missing from our design (high or low score wins option and tournament administrator information). Those two are fairly simple items so I'll leave the plan as it is. However, on my paper that I built this on, I've added in those items in both the UI and the data design.

*There are always going to be areas that you don't fully cover in planning. Don't stress out about getting everything perfect.*

---

## Coding Step 1: Class Library Creation

I love this step. Partly because I get to actually start coding and partly because it is just so very simple. What you do is take your class library model that we did in Planning Step 3 and translate that into classes with properties. That's the majority of the work.

While you are typing out your classes, think again about what each property holds. Maybe you want to tweak the name a bit or adjust the type. Feel free to do so. In the video, I ended up changing my class names to have "Model" at the end (so PersonModel instead of Person for the class name).

Once you are done creating your classes, go back through and add XML comments for each property and class. These will not only help clarify what each property and class does for the next developer, they will also help you



understand what each class does. Being forced to explain what a certain property represents really makes you think about its purpose. If you cannot easily explain it, how can a future developer understand it?

## Key Concepts

### Auto Properties

A property is the recommended way to expose data outside of a class (think public variable). The benefits include the fact that you can limit who has access to a property, you can modify the data it returns without modifying the data it stores, and many more. The only problem with a property is how much space it takes up. Here is an example of a property:

```
private string _firstName;

public string FirstName
{
    get
    {
        return _firstName;
    }
    set
    {
        _firstName = value;
    }
}
```

That's quite a bit of code to create a property that holds a first name. Note, however, that you can modify the get and the set values to modify or restrict the data coming in or going out. That's nice but usually we just need a "public variable": something that is accessible outside of our class and that holds data. That is where an auto-property comes in.

An auto-property is a lean version of a full property. Here is essentially the same FirstName property as above but this time in abbreviated format:

```
public string FirstName { get; set; }
```

Not bad, huh? In just one line, we have a property that contains all of the benefits of a property without all of the extra lines of code that we don't need. Behind the scenes, .NET will translate this into a full property for us. We don't care about that. What we care about is that it is easy to create.

To make it even easier, Visual Studio has a snippet that we can use to shortcut the creation process. Type "prop" and tab twice. This will create an auto-property snippet that we can use. We just need to fill in the type and the name (and we can tab between those two options). Once we are done, we hit enter and the cursor goes to the end of the line.

### XML Comments

An XML comment is a specially-formatted comment that is then used by Visual Studio to create Intellisense hints for the user. Especially in a class library, this style of comments should be used whenever possible.

To create an XML comment, simply put your cursor on the line above the declaration of a property, method, or class. Then, hit the comment slash (/) three times. This will create the structure that you then fill in. Here is an example of our EntryFee property:

```
/// <summary>
/// The amount of money each team needs
/// to put up to enter.
/// </summary>
public decimal EntryFee { get; set; }
```

---

## Coding Step 2: Form Building

Quick confession here: I am terrible at user interface design. I can tell you when something looks good but if you ask me to make something from scratch, I'm going to struggle. I use all the help I can get. That is why, in this section I'm going to cover specifically how I come up with designs and apply them to my user interface. Dragging and dropping items onto the form isn't something I need to tell you how to do.

If you are like me, your first inclination might be to get a kit that will do the design for you. Companies like Telerik have amazing-looking packages that take even the most boring form and seemingly transform it into an interface worthy of Star Trek. The problem is, those packages aren't free. And even if you are fortunate enough to be able to get one for a reasonable price, it still takes work to make your forms look as good as the examples.

Here is my philosophy on form design – I make it look as good as I can without stressing out about it. I spend more of my time making sure the application does the job well. My users seem to be fine with that. I've never won an award for how my applications look but I have won awards for how my applications worked.

So how do I make my application look good without being a designer? I follow these steps:

*I make it look as good as I can without stressing about it. I spend more of my time making sure the application does the job well.*

### Step 1 – Change the defaults

I start by changing the default background, the default font, and the default icon of the form I am working on. Windows Forms are especially bad with their defaults. I usually start with a white background although you could flip the script and go for a dark grey background for your application. As for the font, either use one of the alternate fonts provided by Windows (safer for when you are distributing your application to other computers) or you can go to Google to get a better font for free (make sure you check the licensing to be sure the font is free for your use).

### Step 2 – Pick your color scheme

It sounds strange but I typically do this after picking my background color. I find that the best background colors are either almost dark, almost white, or white. What I choose will have to do with what they application will be used for and what I'm in the mood for. From this, I'll then figure out what colors are best suited to complimenting my design.

For this, I typically go to a color scheme website or two in order to find out what color pairings people are using. I might even build my own color package.

### Step 3 – Determine how to use the colors

Just because you have a color scheme doesn't mean you need to put everything in color. I find that black and white are a great foundation to just accent. Figure out where you are going to add a pop of color. Keep it simple.

### Step 4 – Create a demo

When you are building out a design, create a demo of how the different pieces would look on a form. Take just a few minutes, drag each type of control or item onto your form that you might use for the project and just look at it. Does everything go together? Does it seem to look right? Keep tweaking until you are satisfied.

### Step 5 – Document your design

Once you are happy with your design tweaks, document them. That way, when you create a new form, you know exactly what you need to do to make the current form look like all of the rest of the forms.

For example, here is the design documentation for the tournament tracker:

- Form
  - BackColor: White
  - Icon: Custom (clipboard)
  - Font: Segoe UI 16pt
- Label
  - ForeColor: 51, 153, 255
  - Font: 20pt
- ListBox
  - BorderStyle: FixedSingle

- Button
  - FlatStyle: Flat
  - FlatAppearance:
    - BorderColor: Silver
    - MouseDown: 102, 102, 102
    - MouseOver: 242, 242, 242
  - Font: SemiBold
  - ForeColor: 51, 153, 255
- Checkbox
  - FlatStyle: Flat
  - ForeColor: 51, 153, 255
  - Font: 20pt
- Header
  - Font: Light 28pt

The end result of playing around with these settings was a design I could be happy with. Is it perfect? Nope. Does it look nice? Yep.

### Design Tools

There are a lot of tools that I use to make my designs. Here are a few of the big ones (all of these are free to use):

- [Syncfusion Metro Studio](#) – creates great icons
- [ColourLovers.com](#) – find great color palettes that are user-ranked
- [Fonts.Google.com](#) – pick great fonts to make your application look great
- [Color.Adobe.com](#) – find matching colors, complementing colors, shades of a color and more

---

## Coding Step 3 – Build the Database

OK, so building the database might not be coding in C#, but it is part of the process we need to go through. If you have never done a database before, it can be a bit tricky. Never fear though, it isn't really that hard.

Some tutorials will take you through the Visual Studio database section to add and update the database but I avoid that for two reasons. First, Visual Studio adds its own confusion to the mix here. I've seen things go wrong because Visual Studio was used when it would have been much easier and less error-prone if you used the full SQL Server Management Studio (SSMS). Second, there will be times that you need to create a database not yet associated with a C# project. In that case, it makes no sense to open up Visual Studio to manage a SQL database.

If you don't have SQL Server on your computer, Microsoft has made it really easy to try it out. As a developer, you can get SQL Server Developer Edition absolutely free. It has all of the Enterprise features, it just can't be used in production. For development, though, you can go crazy with it. To get it, go to this link: <https://www.microsoft.com/en-us/sql-server/sql-server-editions-developers> or search the web for "SQL Server Developer Edition". You will need to sign up for the Visual Studio Dev Essentials to download it, but it is free and will give you access to other free software as well.

## Key Concepts

### Tables

Creating tables is both simple and hard. It is simple because the creation form is so easy to use. It is hard because you need to pay attention to what you are doing and create the right fields for any given table. For instance, here is our table for the Tournaments data:

OFFICETOWER\SQL2...dbo.Tournaments			
	Column Name	Data Type	Allow Nulls
🔑	id	int	<input type="checkbox"/>
	TournamentName	nvarchar(200)	<input type="checkbox"/>
	EntryFee	money	<input type="checkbox"/>
	Active	bit	<input type="checkbox"/>
			<input type="checkbox"/>

Note that we do not have information about the teams that are in the tournament or the prizes to be awarded. On the other hand, we do have a field called id that holds our unique, auto-generated number to identify each row in the table. We also carefully chose what Data Type to select for each column based upon the data that was going into that column.

## Stored Procedures

How to get data out of a SQL database is one of those silly territorial battles that rages on the Internet. Don't get sucked into it. There is more than one way to transfer data to and from your application and most ways can be right depending on the situation. Let's digress for a minute to discuss these different methods so we can better understand why I chose to go the route I did.

## Entity Framework

The Entity Framework route is a somewhat popular one because it is Microsoft's method for talking to SQL. The high-level overview is that you connect your Visual Studio to SQL and then it builds out the classes you need. You can then just tell EF to update your class or get a list of class or delete a class and magic happens and it just works. That's an oversimplification of the process but it is enough for this explanation. The issues I see with this route are that it can be very complicated to debug when something goes wrong and it is slow. Like "noticeable to the end-user"-slow.

## Direct ADO

This is where you do direct calls to SQL on your own. It isn't overly complicated to do and it is as fast as it gets. The downsides are that it can be more susceptible to SQL injection attacks and it takes more work on the coding end to get the data into a usable form (it comes back in DataTables, which need to be manually converted over to a List<model>).

## Lightweight ORMs

In my video [C# Data Access: SQL Databases](#) on YouTube, I used a lightweight ORM (Object Relational Mapping) called Dapper. The reason I did was because Dapper sits in between Entity Framework and ADO. It has the speed of ADO but a lot of the benefits of EF. In short, it is the ideal solution (in my mind) for most situations. One of the things I love about it is that it handles stored procedures very well. That, in my mind, is crucial and it is something EF does poorly.

OK, so that's my opinion on ways to get data out of SQL, but the question is why does it matter what I use when I design my SQL database? The reason

it matters is because Entity Framework does not handle stored procedures very well at all. It likes to write custom SQL for its calls. If you were going to use EF in your C# application, you wouldn't spend a lot of time building stored procedures. And why do I spend so much time building stored procedures?

Stored procedures are powerful, pre-compiled code in your SQL database that allow you to get access to exactly what you want as fast as possible. They also limit what data can be

passed in and what data can be returned. This is especially good when your database has sensitive data in it.

Furthermore, you can lock your database down to the point that the

application user can only run stored procedures. No viewing tables, no inserting data directly, and even no connecting to the database. With stored procedures as your only data access method, you can limit your databases exposure.

*With stored procedures...,  
you can limit your  
databases exposure.*

That is why stored procedures and Dapper are my defaults when talking about setting up SQL access from an application. I may choose something different based upon the situation, but unless I find a specific reason to change, I'll stick with my defaults.

---

## Coding Step 4 – Wiring Up the Forms

There is a LOT of work in the code-behind of forms. This is where the real "action" is in your system. As you saw from the first three coding steps, you pretty much just follow your planning guide to create the required sections. That makes for very quick progress in your application (which is always satisfying to see). This step, however, is less direct. Sure, we did logic design during the planning phase but nothing fully prepares you for creating the logic except...creating the logic.



## Key Concepts

There are a number of key concepts that we will cover in the training. This guide isn't meant to be a complete duplicate of the videos. Instead, I'll point out the concepts that are important and where you can get more information about them (if necessary).

## Linq

This one comes up a number of times. Whenever you have a list of something and you need to filter, sort, or manipulate it, Linq is probably going to be a good option. We do not use the full Linq potential (basically doing a SQL-like query on your list with a from, where, select, etc.) Instead, we use the methods that allow us to do one specific action (where, orderby, etc.)

For example, this is how we find one record in a List:

```
model.Prizes.Where(x => x.PlaceNumber == 1).FirstOrDefault()
```

Note that by default, a Where extension like we have here returns an `IEnumerable<T>` (a type of list). If we want to have a list returned, we would need to add `(.ToList())` at the end. In this case, though, we wanted just the first record or, if no record existed that matched our criteria, we wanted the default value.

In this course, we also use other list manipulators like `OrderBy`, `Any`, `All`, and a few others. If you want more information about how Linq works, you can check out my video on the topic: [Linq 101](#).

## Dapper

Getting data out of a SQL database can seem intimidating. If you look at the code for straight ADO, it might look intimidating as well. However, there is a much easier way.

The Dapper library, brought to you by the folks at StackOverflow (who use it in their systems), is a lightweight ORM. That means that it talks to the database and translates the data to and from objects (class instances). If you want to know more about Dapper, you can watch my [C# Data Access: SQL Databases](#) video.

The basics we use in this video include how to connect to the database:

```
using (IDbConnection connection = new System.Data.SqlClient.
SqlConnection(GlobalConfig.CnnString(db)))
{
}
```

How to query the database for information:

```
output = connection.Query<PersonModel>("dbo.spPeople_GetAll"
).ToList();
```

How to write information to the database:

```
var p = new DynamicParameters();
p.Add("@FirstName", model.FirstName);
p.Add("@LastName", model.LastName);
p.Add("@EmailAddress", model.EmailAddress);
p.Add("@CellphoneNumber", model.CellphoneNumber);
p.Add("@id", 0, dbType: DbType.Int32, direction: ParameterDi
rection.Output);

connection.Execute("dbo.spPeople_Insert", p, commandType: Co
mmandType.StoredProcedure);
```

And how to capture variables that have been marked as output variables:

```
model.Id = p.Get<int>("@id");
```

That isn't a lot of code and yet that is really the only parts we needed to worry about with Dapper. Sure, there is a lot more it can do, but we didn't need to make use of those parts.

### Debugging

We were fortunate in this video series. We got to debug a rather large and complicated bug, as well as a few smaller ones. In general, it is better to write smaller pieces of your application and test them so that you aren't then

faced with too many lines of code to review when you hit a bug. However, this isn't always possible. It is also likely that some of the debugging you will do will be against existing applications. In this case, you might not know any of the code, so debugging skills are especially important there.

One of the things I demonstrated in the video was the use of advanced breakpoints, where it didn't break until after the hit counter reached a certain number. If you would like to know more about how to use the advanced breakpoint features, watch my video on [C# Debugging: Breakpoints](#).

*The key to good debugging...is patience. Keep pulling the thread. Don't try to solve the mystery in one big declaration.*

The key to good debugging, though, is patience. Keep pulling on the thread. Don't try to solve the mystery in one big declaration. Instead, pick up clues one at a time. Get closer with each step until you have finally cornered your culprit. Then, before just changing things right away, plan out the best course of action to not only fix the problem but to also not create further problems.

### Error Handling

The first thing people think of when they hear C# error handling is usually exception handling (the try/catch block). There is a lot more to error handling than that. The first thing you need to do is anticipate your errors. How could things possibly go wrong. If you are including user input, there is a lot that could go wrong. Never trust the user.

So, the first step in error handling is actually error prevention. You do this by screening the input users give you. If a text field should contain a number, don't just convert the string to a number. Use TryParse to check to see if the parse will work:

```
bool placeNumberValidNumber = int.TryParse(placeNumberValue.  
Text, out placeNumber);
```

If there might be a critical error you cannot prevent (missing database, bad lookup, etc.), that's when you employ the try/catch method to attempt something and fail gracefully if the attempt does not work:

```
try
{
    TournamentLogic.UpdateTournamentResults(tournament);
}
catch (Exception ex)
{
    MessageBox.Show($"The application had the following error: { ex.Message }");
    return;
}
```

There are a couple things to note here. First, we are only capturing the generic exception. We could capture multiple different exception types if we wanted. Second, the catch block is a great place to log the exception. You have all of the relevant information. Just store it for later use. Finally, if you do catch an exception, do something about it. Don't just eat the exception. In the above example, we are alerting the user and stopping the method from continuing. You could also throw the exception (use just "throw;" - don't use "throw ex;"). In the end, just do something with it.

## Events

An event can be used to alert any listening code that an action has occurred that they are waiting for. For example, the button class has a click event. When a user clicks the button, that fires the button click event. If anyone is listening, the specified code runs. We use this all the time to perform actions when certain actions occur.

In C#, we have the option of adding our own events. In our project, we will add our own event to let the listeners know when the tournament has been finished. To declare our event, we use the following code in our class:

```
public event EventHandler<DateTime> OnTournamentComplete;
```

That sets up the event that can be subscribed to. Next, we need to fire off the event when something happens in our class that satisfies what the event represents. In this case, when our class determines that our tournament is complete, it should call the following code:

```
OnTournamentComplete?.Invoke(this, DateTime.Now);
```

Here we are checking to make sure the event has a subscriber (otherwise it is an empty event that cannot be called) and then we are calling it. We pass through who called it and, in our case, the `DateTime.Now`. Was that last part necessary? Nope. We just didn't need to use that variable. We could have passed in a new `EventArgs` but instead I decided to show you that you can pass anything through that type.

### Email

Emailing is actually built right into C#. You need to include the following using statement to make it easier to use but you don't even need to bring in a dll to do it:

```
using System.Net.Mail;
```

The actual sending of an email can be as simple as the following two lines:

```
SmtplibClient client = new SmtplibClient();  
client.Send(mail);
```

That creates a new mail client and sends out an email. It is dependent on having information set up in the `app.config/web.config` file. That information includes which mail server to use, the port, and the authentication information. You could hard-wire it into your code, but that wouldn't be wise. Here is a peek at our `app.config` file from the course:

```
<system.net>
  <mailSettings>
    <smtp deliveryMethod="Network">
      <network host="127.0.0.1" userName="Tim" password=""
port="25" enableSsl="false"/>
    </smtp>
  </mailSettings>
</system.net>
```

Note that actual values need to be included for all of these values. For our app, it didn't matter that much because we were using Papercut to capture the emails and display them instead of just sending them. You can get more information about Papercut on GitHub at:

<https://github.com/changemakerstudios/papercut>

The only thing left is to actually populate that "mail" object before we send it. That isn't too complicated either. The big things are creating a from address that has a name, not just an email address and adding our email addresses to the To, Cc, and Bcc properties by adding them, not just assigning them a value. Here is the code we build in the lesson:

```
MailAddress fromMailAddress = new MailAddress(GlobalConfig.AppKeyLookup("senderEmail"), GlobalConfig.AppKeyLookup("senderDisplayName"));

MailMessage mail = new MailMessage();
foreach (string email in to)
{
    mail.To.Add(email);
}
foreach (string email in bcc)
{
    mail.Bcc.Add(email);
}
mail.From = fromMailAddress;
mail.Subject = subject;
mail.Body = body;
mail.IsBodyHtml = true;
```

When you put it all together, you get an application that can send out emails when necessary without a lot of code. Wrap that in a class and you have easy-to-use code that can be called with just a simple method call to send out an email.

## SMS

Texting, our bonus lesson, is actually really easy to implement. It isn't free, since we actually have to send out messages over the cell towers. However, it is extremely cheap and we can build our application for free.

In the lesson, we use a service called Twilio to send out our messages. Once we add the NuGet package for Twilio, we simply set up our client by passing in the Sid and authToken we get from Twilio on our dashboard like so:

```
string accountId = "";
string authToken = "";
TwilioClient.Init(accountSid, authToken);
```

In our application, we pulled those values from the app.config file. We then send the message using the following code:

```
var message = MessageResource.Create(
    to: new PhoneNumber(to),
    from: new PhoneNumber(fromPhoneNumber),
    body: textMessage
);
```

You will need to add the following three using statements to get this to work:

```
using Twilio;
using Twilio.Rest.Api.V2010.Account;
using Twilio.Types;
```

That's all it takes.

# WRAPPING UP

Man, there is a lot to cover. This companion document barely seems to scratch the surface of what we covered in this course. In some ways, you could look at C# and be intimidated. But hopefully as we have gone through this course you've noticed that it isn't really about complex code and tons of syntax. Instead, it is about taking the "simple" things and using them in new and interesting ways.

I often liken it to LEGOs. You may only have a few different types of bricks but with a little imagination, you can combine them in really cool ways. C# syntax is like the different LEGO pieces. Our logic is how we put those pieces together. Are the fun-looking pieces neat to use? Absolutely! Are they necessary to create something powerful? Nope.

Well, that's it. Hopefully you have found value watching and following along as we built an entire application from start to finish. Now take what you have learned and be awesome!