Name:

Date:

C# Lab

Invaders

This lab gives you a spec that describes a program for you to build, using the knowledge you've gained throughout this book.

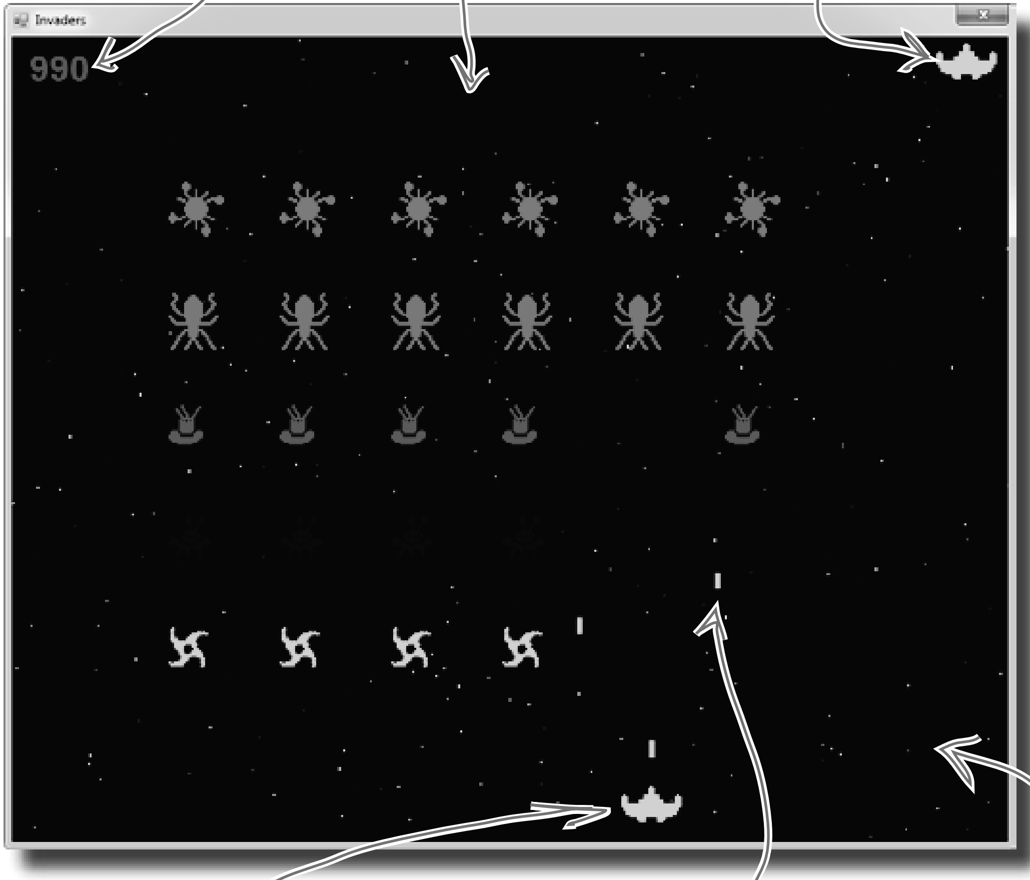
This project is bigger than the ones you've seen so far. So read the whole thing before you get started, and give yourself a little time. And don't worry if you get stuck—there's nothing new in here, so you can move on in the book and come back to the lab later.

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 grandfather of video games

In this lab you'll pay homage to one of the most popular, revered, and replicated icons in video game history, a game that needs no further introduction. **It's time to build Invaders.**



As the player destroys the invaders, the score goes up. It's displayed in the upper left-hand corner.

The invaders attack in waves of 30. The first wave moves slowly and fires a few shots at a time. The next wave moves faster, and fires more shots more frequently. If all 30 invaders in a wave are destroyed, the next wave attacks.

The player starts out with three ships. The first ship is in play, and the other two are kept in reserve. His spare ships are shown in the upper right-hand corner.

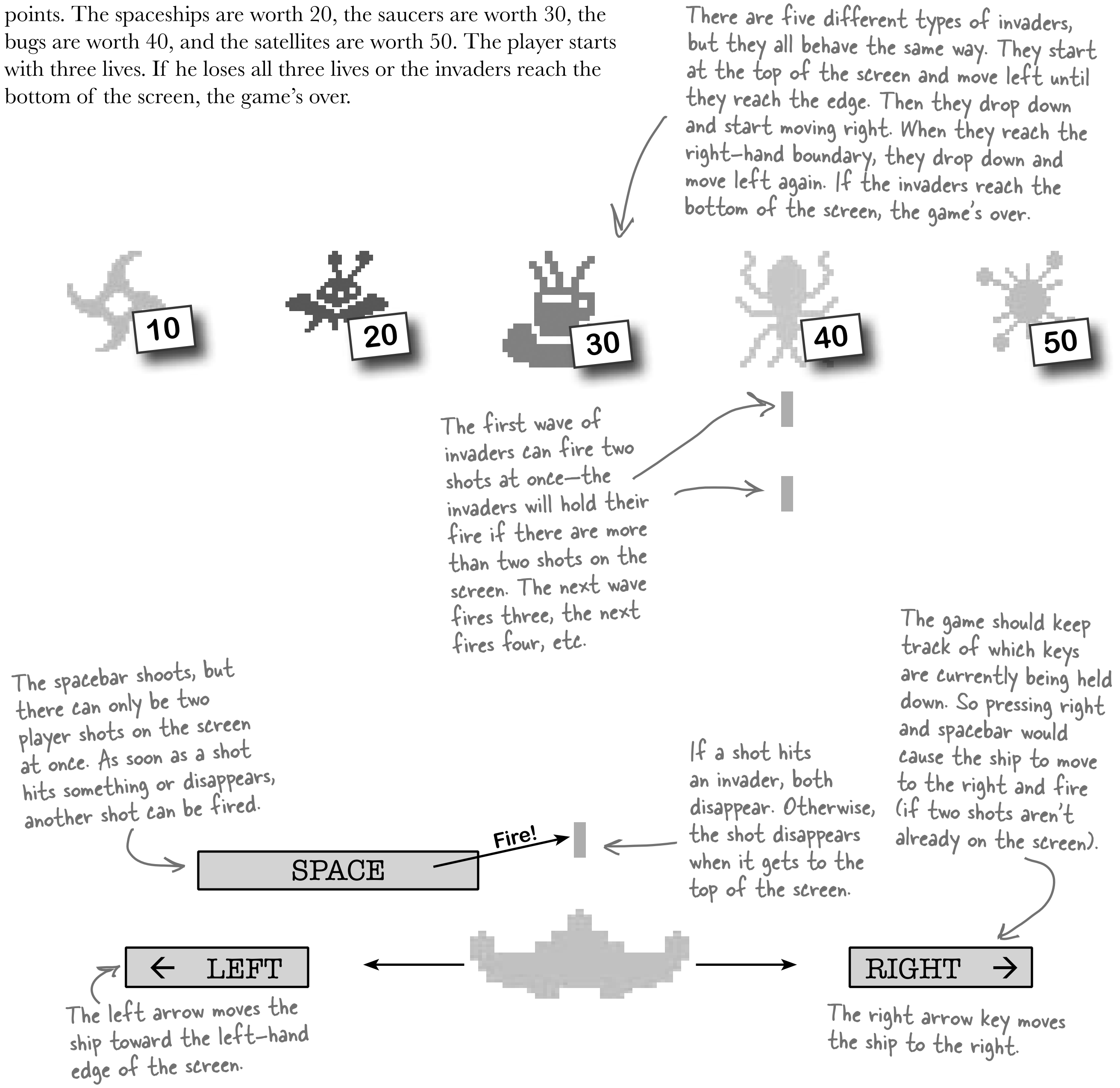
The player moves the ship left and right, and fires shots at the invaders. If a shot hits an invader, the invader is destroyed and the player's score goes up.

The invaders return fire. If one of the shots hits the ship, the player loses a life. Once all lives are gone, or if the invaders reach the bottom of the screen, the game ends and a big "GAME OVER" is displayed in the middle of the screen.

The multicolored stars in the background twinkle on and off, but don't affect gameplay at all.

Your mission: defend the planet against wave after wave of invaders

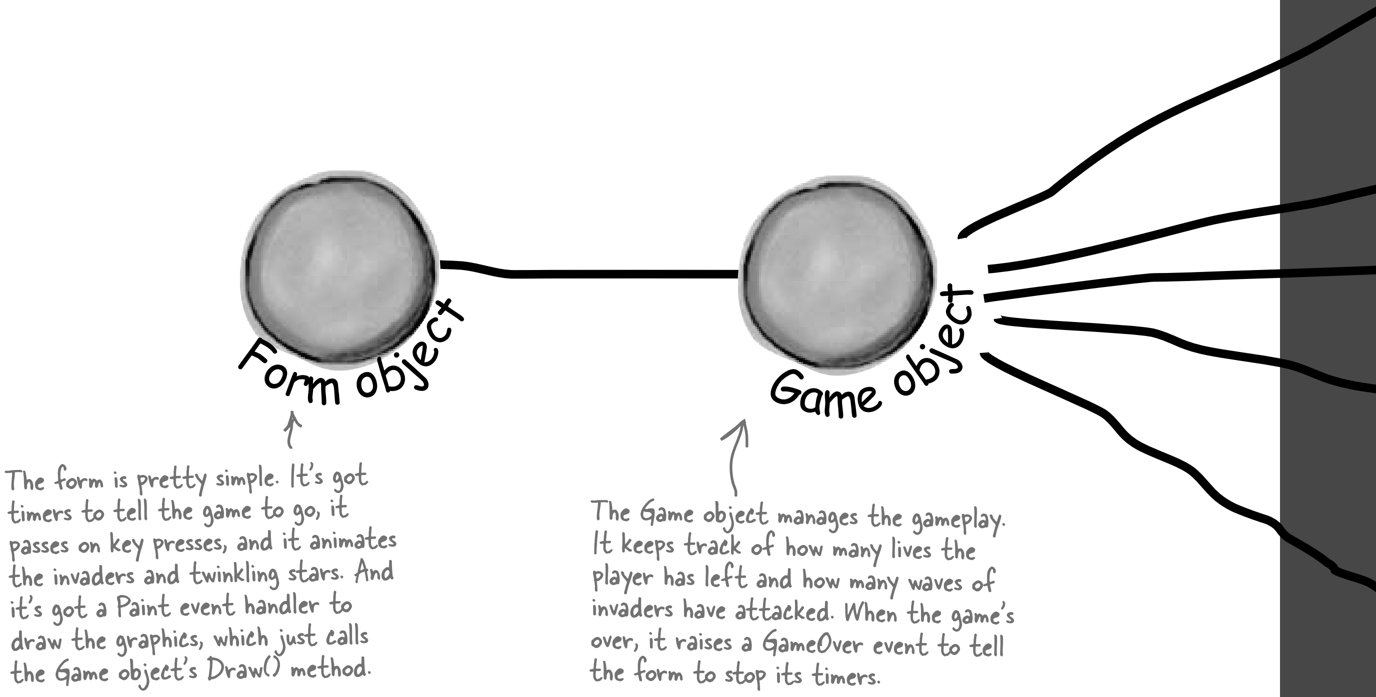
The invaders attack in waves, and each wave is a tight formation of 30 individual invaders. As the player destroys invaders, his score goes up. The bottom invaders are shaped like stars and worth 10 points. The spaceships are worth 20, the saucers are worth 30, the bugs are worth 40, and the satellites are worth 50. The player starts with three lives. If he loses all three lives or the invaders reach the bottom of the screen, the game's over.

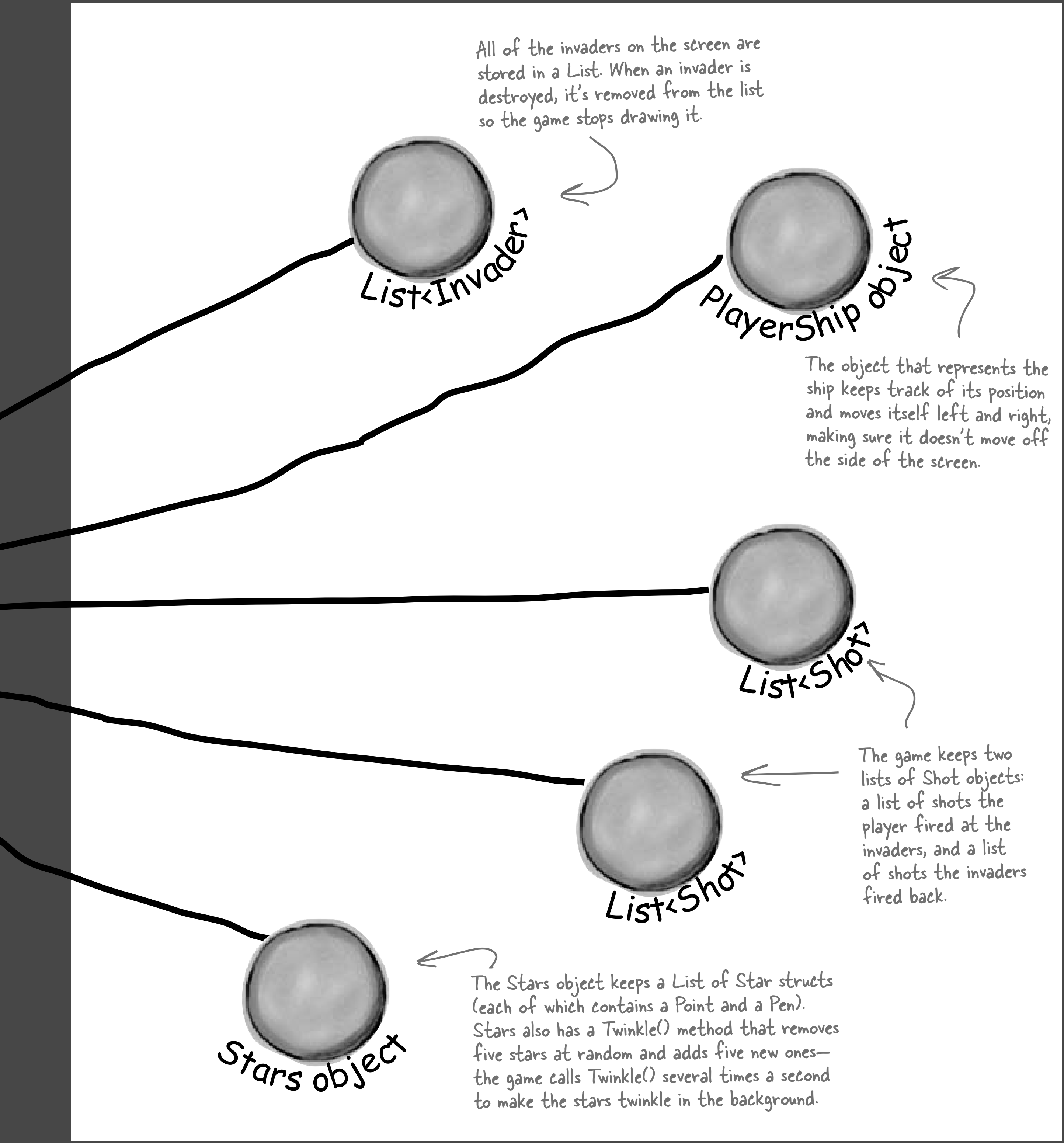

There are five different types of invaders, but they all behave the same way. They start at the top of the screen and move left until they reach the edge. Then they drop down and start moving right. When they reach the right-hand boundary, they drop down and move left again. If the invaders reach the bottom of the screen, the game's over.
10
20
30
40
50
The first wave of invaders can fire two shots at once—the invaders will hold their fire if there are more than two shots on the screen. The next wave fires three, the next fires four, etc.
The spacebar shoots, but there can only be two player shots on the screen at once. As soon as a shot hits something or disappears, another shot can be fired.
SPACE
Fire!
If a shot hits an invader, both disappear. Otherwise, the shot disappears when it gets to the top of the screen.
The game should keep track of which keys are currently being held down. So pressing right and spacebar would cause the ship to move to the right and fire (if two shots aren't already on the screen).
← LEFT
RIGHT →
The left arrow moves the ship toward the left-hand edge of the screen.
The right arrow key moves the ship to the right.

The architecture of Invaders

Invaders needs to keep track of a wave of 30 invaders (including their location, type, and score value), the player's ship, shots that the player and invaders fire at each other, and stars in the background. As in the Quest lab, you'll need a `Game` object to keep up with all this and coordinate between the form and the game objects.

Here's an overview of what you'll need to create:


Form object
Game object
The form is pretty simple. It's got timers to tell the game to go, it passes on key presses, and it animates the invaders and twinkling stars. And it's got a Paint event handler to draw the graphics, which just calls the Game object's Draw() method.
The Game object manages the gameplay. It keeps track of how many lives the player has left and how many waves of invaders have attacked. When the game's over, it raises a GameOver event to tell the form to stop its timers.

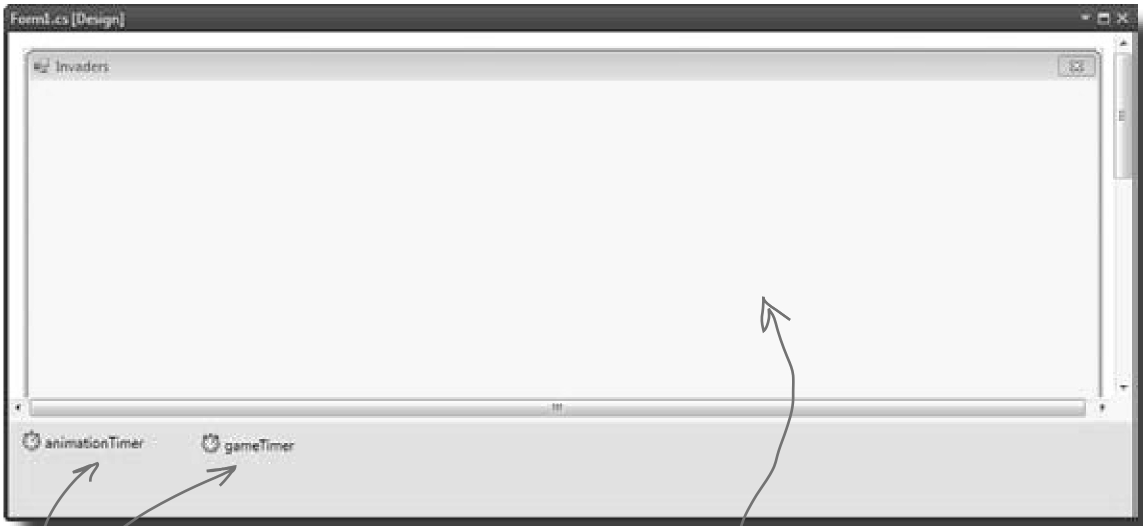

All of the invaders on the screen are stored in a List. When an invader is destroyed, it's removed from the list so the game stops drawing it.
List<Invader>
PlayerShip object
The object that represents the ship keeps track of its position and moves itself left and right, making sure it doesn't move off the side of the screen.
List<Shot>
List<Shot>
The game keeps two lists of Shot objects: a list of shots the player fired at the invaders, and a list of shots the invaders fired back.
Stars object
The Stars object keeps a List of Star structs (each of which contains a Point and a Pen). Stars also has a Twinkle() method that removes five stars at random and adds five new ones—the game calls Twinkle() several times a second to make the stars twinkle in the background.

Design the Invaders form

The Invaders form has only two controls: a timer to trigger animation (making the stars twinkle and the invaders animate by changing each invader picture to a different frame), and a timer to handle gameplay (the invaders marching left and right, the player moving, and the player and invaders shooting at each other). Other than that, the only intelligence in the form is an event handler to handle the game's `GameOver` event, and `KeyUp` and `KeyDown` event handlers to manage the keyboard input.

The form fires a KeyDown event any time a key is pressed, and it fires a KeyUp event whenever a key is released.

When the form initializes its Game object, it passes its ClientRectangle to it so it knows the boundaries of the form. So you can change the size of the battlefield just by changing the size of the form.



You should add two timers: animationTimer and gameTimer.

Set the form's FormBorderStyle property to FixedSingle and its DoubleBuffered property to true, turn off its MinimizeBox and MaximizeBox properties, set its title, and then stretch it out to the width you want the game area to be.

The animation timer handles the eye candy

animationTimer

The stars in the game's background and the invader animation don't affect gameplay, and they continue when the game is paused or stopped. So we need a separate timer for those.

Add code for the animation timer's tick event

Your code should have a counter that cycles from 0 to 3 and then back down to 0. That counter is used to update each of the four-cell invader animations (creating a smooth animation). Your handler should also call the `Game` object's `Twinkle()` method, which will cause the stars to twinkle. Finally, it needs to call the form's `Refresh()` method to repaint the screen.

Try a timer interval of 33ms, which will give you about 30 frames per second. Make sure you set the game timer to a shorter interval, though. The ship should move and gameplay should occur more quickly than the stars twinkle.

Animation occurs even when gameplay doesn't. That means that the stars twinkle and the invaders animate even if the game is over, paused, or hasn't been started.

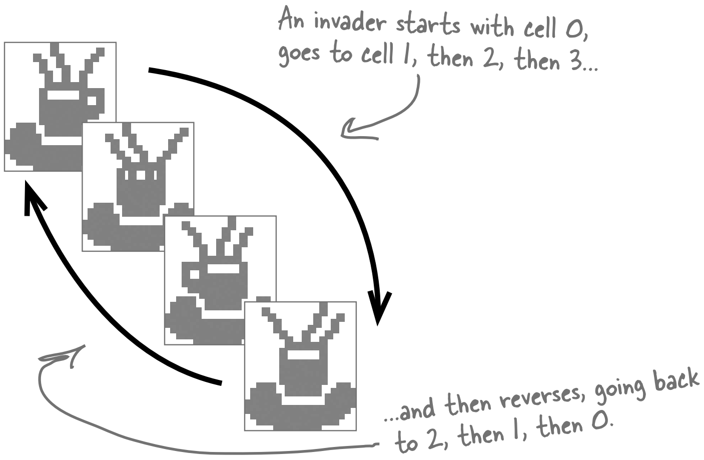
Adjust the timers for smooth animation

With a 33ms interval for animation, set the game timer to 10ms. That way, the main gameplay will occur more quickly than the animation (which is really just background eye candy). At the same time, the `Go()` method in `Game` (fired by the game timer, which we'll talk about in a little bit) can take a lot of CPU cycles. If the CPU is busy handling gameplay, the animation timer will just wait until the CPU gets to it, and then fire (and animate the stars and invaders).

Alternately, you can just set both timers to an interval of 5ms, and the game will run and animate about as fast as your system can handle (although on fast machines, animation could get annoyingly quick).

If the animation timer is set to 33ms, but the Game object's Go() method takes longer than that to run, then animation will occur once Go() completes.

We tried things out on a slow machine, and found that setting the animation interval to 100ms and the gameplay timer interval to 50ms gave us a frame rate of about 10 frames per second, which was definitely playable. Try starting there and reducing each interval until you're happy.


An invader starts with cell 0, goes to cell 1, then 2, then 3...
...and then reverses, going back to 2, then 1, then 0.

Respond to keyboard input

Before we can code the game timer, we need to write event handlers for the KeyDown and KeyUp events. KeyDown is triggered when a key is pressed, and KeyUp when a key is released. For most keys, we can simply take action by firing a shot or quitting the game.

For some keys, like the right or left arrow, we'll want to store those in a list that our game timer can then use to move the player's ship. So we'll also need a list of pressed keys in the form object:

So if the player's holding down the left arrow and spacebar at the same time, the list will contain Keys.Left and Keys.Space.

We need a list of keys so we can track which keys have been pressed. Our game timer will need that list for movement in just a bit.

```
    List<Keys> keysPressed = new List<Keys>();

    private void Form1_KeyDown(object sender, KeyEventArgs e) {
      if (e.KeyCode == Keys.Q)
        Application.Exit();

      if (gameOver)
        if (e.KeyCode == Keys.S) {
          // code to reset the game and restart the timers
          return;
        }

      if (e.KeyCode == Keys.Space)
        game.FireShot();
      if (keysPressed.Contains(e.KeyCode))
        keysPressed.Remove(e.KeyCode);
      keysPressed.Add(e.KeyCode);
    }

    private void Form1_KeyUp(object sender, KeyEventArgs e) {
      if (keysPressed.Contains(e.KeyCode))
        keysPressed.Remove(e.KeyCode);
    }
```

The 'Q' key quits the game.

The Keys enum defines all the keys you might want to check key codes against.

If the game has ended, reset the game and start over.

But we only want this to work if the game's over. Pressing S shouldn't restart a game that's already in progress.

You'll need to fill in this code.

The spacebar fires a shot.

By removing the key and then re-adding it, the key becomes the last (most current) item in the list.

The key that's pressed gets added to our key list, which we'll use in a second.

We want the most current key pressed to be at the very top of the list, so that if the player mashes a few keys at the same time, the game responds to the one that was hit most recently. Then, when he lets up one key, the game responds to the next one in the list.

When a key is released, we remove it from our list of pressed keys.

Flip back to the KeyGame project you built in Chapter 4. You used a KeyDown event handler there, too!

The game timer handles movement and gameplay

gameTimer

The main job of the form's game timer is to call `Go()` in the `Game` class. But it also has to respond to any keys pressed, so it has to check the `keysPressed` list to find any keys caught by the `KeyDown` and `KeyUp` events:

Players "mash" a bunch of keys at once. If we want the game to be robust, it needs to be able to handle that. That's why we're using the keysPressed list.

This timer makes the game advance by one frame. So the first thing it does is call the Game object's Go() method to let gameplay continue.

```
private void gameTimer_Tick(object sender, EventArgs e)
{
    game.Go();
    foreach (Keys key in keysPressed)
    {
        if (key == Keys.Left)
        {
            game.MovePlayer(Direction.Left);
            return;
        }
        else if (key == Keys.Right)
        {
            game.MovePlayer(Direction.Right);
            return;
        }
    }
}
```

keysPressed is your List<Keys> object managed by the KeyDown and KeyUp event handlers. It contains every key the player currently has pressed.

The keysPressed list has the keys in the order that they're pressed. This foreach loop goes through them until it finds a Left or Right key, then moves the player and returns.

The KeyUp and KeyDown events use the Keys enum to specify a key. We'll use Keys.Left and Keys.Right to move the ship.

```
enum Direction {
    Left,
    Right,
    Up,
    Down,
}
```

Shots move up and down, the player moves left and right, and the invaders move left, right, and down. You'll need this enum to keep all those directions straight.

The KeyDown event handler just handles the space, S, and Q keystrokes without adding them to the keysPressed list. What would happen if you moved the code for firing the shot when the space key is pressed to this event handler?

One more form detail: the GameOver event

Add a private `bool` field called `gameOver` to the form that's `true` only when the game is over. Then add an event handler for the `Game` object's `GameOver` event that stops the game timer (but not the animation timer, so the stars still twinkle and the invaders still animate), sets `gameOver` to `true`, and calls the form's `Invalidate()` method.

Here's an example of adding another event to a form without using the IDE. This is all manual coding.

When you write the form's `Paint` event handler, have it check `gameOver`. If it's `true`, have it write GAME OVER in big yellow letters in the middle of the screen. Then have it write "Press S to start a new game or Q to quit" in the lower right-hand corner. You can start the game out in this state, so the user has to hit S to start a new game.

The game over event and its delegate live in the Game class, which you'll see in just a minute.

The form's game timer tells the game to Go()

In addition to handling movement left and right, the main job of the game timer is to call the `Game` object's `Go()` method. That's where all of the gameplay is managed. The `Game` object keeps track of the state of the game, and its `Go()` method advances the game by one frame. That involves:

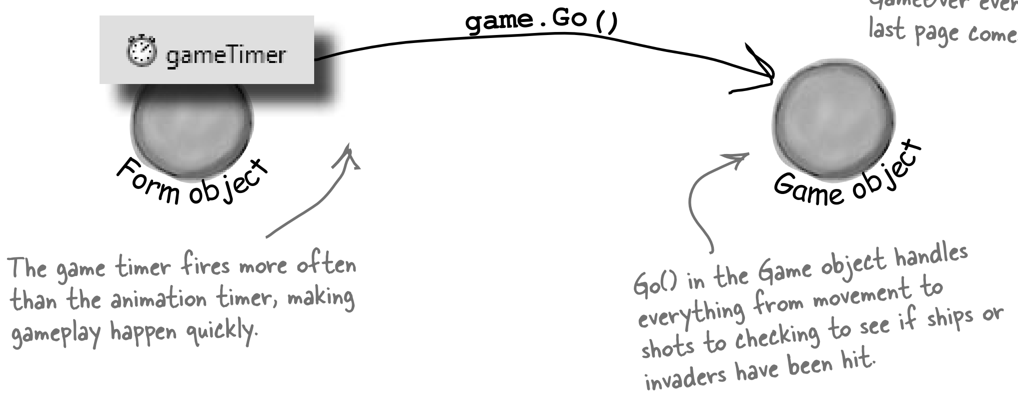
1. **Checking to see if the player died**, using its `Alive` property. When the player dies, the game shows a little animation of the ship collapsing (using `DrawImage()` to squish the ship down to nothing). The animation is done by the `PlayerShip` class, so `Go()` just needs to check to see if it's dead. If it is, it `returns`—that way, it keeps the invaders from moving or shooting while the player gets a small break (and watches his ship get crushed).

2. **Moving each of the shots.** Shots fired by the invaders move down, and shots fired by the player move up. `Game` keeps two `List<Shot>` objects, one for the invaders' shots and one for the player's. Any shot that's moved off the screen needs to be removed from the list.

3. **Moving each of the invaders.** `Game` calls each `Invader` object's `Move()` method, and tells the invaders which way to move. `Game` also keeps up with where the invaders are in case they need to move down a row or switch directions. Then, `Game` checks to see if it's time for the invaders to return fire, and if so, it adds new `Shot` objects to the `List<>`.

4. **Checking for hits**. If a player's shot hit any invaders, `Game` removes the invaders from the appropriate `List<>`. Then `Game` checks to see if any of the invader shots have collided with the player's ship, and if so, it kills the player by setting its `Alive` property to `false`. If the player's out of lives, then `Game` raises the `GameOver` event to tell the form that the game's over. The form's `GameOver` event handler stops its game timer, so `Go()` isn't called again.

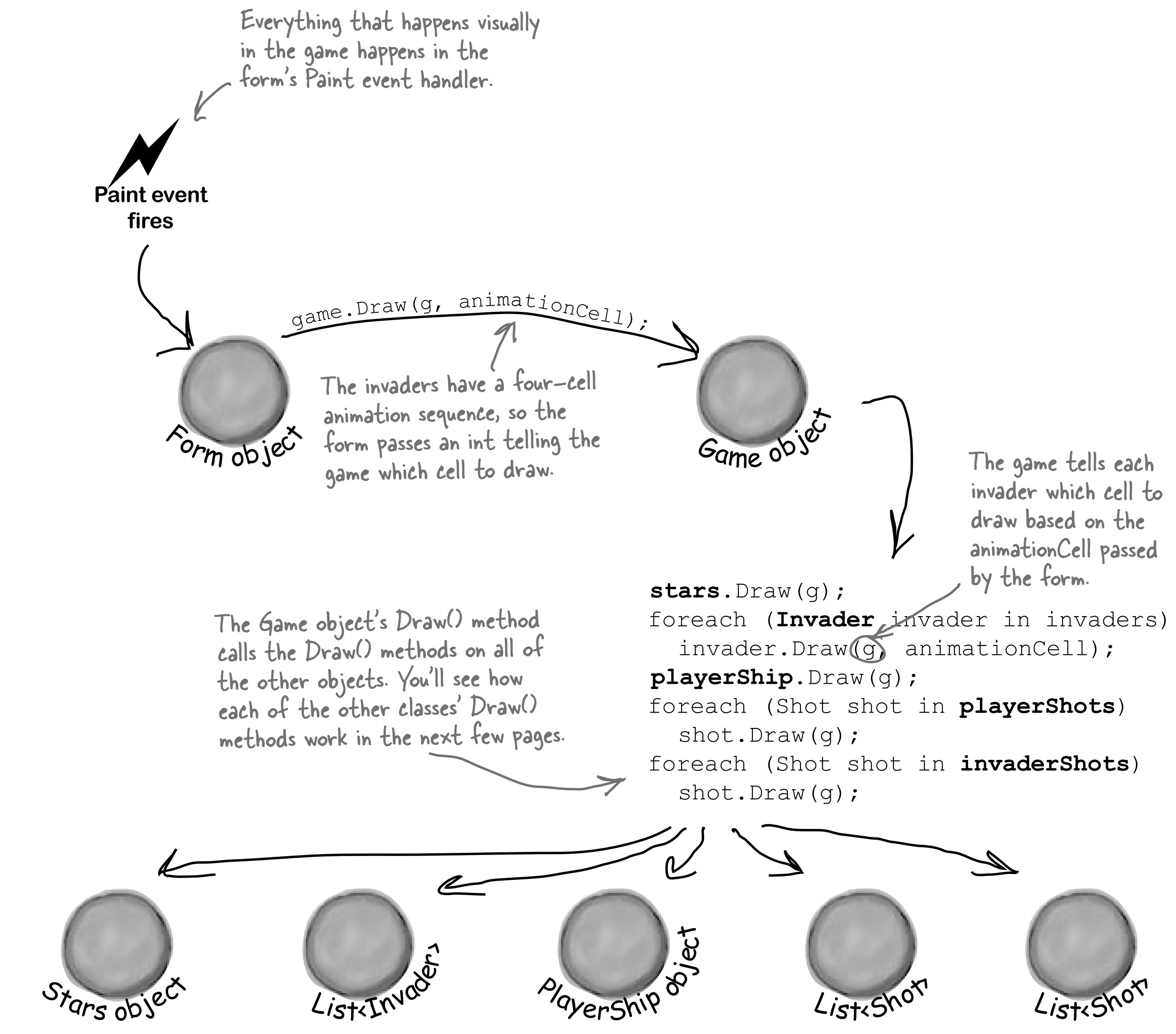
Here's where that GameOver event from the last page comes into play.


gameTimer
game.Go()
Form object
Game object
The game timer fires more often than the animation timer, making gameplay happen quickly.
Go() in the Game object handles everything from movement to shots to checking to see if ships or invaders have been hit.

Taking control of graphics

In earlier labs, the form used controls for the graphics. But now that you know how to use `Graphics` and double-buffering, the `Game` object should handle a lot of the drawing.

So the form should have a `Paint` event handler (make sure you set the form's `DoubleBuffered` property to `true`!). You'll delegate the rest of the drawing to the `Game` object by calling its `Draw()` method every time the form's `Paint` event fires.


Everything that happens visually in the game happens in the form's Paint event handler.
Paint event fires
game.Draw(g, animationCell);
The invaders have a four-cell animation sequence, so the form passes an int telling the game which cell to draw.
Form object
Game object
The game tells each invader which cell to draw based on the animationCell passed by the form.
The Game object's Draw() method calls the Draw() methods on all of the other objects. You'll see how each of the other classes' Draw() methods work in the next few pages.
Stars object
List<Invader>
PlayerShip object
List<Shot>
List<Shot>

```
stars.Draw(g);
foreach (Invader invader in invaders)
  invader.Draw(g, animationCell);
playerShip.Draw(g);
foreach (Shot shot in playerShots)
  shot.Draw(g);
foreach (Shot shot in invaderShots)
  shot.Draw(g);
```

Building the Game class

The Game class is the controller for the Invaders game. Here's a start on what this class should look like, although there's lots of work still for you to do.

```
class Game {
  private int score = 0;
  private int livesLeft = 2;
  private int wave = 0;
  private int framesSkipped = 0;

  private Rectangle boundaries;
  private Random random;

  private Direction invaderDirection;
  private List<Invader> invaders;

  private PlayerShip playerShip;
  private List<Shot> playerShots;
  private List<Shot> invaderShots;

  private Stars stars;


  public event EventHandler GameOver;

  // etc...
}
```

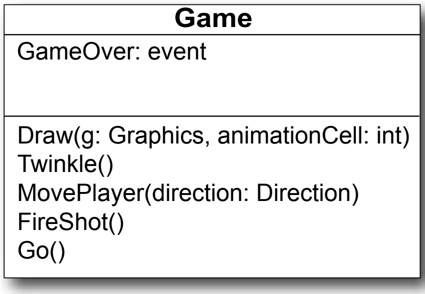
The score, livesLeft, and wave fields keep track of some basic information about the state of the game.

You'll use the frame field to slow down the invaders early on in the game—the first wave should **skip 6 frames** before they move to the left, the next wave should skip 5, the next should skip 4, etc.

This List<> of Invader objects keeps track of all of the invaders in the current wave. When an invader is destroyed, it's removed from the list. The game checks periodically to make sure the list isn't empty—if it is, it sends in the next wave of invaders.

This Stars object keeps track of the multicolored stars in the background.

The Game object raises its GameOver event when the player dies and doesn't have any more lives left. You'll build the event handler method in the form, and hook it into the Game object's GameOver event.


Game
GameOver: event
Draw(g: Graphics, animationCell: int)
Twinkle()
MovePlayer(direction: Direction)
FireShot()
Go()

Most of these methods combine methods on other objects to make a specific action occur.

Remember, these are the public methods. You may need a lot more private methods to structure your code in a way that makes sense to you.

The Game class methods

The `Game` class has five public methods that get triggered by different events happening in the form.

1. **The `Draw()` method draws the game on a `Graphics` object**
   The `Draw()` method takes two parameters: a `Graphics` object and an integer that contains the animation cell (a number from 0 to 3). First, it should draw a black rectangle that fills up the whole form (using the display rectangle stored in `boundaries`, received from the form). Then the method should draw the stars, the invaders, the player's ship, and then the shots. Finally, it should draw the score in the upper left-hand corner, the player's ships in the upper right-hand corner, and a big "GAME OVER" in yellow letters if `gameOver` is `true`.

2. **The `Twinkle()` method twinkles the stars**
   The form's animation timer event handler needs to be able to twinkle the stars, so the `Game` object needs a one-line method to call `stars.Twinkle()`.

   We'll write code for the Stars object in a few more pages.

3. **The `MovePlayer()` method moves the player**
   The form's keyboard timer event handler needs to move the player's ship, so the `Game` object also needs a two-line method that takes a `Direction` enum as a parameter, checks whether or not the player's dead, and calls `playerShip.Move()` to affect that movement.

4. **The `FireShot()` method makes the player fire a shot at the invaders**
   The `FireShot()` method checks to see if there are fewer than two player shots on screen. If so, the method should add a new shot to the `playerShots` list at the right location.

5. **The `Go()` method makes the game go**
   The form's animation timer calls the `Game` object's `Go()` method anywhere between 10 and 30 times a second (depending on the computer's CPU speed). The `Go()` method does everything the game needs to do to advance itself by a frame:

   ★ The game checks if the player's dead using its `Alive` property. If he's still alive, the game isn't over yet—if it were, the form would have stopped the animation timer with its `Stop()` method. So the `Go()` method won't do anything else until the player is alive again—it'll just return.

   ★ Every shot needs to be updated. The game needs to loop through both `List<Shot>` objects, calling each shot's `Move()` method. If any shot's `Move()` returns false, that means the shot went off the edge of the screen—so it gets deleted from the list.

   ★ The game then moves each invader, and allows them to return fire.

   ★ Finally, it checks for collisions: first for any shot that overlaps an invader (and removing both from their `List<T>` objects), and then to see if the player's been shot. We'll add a `Rectangle` property called `Area` to the `Invader` and `PlayerShip` classes—so we can use the `Contains()` method to see if the ship's area overlaps with a shot.

Filling out the Game class

The problem with class diagrams is that they usually leave out any non-public properties and methods. So even after you've got the methods from page 725 done, you've still got a lot of work to do. Here are some things to think about:

The constructor sets everything up

The `Game` object needs to create all of the other objects—the `Invader` objects, the `PlayerShip` object, the `List` objects to hold the shots, and the `Stars` object. The form passes in an initialized `Random` object and its own `ClientRectangle` struct (so the `Game` can **figure out the boundaries of the battlefield**, which it uses to determine when shots are out of range and when the invaders reach the edge and need to drop and reverse direction). Then, your code should create everything else in the game world.

We'll talk about most of these individual objects over the next several pages of this lab.

Build a NextWave() method

A simple method to create the next wave of invaders will come in handy. It should assign a new `List` of `Invader` objects to the invaders field, add the 30 invaders in 6 columns so that they're in their starting positions, increase the `wave` field by 1, and set the `invaderDirection` field to start them moving toward the right-hand side of the screen. You'll also change the `framesSkipped` field.

Here's an example of a private method that will really help out your Game class organization.

A few other ideas for private methods

Here are a few of the private method ideas you might play with, and see if these would also help the design of your `Game` class:

- ✓ A method to see if the player's been hit (`CheckForPlayerCollisions()`)
- ✓ A method to see if any invaders have been hit (`CheckForInvaderCollisions()`)
- ✓ A method to move all the invaders (`MoveInvaders()`)
- ✓ A method allowing invaders to return fire (`ReturnFire()`)

BRAIN POWER

It's possible to show protected and private properties and methods in a class diagram, but you'll rarely see that put into practice. Why do you think that is?

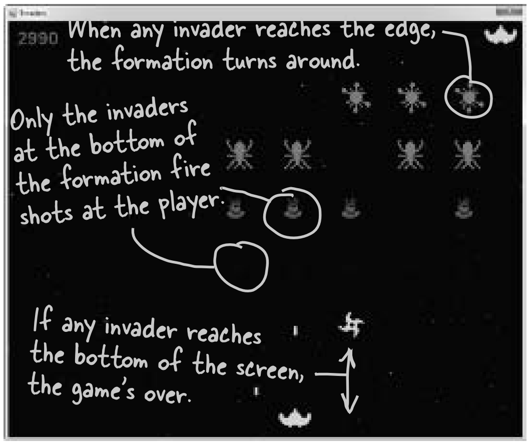
LINQ makes collision detection much easier

This seems really complex when you first read it, but each LINQ query is just a couple of lines of code. Here's a hint: don't overcomplicate it!

You've got collections of invaders and shots, and you need to search through those collections to find certain invaders and shots. Any time you hear collections and searching in the same sentence, you should think LINQ. Here's what you need to do:

1 **Figure out if the invaders' formation has reached the edge of the battlefield**
The invaders need to change direction if any one invader is within 100 pixels of the edge of the battlefield. When the invaders are marching to the right, once they reach the right-hand side of the form the game needs to tell them to drop down and start marching to the left. And when the invaders are marching to the left, the game needs to check if they've reached the left edge. To make this happen, add a private `MoveInvaders()` method that gets called by `Go()`.The first thing it should do is check and update the private `framesSkipped` field, and `return` if this frame should be skipped (depending on the level). Then it should check which direction the invaders are moving. If the invaders are moving to the right, `MoveInvaders()` should use LINQ to search the `invaderCollection` list for any invader whose location's X value is within 100 pixels of the right-hand boundary. If it finds any, then it should tell the invaders to march downward and then set `invaderDirection` equal to `Direction.Left`; if not, it can tell each invader to march to the right. On the other hand, if the invaders are moving to the left, then it should do the opposite, using another LINQ query to see if the invaders are within 100 pixels of the left-hand boundary, marching them down and changing direction if they are.

2 **Determine which invaders can return fire**
Add a private method called `ReturnFire()` that gets called by `Go()`. First, it should `return` if the invaders' shot list already has `wave + 1` shots. It should also `return` if `random.Next(10) < 10 - wave`. (That makes the invaders fire at random, and not all the time.) If it gets past both tests, it can use LINQ to group the invaders by their `Location.X` and sort them `descending`. Once it's got those groups, it can choose a group at random, and use its `First()` method to find the invader at the bottom of the column. All right, now you've got the shooter—you can add a shot to the invader's shot list just below the middle of the invader (use the invader's `Area` to set the shot's location).


2990
When any invader reaches the edge, the formation turns around.
Only the invaders at the bottom of the formation fire shots at the player.
If any invader reaches the bottom of the screen, the game's over.

3 **Check for invader and player collisions**
You'll want to create a method to check for collisions. There are three collisions to check for, and the `Rectangle` struct's `Contains()` method will come in really handy—just pass it any `Point`, and it'll return `true` if that point is inside the rectangle.

- ★ Use LINQ to find any dead invaders by looping through the shots in the player's shot list and selecting any invader where `invader.Area` contains the shot's location. Remove the invader and the shot.
- ★ Add a query to figure out if any invaders reached the bottom of the screen—if so, end the game.
- ★ You don't need LINQ to look for shots that collided with the player, just a loop and the player's `Area` property. (Remember, **you can't modify a collection inside a foreach loop**. If you do, you'll get an `InvalidOperationException` with a message that the collection was modified.)

Crafting the Invader class

Invader
Location: Point InvaderType: ShipType Area: Rectangle Score: int

Draw(g: Graphics, animationCell: int)
Move(direction: Direction)

The `Invader` class keeps track of a single invader. So when the `Game` object creates a new wave of invaders, it adds 30 instances of `Invader` to a `List<Invader>` object. Every time its `Go()` method is called, it calls each invader's `Move()` method to tell it to move. And every time its `Draw()` method is called, it calls each invader object's `Draw()` method. So you'll need to build out the `Move()` and `Draw()` methods. You'll want to add a private method called `InvaderImage()`, too—it'll come in really handy when you're drawing the invader. Make sure you call it inside the `Draw()` method to keep the `image` field up to date:

```
class Invader {
  private const int HorizontalInterval = 10;
  private const int VerticalInterval = 40;

  private Bitmap image;

  public Point Location { get; private set; }

  public ShipType InvaderType { get; private set; }

  public Rectangle Area { get {
    return new Rectangle(location, image.Size); }
  }

  public int Score { get; private set; }

  public Invader(ShipType invaderType, Point location, int score) {
    this.InvaderType = invaderType;
    this.Location = location;
    this.Score = score;
    image = InvaderImage(0);
  }

  // Additional methods will go here
}
```

The HorizontalInterval constant determines how many pixels an invader moves every time it marches left or right. VerticalInterval is the number of pixels it drops down when the formation reaches the edge of the battlefield.

Check out what we did with the Area property. Since we know the invader's location and we know its size (from its image field), we can add a get accessor that calculates a Rectangle for the area it covers...

...which means you can use the Rectangle's **Contains()** method inside a LINQ query to **detect any shots that collided** with an invader.

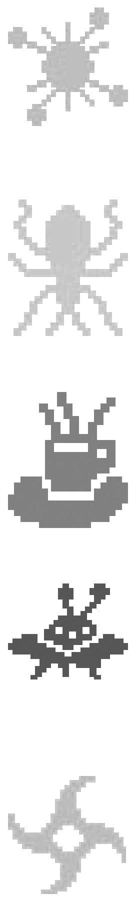
An Invader object uses the ShipType enum to figure out what kind of enemy ship it is.

```
enum ShipType {
    Bug,
    Saucer,
    Satellite,
    Spaceship,
    Star,
  }
```

Build the Invaders' methods

The three core methods for `Invader` are `Move()`, `Draw()`, and `InvaderImage()`. Let's look at each in turn.

There are five types of invaders, and each of them has four different animation cell pictures.



Move the invader ships

First, you need a method to move the invader ships. The `Game` object should send in a direction, using the `Direction` enum, and then the ship should move. Remember, the `Game` object handles figuring out if an invader needs to move down or change direction, so your `Invader` class doesn't have to worry about that.

```
    public void Move(Direction direction) {
      // This method needs to move the ship in the
      //   specified direction
    }
```

Draw the ship—and the right animation cell

Each `Invader` knows how to draw itself. Given a `Graphics` object to draw to, and the animation cell to use, the invader can display itself onto the game board using the `Graphics` object the `Game` gives it.

```
    public void Draw(Graphics g, int animationCell) {
      // This method needs to draw the image of
      //   the ship, using the correct animation cell
    }
```

Get the right invader image

You're going to need to grab the right image based on the animation cell a lot, so you may want to pull that code into its own method. Build an `InvaderImage()` method that returns a specific `Bitmap` given an animation cell.

```
    private Bitmap InvaderImage(int animationCell) {
      // This is mostly a convenience method, and
      //   returns the right bitmap for the specified cell
    }
```

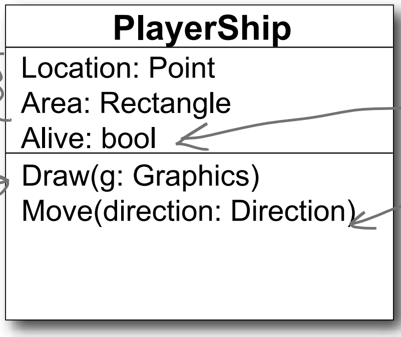
Each invader knows its type. So if you give its InvaderImage() method a number for its animation cell, it can return a Bitmap that's got the right graphic in it.

Remember, you can download these graphics from http://www.headfirstlabs.com/hfcsharp/.

The player's ship can move and die

The `PlayerShip` class keeps track of the player's ship. It's similar to the `Invaders` class, but even simpler.

The Location and Area properties are exactly like the ones in the Invader class.



When the ship's hit with a shot, the game sets the ship's Alive property to false. The game then keeps the invaders from moving until the ship resets its Alive property back to true.

The Draw() method just draws the player's ship in the right location—unless the player died, in which case it draws an animation of the ship getting crushed by the shot.

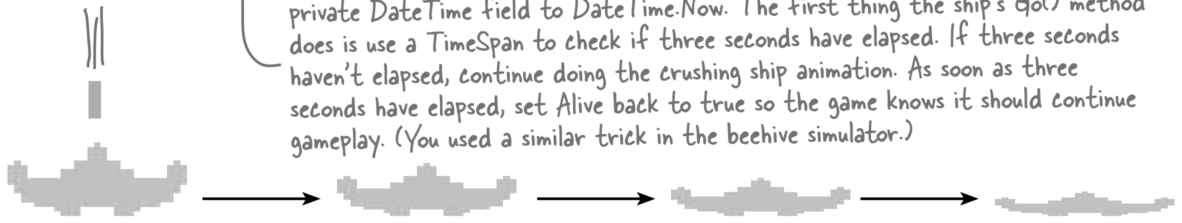
The Move() method takes one parameter, a Direction enum, and moves the player in that direction.

PlayerShip needs to take in a Rectangle with the game's boundaries in its constructor, and make sure the ship doesn't get moved out of the game's boundaries in Move().

Animate the player ship when it's hit

The `Draw()` method should take a `Graphics` object as a parameter. Then it checks its `Alive` property. If it's alive, it draws itself using its `Location` property. If it's dead, then instead of drawing the regular bitmap on the graphics, the `PlayerShip` object uses its private `deadShipHeight` field to animate the player ship slowly getting crushed by the shot. After three seconds of being dead, it should flip its `Alive` property back to `true`.

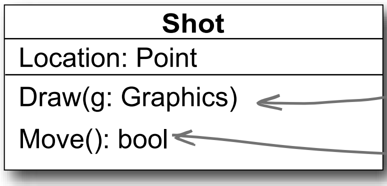
Waiting three seconds is easy—just use the Alive property's set accessor to set a private DateTime field to DateTime.Now. The first thing the ship's Go() method does is use a TimeSpan to check if three seconds have elapsed. If three seconds haven't elapsed, continue doing the crushing ship animation. As soon as three seconds have elapsed, set Alive back to true so the game knows it should continue gameplay. (You used a similar trick in the beehive simulator.)



```
public void Draw(Graphics g) {
  if (!Alive) {
        Reset the deadShipHeight field and draw the ship.
    } else {
        Check the deadShipHeight field. If it's greater than zero, decrease it by 1
        and use DrawImage() to draw the ship a little flatter.
    }
}
```

"Shots fired!"

Game has two lists of Shot objects: one for the player's shots moving up the screen, and one for enemy shots moving down the screen. Shot only needs a few things to work: a Point location, a method to draw the shot, and a method to move. Here's the class diagram:



Draw() handles drawing the little rectangle for this shot. Game will call this every time the screen needs to be updated.

Move() moves the shot up or down, and keeps up with whether the shot is within the game's boundaries.

Here's a start on the Shot class:

```
class Shot {
  private const int moveInterval = 20;
  private const int width = 5;
  private const int height = 15;

  public Point Location { get; private set; }

  private Direction direction;
  private Rectangle boundaries;

  public Shot(Point location, Direction direction,
              Rectangle boundaries) {
    this.Location = location;
    this.direction = direction;
    this.boundaries = boundaries;
  }

  // Your code goes here
}
```

You can adjust these to make the game easier or harder...smaller shots are easier to dodge, faster shots are harder to avoid.

The shot updates its own location in the Move() method, so location can be a read-only automatic property.

Direction is the enum with Up and Down defined.

The game passes the form's display rectangle into the constructor's boundaries parameter so the shot can tell when it's off of the screen.

Your job is to make sure Draw() takes in a Graphics object and draws the shot as a yellow rectangle. Then, Move() should move the shot up or down, and return true if the shot is still within the game boundaries.

Twinkle, twinkle...it's up to you

Here's another hint: start out the project with just a form, a Game class, and Stars class. See if you can get it to draw a black sky with twinkling stars. That'll give you a solid foundation to add the other classes and methods.

The last class you'll need is the `Stars` class. There are 300 stars, and this class keeps up with all of them, causing 5 to display and 5 to disappear every time `Twinkle()` is called.

First, though, you'll need a `struct` for each star:

```
private struct Star {
  public Point point;
  public Pen pen;

  public Star(Point point, Pen pen) {
    this.point = point;
    this.pen = pen;
  }
}
```

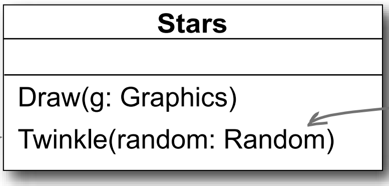
Each star has a point (its location) and a pen (for its color).

All Star does is hold this data...no behavior.

The `Stars` class should keep a `List<Star>` for storing 300 of these `Star structs`. You'll need to build a constructor for `Stars` that populates that list. The constructor will get a `Rectangle` with the display boundaries, and a `Random` instance for use in creating the random `Points` to place each star in a random location.

You can define the Star struct inside Stars.cs, as only Stars needs to use that struct.

Here's the class diagram for `Stars`, with the other methods you'll need:



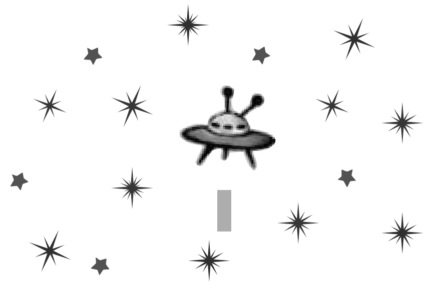
Draw() draws all 300 stars...

...and Twinkle() pulls 5 stars and adds 5 new ones.

Game maintains an instance of Random that all the objects can use.

`Draw()` should draw all the stars in the list, and `Twinkle()` should remove five random stars and add five new stars in their place.

You might also want to create a `RandomPen()` method so you can get a random color for every new star you create. It should return one of the five possible star colors, by generating a number between 0 and 4, and selecting the matching `Pen` object.



And yet there's more to do...

Think the game's looking pretty good? You can take it to the next level with a few more additions:

Add animated explosions
Make each invader explode after it's hit, then briefly display a number to tell the player how many points the invader was worth.

Add a mothership
Once in a while, a mothership worth 250 points can travel across the top of the battlefield. If the player hits it, he gets a bonus.

Add shields
Add floating shields the player can hide behind. You can add simple shields that the enemies and player can't shoot through. Then, if you really want your game to shine, add breakable shields that the player and invaders can blast holes through after a certain number of hits.

Try making the shields last for fewer hits at higher levels of the game.

Add divebombers
Create a special type of enemy that divebombs the player. A divebombing enemy should break formation, take off toward the player, fly down around the bottom of the screen, and then resume its position.

Add more weapons
Start an arms race! Smart bombs, lasers, guided missiles...there are all sorts of weapons that both the player and the invaders can use to attack each other. See if you can add three new weapons to the game.

Add more graphics
You can go to **www.headfirstlabs.com/books/hfcsharp/** to find more graphics files for simple shields, a mothership, and more. We provided blocky, pixelated graphics to give it that stylized '80s look. Can you come up with your own graphics to give the game a new style?

A good class design should let you change out graphics with minimal code changes.

This is your chance to show off! Did you come up with a cool new version of the game? Join the Head First C# forum and claim your bragging rights: www.headfirstlabs.com/books/hfcsharp/