Name:

Date:

C# Lab

A Day at the Races

This lab gives you a spec that describes a program for you to build, using the knowledge you've gained over the last few chapters.

This project is bigger than the ones you've seen so far. So read the whole thing before you get started, and give yourself a little time. And don't worry if you get stuck—there's nothing new in here, so you can move on in the book and come back to the lab later.

We've filled in a few design details for you, and we've made sure you've got all the pieces you need...and nothing else.

It's up to you to finish the job. You can download an executable for this lab from the website...but we won't give you the code for the answer.

The spec: build a racetrack simulator

Joe, Bob, and Al love going to the track, but they're tired of losing all their money. They need you to build a simulator for them so they can figure out winners *before* they lay their money down. And, if you do a good job, they'll cut you in on their profits.

Here's what you're going to build for them....

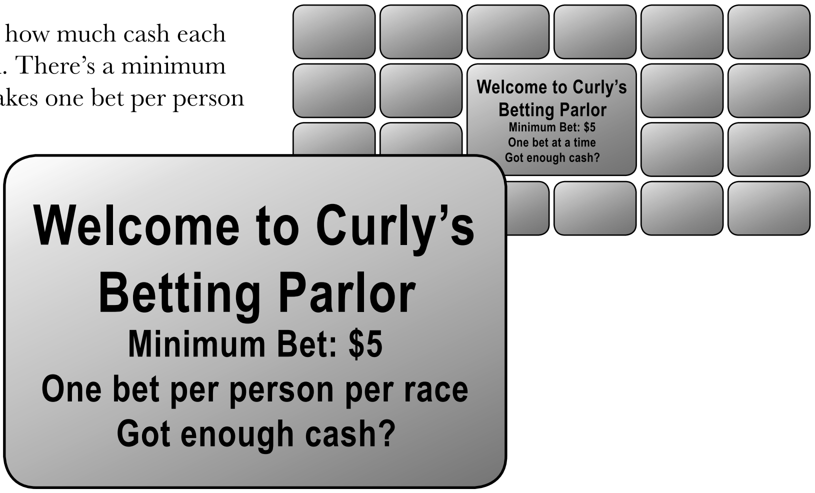
The Guys

Joe, Bob, and Al want to bet on a dog race. Joe starts with 50 bucks, Bob starts with 75 bucks, and Al starts with 45 bucks. Before each race, they'll each decide if they want to bet, and how much they want to put down. The guys can change their bets right up to the start of the race…but once the race starts, all bets are final.

The Betting Parlor

The betting parlor keeps track of how much cash each guy has, and what bet he's placed. There's a minimum bet of 5 bucks. The parlor only takes one bet per person for any one race.

The parlor checks to make sure that the guy who's betting has enough cash to cover his bet—so the guys can't place a bet if they don't have the cash to cover the bet.


Welcome to Curly's Betting Parlor
Minimum Bet: $5
One bet at a time
Got enough cash?
Welcome to Curly's Betting Parlor
Minimum Bet: $5
One bet per person per race
Got enough cash?

Betting

Every bet is double-or-nothing—either the winner doubles his money, or he loses what he bet. There's a minimum bet of 5 bucks, and each guy can bet up to 15 bucks on a single dog. If the dog wins, the bettor ends up with twice the amount that he bet (after the race is complete). If he loses, that amount disappears from his pile.

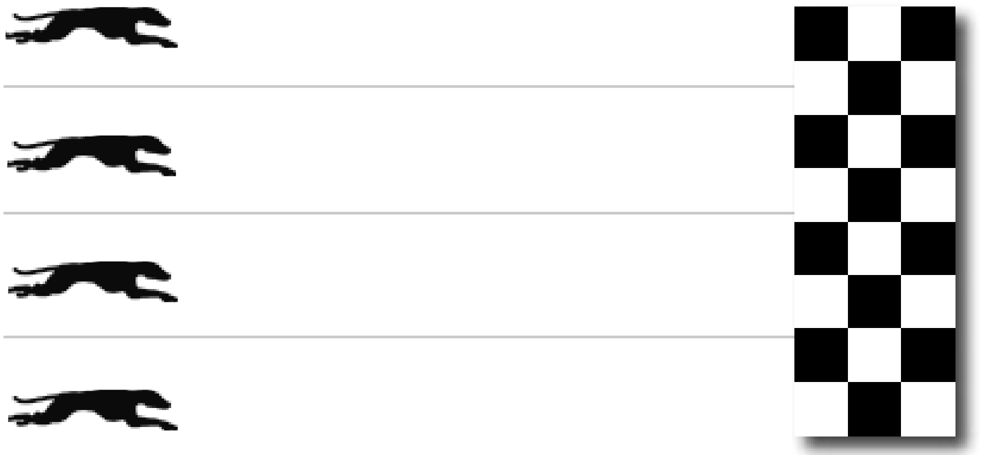
Say a guy places a $10 bet at the window. At the end of the race, if his dog wins, his cash goes up by $10 (because he keeps the original $10 he bet, plus he gets $10 more from winning). If he loses, his cash goes down by $10.

All bets: double-or-nothing
Minimum Bet: $5
Up to $15 per dog
Win: $$ added
Lose: $$ removed

The Race

There are four dogs that run on a straight track. The winner of the race is the first dog to cross the finish line. The race is totally random, there are no handicaps or odds, and a dog isn't more likely to win his next race based on his past performance.

If you want to build a handicap system, by all means do it! It'll be really good practice writing some fun code.



Sound fun? We've got more details coming up... →

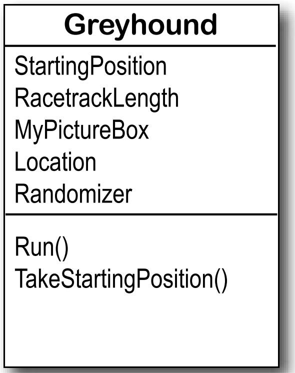
You'll need three classes and a form

You'll build three main classes in the project, as well as a GUI for the simulator. You should have an array of three `Guy` objects to keep track of the three guys and their winnings, and an array of four `Greyhound` objects that actually run the race. Also, each instance of `Guy` should have its own `Bet` object that keeps track of his bet and pays out (or takes back) cash at the end of the race.

We've gotten you started with class descriptions and some snippets of code to work from. You've got to finish everything up.

You'll need to add "using System.Windows.Forms" to the top of the Greyhound and Guy classes. And you'll need to add "using System. Drawing;" to Greyhound, because it uses Point.

We've given you the skeleton of the class you need to build. Your job is to fill in the methods.



```
class Greyhound {
   public int StartingPosition; // Where my PictureBox starts
   public int RacetrackLength; // How long the racetrack is
   public PictureBox MyPictureBox = null; // My PictureBox object
   public int Location = 0; // My Location on the racetrack
   public Random Randomizer; // An instance of Random

   public bool Run() {
     // Move forward either 1, 2, 3 or 4 spaces at random
     // Update the position of my PictureBox on the form
     // Return true if I won the race
   }

   public void TakeStartingPosition() {
     // Reset my location to the start line
  }
}
```

You only need one instance of Random—each Greyhound's Randomizer reference should point to the same Random object.

We've added comments to give you an idea of what to do.

See how the class diagram matches up with the code?

The Greyhound object initializer is pretty straightforward. Just make sure you pass a reference to the right PictureBox on the form to each Greyhound object.

Don't overthink this... sometimes you just need to set a variable, and you're done.

Your object can control things on your form...

The `Greyhound` class keeps track of its position on the racetrack during the race. It also updates the location of the `PictureBox` representing the dog moving down the race track. Each instance of `Greyhound` uses a field called `MyPictureBox` to reference the `PictureBox` control on the form that shows the picture of the dog. Suppose the `distance` variable contains the distance to move the dog forward. Then this code will update the location of `MyPictureBox` by adding `distance` to its `X` value:

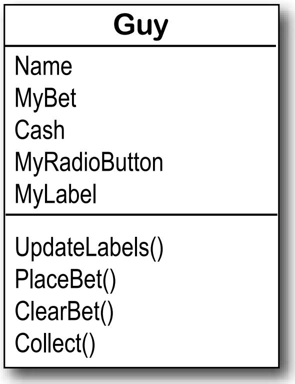
You'll have to make sure the form passes the right picture box into each Greyhound's object initializer.

```
Point p = MyPictureBox.Location;
p.X += distance;
MyPictureBox.Location = p;
```

You get the current location of the picture...

...add the value to move forward to its X coordinate...

...and then update the picture box location on the form.



```
class Guy {
   public string Name; // The guy's name
   public Bet MyBet; // An instance of Bet() that has his bet
   public int Cash; // How much cash he has


   // The last two fields are the guy's GUI controls on the form
   public RadioButton MyRadioButton; // My RadioButton
   public Label MyLabel; // My Label


   public void UpdateLabels() {
     // Set my label to my bet's description, and the label on my
     // radio button to show my cash ("Joe has 43 bucks")
   }v

   public void ClearBet() { } // Reset my bet so it's zero

   public bool PlaceBet(int Amount, int Dog) {
     // Place a new bet and store it in my bet field
     // Return true if the guy had enough money to bet
   }

   public void Collect(int Winner) { } // Ask my bet to pay out
}
```

Once you set MyLabel to one of the labels on the form, you'll be able to change the label's text using MyLabel.Text. And the same goes for MyRadioButton!

When you initialize the Guy object, make sure you set its MyBet field to null, and call its UpdateLabels() method as soon as it's initialized.

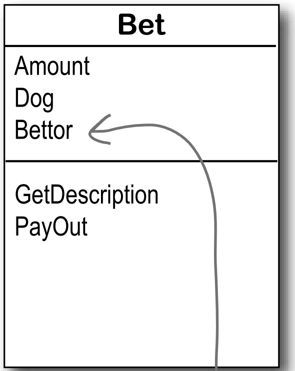
Add your code here.

Remember that bets are represented by instances of Bet.

The key here is to use the Bet object...let it do the work.

This is the object that Guy uses to represent bets in the application.

The object initializer for Bet just sets the amount, dog, and bettor.



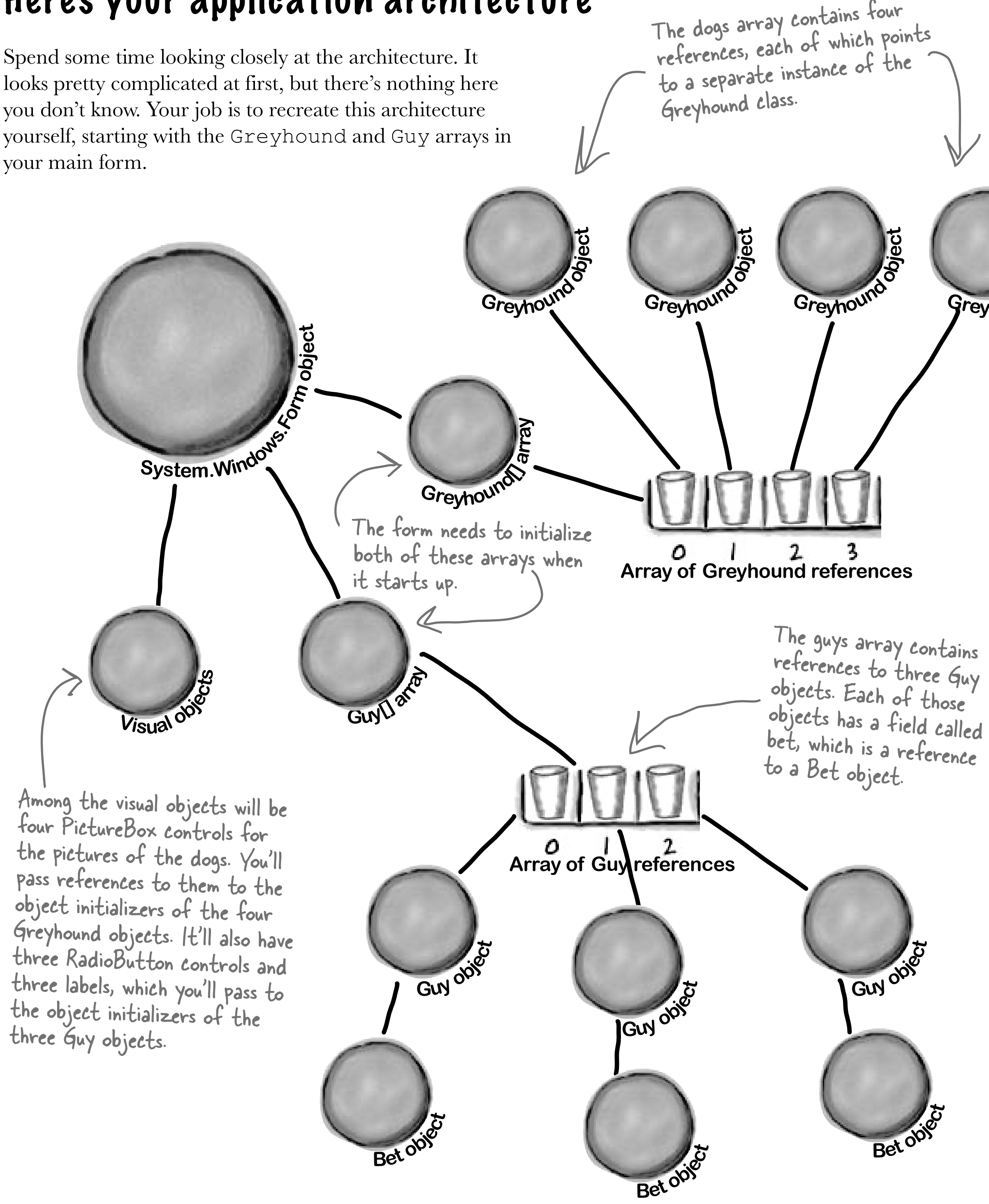
```
class Bet {
   public int Amount; // The amount of cash that was bet
   public int Dog; // The number of the dog the bet is on
   public Guy Bettor; // The guy who placed the bet

   public string GetDescription() {
     // Return a string that says who placed the bet, how much
     // cash was bet, and which dog he bet on ("Joe bets 8 on
     // dog #4"). If the amount is zero, no bet was placed
     // ("Joe hasn't placed a bet").
   }

   public int PayOut(int Winner) {
     // The parameter is the winner of the race. If the dog won,
     // return the amount bet. Otherwise, return the negative of
     // the amount bet.
   }
}
```

This is a common programming task: assembling a string or message from several individual bits of data.

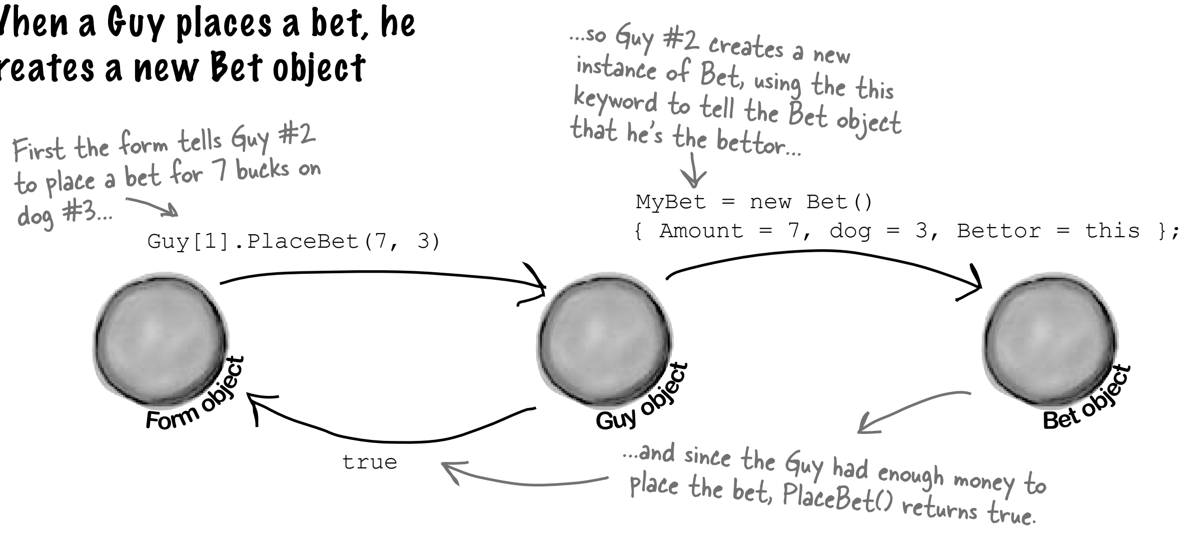
Hint: You'll instantiate Bet in the Guy code. Guy willl use the **this** keyword to pass a reference to himself to the Bet's initializer.

Here's your application architecture

Spend some time looking closely at the architecture. It looks pretty complicated at first, but there's nothing here you don't know. Your job is to recreate this architecture yourself, starting with the `Greyhound` and `Guy` arrays in your main form.


The dogs array contains four references, each of which points to a separate instance of the Greyhound class.
Greyhound object
Greyhound object
Greyhound object
Greyhound object
System.Windows.Form object
Greyhound[] array
0 1 2 3
Array of Greyhound references
The form needs to initialize both of these arrays when it starts up.
Visual objects
Guy[] array
The guys array contains references to three Guy objects. Each of those objects has a field called bet, which is a reference to a Bet object.
Among the visual objects will be four PictureBox controls for the pictures of the dogs. You'll pass references to them to the object initializers of the four Greyhound objects. It'll also have three RadioButton controls and three labels, which you'll pass to the object initializers of the three Guy objects.
0 1 2
Array of Guy references
Guy object
Guy object
Guy object
Bet object
Bet object
Bet object

When a Guy places a bet, he creates a new Bet object



First the form tells Guy #2 to place a bet for 7 bucks on dog #3...

```
Guy[1].PlaceBet(7, 3)
```

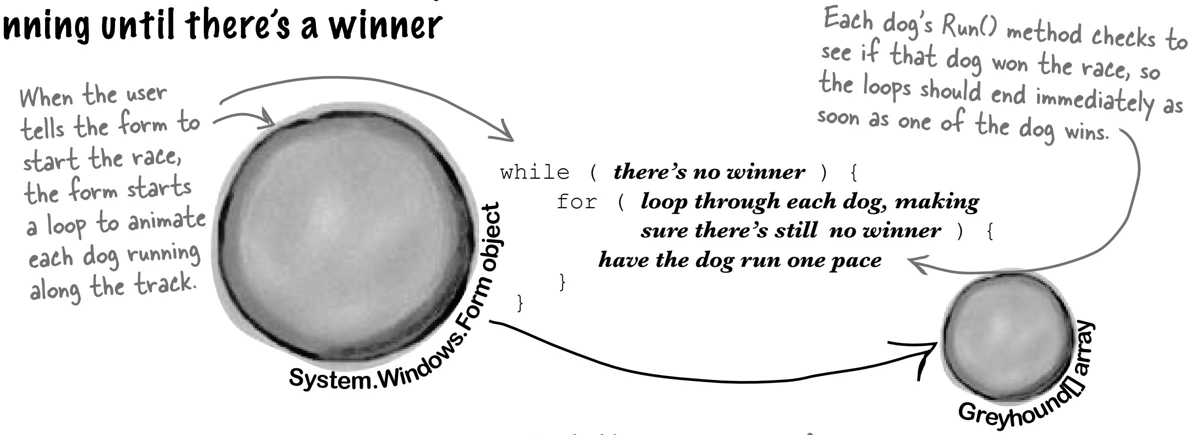
...so Guy #2 creates a new instance of Bet, using the this keyword to tell the Bet object that he's the bettor...

```
MyBet = new Bet()
{ Amount = 7, dog = 3, Bettor = this };
```

```
true
```

...and since the Guy had enough money to place the bet, PlaceBet() returns true.

The form tells the dogs to keep running until there's a winner

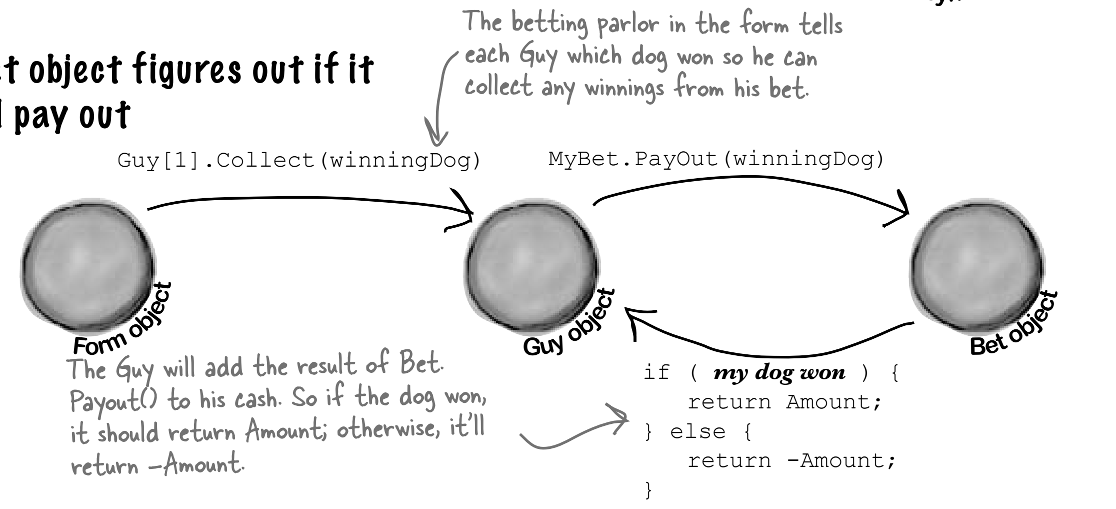


When the user tells the form to start the race, the form starts a loop to animate each dog running along the track.

```
while ( there's no winner ) {
    for ( loop through each dog, making
            sure there's still no winner ) {
        have the dog run one pace
    }
}
```

Each dog's Run() method checks to see if that dog won the race, so the loops should end immediately as soon as one of the dog wins.

The Bet object figures out if it should pay out



The betting parlor in the form tells each Guy which dog won so he can collect any winnings from his bet.

```
Guy[1].Collect(winningDog)
```

```
MyBet.PayOut(winningDog)
```

The Guy will add the result of Bet. Payout() to his cash. So if the dog won, it should return Amount; otherwise, it'll return -Amount.

```
if ( my dog won ) {
    return Amount;
} else {
    return -Amount;
}
```

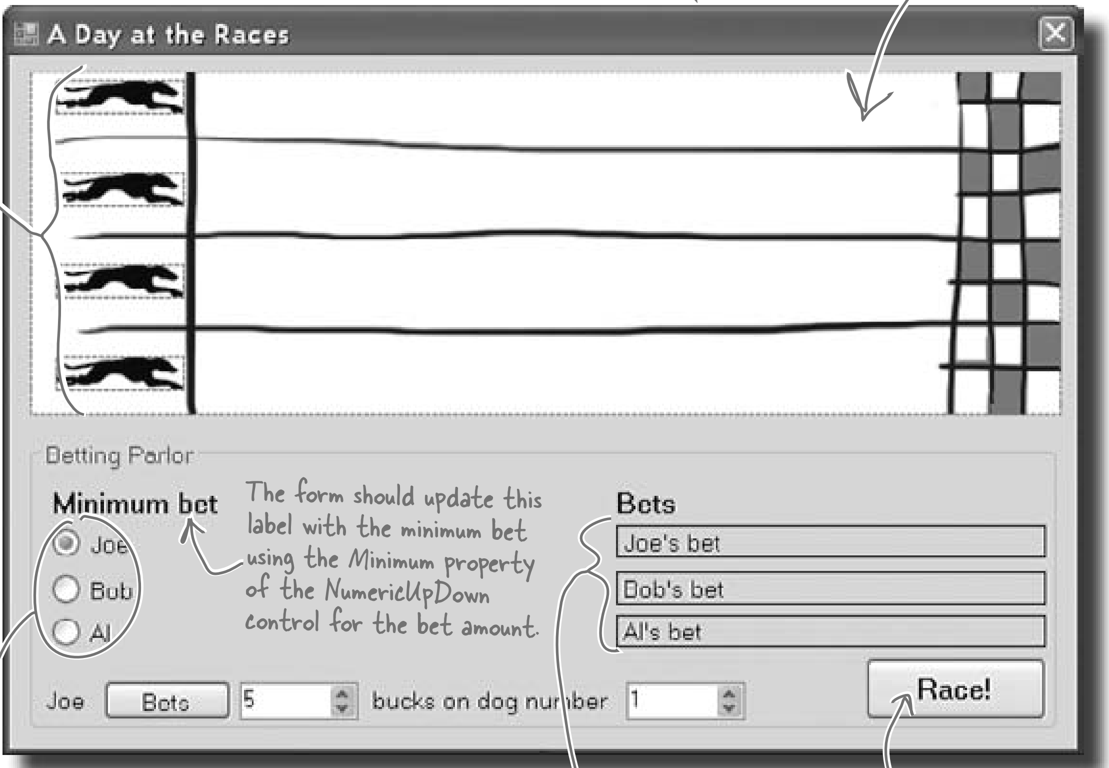
Here's what your GUI should look like

The graphical user interface for the "Day at the Races" application consists of a form that's divided into two sections. The top is the racetrack: a `PictureBox` control for the track, and four more for the dogs. The bottom half of the form shows the betting parlor, where three guys (Joe, Bob, and Al) can bet on the outcome of the race.

Each of the four dogs has its own PictureBox control. When you initialize each of the four Greyhound objects, each one's MyPicturebox field will have a reference to one of these objects. You'll pass the reference (along with the racetrack length and starting position) to the Greyhound's object initializer.

You'll use the Length property of the racetrack PictureBox control to set the racetrack length in the Greyhound object, which it'll use to figure out if it won the race.

Make sure you set each PictureBox's SizeMode property to Zoom.



The form should update this label with the minimum bet using the Minimum property of the NumericUpDown control for the bet amount.

All three guys can bet on the race, but there's only one betting window so only one guy can place a bet at a time. These radio buttons are used to select which guy places the bet.

When a Guy places a bet, it overwrites any previous bet he placed. The current bets show up in these label controls. Each label has AutoSize set to False and BorderStyle set to FixedSingle.

Once all bets are placed, click this button to start the race.

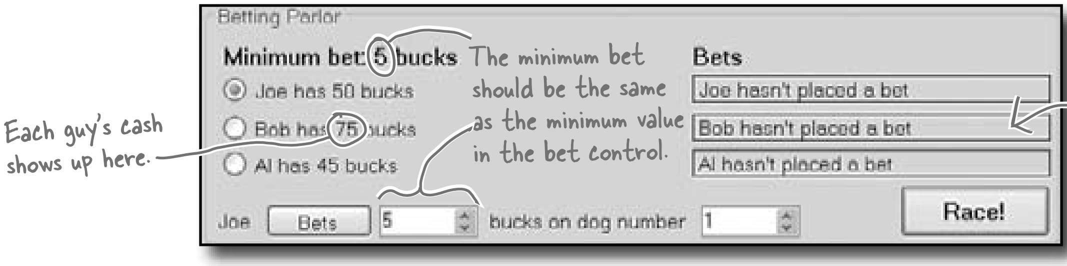
You can download the graphics files from www.headfirstlabs.com/books/hfcsharp/

Placing bets

Use the controls in the Betting Parlor group box to place each guy's bet. There are three distinct stages here:

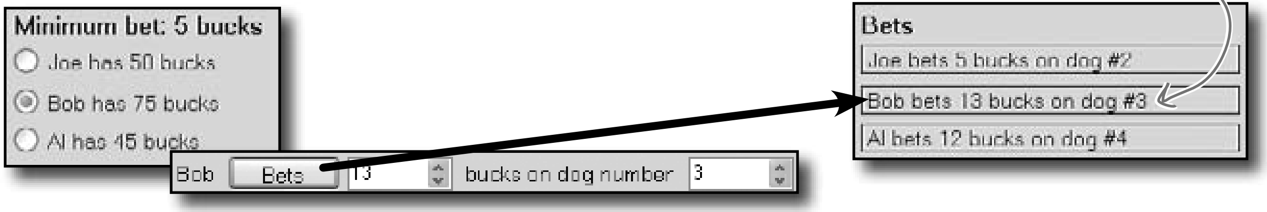
When a guy places a bet, his Guy object updates this label using the MyLabel reference. He also updates the cash he has using his MyRadioButton reference.

1 No bets have been placed yet
When the program first starts up, or if a race has just finished, no bets have been placed in the betting parlor. You'll see each guy's total cash next to his name on the left.



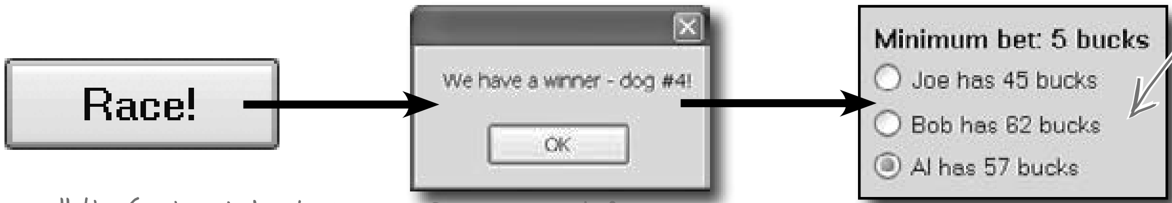
Once Bob places his bet, his Guy object updates this label and the radio button text.

2 Each guy places his bets
To place a bet, select the guy's radio button, select an amount and a dog, and click the Bets button. His `PlaceBet()` method will update the label and radio button.



Since Al bet 12 bucks on the winning dog, his cash goes up by 12. The other two guys lose the money they bet.

3 After the race, each guy collects his winnings (or pays up!)
Once the race is complete and there's a winner, each `Guy` object calls his `Collect()` method and adds his winnings or losses to his cash.

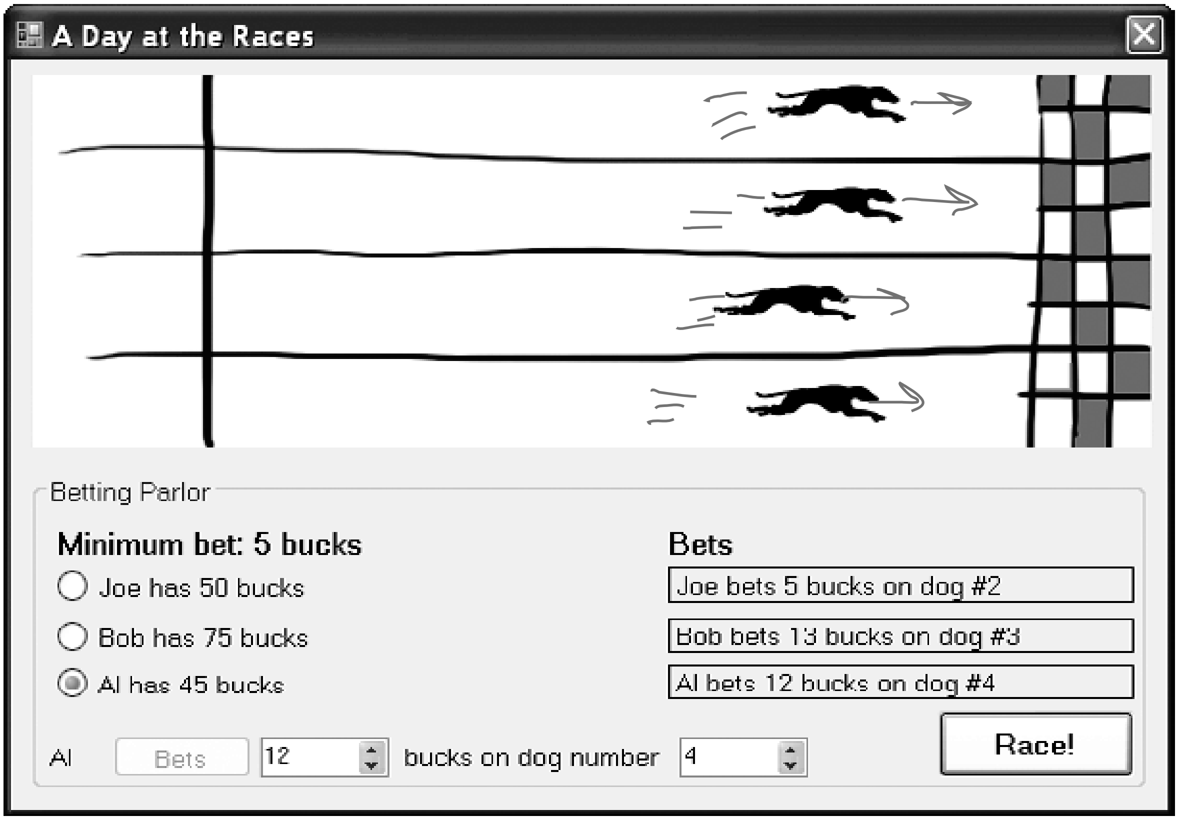


Make sure all the Greyhound objects share one Random object! If each dog creates its own new instance of Random, you might see a bug where all of the dogs generate the same sequence of random numbers.

The Finished Product

You'll know your "Day at the Races" application is done when your guys can place their bets and watch the dogs race.

During the race, the four dog images run across the racetrack until one of them wins the race.



During the race, no bets can be placed...and make sure you can't start a new race while the dogs are running!

You can download a finished executable, as well as the graphics files for the four dogs and the racetrack, from the Head First Labs website: www.headfirstlabs.com/books/hfcsharp

But you won't find the source code! In real life, you don't get a solution to your programming problems. Here's your chance to really test your C# knowledge and see just how much you've learned!