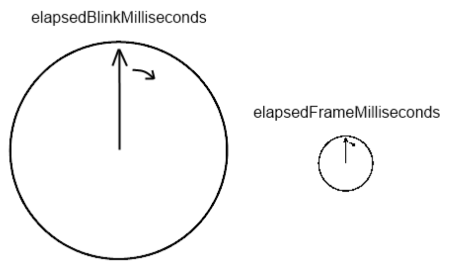
MonoGame Increment 5

This increment finishes the game. When the correct tile is clicked, it blinks for a while, then a new game is started. The game also includes sound effects.

This increment has a lot more steps than the previous increments, in part because there are no programming assignments due in Week 7.

39. Next we want to make it so that when the correct tile is clicked, that tile starts blinking. Add all the blinking number .png files to the GameProjectContent project (Chapter 5, Week 2)
40. Declare a `Texture2D` field in the `NumberTile` class to hold the blinking tile texture (Chapter 12, Week 6 for fields)
41. In the `NumberTile` `LoadContent` method, add code to load the blinking tile texture into the field from the previous step. Note that you can use the string for the tile number to help build the name of the texture (all the blinking textures start with “blinking” followed by the number string) (Chapter 6, Week 3)
42. Now we have two textures we need to use for drawing: one for normal and shrinking tiles and the other for blinking tiles. That means we’ll need to keep track of the current texture we’re using so we draw the appropriate one. Declare a `Texture2D` field in the `NumberTile` class to hold the current texture we’ll be drawing (Chapter 12, Week 6)
43. In the `NumberTile` `LoadContent` method, add code to set the current texture to the non-blinking texture (Chapter 3, Week 1)
44. In the `NumberTile` `Draw` method, change the code to draw the current texture (Chapter 5, Week 2)
45. In the `NumberTile` `Update` method, add two lines of code just after the line where you set the tile is blinking field to `true`. Your first new line of code should set the current texture to the blinking texture. Your second new line of code should set the `X` property for your source rectangle to 0 (Chapter 5, Week 2)
46. In the `NumberTile` `Update` method, change the original if clause (that checked if the tile is shrinking) for the if statement from Step 37 to an else if instead. Now add an if clause before that else if clause to check if the tile is blinking. You'll now have an if statement with an if clause that checks if the tile is blinking, an else if clause that checks if the tile is shrinking, and an else clause that highlights/unhighlights the tile and lets the player click the mouse button on the tile. Basically, we’re doing different update processing on the tile based on the current tile state (blinking, shrinking, or normal). (Chapter 7, Week 3)
47. You don’t have to do anything in this step, it’s just explaining how the next two steps will work. For our blinking animation, we need to have two timers. We’ll use one timer, `elapsedBlinkMilliseconds`, to tell us when the blinking animation should stop. We’ll use the other (shorter) timer, `elapsedFrameMilliseconds`, to tell us when it’s time to change frames in the animation. See the picture below.



48. Inside the if clause you added in Step 46, add code to determine when to make the tile invisible (after it’s done blinking). Note that I've already included `elapsedBlinkMilliseconds` and `TOTAL_BLINK_MILLISECONDS` fields for you to use for this. You’ll need to update the elapsed blink milliseconds field by adding `gameTime.ElapsedGameTime.Milliseconds` to it before deciding whether or not to stop the blinking animation (Chapter 7, Week 3 for timers)
49. After the code you added in the previous step (but still inside the if clause you added in Step 48), add code to update the animation frame as appropriate. I've already included `elapsedFrameMilliseconds` and `TOTAL_FRAME_MILLISECONDS` fields for you to use for this. Add

`gameTime.ElapsedGameTime.Millisconds` to the elapsed frame milliseconds field, then use the new value to decide when to change to a new animation frame. To change frames in the blinking animation, you need to move the source rectangle left or right (depending on where the source rectangle currently is) after each frame ends to change the texels being displayed (Chapter 7, Week 3)

50. You don't have to do anything for this step, it's just a check on your progress. At this point, clicking the 8 tile should make it blink between yellow and green for a few seconds, then the tile should disappear. Clicking on all the other tiles should make them shrink down until they're no longer visible
51. You don't have to do anything for this step either; it's just explaining what you'll be doing next. Now we're faced with a difficult problem. When the correct number is guessed, we're supposed to reset the board and start a new game. We allocated the responsibility of determining whether or not the correct number was guessed to each individual number tile (this was the correct design decision), but how does the number tile let the game know that the correct number was guessed? The `NumberTile` class doesn't know anything about the `Game1` or `NumberBoard` classes, so how can it communicate this information to them? There's a very slick way to do this in C# using something called *event handlers*, but that's not introductory-level material. Instead, I've made the `NumberBoard` `Update` method and the `NumberTile` `Update` method return a `bool` instead of the typical `void`. A `false` will mean the correct number hasn't been guessed yet and a `true` will mean it has. This isn't the best general solution (event handlers are), but it's the best solution using what we know at this point
52. In the `NumberBoard` `Update` method, we need to return `true` if one of the `NumberTile` `Update` calls returns `true` (indicating that the correct number was guessed), otherwise we need to return `false`. Change the call to the `NumberTile` `Update` method to put the result into a `bool` variable; the syntax is similar to when you called the `Deck` `TakeTopCard` method and put the returned value into a `Card` variable (in Lab 4). If this `bool` variable is `true` after the call to the method, immediately return `true` from the `NumberBoard` `Update` method. Doing it this way ensures the method immediately returns `true` if the correct number is guessed and also lets us skip the updates for the rest of the number tiles. There are more complicated ways to do this to make sure all the tiles get updated (to make sure we don't skip updating shrinking animations for 1/60 of a second, for example, for tiles that are currently shrinking), but this approach works fine here (Chapter 13, Week 6 for method return)
53. You don't have to do anything for this step, but you might be wondering if the `NumberBoard` `Update` method returns `false` correctly. The answer is yes, because the only way we get to the last line of code in the method is if none of the `NumberTile` `Update` method calls return `true`, which means the correct number wasn't guessed on this update. Slick, huh?
54. In the `Game1` `Update` method, change the call to the `NumberBoard` `Update` method to declare a variable to tell whether or not the correct number has been guessed and put the returned value from the method call into that variable (Chapter 4, Week 2)
55. Cut the code you added to the `Game1` `LoadContent` method to calculate the board size and actually create the board and paste that code into the `StartGame` method I provided at the end of the `Game1` class. Call the `StartGame` method from the `Game1` `LoadContent` method where you used to have that code. You'll see why we need to do this soon (Chapter 4, Week 2)
56. In the `Game1` `Update` method, add an if statement right after the call to the `NumberBoard` `Update` method to check if the correct number has been guessed. If it has, call the `StartGame` method (Chapter 7, Week 3)
57. At this point, we never actually end up restarting a game. That's because the `NumberTile` `Update` method always returns `false`. In the code in that method that sets visible for a blinking tile to `false` (see step 48), change the code that sets visible to `false` to return `true`; instead. We don't want to return `true` from this method right when the user picks the correct number because then they won't get to see any blinking, so we return `true` when the blinking is done (Chapter 13, Week 6 for return)
58. At this point, our game always has 8 as the correct number. Declare a `Random` field in the `Game1` class and use `new Random()` to instantiate the object for the field (Chapter 12, Week 6)
59. In the `StartGame` method, declare a variable to hold the correct number for the current game and set it to a random number (from 1 to 9 inclusive) you generate using the field from the previous step. Pass that number as the correct number argument when you call the `NumberBoard` constructor in the `StartGame` method (Chapter 4, Week 2)

60. In the `NumberBoard` constructor, change the call to the `NumberTile` constructor to pass the `correctNumber` parameter as the correct number argument (instead of 8 or whatever hard-coded number you used) (Chapter 4, Week 2)

    Because MonoGame doesn't support using XACT, this is where MonoGame users start following different steps from XNA users. You should find the 12.3 XNA Audio Without XACT lecture helpful as you complete the following steps.

61. Holy smokes! The game is – finally – almost done. It should all be working properly for you by this point, with the sound effects the only thing that are missing. Download the RequiredProjectWavFiles.zip file from the Required Assessment Materials course page. Add the .wav files to your project as described in the Adding Content to a MonoGame Project link on the MonoDevelop Resources course page. Note that it's fine that these are wav files, not xnb files. (Week 7)
62. You don't have to do anything for this step, I'm just letting you know that we'll be removing the `SoundBank` parameters from the `NumberBoard` and `NumberTile` constructors because we'll be using `SoundEffect` objects to play our sound effects, not a `SoundBank` object. (Week 7)
63. In the `NumberBoard` constructor header, remove the `SoundBank soundBank` parameter (including the comma before it). Remove the last comment line in the documentation comment for the constructor.
64. Compile the project. You'll see we get a compilation error when we try to call the `NumberBoard` constructor in the `Game1` `StartGame` method because we're now trying to provide one too many arguments in the call. Remove the final argument (and the comma before it) from the constructor call; I've been using `null` for the final argument. The code should now compile and run without any errors.
65. In the `NumberTile` constructor header, remove the `SoundBank soundBank` parameter (including the comma before it). Remove the last comment line in the documentation comment for the constructor.
66. Compile the project. You'll see we get a compilation error when we try to call the `NumberTile` constructor in the `NumberBoard` constructor because we're now trying to provide one too many arguments in the call. Remove the final argument (and the comma before it) from the constructor call; I've been using `null` for the final argument. The code should now compile and run without any errors.

    Note: As an alternative to following Steps 67-70 below, it's certainly possible to use a single `SoundEffect` field in the `NumberTile` class; you'd just have to load the appropriate sound effect in the `NumberTile` constructor based on the `isCorrectNumber` flag you set at the end of the constructor and play that sound effect at the appropriate places in the `NumberTile` `Update` method. I did it the way I described in Steps 67-70 to more closely parallel the structure for the Visual Studio XNA solution, but you can of course do it with the single `SoundEffect` field instead if you prefer.

67. Declare two new fields in the `NumberTile` class to hold `SoundEffect` objects. One of these fields will hold the sound effect for a correct guess and one will hold the sound effect for an incorrect guess. (Chapter 12, Week 6)
68. In the `NumberTile` constructor, change the comment that says `// set sound bank field` to `// load sound effects`. Use the constructor's `contentManager` parameter to load the explosion asset into your correct guess sound effect field and the loser asset into your incorrect guess sound effect field. (Week 7)
69. Run your code to make sure the sound effects are loaded without crashing the program.
70. In the `NumberTile` `Update` method, add code to play the correct guess sound effect when the player picks a tile corresponding to the correct number (right after you set the blinking flag to `true`) and to play the incorrect guess sound effect when the player picks an incorrect tile (right after you set the shrinking flag to `true`) (Week 7)
71. In the fields for the `Game1` class, change the comment that says `// audio components` to `// new game sound effect`. Add a field below the comment to hold a `SoundEffect` object for the new game sound effect
72. In the `Game1` `LoadContent` method, change the comment that says `// load audio content` to `// load sound effect`. Use `Content` to load the applause asset into your new game sound effect field. (Week 7)

73. In the `Game1` `Update` method, add code to play the new game sound effect before calling the `StartGame` method. We do this here rather than in the `StartGame` method because we don't want this cue to play for the first game (Week 7)
74. Change the if statement at the top of the `Game1` `Update` method to exit if the <Esc> key is pressed (Chapter 15, Week 7)
75. That's it, you're done!