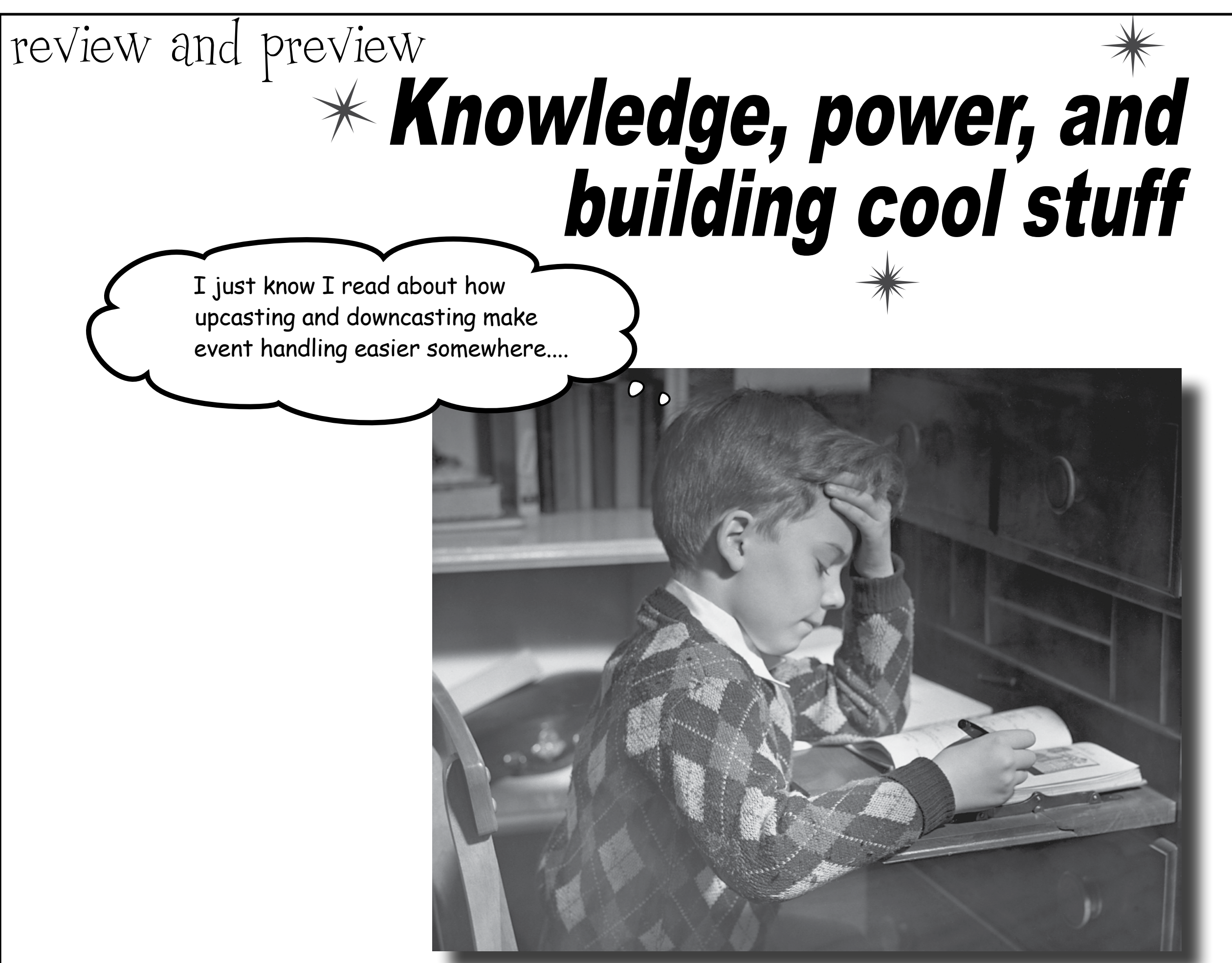
This is the GDI+ Graphics bonus PDF download for Head First C#. GDI+ is the API that allows WinForms programs to draw and print graphics. This PDF divided into three sections. The first section is a review/preview chapter, which was originally published as Chapter 12 in the second edition of Head First C#. In the first scetion, you'll build a beehive simulator. That simulator will serve as the basis for the project that you'll build in the second section, in which you'll learn about the specific GDI+ graphics methods, classes, and structs, and how to use them in your programs.

review and preview

Knowledge, power, and building cool stuff



Learning's no good until you BUILD something.

Until you've actually written working code, it's hard to be sure if you really *get* some of the tougher concepts in C#. In this chapter, we're going to use what we've learned to do just that. We'll also get a preview of some of the new ideas coming up soon. And we'll do all that by building phase I of a **really complex application** to make sure you've got a good handle on what you've already learned from earlier chapters. So buckle up…it's time to **build some software**!

Did you find an error in this PDF? Please submit it using the Errata page for Head First C# (3rd edition) so we can fix it and upload an updated PDF as quickly as possible!

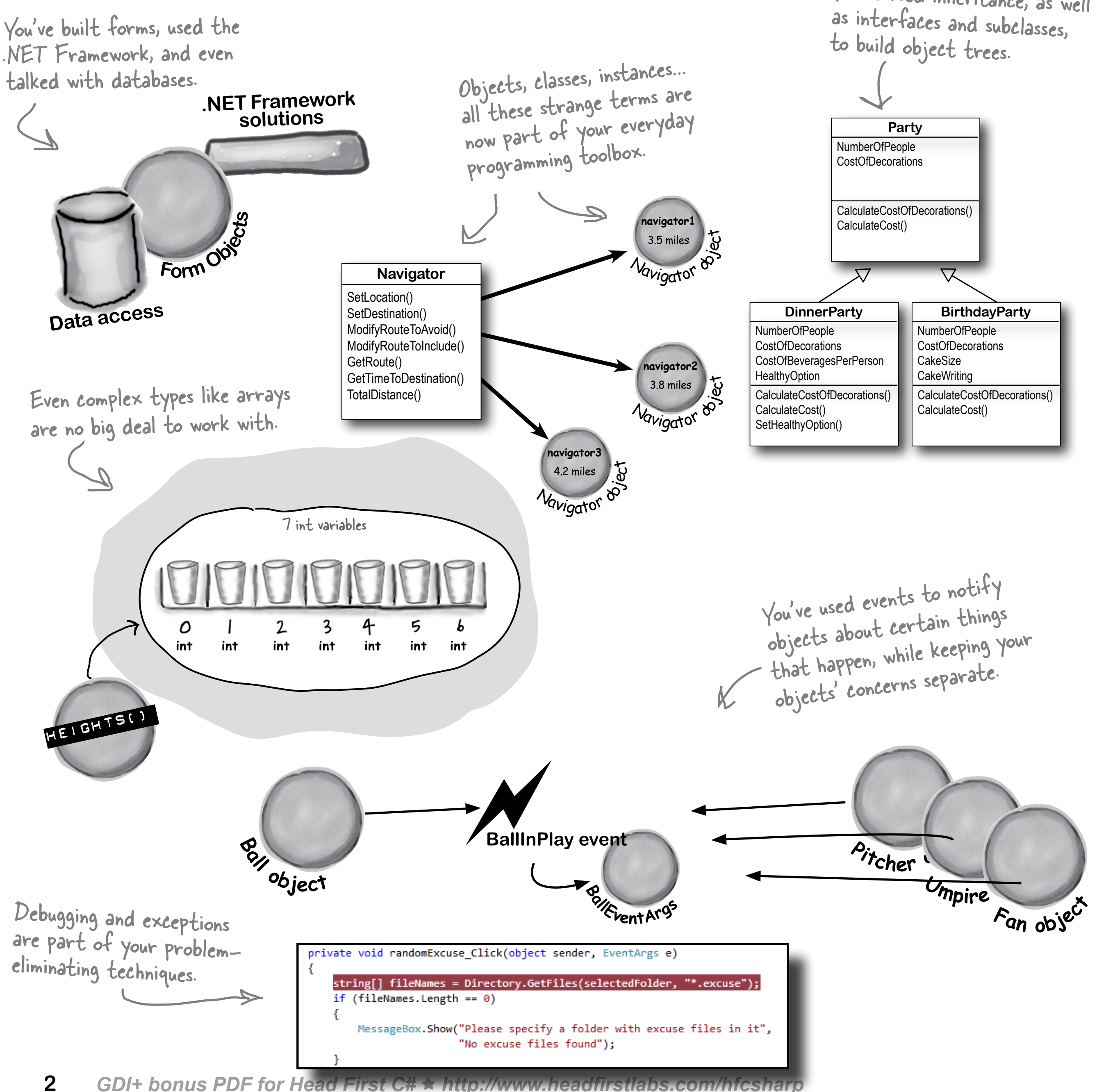
http://www.oreillynet.com/oreilly/authors/errata.csp?b=0636920027812

www.itbook.store/books/9781449343507

You've come a long way, ~~baby~~

[note from human resources: "baby" is no longer politically correct. Please use age-challenged or infant to avoid offending readers.]

We've come a long way since we first used the IDE to help us rescue the Objectville Paper Company. Here's just a few of the things you've done over the last several hundred pages:




```
private void randomExcuse_Click(object sender, EventArgs e)
{
    string[] fileNames = Directory.GetFiles(selectedFolder, "*.excuse");
    if (fileNames.Length == 0)
    {
        MessageBox.Show("Please specify a folder with excuse files in it",
                   "No excuse files found");
    }
```


www.itbook.store/books/9781449343507

We've also become beekeepers

Back in Chapter 6, we built some bee classes. Remember these?

We had different bees doing different jobs...

Beehive Management System
Worker Bee Job Assignments
Worker bee job
Hive maintenance
Shifts
3
Assign this job to a bee
Work the next shift
Report for shift #1
Worker #1 is not working
Worker #2 is not working
Worker #3 is not working
Worker #4 is not working
Total honey consumption: 65.375 units

The queen bee says...
No workers are available to do the job 'Sting patrol'
OK

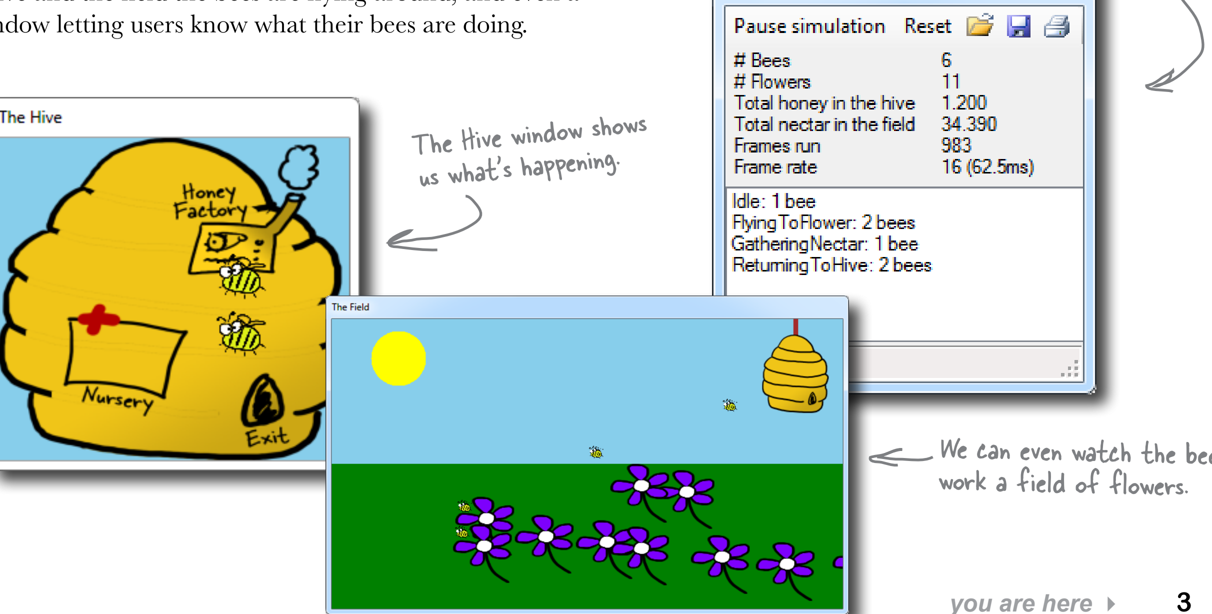
The queen bee says...
The job 'Hive maintenance' will be done in 3 shifts
OK

...and even shifts that the bees worked on.

But we can do a lot better now...

You've learned a lot since Chapter 6, though. So let's start from scratch, and build an **animated beehive simulator** over the next few chapters. We'll end up with a user interface that shows us the hive and the field the bees are flying around, and even a stats window letting users know what their bees are doing.

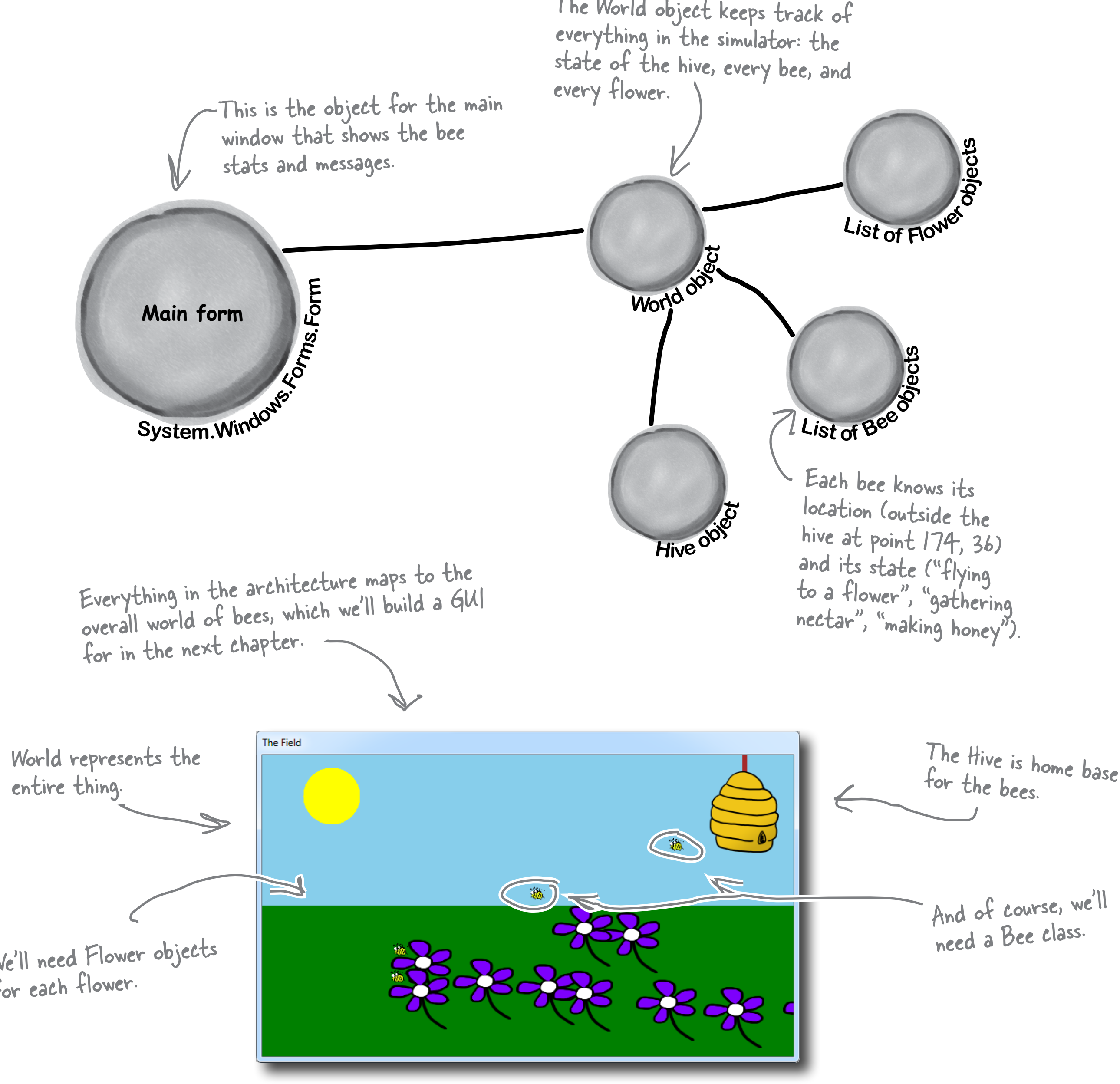
The stats window lets us monitor the simulation in detail.



www.itbook.store/books/9781449343507

The beehive simulator architecture

Here's the architecture for the bee simulator. Even though the simulator will be controlling a lot of different bees, the overall object model is pretty simple.



www.itbook.store/books/9781449343507

Building the beehive simulator

Of course, we've never built anything this complex before, so it's going to take us a couple of chapters to put all the pieces together. Along the way, you'll add timers, LINQ, and a lot of graphical skill to your toolkit.

Here's what you're going to do in this chapter (more to come in the next):

1. **Build a Flower class that ages, produces nectar, and eventually wilts and dies.**
2. **Build a Bee class that has several different states (gathering nectar from a flower, returning to the hive), and knows what to do based on its state.**
3. **Build a Hive class that has an entrance, exit, nursery for new bees, and honey factory for turning collected nectar into honey.**
4. **Build a World class that manages the hive, flowers, and bees at any given moment.**
5. **Build a main form that collects statistics from the other classes and keeps the world going.**

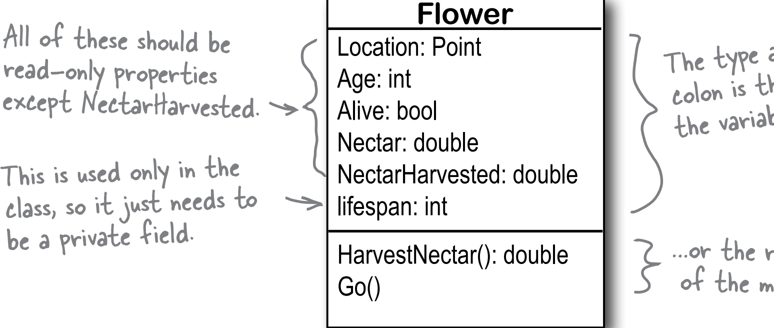
www.itbook.store/books/9781449343507

Let's jump right into some code. First up, we need a `Flower` class. The `Flower` class has a location defined by a point, an age, and a lifespan. As time goes on, the flower gets older. Then, when its age reaches its lifespan, the flower dies. It's your job to put all this into action.

A class "skeleton" is just its field, property, and method declarations, with no implementation.

Write the skeleton code for `Flower`

Below is the class diagram for `Flower`. Write the basic class skeleton. `Location`, `Age`, `Alive`, `Nectar`, and `NectarHarvested` are **automatic properties**. `NectarHarvested` is writable; the other four are **read-only**. For now, leave the methods blank; we'll come back to those in a minute.



Add several constants to the class

We need lots of constants for flowers. Add six to your `Flower` class:

- `LifeSpanMin`, the shortest flower lifespan
- `LifeSpanMax`, the longest flower lifespan
- `InitialNectar`, how much nectar a flower starts with
- `MaxNectar`, how much nectar a flower can hold
- `NectarAddedPerTurn`, how much nectar gets added each time the flower grows older
- `NectarGatheredPerTurn`, how much nectar gets collected during a cycle

FYI, you don't usually show constants in a class diagram.

You should be able to figure out the types for each constant based on their values. Flowers live between 15,000 and 30,000 cycles, and have 1.5 units of nectar when they start out. They can store up to 5 units of nectar. In each cycle of life, a flower adds 0.01 units of nectar, and in a single cycle, 0.3 units can be collected.

Since this simulator will be animated, we'll be drawing it frame by frame. We'll use the words "frame," "cycle," and "turn" interchangeably.

www.itbook.store/books/9781449343507

You'll need to add `using System.Drawing;` to the top of any class file that uses a `Point`.

3 Build the constructor

The constructor for `Flower` should take in a `Point`, indicating the flower's location, and an instance of the `Random` class. You should be able to use those arguments to set the location of the flower, and then set its age to 0, set the flower to alive, and set its nectar to the initial amount of nectar for a flower. Since no nectar has been harvested yet, set that variable correctly, as well. Finally, figure out the flower's lifespan. Here's a line of code to help you:

```
lifeSpan = random.Next(LifeSpanMin, LifeSpanMax + 1);
```

This will only work if you've got your variables and constants named right, as well as the argument to the Flower constructor.

4 Write code for the `HarvestNectar()` method

Every time this method is called, it should check to see if the nectar gathered every cycle is larger than the amount of nectar left. If so, return 0. Otherwise, you should remove the amount collected in a cycle from the nectar the flower has left, and return how much nectar was collected. Oh, and don't forget to add that amount to the `NectarHarvested` variable, which keeps up with the total nectar collected from this particular flower.

Hint: You'll use NectarGatheredPerTurn, Nectar, and NectarHarvested in this method, but nothing else.

5 Write code for the `Go()` method

This is the method that makes the flower go. Assume every time this method is called, one cycle passes, so update the flower's age appropriately. You'll also need to see if the age is greater than the flower's lifespan. If so, the flower dies.

Assuming the flower stays alive, you'll need to add the amount of nectar each flower gets in a cycle. Be sure and check against the maximum nectar your flower can store, and don't overrun that.

The final product will be **animated**, with little pictures of bees flying around. The Go() method will be called once every **frame**, and there will be several frames run per second.

Answers on the next page...try and finish your code and compile it before peeking.

www.itbook.store/books/9781449343507

Exercise Solution

Your job was to build the `Flower` class for our beehive simulator.

Flower
Location: Point Age: int Alive: bool Nectar: double NectarHarvested: double lifespan: int
HarvestNectar(): double Go()

```
class Flower {
    private const int LifeSpanMin = 15000;
    private const int LifeSpanMax = 30000;
    private const double InitialNectar = 1.5;
    private const double MaxNectar = 5.0;
    private const double NectarAddedPerTurn = 0.01;
    private const double NectarGatheredPerTurn = 0.3;
    public Point Location { get; private set; }
    public int Age { get; private set; }
    public bool Alive { get; private set; }
    public double Nectar { get; private set; }
    public double NectarHarvested { get; set; }
    private int lifeSpan;

    public Flower(Point location, Random random) {
        Location = location;
        Age = 0;
        Alive = true;
        Nectar = InitialNectar;
        NectarHarvested = 0;
        lifeSpan = random.Next(LifeSpanMin, LifeSpanMax + 1);
    }

    public double HarvestNectar() {
        if (NectarGatheredPerTurn > Nectar)
            return 0;
        else {
            Nectar -= NectarGatheredPerTurn;
            NectarHarvested += NectarGatheredPerTurn;
            return NectarGatheredPerTurn;
        }
    }

    public void Go() {
        Age++;
        if (Age > lifeSpan)
            Alive = false;
        else {
            Nectar += NectarAddedPerTurn;
            if (Nectar > MaxNectar)
                Nectar = MaxNectar;
        }
    }
}
```

Location, Age, Alive, and Nectar are all readonly automatic properties.

NectarHarvested will need to be accessible to other classes.

Flowers have random lifespans, so the field of flowers doesn't all change at once.

A bee calls HarvestNectar() to get nectar out of a flower. A bee can only harvest a little bit of nectar at a time, so he'll have to sit near the flower for several turns until the nectar's all gone.

As part of the simulator's animation, the Go() method will be called each frame. This makes the flower age just a tiny little bit per frame. As the simulator runs, those tiny bits will add up over time.

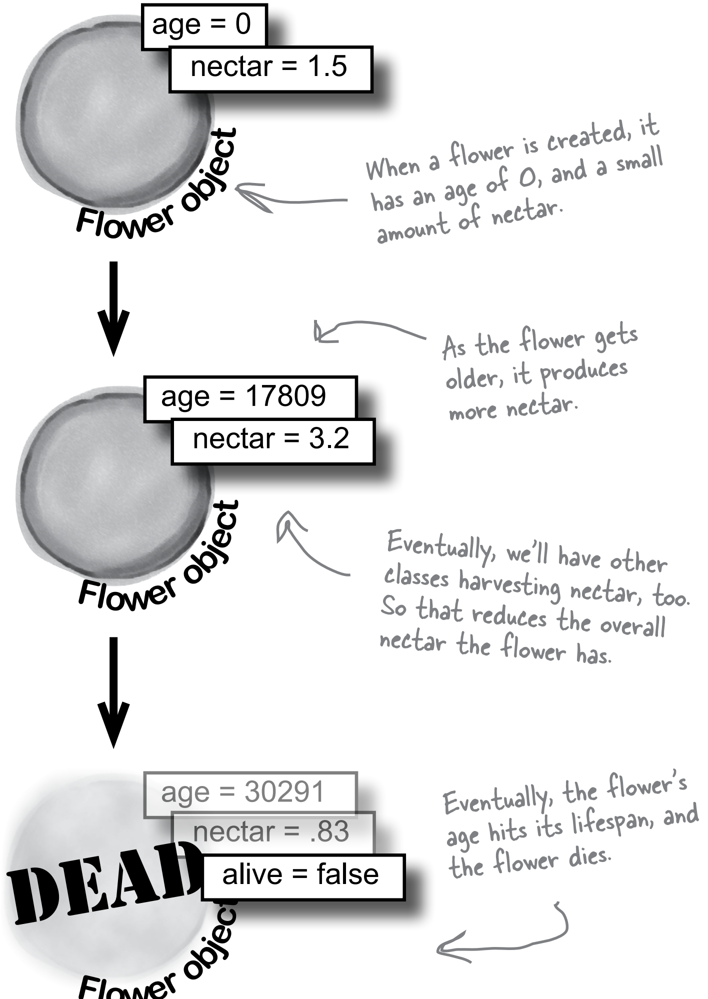
Make sure the flower stops adding nectar after it's dead.

`Point` lives in the `System.Drawing` namespace, so make sure you added `using System.Drawing;` to the top of the class file.

www.itbook.store/books/9781449343507

Life and death of a flower

Our flower goes through a basic turn, living, adding nectar, having nectar harvested, and eventually dying:



there are no Dumb Questions

Q: It doesn't look like NectarHarvested is used anywhere in the class, except where we increment it. What's that variable for?

A: Good catch! We're planning ahead a bit. Eventually, the simulator will keep an eye on flowers, and how much total nectar has been harvested, for our statistics monitor. So leave it in, and our other classes will use it shortly.

Q: Why all the read-only automatic properties?

A: Remember Chapter 5, and hiding our privates? Always a good practice. Flowers can take care of those values, so we've made them read-only. Other objects, like bees and the hive, should be able to read those properties, but not change them. But remember, they're only read-only outside of the class—code inside the class can access the private set accesor.

Q: My code looks different. Did I do something wrong?

A: You might have your code in each method in a different order, but as long as your code ***functions*** the same way as ours does, you'll be OK. That's another aspect of encapsulation: the internals of each class aren't important to other classes, as long as each class does what it's supposed to do.

If `Go()` increases the age of the `Flower` by 1, and the lifespan range is between 15,000 and 30,000, that means `Go()` will get called at least 15,000 times for each flower before it dies. How would you handle calling the method that many times? What if there are 10 flowers? 100? 1,000?

www.itbook.store/books/9781449343507

Now we need a Bee class

With flowers ready to be harvested, we need a Bee class. Below is the basic code for Bee. The Bee knows its age, whether or not it's in the hive, and how much nectar it can collect. We've also added a method to move the bee toward a specific destination point.

```
class Bee {
  private const double HoneyConsumed = 0.5;
  private const int MoveRate = 3;
  private const double MinimumFlowerNectar = 1.5;
  private const int CareerSpan = 1000;

  public int Age { get; private set; }
  public bool InsideHive { get; private set; }
  public double NectarCollected { get; private set; }

  private Point location;
  public Point Location { get { return location; } }

  private int ID;
  private Flower destinationFlower;

  public Bee(int id, Point location) {
    this.ID = id;
    Age = 0;
    this.location = location;
    InsideHive = true;
    destinationFlower = null;
    NectarCollected = 0;
  }

  public void Go(Random random) {
    Age++;
  }
```

Like the Flower class, there are several bee-specific constants we need to define.

MinimumFlowerNectar is how the bee figures out which flowers are eligible for harvesting.

We used a backing field for location. If we'd used an automatic property, MoveTowardsLocation() wouldn't be able to set its members directly ("Location.X -= MoveRate").

Each bee will be assigned its own **unique** ID number.

A bee needs an ID and an initial location.

Bees start out inside the hive, they don't have a flower to go to, and they don't have any nectar.

We'll have to add a lot more code to Go() before we're done, but this will get us started.

www.itbook.store/books/9781449343507

Here we used Math.Abs() to calculate the absolute value of the difference between the destination and the current location.

```
private bool MoveTowardsLocation(Point destination) {
    if (Math.Abs(destination.X - location.X) <= MoveRate &&
        Math.Abs(destination.Y - location.Y) <= MoveRate)
        return true;

    if (destination.X > location.X)
        location.X += MoveRate;
    else if (destination.X < location.X)
        location.X -= MoveRate;

    if (destination.Y > location.Y)
        location.Y += MoveRate;
    else if (destination.Y < location.Y)
        location.Y -= MoveRate;

    return false;
}
```

If the bee reached its destination, the method returns true; otherwise, it returns false.

This method starts by figuring out if we're already within our MoveRate of being at the destination.

If we're not close enough, then we move toward the destination by our move rate.

We return false, since we're not yet at the destination point. We need to keep moving.

The MoveTowardsLocation() destination moves the bee's current location by **changing the X and Y values** of its location field. It returns true if the bee's reached its destination.

Bees have lots of things they can do. Below is a list. Create a new enum that `Bee` uses called `BeeState`. You should also create a read-only automatic property called `CurrentState` for each `Bee` to track that bee's state. Set a bee's initial state to idle, and in the `Go()` method, add a switch statement that has an option for each item in the enum.

The enum item	What the item means
Idle	The bee isn't doing anything
FlyingToFlower	The bee's flying to a flower
GatheringNectar	The bee's gathering nectar from a flower
ReturningToHive	The bee's heading back to the hive
MakingHoney	The bee's making honey
Retired	The bee's hung up his wings

www.itbook.store/books/9781449343507

Bees have lots of things they can do. Below is a list. Create a new enum that `Bee` uses called `BeeState`. You should also create a private `currentState` field for each `Bee` to track that bee's state. Set a bee's initial state to idle, and in the `Go()` method, add a switch statement that has an option for each item in the enum.

```
enum BeeState {
  Idle,
  FlyingToFlower,
  GatheringNectar,
  ReturningToHive,
  MakingHoney,
  Retired
}
```

Here's the enum with all the different bee states.

```
class Bee {
  // constant declarations
  // variable declarations

  public BeeState CurrentState { get; private set; }

  public Bee(int ID, Point initialLocation) {
    this.ID = ID;
    Age = 0;
    location = initialLocation;
    InsideHive = true;
    CurrentState = BeeState.Idle;
    destinationFlower = null;
    NectarCollected = 0;
  }
```

We also need a variable to track the state of each bee.

The bee starts out idle.

Did you remember to add `using System.Drawing;` to the top of the class file (because it uses `Point`)?

www.itbook.store/books/9781449343507

We've filled out a few of the states. It's OK if you didn't come up with this code, but go ahead and add it in now.

```
public void Go(Random random) {
    Age++;
    switch (CurrentState) {
      case BeeState.Idle:
        if (Age > CareerSpan) {
          CurrentState = BeeState.Retired;
        } else {
          // What do we do if we're idle?
        }
        break;
      case BeeState.FlyingToFlower:
         // move towards the flower we're heading to
         break;
      case BeeState.GatheringNectar:
        double nectar = destinationFlower.HarvestNectar();
        if (nectar > 0)
          NectarCollected += nectar;
        else
          CurrentState = BeeState.ReturningToHive;
        break;
      case BeeState.ReturningToHive:
        if (!InsideHive) {
          // move towards the hive
        } else {
          // what do we do if we're inside the hive?
        } break;
      case BeeState.MakingHoney:
        if (NectarCollected < 0.5) {
          NectarCollected = 0;
          CurrentState = BeeState.Idle;
        } else {
          // once we have a Hive, we'll turn the nectar into honey
        }
        break;
      case BeeState.Retired:
        // Do nothing! We're retired!
        break;
    }
  }
}
```

Here's the switch() statement to handle each bee's state.

If the age reaches the bee's lifespan, the bee retires. But he'll finish the current job before he does.

We'll fill this code in a bit later.

Here, we harvest nectar from the flower we're working....

...and if there's nectar left, add it to what we've already collected...

...but if there's no nectar left, head for the hive.

You should have each of these states covered.

Returning to the hive is different based on whether we're already in the hive or not.

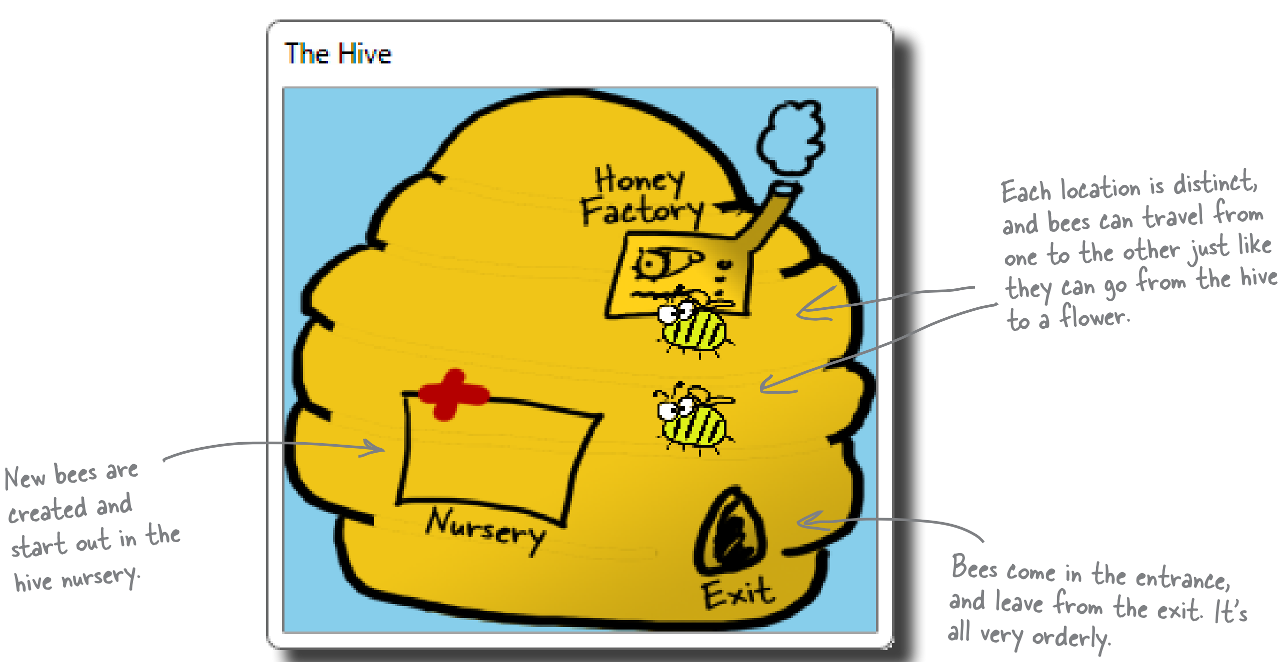
The bee adds half a unit of nectar to the honey factory at a time. If there's not enough nectar to add, the factory can't use it so the bee just discards it.

www.itbook.store/books/9781449343507

P. A. H. B. (Programmers Against Homeless Bees)

We've got bees, and flowers full of nectar. We need to write code so the bees can collect nectar, but before that happens, where do the bees get created in the first place? And where do they take all that nectar? That's where a `Hive` class comes in.

The hive isn't just a place for bees to come back to, though. It has several locations within it, all with different points in the world. There's the entrance and the exit, as well as a nursery for birthing more bees and a honey factory for turning nectar into honey.



The hive runs on honey

The other big part that the hive plays is keeping up with how much honey it has stored up. It takes honey for the hive to keep running, and if new bees need to be created, that takes honey, too. On top of that, the honey factory has to take nectar that bees collect and turn that into honey. For every unit of nectar that comes in, .25 units of honey can be created.

Think about this for a second...as time passes, the hive uses honey to run, and to create more bees. Meanwhile, other bees are bringing in nectar, which gets turned into honey, which keeps things going longer.

It's up to you (with some help) to model all of this in the simulator code.

www.itbook.store/books/9781449343507

It's up to you to write the code for `Hive`.

Hive
Honey: double locations: Dictionary<string, Point> beeCount: int
InitializeLocations() AddHoney(Nectar: double): bool ConsumeHoney(amount: double): bool AddBee(random: Random) Go(random: Random) GetLocation(location: string): Point

1 Write the skeleton code for `Hive`

Like we did with the `Flower` class, you should start with a basic skeleton for `Hive`. The class diagram is shown to the right. Make `Honey` a read-only automatic property, `locations` should be private, and `beeCount` is only used internally, so can be a private field.

2 Define the constants for the `Hive`

You need a constant for the initial number of bees (6), the amount of honey the hive starts with (3.2), the maximum amount of honey the hive can store (15), the ratio of units of nectar produced from units of honey (.25), the maximum number of bees (8), and the minimum honey required for the hive to birth new bees (4).

You'll have to figure out good names for each, as well as the types. For types, don't just think about initial values, but also the values these constants will be used with. Doubles pair best with other doubles, and ints with other ints.

3 Write the code to work with `Locations`

First, write the `GetLocation()` method. It should take in a string, look up that string in the `locations` dictionary, and return the associated point. If it's not there, throw an `ArgumentException`.

Then, write the `InitializeLocations()` method. This method should set up the following locations in the hive:

- Entrance, at (600, 100)
- Nursery, at (95, 174)
- HoneyFactory, at (157, 98)
- Exit, at (194, 213)

Each of these maps to a location within the 2D space that our hive takes up. Later on, we'll have to make sure the simulator makes the hive cover all these points.

In this simulation, we're just assuming one hive, with fixed points. If you wanted multiple hives, you might make the points relative to the hive, instead of the overall world.

4 Build the `Hive` constructor

When a hive is constructed, it should set its honey to the initial amount of honey all hives have. It should set up the locations in the hive, and also create a new instance of `Random`. Then, `AddBee()` should be called—passing in the `Random` instance you just created—once for each bee that starts out in the hive.

AddBee() needs a Random object because it adds a random value to the Nursery location—that way the bees don't start on top of each other.

www.itbook.store/books/9781449343507

Exercise Solution

Your job was to start building the Hive class.

Make sure you add "using System.Drawing;" because this code uses Point.

You might have different names for your constants. That's OK, as long as you're consistent in the rest of your code.

We made MaximumHoney a double, since it can range from InitialHoney (3.2) to this value. Since InitialHoney will need to be a double, it's best to make this a double, too.

```
class Hive {
  private const int InitialBees = 6;
  private const double InitialHoney = 3.2;
  private const double MaximumHoney = 15.0;
  private const double NectarHoneyRatio = .25;
  private const double MinimumHoneyForCreatingBees = 4.0;
  private const int MaximumBees = 8;

  private Dictionary<string, Point> locations;
  private int beeCount = 0;

  public double Honey { get; private set; }

  private void InitializeLocations() {
    locations = new Dictionary<string, Point>();
    locations.Add("Entrance", new Point(600, 100));
    locations.Add("Nursery", new Point(95, 174));
    locations.Add("HoneyFactory", new Point(157, 98));
    locations.Add("Exit", new Point(194, 213));
  }

  public Point GetLocation(string location) {
    if (locations.Keys.Contains(location))
       return locations[location];
    else
       throw new ArgumentException("Unknown location: " + location);
  }

  public Hive() {
    Honey = InitialHoney;
    InitializeLocations();
    Random random = new Random();
    for (int i = 0; i < InitialBees; i++)
      AddBee(random);
  }

  public bool AddHoney(double nectar) { return true; }
  public bool ConsumeHoney(double amount) { return true; }
  private void AddBee(Random random) { }
  public void Go(Random random) { }
}
```

Remember dictionaries? Ours stores a location, keyed with a string value.

Don't forget to create a new instance of Dictionary, or this won't work.

The rest of this method is pretty straightforward.

This method protects other classes from working with our locations dictionary and changing something they shouldn't. It's an example of encapsulation.

You should have called AddBee() once for each bee that a hive starts with.

We don't have code for these yet, but you should have built empty methods as placeholders.

You could also throw a NotImplementedException in any method you haven't implemented yet. That's a great way to keep track of code you still have to build.

www.itbook.store/books/9781449343507

Real code is built bit by bit

It would be nice if you could write all the code for a single class at one time, compile it, test it, and put it away, and ***then*** start on your next class. Unfortunately, that's almost never possible.

More often than not, you'll write code just the way we are in this chapter: piece by piece. We were able to build pretty much the entire `Flower` class, but when it came to `Bee`, we've still got some work to do (mostly telling it what to do for each state).

And now, with `Hive`, we've got lots of empty methods to fill in. Plus, we haven't hooked any `Bees` up to the `Hive`. And there's still that nagging problem about how to call the `Go()` method in all these objects thousands of times....

First you design, then you build

We started out the project knowing exactly what we wanted to build: a beehive simulator. And we know a lot about how the bees, flowers, hive, and world all work together. That's why we started out with the **architecture**, which told us how the classes would work with each other. Then we could move on to each class, designing them individually.

Projects always go a lot more smoothly if you have a good idea of what you're building **before** you start building it. That seems pretty straightforward and common-sense. But it makes all the difference in the final product.

www.itbook.store/books/9781449343507

Filling out the Hive class

Let's get back to the Hive class, and fill in a few of those missing methods:

```
class Hive {
  // constant declarations
  // variable declarations

  // InitializeLocations()
  // GetLocation()
  // Hive constructor

  public bool AddHoney(double nectar) {
    double honeyToAdd = nectar * NectarHoneyRatio;
    if (honeyToAdd + Honey > MaximumHoney)
        return false;
    Honey += honeyToAdd;
    return true;
  }
  public bool ConsumeHoney(double amount) {
    if (amount > Honey)
      return false;
    else {
      Honey -= amount;
      return true;
    }
  }
  private void AddBee(Random random) {
    beeCount++;
    int r1 = random.Next(100) - 50;
    int r2 = random.Next(100) - 50;
    Point startPoint = new Point(locations["Nursery"].X + r1,
                                 locations["Nursery"].Y + r2);
    Bee newBee = new Bee(beeCount, startPoint);
    // Once we have a system, we need to add this bee to the system
  }
  public void Go(Random random) { }
}
```

First, we figure out how much honey this nectar can be converted to...

...and then see if there's room in the hive for that much more honey.

If there's room, we add the honey to the hive.

This method takes an amount of honey, and tries to consume it from the hive's stores.

If there's not enough honey in the hive to meet the demand, we return false.

If there's enough, remove it from the hive's stores and return true.

This is private... only Hive instances can create bees.

This creates a point within 50 units in both the X and Y direction from the nursery location.

Add a new bee, at the designated location.

We'll finish AddBee() and fill in the Go() method soon....

www.itbook.store/books/9781449343507

The hive's Go() method

We've already written a Go() method for Flower, and a Go() method for Bee (even though we've got some additional code to add in). Here's the Go() method for Hive:

```
public void Go(Random random) {
  if (Honey > MinimumHoneyForCreatingBees)
    AddBee(random);
}
```

The only constraint (at least for now) is the hive must have enough honey to create more bees.

The same instance of Random that got passed to Go() gets sent to the AddBee() method.

Unfortunately, this isn't very realistic. Lots of times in a busy hive, the queen doesn't have time to create more bees. We don't have a QueenBee class, but let's assume that when there's enough honey to create bees, a new bee actually gets created 10% of the time. We can model that like this:

```
public void Go(Random random) {
  if (Honey > MinimumHoneyForCreatingBees
      && random.Next(10) == 1) {
    AddBee(random);
  }
}
```

This is an easy way to simulate a 1 in 10 chance of a bee getting created. It comes up with a random number between 0 and 9. If the number is 1, then create the bee.

there are no Dumb Questions

Q: So the hive can create an infinite number of bees?

A: Right now it can—or, at least, it's got a very large limit—but you're right, that's not very realistic. Later on, we'll come back to this, and add a constraint that only lets so many bees exist in our simulator world at one time.

Q: Couldn't we assign that instance of Random to a property of the class, instead of passing it on to AddBee()?

A: You sure could. Then AddBee could use that property, rather than a parameter passed in. There's not really a right answer to this one; it's up to you.

One reason to leave it out is so that you can save the Random seed—that way you can rerun a specific simulation...if you feel like doing that later!

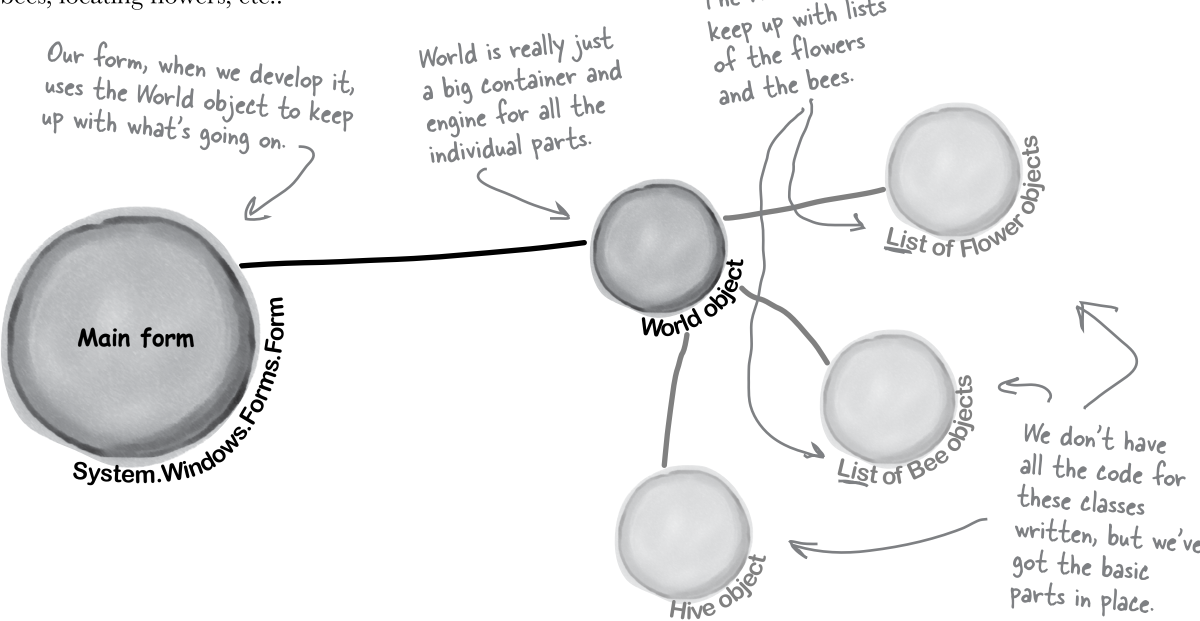
Q: I still don't understand how all of these Go() methods are getting called.

A: That's OK, we're just about to get to that. First, though, we need one more object: the World class, which will keep track of everything that's going on in the hive, track all the bees, and even keep up with flowers.

www.itbook.store/books/9781449343507

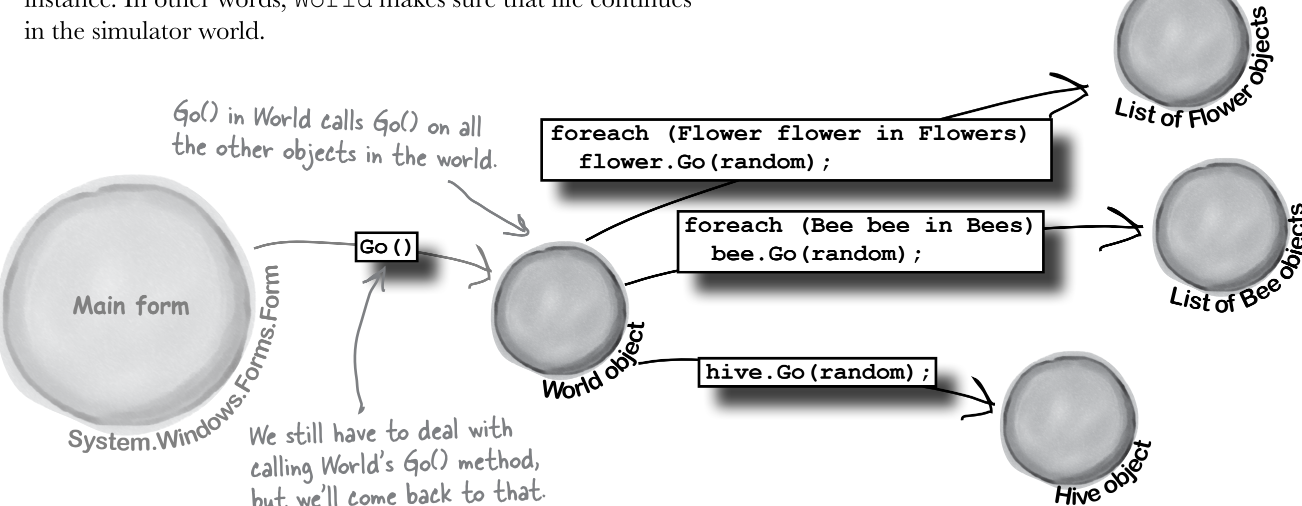
We're ready for the World

With the `Hive`, `Bee`, and `Flower` classes in place, we can finally build the `World` class. `World` handles coordination between all the individual pieces of our simulator: keeping up with all the bees, telling the hive if there is room for more bees, locating flowers, etc.:



The World object keeps everything Go()ing

One of the biggest tasks of the `World` object is, for each turn in the simulator, to call `Go()` on every `Flower`, `Bee`, and `Hive` instance. In other words, `World` makes sure that life continues in the simulator world.



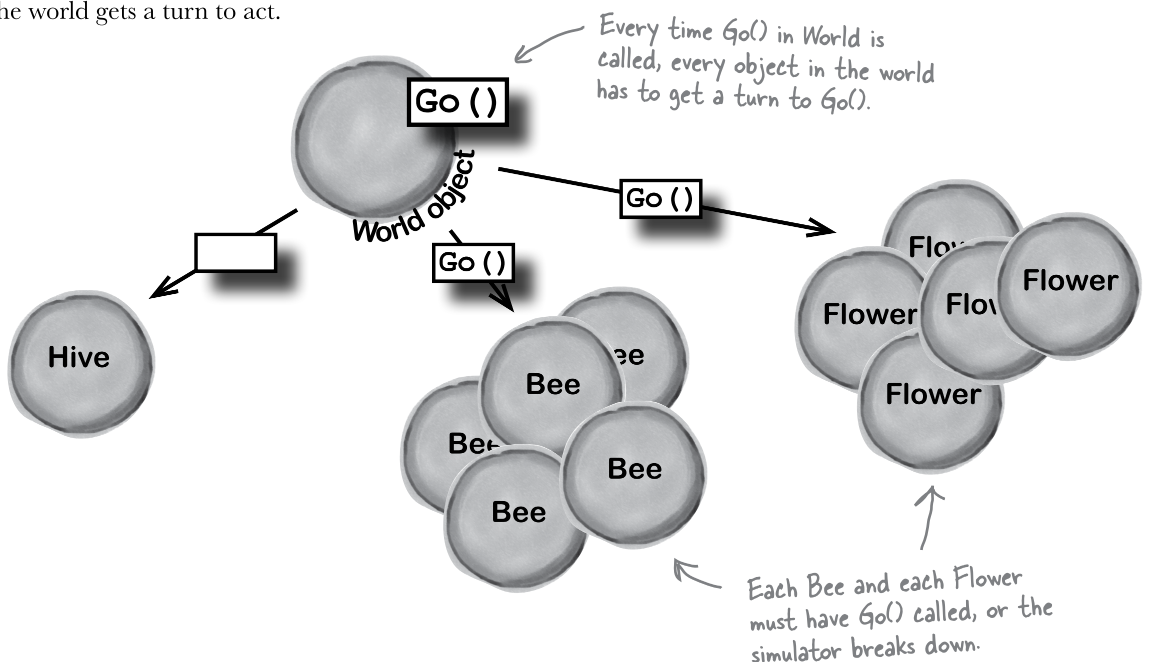
www.itbook.store/books/9781449343507

We're building a turn-based system

Each "turn" will be drawn as a single frame of animation, so the world only needs to change a tiny little bit each turn.

Our Go() methods in each object are supposed to run each **turn**, or **cycle**, of our simulator. A turn in this case just means an arbitrary amount of time: for instance, a turn could be every 10 seconds, or every 60 seconds, or every 10 minutes.

The main thing is that a turn affects every object in the world. The hive ages by one "turn," checking to see if it needs to add more bees. Then each bee takes a turn, moving a very small distance toward its destination or doing one small action, and getting older. Then each flower takes a turn, manufacturing a little nectar and getting older too. And that's what World does: it makes sure that every time its Go() method is called, every object in the world gets a turn to act.



Sharpen your pencil

One of the big object-oriented principles we've been using in the simulator is encapsulation (flip back to Chapter 5 for a refresher). See if you can look over the code we've developed so far and come up with two examples of encapsulation for each class you've built.

Hive	Bee	Flower
1.	1.	1. ....
2.	2.	2. ....

www.itbook.store/books/9781449343507

Here's the code for World

The `World` class is actually one of the simpler classes in our simulator. Here's a starting point for the code. But if you look closely, you'll notice that it's missing a few things (which you'll add in just a minute).

Encapsulation alert!

Take a look at the public Hive, Bees, and Flowers fields. Another class could accidentally reset any of those to null, which would cause serious problems! Can you think of a way to use properties or methods to encapsulate them better?

```
using System.Drawing;
class World {
    private const double NectarHarvestedPerNewFlower = 50.0;
    private const int FieldMinX = 15;
    private const int FieldMinY = 177;
    private const int FieldMaxX = 690;
    private const int FieldMaxY = 290;

    public Hive Hive;
    public List<Bee> Bees;
    public List<Flower> Flowers;

    public World() {
        Bees = new List<Bee>();
        Flowers = new List<Flower>();
        Random random = new Random();
        for (int i = 0; i < 10; i++)
            AddFlower(random);
    }

    public void Go(Random random) {
        Hive.Go(random);

        for (int i = Bees.Count - 1; i >= 0; i--) {
            Bee bee = Bees[i];
            bee.Go(random);
            if (bee.CurrentState == BeeState.Retired)
                Bees.Remove(bee);
        }

        double totalNectarHarvested = 0;
        for (int i = Flowers.Count - 1; i >= 0; i--) {
            Flower flower = Flowers[i];
            flower.Go();
            totalNectarHarvested += flower.NectarHarvested;
            if (!flower.Alive)
                Flowers.Remove(flower);
        }
```

These define the bounds of the field, which is where flowers can live.

Every world has one hive, a list of bees, and a list of flowers.

When we create a new world, we initialize our lists, create a new hive, and then add 10 initial flowers.

This is easy...we just tell the Hive to Go(), passing in a Random instance.

We run through all the current bees and tell them Go().

If a bee's retired, we need to take it out of the world.

We run through each flower and tell it to Go().

We need to keep up with how much nectar's been collected this turn, too. So we get that by summing up the nectar collected from each flower.

Just like bees, we remove any flowers that die during this turn.

www.itbook.store/books/9781449343507

Sharpen your pencil Solution

Here are the ones we came up with. Did you come up with any others?

One of the big object-oriented principles we've been using in the simulator is encapsulation (flip back to Chapter 5 for a refresher). See if you can look over the code we've developed so far and come up with two examples of encapsulation for each class you've built.

Hive

1. The hive's Locations dictionary is private
2. It gives the bees a method to add honey

Bee

1. The bee's location is read-only
2. So is its age. So other classes can't write to them

Flower

1. The flower provides a method to gather nectar
2. And it keeps its alive boolean private

```
        if (totalNectarHarvested > NectarHarvestedPerNewFlower) {
            foreach (Flower flower in Flowers)
                flower.NectarHarvested = 0;
            AddFlower(random);
        }
    }

    private void AddFlower(Random random)
    {
        Point location = new Point(random.Next(FieldMinX, FieldMaxX),
                                   random.Next(FieldMinY, FieldMaxY));
        Flower newFlower = new Flower(location, random);
        Flowers.Add(newFlower);
    }
}
```

Bees **pollinate** flowers as they harvest nectar. Once they've harvested enough nectar from the flowers, they've pollinated enough for the world to add a new flower.

If there's enough nectar in the field, the world adds a new flower.

This handles coming up with a random location in the field...

...and then adding a new flower in that location.

there are no Dumb Questions

Q: Why don't you use `foreach` loops to remove dead flowers and retired bees?

A: Because you **can't remove items** from a collection from inside a `foreach` loop that's iterating on it. If you do, .NET will throw an exception.

Q: OK, then why does each of those `for` loops start at the end of the list and count down to 0?

A: Because each loop needs to preserve the numbering of the list. Let's say you started at the beginning of a list of five flowers, and your loop discovered that one of the flowers in the middle was dead. If it removes the flower at index #3, now the list only has 4 flowers in it, and there's a new flower at index #3—and that flower will end up getting skipped, because the next time through the loop it'll look at index #4.

If the loop starts at the end, then the flower that moves into the empty slot will already have been looked at by the loop, so there's no chance of missing a flower.

www.itbook.store/books/9781449343507

With all four of our core classes in place, we've got some work to do to tie them all together. Follow the steps below, and you should have working `Bee`, `Hive`, `Flower`, and `World` classes. But beware: you'll have to make changes to almost every class, in several places, before you're done.

1. **Update `Bee` to take in a `Hive` and `World` reference.**
   Now that we've got a class for `Hive` and a class for `World`, `Bee` objects need to know about both. Update your code to take in references to a bee's hive and world as parameters to its constructor and save those references for later use.

2. **Update `Hive` to take in a `World` reference.**
   Just as a `Bee` needs to know about its `Hive`, a `Hive` needs to know about its `World`. Update `Hive` to take in a `World` reference in its constructor, and save that reference. You should also update the code in `Hive` that creates new bees to pass into the `Bee` a reference to itself (the `Hive`) and the `World`.

3. **Update `World` to pass itself into a new Hive.**
   Update your `World` class so that when it creates a new `Hive`, it passes in a reference to itself.

STOP! At this point, you should be able to compile all of your code. If you can't, check through it and correct any mistakes before continuing on.

4. **Place an upper limit on the bees that `Hive` can create.**
   The `Hive` class has a `MaximumBees` constant that determines how many bees the `Hive` can support (inside and outside the hive, combined). Now that the `Hive` has access to the `World`, you should be able to enforce that constraint.

5. **When the `Hive` creates bees, let the `World` know.**
   The `World` class uses a `List` of bee objects to keep up with all the bees that exist. When the `Hive` creates a new `Bee`, make sure that `Bee` gets added to the overall list that the `World` is keeping up with.

Hint: Look at code near where you create or add bees. There are two places where code related to this occurs in Hive, so be careful.

www.itbook.store/books/9781449343507

there are no Dumb Questions

Q: Why did you throw an exception in the `Hive` class's `GetLocation()` method?

A: Because we needed a way to deal with bad data passed into the parameter. The hive has a few locations, but the parameter to `GetLocations()` can pass any string. What happens if there's a bug in the program that causes an invalid string (like an empty string, or the name of a location that's not in the locations dictionary) to be sent as the parameter? What should the method return?

When you've got an invalid parameter and it's not clear what to do with it, it's always a good idea to throw a new `ArgumentException`. Here's how the `GetLocation()` method does it:

```
throw new ArgumentException(
      "Unknown location: " + location);
```

This statement causes the `Hive` class to throw an `ArgumentException` with the message "Unknown location:" that contains the location that it couldn't find.

The reason this is useful is that it immediately alerts you if a bad location parameter is passed to the method. And by including the parameter in the exception message, you're giving yourself some valuable information that will help you debug the problem.

Q: What's the point of storing all the locations in a `Point` if we're not drawing anything?

A: Every bee has a location, whether or not you draw it on the screen in that location. The job of the `Bee` object is to keep track of where it is in the world. Each time its `Go()` method is called, it needs to move a very small distance toward its destination.

Now, even though we may not be drawing a picture of the bee yet, the bee still needs to keep track of where it is inside the hive or in the field, because it needs to know if it's arrived at its destination.

Q: Then why use `Point` to store the location, and not something else? Aren't `Points` specifically for drawing?

A: Yes, a `Point` is what all of the visual controls use for their `Location` properties. Plus, it'll come in handy when we do the animation. However, just because .NET uses them that way, that doesn't mean it's not also useful for us to keep track of locations. Yes, we could have created our own `BeeLocation` class with integer fields called X and Y. But why reinvent the wheel when C# and .NET give us `Point` for free?

It's almost always easier to repurpose or extend an existing class that does MOSTLY what you want it to do, rather than creating an all-new class from scratch.

www.itbook.store/books/9781449343507

With all four of our core classes in place, we've got some work to do to tie them all together. Follow the steps below, and you should have working `Bee`, `Hive`, `Flower`, and `World` classes. Here's how we made the changes to put this into place.

1 Update `Bee` to take in a `Hive` and `World` reference.

Now that we've got a class for `Hive` and a class for `World`, `Bee` objects need to know about both. Update your code to take in references to a bee's hive and world in the constructor and save those references for later use.

```
class Bee {
  // existing constant declarations
  // existing variable declarations
  private World world;
  private Hive hive;

  public Bee(int ID, Point InitialLocation, World world, Hive hive) {
    // existing code
    this.world = world;
    this.hive = hive;
  }
}
```

This is pretty straightforward...take these in, assign them to private fields.

2 Update Hive to take in a World reference.

Just as a `Bee` needs to know about its `Hive`, a `Hive` needs to know about its `World`. Update `Hive` to take in a `World` reference in its constructor, and save that reference. You should also update the code in `Hive` that creates new bees to pass into the `Bee` a reference to itself (the `Hive`) and the `World`.

```
class Hive {
  private World world;

  public Hive(World world) {
    this.world = world;
    // existing code
  }
  public void AddBee(Random random) {
    // other bee creation code
    Bee newBee = new Bee(beeCount, startPoint, world, this);
  }
}
```

More basic code...get the reference, set a private field. You want to assign the world FIRST because the rest of the constructor needs to use it.

New bees need a reference to the world, and to the hive, now.

www.itbook.store/books/9781449343507

If you're having trouble getting this running, you can download the code for this exercise (and all the others, too) from:

http://www.headfirstlabs.com/books/hfcsharp/

3 **Place an upper limit on the bees that `Hive` can create.**

The `Hive` class has a `MaximumBees` constant that determines how many bees the `Hive` can support (inside and outside the hive, combined). Now that the `Hive` has access to the `World`, you should be able to enforce that constraint.

```
public void Go(Random random) {
  if (world.Bees.Count < MaximumBees
        && Honey > MinimumHoneyForCreatingBees
        && random.Next(10) == 1) {
    AddBee(random);
  }
}
```

We can use the World object to see how many total bees there are, and compare that to the maximum bees for this hive.

We put that comparison first. If there's no room for bees, no sense in seeing if there's enough honey to create bees.

4 **When the `Hive` creates bees, let the `World` know.**

The `World` class keeps up with all the bees that exist. When the `Hive` creates a new `Bee`, make sure that `Bee` gets added to the overall list that the `World` is keeping up with.

```
private void AddBee(Random random) {
  beeCount++;
  // Calculate the starting point
  Point startPoint = // start the near the nursery
  Bee newBee = new Bee(beeCount, startPoint, world, this);
  world.Bees.Add(newBee);
}
```

We add the new bee to the world's overall bee list.

This demonstrates one of the reasons we need a World reference in the Hive class.

5 **Update `World` to pass itself into a new `Hive`.**

Update your `World` class so that when it creates a new `Hive`, it passes in a reference to itself.

```
public World() {
  Bees = new List<Bee>();
  Flowers = new List<Flower>();
  Hive = new Hive(this);
  Random random = new Random();
    for (int i = 0; i < 10; i++)
      AddFlower(random);
}
```

This passes in the reference to the Hive.

www.itbook.store/books/9781449343507

Giving the bees behavior

The one big piece of code that's missing in our current classes is the Bee's `Go()` method. We were able to code a few of the states earlier, but there are plenty left (`Idle` is incomplete, `FlyingToFlower`, and part of `MakingHoney`).

Let's finish up those remaining states now:

```
public void Go(Random random) {
    Age++;
    switch (CurrentState) {
        case BeeState.Idle:
            if (Age > CareerSpan) {
                CurrentState = BeeState.Retired;
            } else if (world.Flowers.Count > 0
                  && hive.ConsumeHoney(HoneyConsumed)) {
                Flower flower =
                  world.Flowers[random.Next(world.Flowers.Count)];
                if (flower.Nectar >= MinimumFlowerNectar && flower.Alive) {
                    destinationFlower = flower;
                    CurrentState = BeeState.FlyingToFlower;
                }
            }
            break;
        case BeeState.FlyingToFlower:
            if (!world.Flowers.Contains(destinationFlower))
                CurrentState = BeeState.ReturningToHive;
            else if (InsideHive) {
                if (MoveTowardsLocation(hive.GetLocation("Exit"))) {
                    InsideHive = false;
                    location = hive.GetLocation("Entrance");
                }
            }
            else
                if (MoveTowardsLocation(destinationFlower.Location))
                    CurrentState = BeeState.GatheringNectar;
            break;
        case BeeState.GatheringNectar:
            double nectar = destinationFlower.HarvestNectar();
            if (nectar > 0)
                NectarCollected += nectar;
            else
                CurrentState = BeeState.ReturningToHive;
            break;
```

If we're idle, we want to go find another flower to harvest from.

See if there are flowers left, and then consume enough honey to keep on going. Otherwise, we're stuck.

We need another living flower with nectar.

Assuming that all works out, go to the new flower.

Make sure the flower hasn't died as we're heading toward it.

That's why we passed a reference to the hive to the Bee constructor.

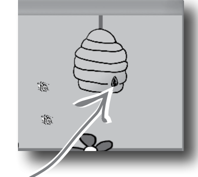
If we can get to the exit, then we're out of the hive. Update our location. Since we're now on the field form, we should fly out near the entrance.

If we're out of the hive, and the flower is alive, get to it and start gathering nectar.

www.itbook.store/books/9781449343507

This is the exit. When the hive stores its "Exit" location, it corresponds to the point on the Hive form that shows the picture of the exit.

This is the entrance. When the bees fly back to the hive, they fly toward the entrance of the hive on the field form.



That's why the location dictionary stores two separate "Exit" and "Entrance" locations.

```
        case BeeState.ReturningToHive:
            if (!InsideHive) {
                if (MoveTowardsLocation(hive.GetLocation("Entrance"))) {
                    InsideHive = true;
                    location = hive.GetLocation("Exit");
                }
            }
            else
                if (MoveTowardsLocation(hive.GetLocation("HoneyFactory")))
                    CurrentState = BeeState.MakingHoney;
            break;
        case BeeState.MakingHoney:
            if (NectarCollected < 0.5) {
                NectarCollected = 0;
                CurrentState = BeeState.Idle;
            }
            else
                if (hive.AddHoney(0.5))
                    NectarCollected -= 0.5;
                else
                    NectarCollected = 0;
            break;
        case BeeState.Retired:
            // Do nothing! We're retired!
            break;
    }
}
```

If we've made it to the hive, update our location and the insideHive status.

If we're already in the hive, head to the honey factory.

Try and give this nectar to the hive.

If the hive could use the nectar to make honey...

...remove it from the bee.

If the hive's full, AddHoney() will return false, so the bee just dumps the rest of the nectar so he can fly out on another mission.

Once the bee's retired, he just has to wait around until the Hive removes him from the list. Then he's off to Miami!

BRAIN POWER

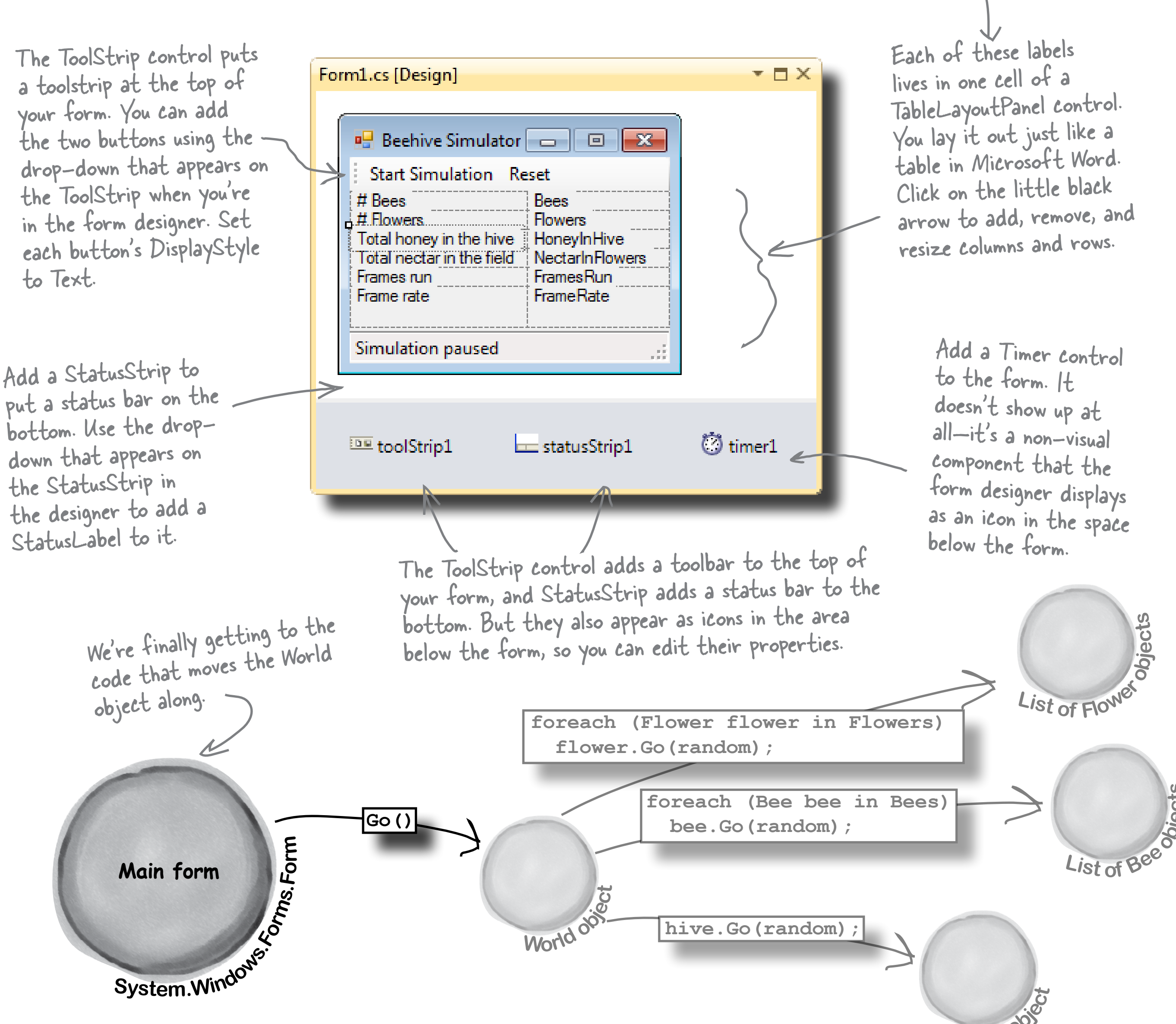
Suppose you wanted to change the simulator so it took two turns to reach a flower, and two turns to go from a flower back to the hive. Without writing any code, which ***methods*** of which classes would you have to change to put this new behavior into place?

www.itbook.store/books/9781449343507

The main form tells the world to Go()

OK, so you know that the world advances by one frame every time its `Go()` method is called. But what calls that `Go()` method? Why, the main form, of course! Time to lay it out.

Go ahead and add a new form to your project. Make it look like the form below. We're using some new controls, but we'll explain them all over the next several pages.



www.itbook.store/books/9781449343507

We can use World to get statistics

Now we want to update all these controls. But we don't need click handlers for each one; instead, let's use a single method that will update the different statistics in the simulator window (we'll explain framesRun shortly):

This indicates how long passes for a turn...we'll have to send this parameter in from somewhere else, in just a few pages.

```
private void UpdateStats(TimeSpan frameDuration) {
    Bees.Text = world.Bees.Count.ToString();
    Flowers.Text = world.Flowers.Count.ToString();
    HoneyInHive.Text = String.Format("{0:f3}", world.Hive.Honey);
    double nectar = 0;
    foreach (Flower flower in world.Flowers)
        nectar += flower.Nectar;
    NectarInFlowers.Text = String.Format("{0:f3}", nectar);
    FramesRun.Text = framesRun.ToString();
    double milliSeconds = frameDuration.TotalMilliseconds;
    if (milliSeconds != 0.0)
        FrameRate.Text = string.Format("{0:f0} ({1:f1}ms)",
                              1000 / milliSeconds, milliSeconds);
    else
        FrameRate.Text = "N/A";
}
```

Most of this just involves getting data from the world and updating labels.

Be sure you match your label names on the form with your code.

Print the first parameter as a number with no decimals, then a space, then print the second parameter with one decimal followed by the letters "ms" (in parentheses)

The frame rate is the number of frames run per second. We're using a TimeSpan object to store how long it took to run the frame. We divide 1000 by the number of milliseconds it took to run the frame—that gives us the total number of milliseconds it took to run the last frame.

Add this method into Form1.

Whoa! Where did that World object come from...we haven't created that yet, have we? And what's all that time and frame stuff?

We'll talk more about this when we create that TimeSpan object.

This code uses the same String.Format() method you used in the hex dump. But instead of printing in hex using "x2", you use "f3" to display a number with three decimal places.

Let's create a World

You're right, we need to create the World object. Add this line to your form's constructor:

```
public Form1() {
  InitializeComponent();
  world = new World();
}
```

Go ahead and add a private World field to your form called world.

That just leaves all the time-related code. We've always said we needed a way to run Go() in World over and over… sounds like we need some sort of timer.

www.itbook.store/books/9781449343507

Timers fire events over and over again

Take a minute and create a new project so you can see how timers work. Then we'll get back to the simulator and put your new knowledge to work.

Remember how you used a loop to animate the greyhounds? Well, there's a better way to do it. A **timer** is an especially useful component that triggers an event over and over again, up to a thousand times a second.

1 Create a new project with a timer and three buttons

You don't have to close your current project—just pop open a new instance of Visual Studio and start up a new project. Drag a timer and three buttons onto the form. Click on the timer icon at the bottom of the designer and set its Interval property to 1000. That number is measured in milliseconds—it tells the timer to fire its tick event once a second.

You can also just double-click on the Timer icon to add the event handler instead of using the Properties window.

2 Open the IDE's Properties window and click on the `Events` button.

(Remember, the Events button looks like a lightning bolt, and it lets you manage the events for any of your form's controls.) The timer control has exactly one event, Tick. **Click on the Timer** icon in the designer, then **double-click on its row** in the Events page and the IDE will create a new event handler method for you and hook it up to the property automatically.

Properties
timer1 System.Windows.Forms.Timer
Tick | timer1_Tick
Tick
Occurs whenever the specified interval time elapses.

The Events button in the Properties window lets you work with all the events for each of your controls.

The bottom of the window has a description of the event.

The Timer control has one event called Tick. If you double-click here, the IDE creates an event handler method for you automatically.

3 Add code to the `Tick` event and to your buttons

Here's some code that will help you get a sense of how the timer works:

```
private void timer1_Tick(object sender, EventArgs e) {
    Console.WriteLine(DateTime.Now.ToString());
}
private void toggleEnabled_Click(object sender, EventArgs e) {
    if (timer1.Enabled)
        timer1.Enabled = false;
    else
        timer1.Enabled = true;
}
private void startTimer_Click(object sender, EventArgs e){
    timer1.Start();
    Console.WriteLine("Enabled = " + timer1.Enabled);
}
private void stopTimer_Click(object sender, EventArgs e) {
    timer1.Stop();
    Console.WriteLine("Enabled = " + timer1.Enabled);
}
```

This statement writes the current date and time to the output. Check the output window to make sure the tick event is fired once a second (every 1000 milliseconds).

These buttons let you play with the Enabled property and the Start() and Stop() methods. The first one switches Enabled between true and false, and the other two call the Start() and Stop() methods.

The timer's Enabled property starts and stops the timer.

The timer's Start() method starts the timer and sets Enabled to true. The Stop() method stops the timer and sets Enabled to false.

www.itbook.store/books/9781449343507

The timer's using an event handler behind the scenes

The timer's Tick event is an average, everyday event handler, just like the ones to handle button clicks.

How do C# and .NET tell the timer what to do every tick? How does the `timer1_Tick()` method get run every time your timer ticks? Well, we're back to **events** and **delegates**, just like we talked about in the last chapter. Use the IDE's "Go To Definition" feature to remind yourself how the `EventHandler` delegate works:

Right-click on your timer1 variable and select "Go To Definition"

The "Go To Definition" feature will cause the IDE to automatically jump to the location in the code where the `timer1` variable is defined. The IDE will jump you to the code it created to add `timer1` as a property in the `Form1` object in `Form1.Designer.cs`. Scroll up in the file until you find this line:

```
this.timer1.Tick += new System.EventHandler(this.timer1_Tick);
```

This is the Tick event of your timer control. You've set this to occur every 1000 milliseconds.

Here's one of the System's delegates: the basic event handler. It's a delegate...a pointer to one or more methods.

Here's the method you just wrote, timer1_Tick(). You're telling the delegate to point to that method.

5 **Now right-click on EventHandler and select "Go To Definition"**

The IDE will automatically jump to the code that defines `EventHandler`. Take a look at the name of the new tab that it opened to show you the code: "EventHandler [from metadata]". This means that the code to define `EventHandler` isn't in your code. It's built into the .NET Framework, and the IDE generated a "fake" line of code to show you how it's represented:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Each event is of type EventHandler. So our Tick event now points to the timer1_Tick() method.

Here's why every event in C# generally takes an Object and EventArgs parameter—that's the form of the delegate that C# defines for event handling.

What code would you write to run the World's Go() method 10 times a second in our beehive simulator?

www.itbook.store/books/9781449343507

Add a timer to the simulator

Let's add a timer to the simulator. You've already got a timer control, probably called `timer1`. Instead of using the IDE to generate a `timer1_Tick()` method, though, we can wire the timer to an event handler method called `RunFrame()` manually:

DateTime & TimeSpan

.NET uses the DateTime class to store information about a time, and its Now property returns the current date and time. If you want to find the difference between two times, use a TimeSpan object: just subtract one DateTime object from another, and that'll return a TimeSpan object that holds the difference between them.

TimeSpan has properties like Days, Hours, Seconds, and Milliseconds that let you measure the span in different units.

```
public partial class Form1 : Form {
  World world;
  private Random random = new Random();
  private DateTime start = DateTime.Now;
  private DateTime end;
  private int framesRun = 0;

  public Form1() {
    InitializeComponent();
    world = new World();

    timer1.Interval = 50;
    timer1.Tick += new EventHandler(RunFrame);
    timer1.Enabled = false;
    UpdateStats(new TimeSpan());
  }

  private void UpdateStats(TimeSpan frameDuration) {
    // Code from earlier to update the statistics
  }

  public void RunFrame(object sender, EventArgs e) {
    framesRun++;
    world.Go(random);
    end = DateTime.Now;
    TimeSpan frameDuration = end - start;
    start = end;
    UpdateStats(frameDuration);
  }
}
```

You should have a World property from earlier.

These will be used to figure out how long the simulator's been running at any given point.

We want to keep up with how many frames—or turns—have passed.

Run every 50 milliseconds.

We set the handler to our own method, RunFrame().

A second is 1000 milliseconds, so our timer will tick 20 times a second.

Timer starts off.

We also start out by updating stats, with a new TimeSpan (0 time elapsed).

Increase the frame count, and tell the world to Go().

Next, we figure out the time elapsed since the last frame was run.

Finally, update the stats again, with the new time duration.

www.itbook.store/books/9781449343507

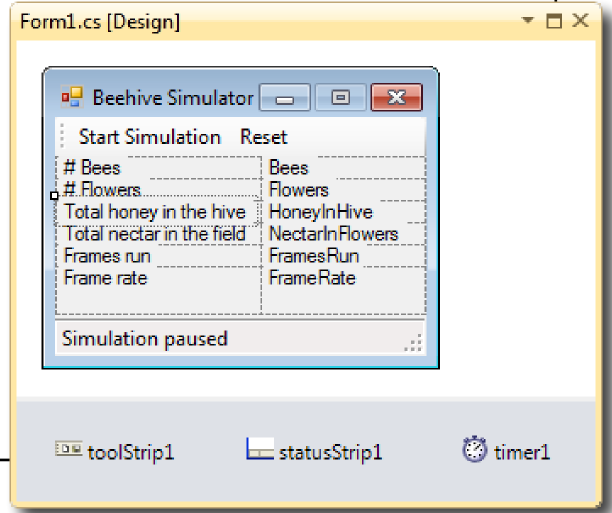
Exercise

Your job is to write the event handlers for the **Start Simulation** and **Reset** buttons in the ToolStrip. Here's what each button should do:

If you haven't dragged a ToolStrip and StatusStrip out of the toolbox and onto your form, do it now.

1. Initially, the first button should read "Start Simulation." Pressing it causes the simulation to start, and the label to change to "Pause Simulation." If the simulation is paused, the button should read, "Resume simulation."

2. The second button should say "Reset." When it's pressed, the world should be recreated. If the timer is paused, the text of the first button should change from "Resume simulation" to "Start Simulation."



There's no single answer to this question—we just want you to think about what's left to do.

Just double-click on a ToolStrip button in the designer to make the IDE add its event handler, just like a normal button.

Sharpen your pencil

What do you think is left to be done in this phase of the simulator? Try running the program. Write down everything you think we still need to take care of before moving on to the graphical stuff.

....

....

....

....

....

....

....

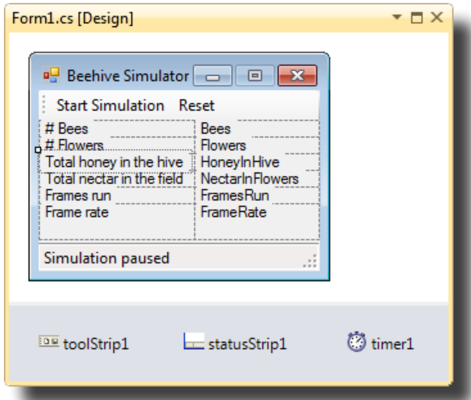
there are no Dumb Questions

Q: We've been using the term "turn," but now you're talking about frames. What's the difference?

A: Semantics, really. We're still dealing in turns: little chunks of time where every object in the world gets to act. But since we'll soon be putting some heavy-duty graphics in place, we've started using "frame," as in a graphical game's frame-rate.

www.itbook.store/books/9781449343507

Your job was to write the event handlers for the Start Simulation and Reset buttons.



The existing code from the form remains unchanged.

```
private void startSimulation_Click(object sender, EventArgs e) {
  if (timer1.Enabled) {
      toolStrip1.Items[0].Text = "Resume simulation";
      timer1.Stop();
  } else {
      toolStrip1.Items[0].Text = "Pause simulation";
      timer1.Start();
  }
}

private void reset_Click(object sender, EventArgs e) {
  framesRun = 0;
  world = new World();
  if (!timer1.Enabled)
      toolStrip1.Items[0].Text = "Start simulation";
}
}
```

Be sure your form's control names match up with what you use in your code.

Toggle the timer, and update the message.

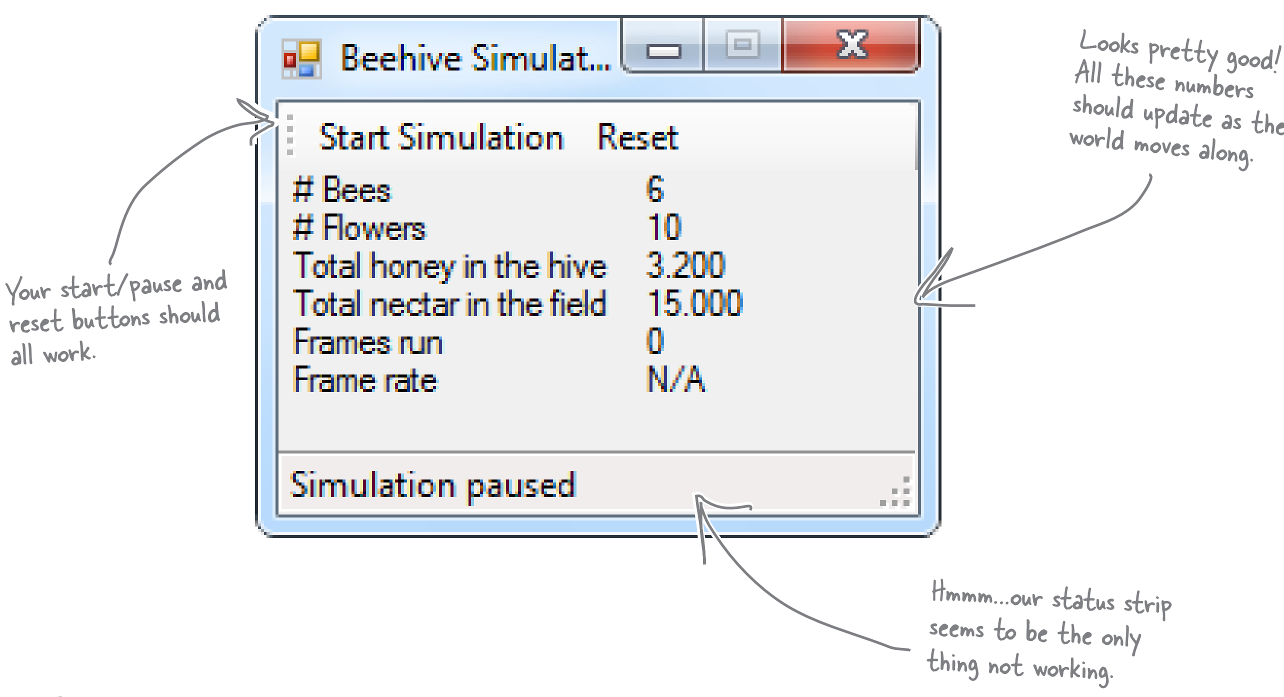
Resetting the simulator is just a matter of recreating the World instance and resetting framesRun.

The only time we need to change the first button's label is if it says, "Resume simulation." If it says, "Pause simulation," it doesn't need to change.

www.itbook.store/books/9781449343507

Test drive

You've done a ton of work. Compile your code, fix any typos, and run the simulator. How's it look?



Here's your chance to put together everything you've learned. We need to allow bees to tell our simulator what they're doing. When they do, we want our simulator to update the status message in the simulator.

This time, it's up to you to not only write most of the code, but to figure out what code you need to write. How can you have a method in your simulator that gets called every time a bee changes its state?

To give you a little help, we've written the method to add to the form. The `Bee` class should call this method any time its state changes:

```
private void SendMessage(int ID, string Message) {
  statusStrip1.Items[0].Text = "Bee #" + ID + ": " + Message;
}
```

* OK, one more hint. You'll need to make changes to all but one of your classes to make this work.

www.itbook.store/books/9781449343507

Your job was to come up with a way for bees to let the simulator know about what they're doing.

Here's what we added to the Bee class.

We used a **callback** to hook each individual bee object up to the form's SendMessage() method.

It uses a delegate called BeeMessage that takes a bee ID and a message. The bee uses it to send messages back to the form.

```
class Bee {
  // all our existing code
  public BeeMessage MessageSender;

  public void Go(Random random) {
    Age++;
    BeeState oldState = CurrentState;
    switch (currentState) {
     // the rest of the switch statement is the same
    }
    if (oldState != CurrentState
        && MessageSender != null)
           MessageSender(ID, CurrentState.ToString());
  }
}
```

If the status of the Bee changed, we call back the method our BeeMessage delegate points to, and let that method know about the status change.

Here are the changes we made to the Hive.

Hive needs a delegate too, so it can pass on the methods for each bee to call when they're created in AddBee().

```
class Hive {
  // all our existing code
  public BeeMessage MessageSender;

  public Hive(World world, BeeMessage MessageSender) {
    this.MessageSender = MessageSender;
    // existing constructor code
  }

  public void AddBee(Random random) {
    // existing AddBee() code
    Bee newBee = new Bee(beeCount, startPoint, world, this);
    newBee.MessageSender += this.MessageSender;
    world.Bees.Add(newBee);
}
```

AddBee() now has to make sure that each new bee gets the method to point at.

www.itbook.store/books/9781449343507

```
public delegate void BeeMessage(int ID, string Message);
```

BeeMessage is our delegate. It's also a match with the SendMessage() method we wrote in the form. Add it to its own file called BeeMessage.cs—it should be in the namespace, but outside of any class.

The World class required some changes as well.

```
class World {
  // all our existing code

  public World(BeeMessage messageSender) {
    Bees = new List<Bee>();
    Flowers = new List<Flower>();
    Hive = new Hive(this, messageSender);
    Random random = new Random();
    for (int i = 0; i < 10; i++)
        AddFlower(random);
  }
}
```

World doesn't need to have a delegate of its own. It just passes on the method to call to the Hive instance.

Last but not least, here's the updated form. Anything not shown stayed the same.

```
public partial class Form1 : Form {
  // variable declarations

  public Form1() {
    InitializeComponent();
    world = new World(new BeeMessage(SendMessage));
    // the rest of the Form1 constructor
  }

  private void reset_Click(object sender, EventArgs e) {
    framesRun = 0;
    world = new World(new BeeMessage(SendMessage));
    if (!timer1.Enabled)
        toolStrip1.Items[0].Text = "Start simulation";
  }

  private void SendMessage(int ID, string Message) {
    statusStrip1.Items[0].Text = "Bee #" + ID + ": " + Message;
  }
}
```

We create a new delegate from the Bee class (make sure you declared BeeMessage public), and point it at our SendMessage() method.

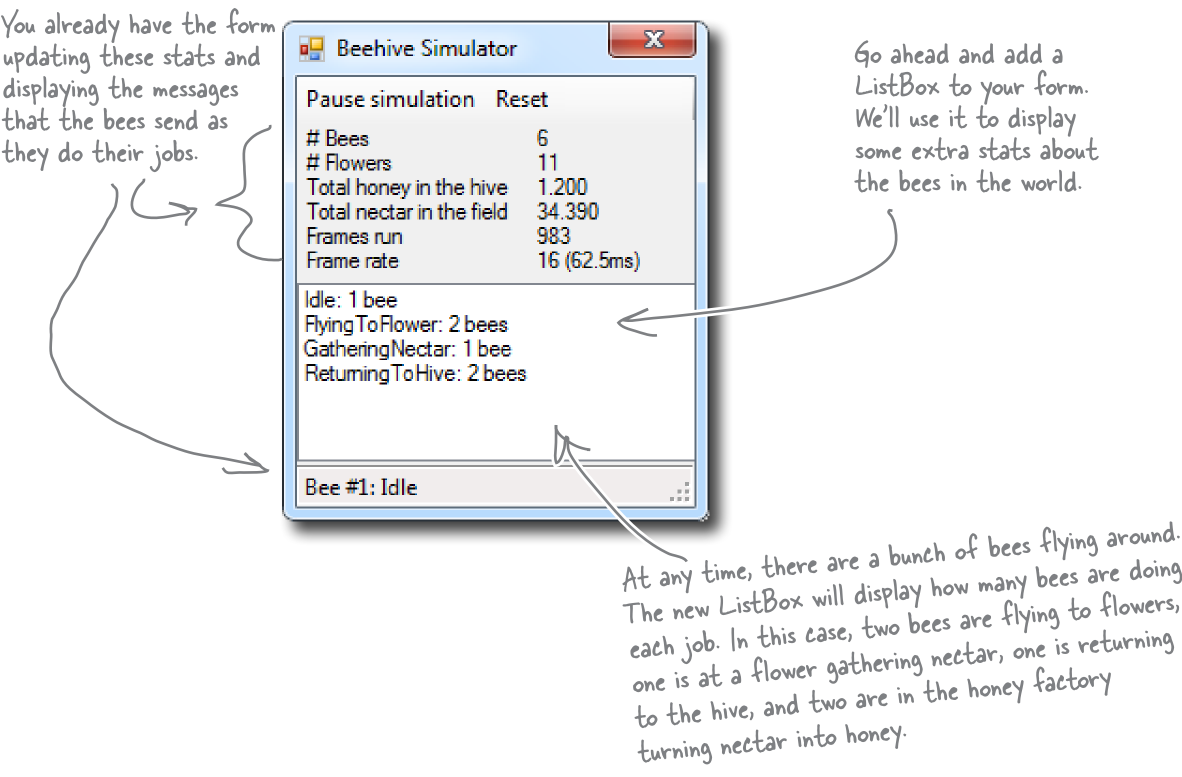
Same thing here...create the world with the method for bees to call back.

This is the method we gave you...be sure to add it in, too.

www.itbook.store/books/9781449343507

Let's work with groups of bees

Your bees should be buzzing around the hive and the field, and your simulation should be running! How cool is that? But since we don't have the visual part of the simulator working yet—that's what we're doing in the next chapter—all the information we have so far is the messages that the bees are sending back to the main form with their callbacks. So let's add more information about what the bees are doing.

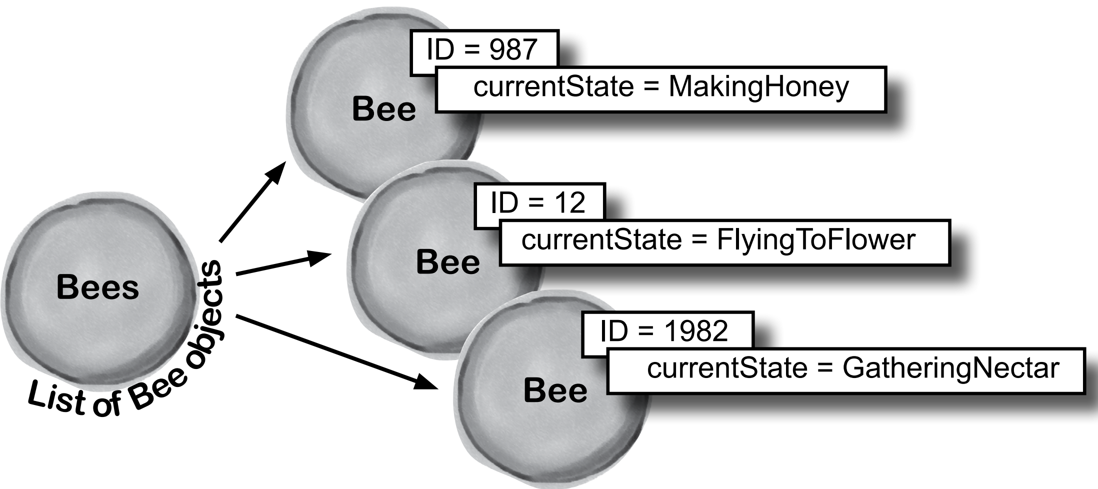


You know enough to gather the information you'd need to populate that `ListBox`—take a minute and think through how that would work. But it's a little more complex than it seems at first. What would you need to do to figure out how many bees are in each of the various `Bee.State` states?

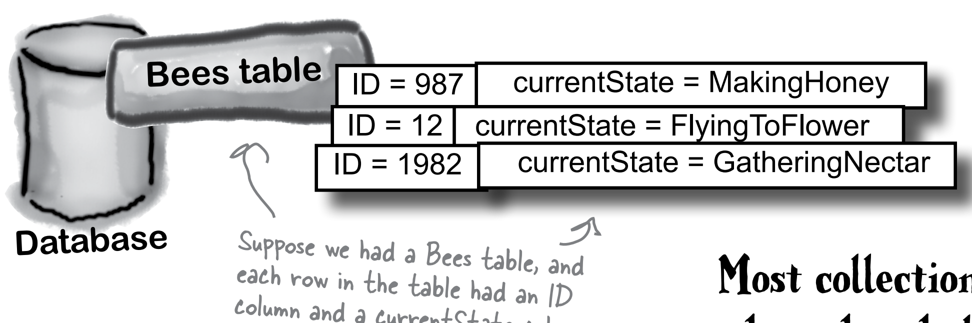
www.itbook.store/books/9781449343507

A collection collects...DATA

Our bees are stored in a `List<Bee>`, which is one of the collection types. And collection types really just store data…a lot like a database does. So each bee is like a row of data, complete with a state, and ID, and so on. Here's how our bees look as a collection of objects:



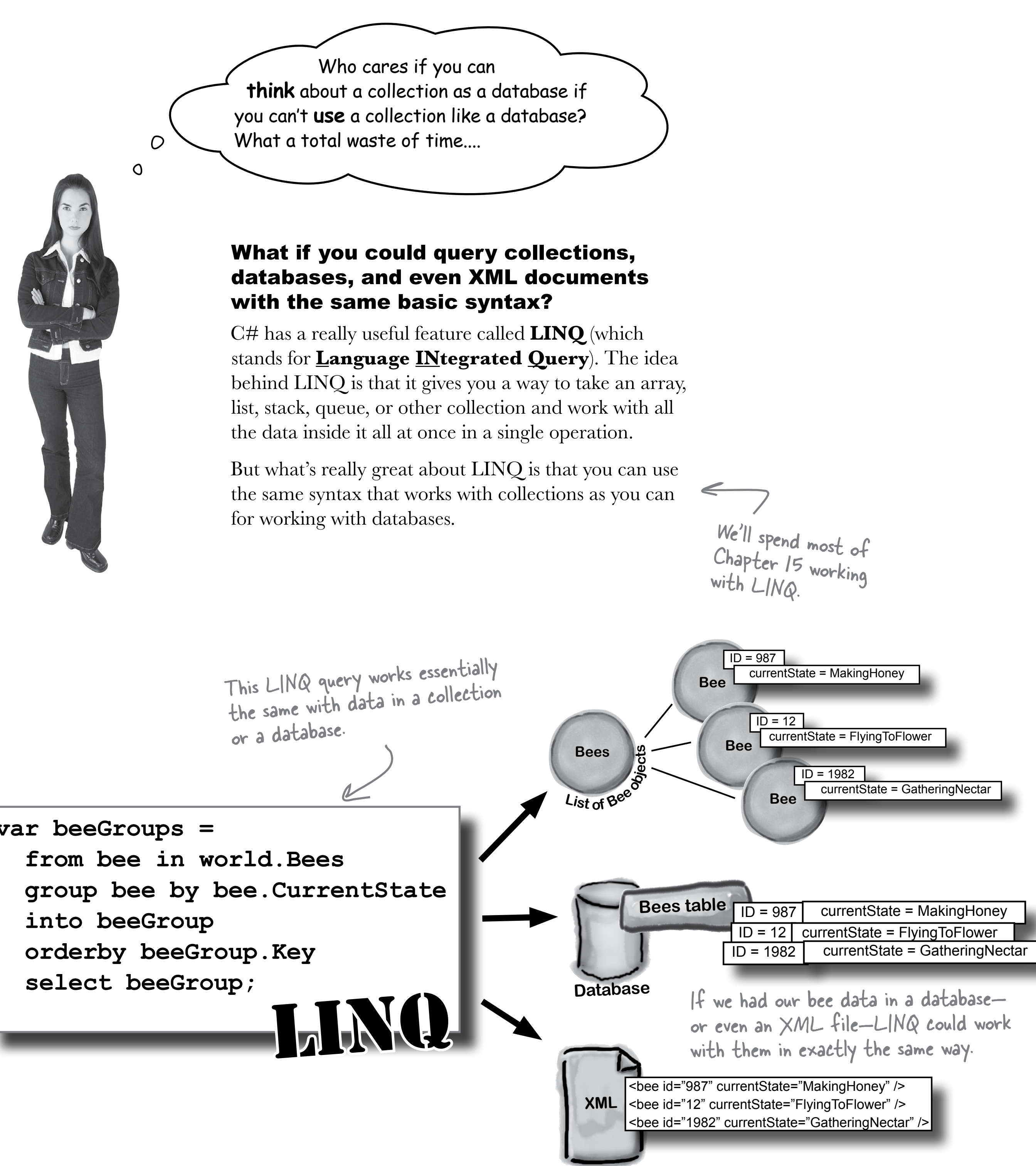
There's a lot of data in the `Bee` objects' fields. You can *almost* think of a collection of objects the same way you think of rows in a database. Each object holds data in its fields, the same way each row in a database holds data in its columns.



Suppose we had a Bees table, and each row in the table had an ID column and a currentState column.

Most collections—especially when they hold objects—can be thought of as data stores, just like a database.

www.itbook.store/books/9781449343507



What if you could query collections, databases, and even XML documents with the same basic syntax?

C# has a really useful feature called **LINQ** (which stands for **Language INtegrated Query**). The idea behind LINQ is that it gives you a way to take an array, list, stack, queue, or other collection and work with all the data inside it all at once in a single operation.

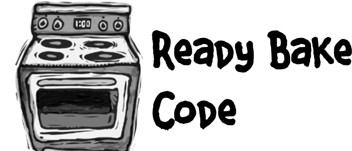
But what's really great about LINQ is that you can use the same syntax that works with collections as you can for working with databases.

```
var beeGroups =
  from bee in world.Bees
  group bee by bee.CurrentState
  into beeGroup
  orderby beeGroup.Key
  select beeGroup;
```


www.itbook.store/books/9781449343507

LINQ makes working with data in collections and databases easy

We're going to spend an entire chapter on LINQ before long, but we can use LINQ and some Ready Bake Code to add some extra features to our simulator. Ready Bake Code is code you should type in, and it's OK if you don't understand it all. You'll learn how it all works in Chapter 15.




```
private void SendMessage(int ID, string Message) {
    statusStrip1.Items[0].Text = "Bee #" + ID + ": " + Message;
    var beeGroups =
      from bee in world.Bees
      group bee by bee.CurrentState into beeGroup
      orderby beeGroup.Key
      select beeGroup;
    listBox1.Items.Clear();
    foreach (var group in beeGroups) {
        string s;
        if (group.Count() == 1)
            s = "";
        else
            s = "s";
        listBox1.Items.Add(group.Key.ToString() + ": "
                         + group.Count() + " bee" + s);
        if (group.Key == BeeState.Idle
         && group.Count() == world.Bees.Count()
         && framesRun > 0) {
            listBox1.Items.Add("Simulation ended: all bees are idle");
            toolStrip1.Items[0].Text = "Simulation ended";
            statusStrip1.Items[0].Text = "Simulation ended";
            timer1.Enabled = false;
        }
    }
}
```

This is a LINQ query. It takes all the bees in the Bees collection, and groups them by their CurrentState property.

The group's Key is the bee's CurrentState, so that's the order the states will be displayed on the form.

Make sure this matches the list box control's name on your form.

beeGroups is from the LINQ query. We can count the members, and iterate over them.

This bit of code makes sure it says, "1 bee" and "3 bees", keeping the plural right.

Finally, add the group status (its key) and count to the list box.

Here's another nice feature. Since we know how many bees are idle...

...we can see if ALL bees are idle. If so, the hive's out of honey, so let's stop the simulation.

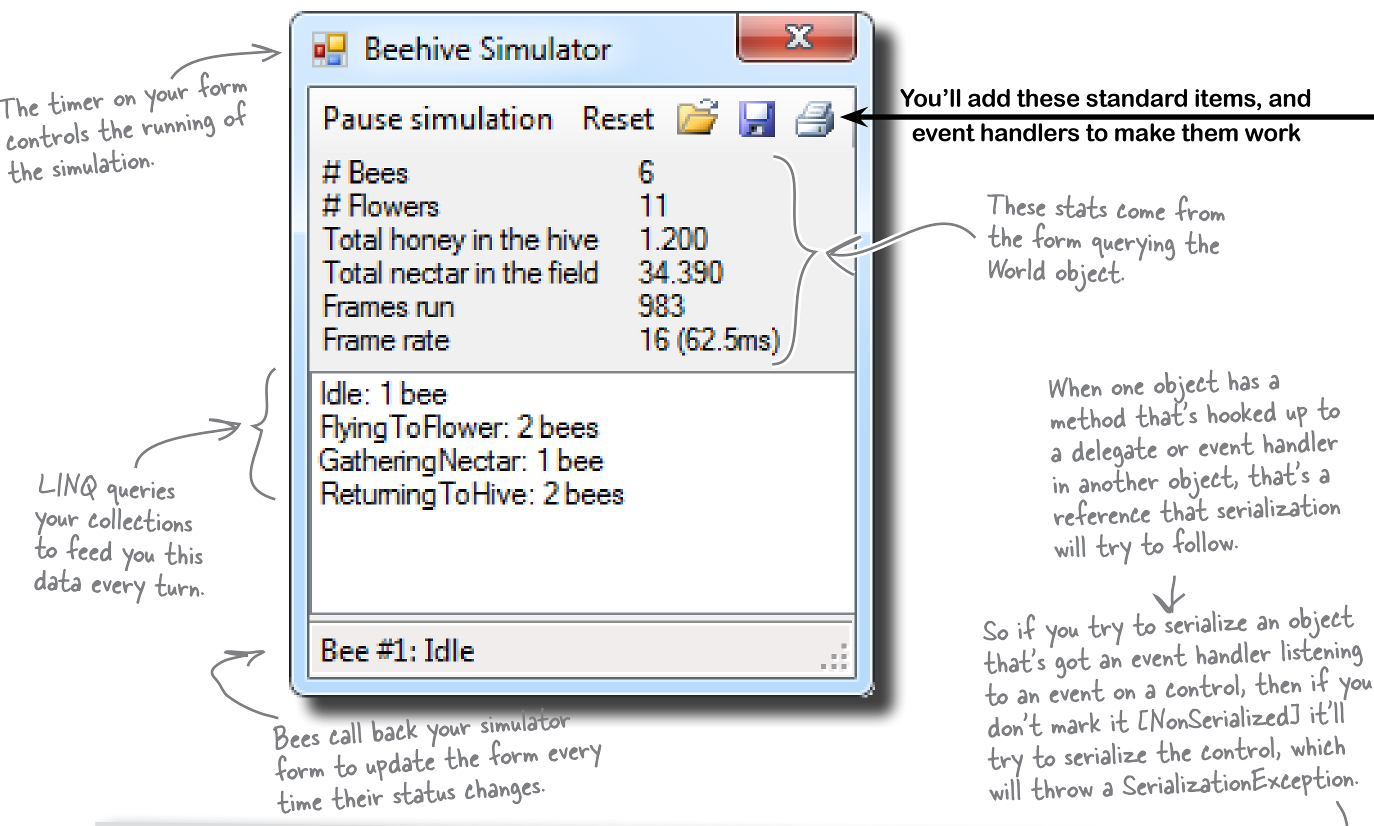
We'll learn a lot more about LINQ in upcoming chapters.

You don't need to memorize LINQ syntax or try to drill all of this into your head right now. You'll get a lot more practice working with LINQ in Chapter 15.

www.itbook.store/books/9781449343507

Test drive (Part 2)

Go ahead and compile your code and run your project. If you get any errors, double-check your syntax, especially with the new LINQ code. Then, fire up your simulator!



[NonSerialized] keeps data from getting serialized

Sometimes you want to serialize part of an object, not all of it. It might have data that you don't want written to the disk. Let's say you're building a system that a user logs into, and you want to save an object that stores the user's options and settings to a file. You might mark the `password` field with the [NonSerialized] attribute. That way, when you Serialize() the object, it will skip that field.

The [NonSerialized] attribute is especially useful when your object has a reference to an object that is not serializable. For example, if you try to serialize a Form, Seralize() will throw a SerializationException. So if our object has a reference to a Form object, then when you try to serialize it the serializer will follow that link and try to serialize the Form, too...which will throw that exception. But if you mark the field that holds the reference with the [NonSerialized] attribute, then Serialize() won't follow the reference at all.

www.itbook.store/books/9781449343507

One final challenge: Open and Save

We're almost ready to take on graphics, and add some visual eye candy to our simulator. First, though, let's do one more thing to this version: allow loading, saving, and printing of bee statistics.

You'll add the **Print** button now—we'll make it print a status page for the hive in the next chapter.

Add the Open, Save, and Print icons

The ToolStrip control has a really useful feature—it can automatically insert picture buttons for standard icons: new, open, save, print, cut, copy, paste, and help. Just right-click on the ToolStrip icon at the bottom of the Form Designer window and select "**Insert Standard Items**". Then click on the first item—that's the "new" icon—and delete it. Keep the next three items, because they're the ones we need (open, save, and print). After that comes a separator; you can either delete it or move it between the Reset button and the save buton. Then delete the rest of the buttons. Make sure you set its **CanOverflow** property to false (so it doesn't add an overflow menu button to the right-hand side of the toolbar) and its **GripStyle** property to Hidden (so it removes the sizing grip from the left-hand side).

Add the button event handlers

The new standard buttons are named `openToolStripButton`, `saveToolStripButton`, and `printToolStripButton`. Just double-click on them to add their event handlers.

Exercise

Add code to make the save and open buttons work.

1. Make the save button serialize the world to a file. The save button should stop the timer (it can restart it after saving if the simulator was running). It should display a Save dialog box, and if the user specifies a filename then it should serialize the `World` object, and the number of frames that have been run.

When you try to serialize the `World` object, it will throw a `SerializationException` with this message: `Type 'Form1' is not marked as serializable`. That's because the serializer found one of the `BeeMessage` fields and tried to follow it. Since the delegate was hooked up to a field on the form, the serializer tried to serialize the form, too.

Fix this problem by adding the **`[NonSerialized]`** attribute to the **`MessageSender`** fields in the Hive and Bee classes, so .NET doesn't try and serialize the code your delegates point to.

2. Make the open button deserialize the world from a file. Take care of the timer just like in the save button: pop up an Open dialog box, and deserialize the world and the number of frames run from the selected file. Then you can hook up the `MessageSender` delegates again and restart the timer (if necessary).

3. Don't forget about exception handling! Make sure the world is intact if there's a problem reading or writing the file. Consider popping up a human-readable error message indicating what went wrong.

www.itbook.store/books/9781449343507

Your job was to make the Save and Open buttons work.

Don't forget the extra using statements.

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

You'll need to make the World, Hive, Flower, and Bee classes serializable. When you serialize the world, .NET will find its references to Hive, Flower, and Bee objects and serialize them, too.

```
[Serializable]
class World {
[Serializable]
class Hive {
[Serializable]
class Flower {
[Serializable]
class Bee {
```

```
[NonSerialized]
public BeeMessage MessageSender;
```

And make sure the MessageSender fields in the Hive and Bee classes are marked [NonSerialized].

Here's the code for the Save button.

```
private void saveToolStripButton_Click(object sender, EventArgs e) {
    bool enabled = timer1.Enabled;
    if (enabled)
        timer1.Stop();

    SaveFileDialog saveDialog = new SaveFileDialog();
    saveDialog.Filter = "Simulator File (*.bees)|*.bees";
    saveDialog.CheckPathExists = true;
    saveDialog.Title = "Choose a file to save the current simulation";
    if (saveDialog.ShowDialog() == DialogResult.OK) {
        try  {
            BinaryFormatter bf = new BinaryFormatter();
            using (Stream output = File.OpenWrite(saveDialog.FileName)) {
                bf.Serialize(output, world);
                bf.Serialize(output, framesRun);
            }
        }
        catch (Exception ex) {
            MessageBox.Show("Unable to save the simulator file\r\n" + ex.Message,
              "Bee Simulator Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }
    if (enabled)
        timer1.Start();
}
```

We decided to use ".bees" as the extension for simulator save files.

Here's where the world is written out to a file.

Remember, when we serialize World, everything it references gets serialized...all the bees, flowers, and the hive.

After we save the file, we can restart the timer (if we stopped it).

www.itbook.store/books/9781449343507

Here's the code for the Open button.

```
private void openToolStripButton_Click(object sender, EventArgs e) {
    World currentWorld = world;
    int currentFramesRun = framesRun;

    bool enabled = timer1.Enabled;
    if (enabled)
        timer1.Stop();

    OpenFileDialog openDialog = new OpenFileDialog();
    openDialog.Filter = "Simulator File (*.bees)|*.bees";
    openDialog.CheckPathExists = true;
    openDialog.CheckFileExists = true;
    openDialog.Title = "Choose a file with a simulation to load";
    if (openDialog.ShowDialog() == DialogResult.OK) {
        try {
            BinaryFormatter bf = new BinaryFormatter();
            using (Stream input = File.OpenRead(openDialog.FileName)) {
                world = (World)bf.Deserialize(input);
                framesRun = (int)bf.Deserialize(input);
            }
        }
        catch (Exception ex) {
      MessageBox.Show("Unable to read the simulator file\r\n" + ex.Message,
      "Bee Simulator Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
            world = currentWorld;
            framesRun = currentFramesRun;
        }
    }

    world.Hive.MessageSender = new BeeMessage(SendMessage);
    foreach (Bee bee in world.Bees)
        bee.MessageSender = new BeeMessage(SendMessage);
    if (enabled)
        timer1.Start();
}
```

Before opening the file and reading from it, save a reference to the current world and framesRun. If there's a problem, you can revert to these and keep running.

Set up the Open File dialog box and pop it up.

using ensures the stream gets closed.

Here's where we deserialize the world and the number of frames run to the file.

If the file operations throw an exception, we restore the current world and framesRun.

Once everything is loaded, we hook up the delegates and restart the timer.

You'll need to get your simulator up and running before you move on to the next chapter. You can download a working version from the Head First Labs website: www.headfirstlabs.com/books/hfcsharp/

www.itbook.store/books/9781449343507

Pages 49 through 106 of this PDF were originally published as the GDI+ chapter in Head First C#, 2nd Edition. The rest of the PDF is Lab #3, in which you'll build the Invaders arcade game. This is the version from the second edition of Head First C#, which was designed to work with GDI+ graphics.

www.itbook.store/books/9781449343507

controls and graphics

Make it pretty

Sometimes you have to take graphics into your own hands.

We've spent a lot of time relying on controls to handle everything visual in our applications. But sometimes that's not enough—like when you want to **animate a picture**. And once you get into animation, you'll end up **creating your own controls** for your .NET programs, maybe adding a little **double buffering**, and even **drawing directly onto your forms**. It all begins with the **Graphics** object, **bitmap**s, and a determination to not accept the graphics status quo.

www.itbook.store/books/9781449343507

You've been using controls all along to interact with your programs

`TextBoxes`, `PictureBoxes`, `Labels`…you've got a pretty good handle by now on how you can use the controls in the IDE's toolbox. But what do you ***really*** know about them? There's a lot more to a control than just dragging an icon onto your form.

You can create your own controls

The controls in the toolbox are really useful for building forms and applications, but there's nothing magical about them. They're just classes, like the classes that you've been writing on your own. In fact, C# makes it really easy for you to create controls yourself, just by inheriting from the right base class.

Your custom controls show up in the IDE's toolbox

There's also nothing mysterious about the toolbox in the IDE. It just looks in your project's classes and the built-in .NET classes for any controls. If it finds a class that implements the right interface, then it displays an icon for it in the toolbox. If you add your own custom controls, they'll show up in the toolbox, too.

You can create a class that inherits from any of the existing control classes—even if it doesn't have any other code in it—and it'll automatically show up in the toolbox.

You can write code to add controls to your form, and even remove controls, while your program's running

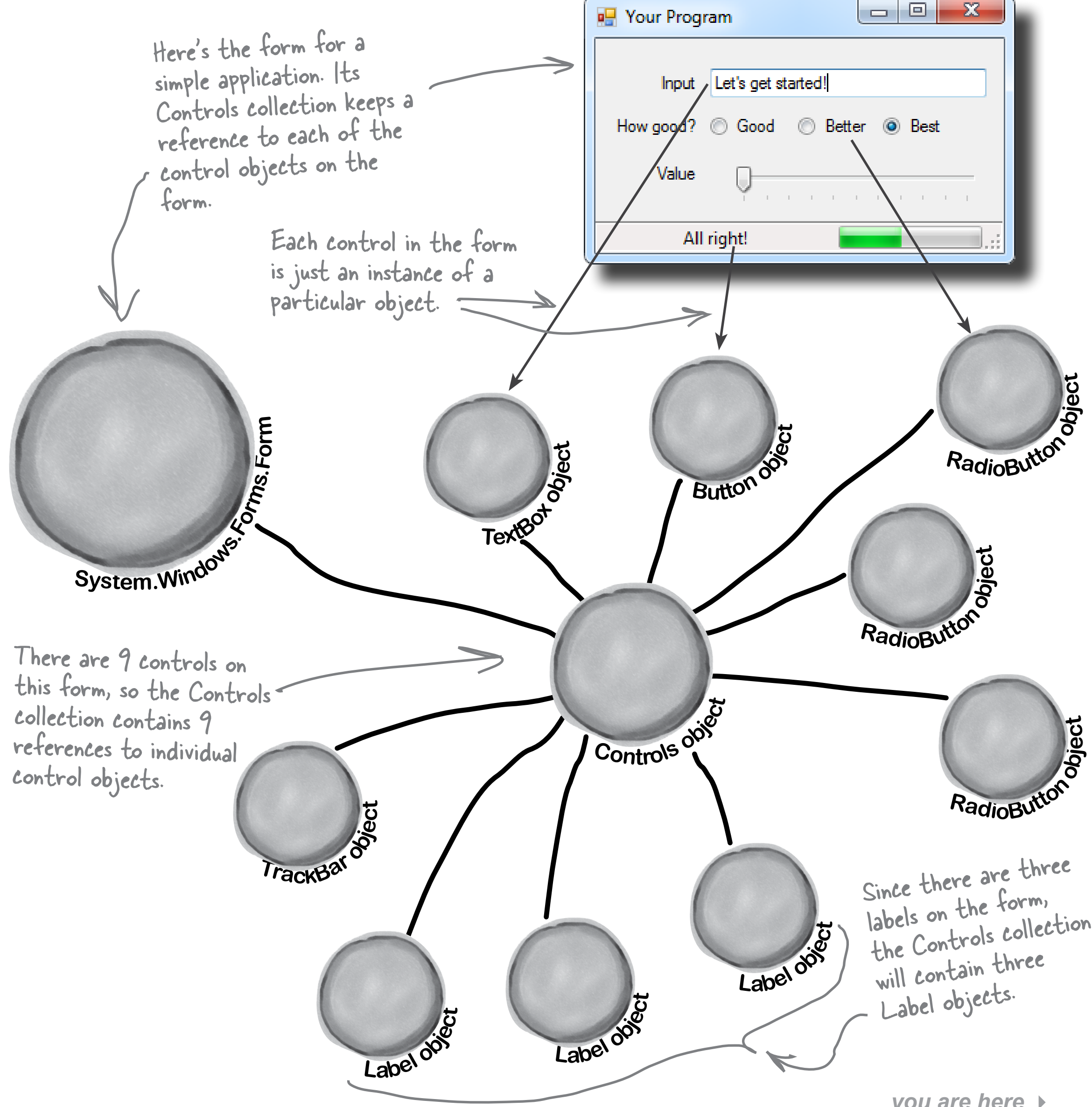
Just because you lay out a form in the IDE's form designer, it doesn't mean that it has to stay like that. You've already moved plenty of `PictureBox` controls around (like when you built the greyhound race). But you can add or remove controls, too. In fact, when you build a form in the IDE, all it's doing is writing the code that adds the controls to the form…which means you can write similar code, and run that code whenever you want.

www.itbook.store/books/9781449343507

Form controls are just objects

You already know how important **controls** are to your forms. You've been using buttons, text boxes, picture boxes, checkboxes, group boxes, labels, and other forms since Chapter 1. Well, it turns out that those controls are just objects, just like everything else you've been working with.

A control is just an object, like any other object—it just happens to know how to draw itself. The `Form` object keeps track of its controls using a special collection called **`Controls`**, which you can use to add or remove controls in your own code.



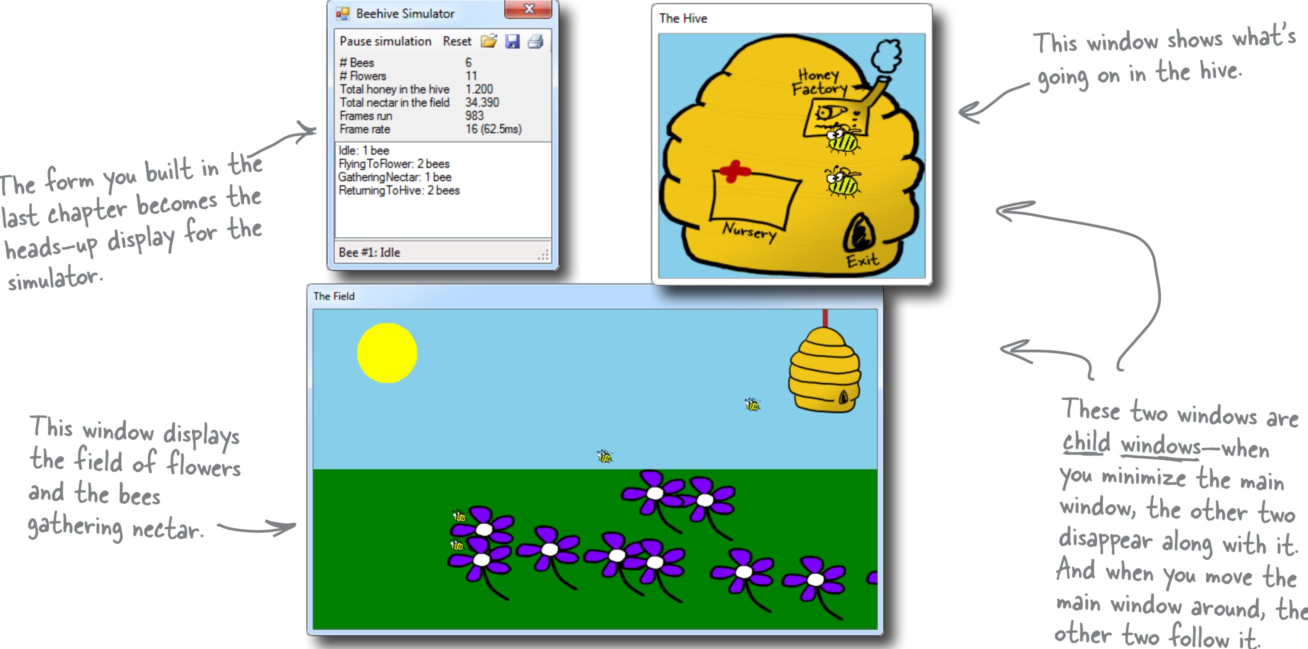
www.itbook.store/books/9781449343507

Use controls to animate the beehive simulator

You've built a cool simulator, but it's not much to look at. It's time to create a really stunning visualization that shows those bees in action. You're about to build a renderer that animates the beehive…and controls are the key.

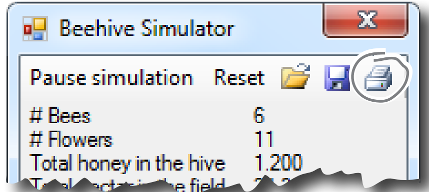
The user interface shows you everything that's going on

Your simulator will have three different windows. You've already built the main "heads-up display" stats window that shows stats about the current simulation and updates from the bees. Now you'll add a window that shows you what's going in inside the hive, and a window that shows the field of flowers where the bees gather nectar.



We'll make the Print button in the stats window work

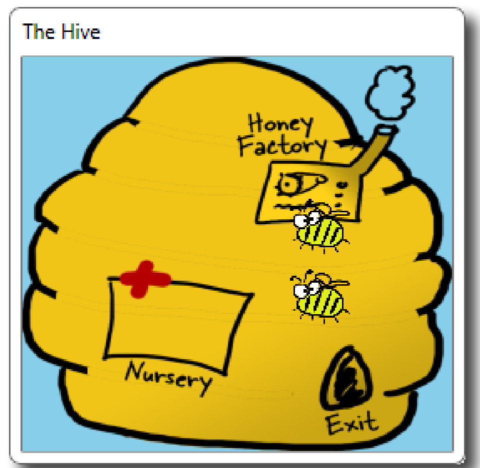
The stats window has working Open and Save buttons, but the Print button doesn't work yet. We'll be able to reuse a lot of the graphics code to get the Print button on the ToolStrip to print an info page about what's going on.



www.itbook.store/books/9781449343507

3 **The hive window shows you what's going on inside the hive**

As the bees fly around the world, you'll need to animate each one. Sometimes they're inside the hive, and when they are, they show up in this window.



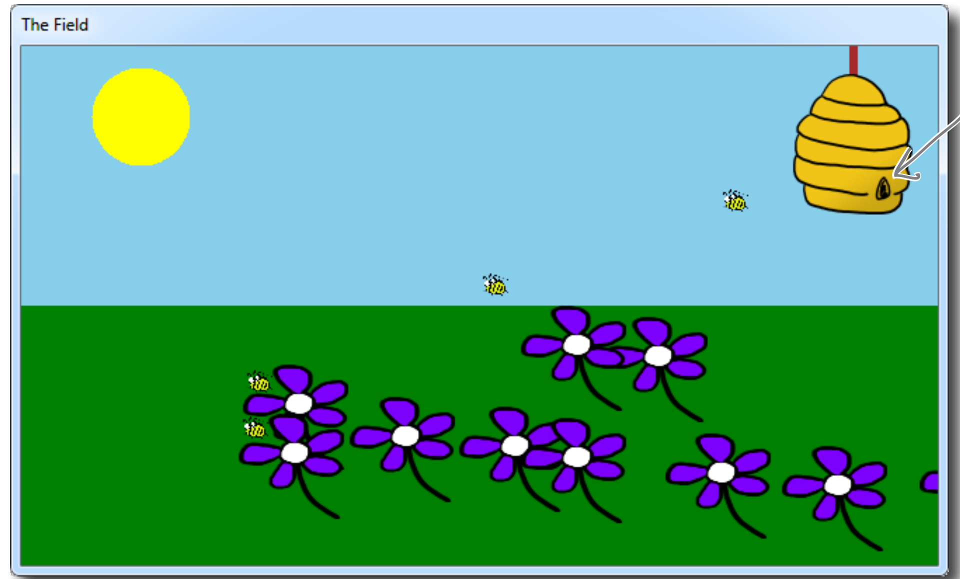
The hive has three important locations in it. The bees are born in the nursery, they have to fly to the exit to leave the hive to gather nectar from the flowers, and when they come back they need to go to the honey factory to make honey.

The hive exit is on the hive form, and the entrance is on the field form. (That's why we put both of them in the hive's locations dictionary.)

4 **The field window is where the bees collect the nectar**

Bees have one big job: to collect nectar from the flowers, and bring it back to the hive to make honey. Then they eat honey to give them energy to fly out and get more nectar.

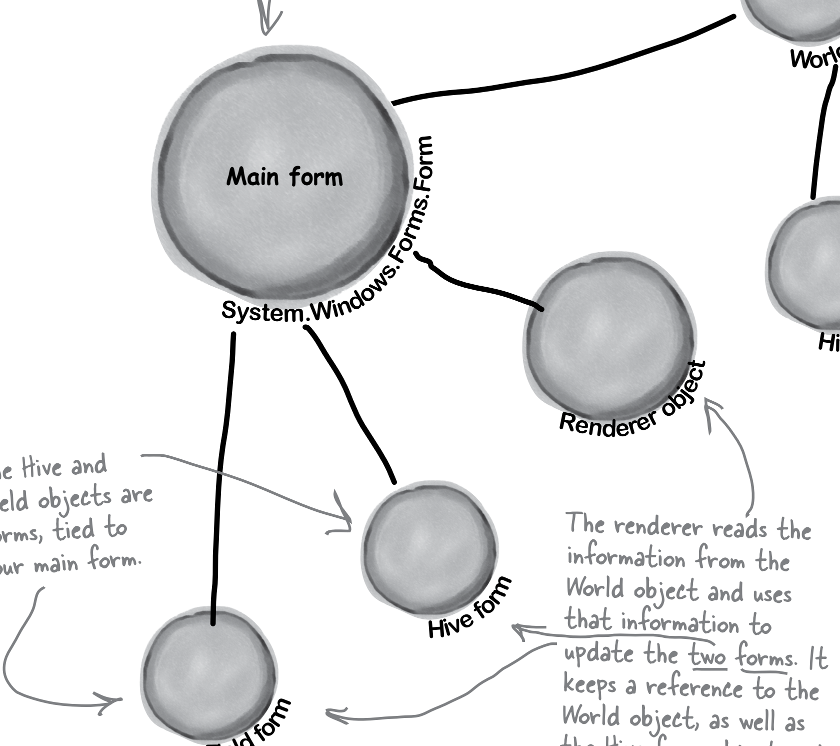
Here's the entrance to the hive. When bees fly into it, they disappear from the field form and reappear near the exit in the hive form.



www.itbook.store/books/9781449343507

Add a renderer to your architecture

We need another class that reads the information in the world and uses it to draw the hive, bees, and flowers on the two new forms. We'll add a class called **`Renderer`** to do exactly that. And since your other classes are well encapsulated, this won't require a lot of changes to your existing code.



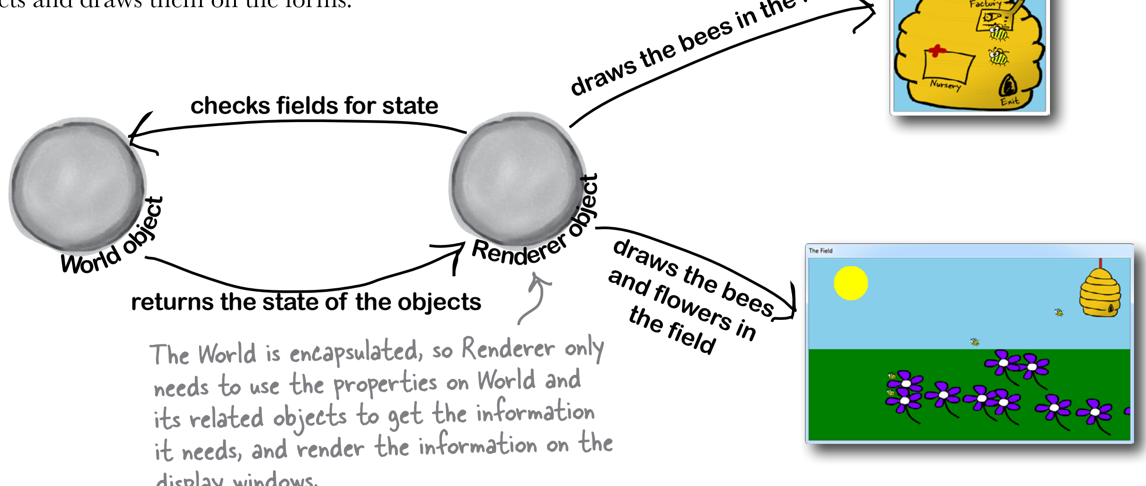
Because Bee, Hive, Flower, and World are <u>well encapsulated</u>, a class that renders those objects can be added <u>without</u> lots of changes to existing code.

> ren-der, verb.
> to represent or depict artistically.
> *Sally's art teacher asked the class to look at all of the shadows and lines in the model and* ***render*** *them on the page.*

www.itbook.store/books/9781449343507

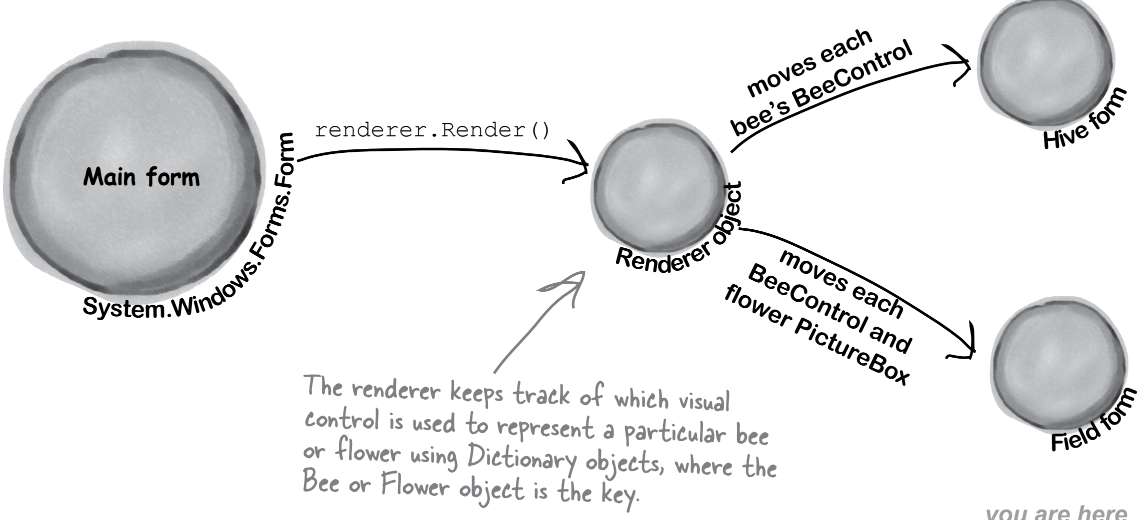
The renderer draws everything in the world on the two forms

The World object keeps track of everything in the simulation: the hive, the bees, and the flowers. But it doesn't actually draw anything or produce any output. That's the job of the Renderer object. It reads all of the information in the World, Hive, Bee, and Flower objects and draws them on the forms.



The simulator renders the world after each frame

After the main form calls the world's Go() method, it should call the renderer's Render() method to redraw the display windows. For example, each flower will be displayed using a PictureBox control. But let's go further with bees and create an animated control. You'll create this new control, called BeeControl, and define its behavior yourself.

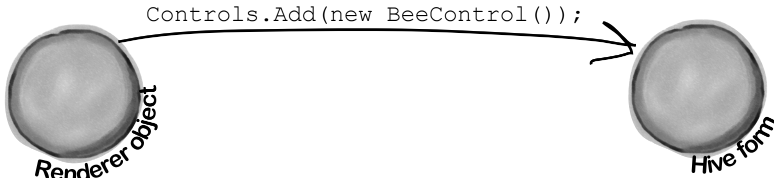


www.itbook.store/books/9781449343507

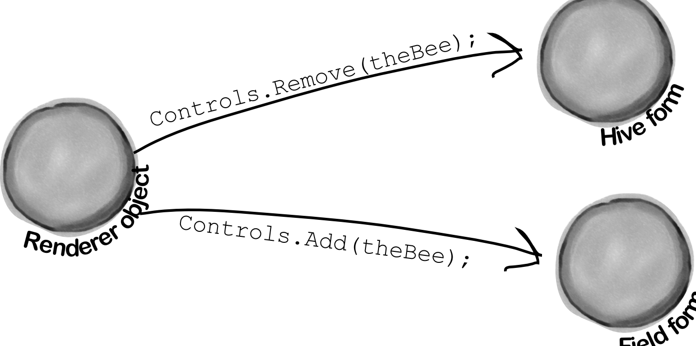
Controls are well suited for visual display elements

When a new bee is added to the hive, we'll want our simulator to add a new `BeeControl` to the `Hive` form and change its location as it moves around the world. When that bee flies out of the hive, our simulator will need to remove the control from the `Hive` form and add it to the `Field` form. And when it flies back to the hive with its load of nectar, its control needs to be removed from the `Field` form and added back to the `Hive` form. And all the while, we'll want the animated bee picture to flap its wings. Controls will make it easy to do all of that.

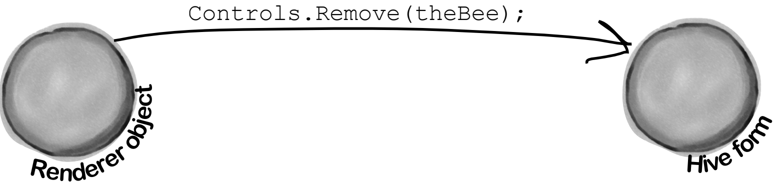
1. The world adds a new bee, and the renderer creates a new `BeeControl` and adds it to the `Hive` form's `Controls` collection.



2. When the bee flies out of the hive and enters the field, the renderer removes the `BeeControl` from the hive's `Controls` collection and adds it to the `Field` form's `Controls` collection.



3. A bee will retire if it's idle and it's gotten too old. If the renderer checks the world's `Bees` list and finds that the bee is no longer there, it removes the control from the `Hive` form.



www.itbook.store/books/9781449343507

Can you figure out what each of these code snippets does? Assume each snippet is inside a form, and write down your best guess.

```
this.Controls.Add(new Button());
```

```
Form2 childWindow = new Form2();
childWindow.BackgroundImage =
          Properties.Resources.Mosaic;
childWindow.BackgroundImageLayout =
          ImageLayout.Tile;
childWindow.Show();
```

If you've got a ListBox on your form, you can use its AddRange() method to add list items.

```
Label myLabel = new Label();
myLabel.Text = "What animal do you like?";
myLabel.Location = new Point(10, 10);
ListBox myList = new ListBox();
myList.Items.AddRange( new object[]
       { "Cat", "Dog", "Fish", "None" } );
myList.Location = new Point(10, 40);
Controls.Add(myLabel);
Controls.Add(myList);
```

You don't need to write down each line, as much as summarize what's going on in the code block.

```
Label controlToRemove = null;
foreach (Control control in Controls) {
    if (control is Label
          && control.Text == "Bobby")
        controlToRemove = control as Label;
}
Controls.Remove(controlToRemove);
controlToRemove.Dispose();
```

Bonus question: Why do you think we didn't put the `Controls.Remove()` statement inside the foreach loop?

Try it out if you want, and write why you think you got the result that .NET gave you.

www.itbook.store/books/9781449343507

Sharpen your pencil Solution

Can you figure out what each of these code snippets does? Assume each snippet is inside a form, and write down what you think it does.

```
this.Controls.Add(new Button());
```

Create a new button and add it to the form. It'll have default values (e.g., the Text property will be empty).

```
Form2 childWindow = new Form2();
childWindow.BackgroundImage =
          Properties.Resources.Mosaic;
childWindow.BackgroundImageLayout =
          ImageLayout.Tile;
childWindow.Show();
```

There's a second Form in the application called Form2, so this creates it, sets its background image to a resource image called "Mosaic", makes the background image so it's tiled instead of stretched, and then displays the window to the user.

```
Label myLabel = new Label();
myLabel.Text = "What animal do you like?";
myLabel.Location = new Point(10, 10);
ListBox myList = new ListBox();
myList.Items.AddRange( new object[]
       { "Cat", "Dog", "Fish", "None" } );
myList.Location = new Point(10, 40);
Controls.Add(myLabel);
Controls.Add(myList);
```

This code creates a new label, sets its text, and moves it to a new position. Then it creates a new list box, adds four items to the list, and moves it just underneath the label. It adds the label and list box to the form, so they both get displayed immediately.

What happens if there's no control named "Bobby" in the Controls collection?

```
Label controlToRemove = null;
foreach (Control control in Controls) {
    if (control is Label
          && control.Text == "Bobby")
        controlToRemove = control as Label;
}
Controls.Remove(controlToRemove);
controlToRemove.Dispose();
```

This loop searches through all the controls on the form until it finds a label with the text "Bobby". Once it finds the label, it removes it from the form.

If you try, .NET will throw an exception. It needs the collection intact, otherwise it'll lose its place and give you unpredictable results. That's why you'd use a for loop for this instead.

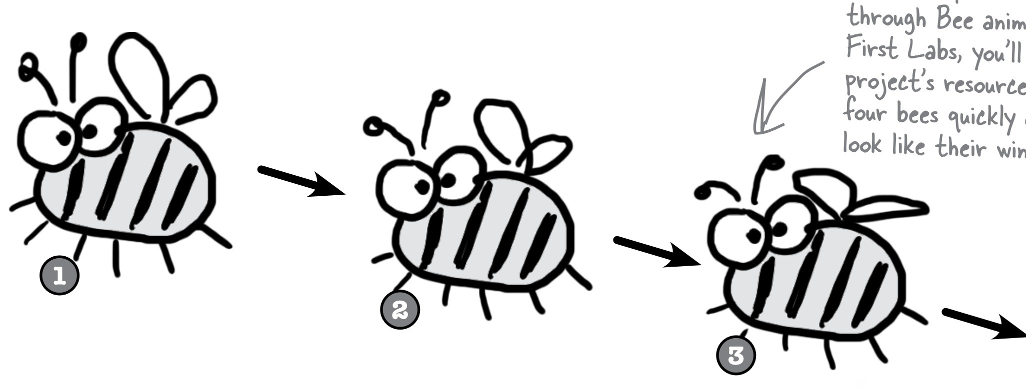
Bonus question: Why do you think we didn't put the Controls.Remove() statement inside the foreach loop?

You can't modify the Controls collection (or any other collection) in the middle of a foreach loop that's iterating through it.

www.itbook.store/books/9781449343507

Build your first animated control

You're going to **build your own control** that draws an animated bee picture. If you've never done animation, it's not as hard as it sounds: you draw a sequence of pictures one after another, and produce the illusion of movement. Lucky for us, the way C# and .NET handle resources makes it really easy for us to do animation.

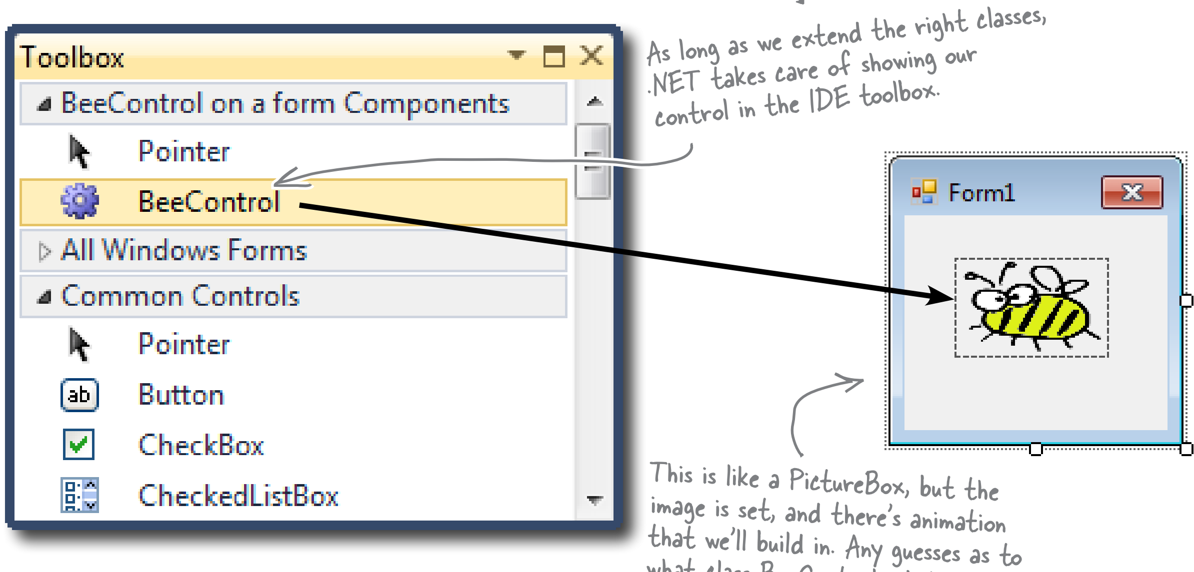


We want a control in the toolbox

If you build `BeeControl` right, it'll appear as a control that you can drag out of your toolbox and onto your form. It'll look just like a `PictureBox` showing a picture of a bee, except that it'll have animated flapping wings.

Download the images for this chapter from the Head First Labs website:

www.headfirstlabs.com/books/hfcsharp/



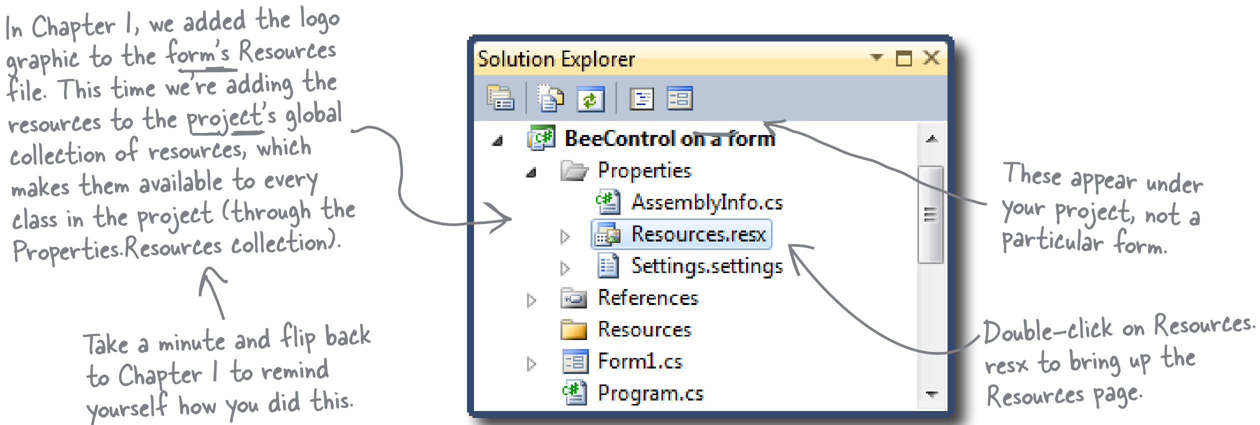
www.itbook.store/books/9781449343507

BeeControl is LIKE a PictureBox...so let's start by INHERITING from PictureBox

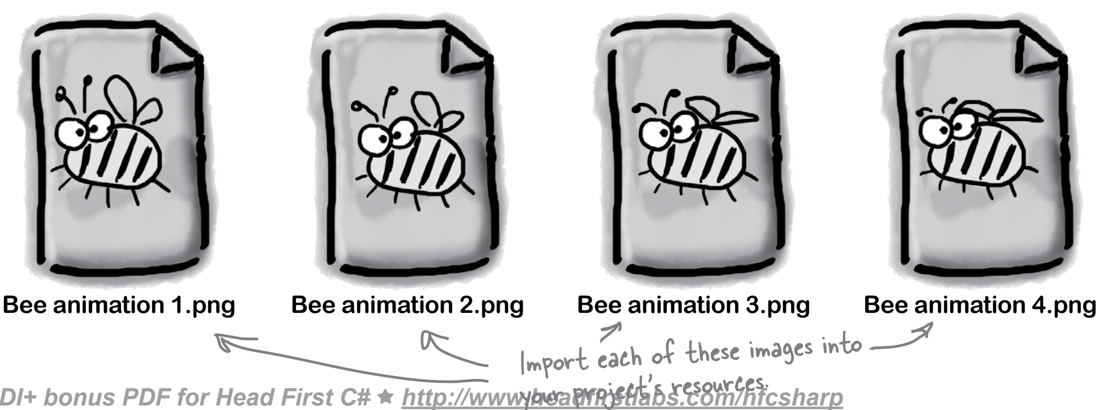
Since every control in the toolbox is just an object, it's easy to make a new control. All you need to do is add a new class to your project that inherits from an existing control, and add any new behavior you want your control to perform.

We want a control—let's call it a **`BeeControl`**—that shows an animated picture of a bee flapping its wings, but we'll start with a control that shows a *non*-animated picture, and then just add animation. So we'll start with a `PictureBox`, and then we'll add code to draw an animated bee on it.

1. **Create a new project** and add the four animation cells to the project's resources, just like you added the Objectville Paper Company logo to your project way back in Chapter 1. But instead of adding them to the *form* resources, add them to the *project's* resources. Find **your project's Resources.resx file** in the Solution Explorer (it's under Properties). Double-click on it to bring up the project's Resources page.



2. We've drawn a four-cell bee animation to import into your resources that you can download from **http://www.headfirstlabs.com/books/hfcsharp/**. Then, go to the Resources page, select "Images" from the first drop-down at the top of the screen, and select "Add Existing File…" from the "Add Resource" drop-down.

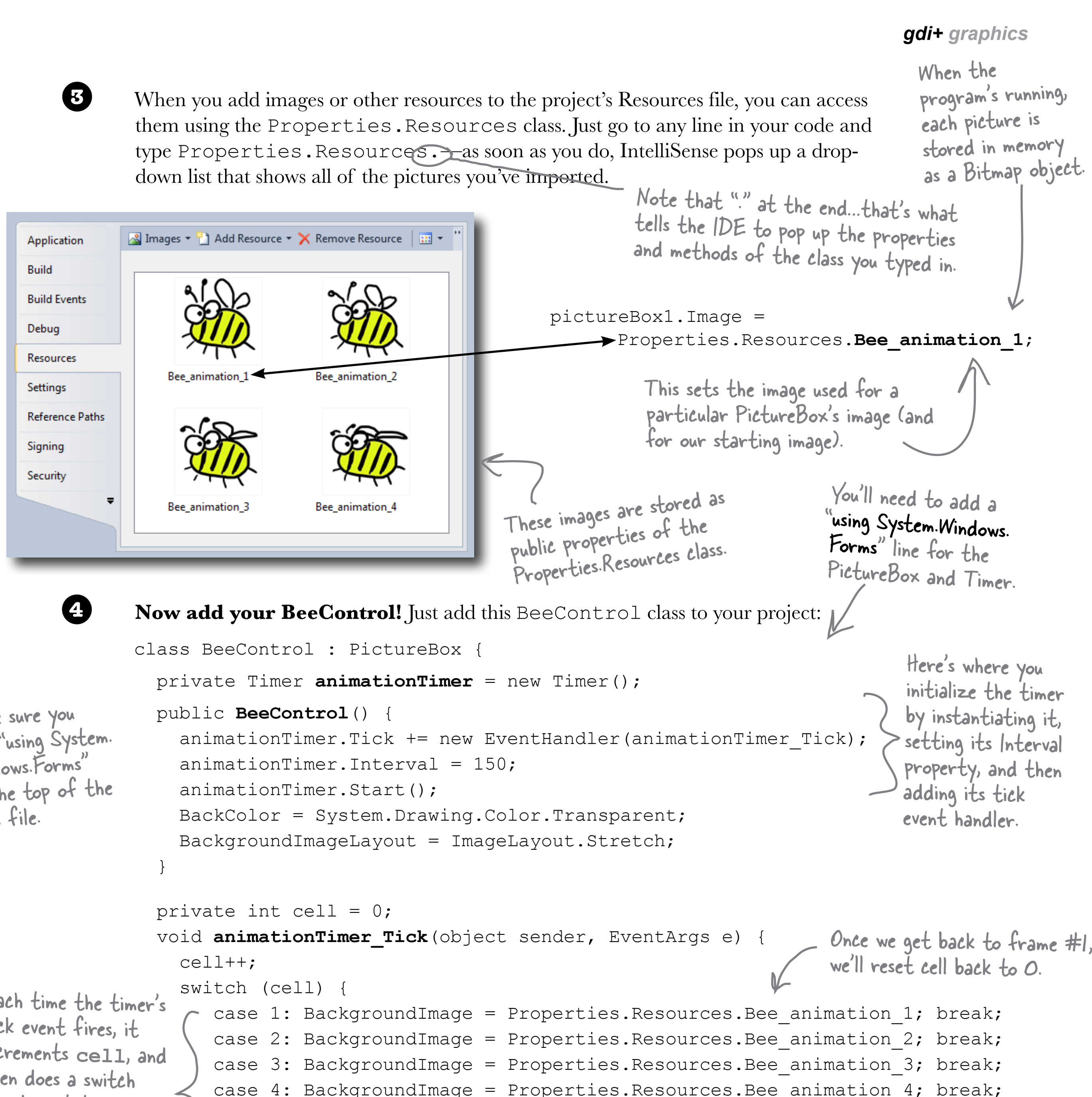


www.itbook.store/books/9781449343507

3 When you add images or other resources to the project's Resources file, you can access them using the `Properties.Resources` class. Just go to any line in your code and type `Properties.Resources.`—as soon as you do, IntelliSense pops up a drop-down list that shows all of the pictures you've imported.

When the program's running, each picture is stored in memory as a Bitmap object.

Note that "." at the end...that's what tells the IDE to pop up the properties and methods of the class you typed in.




```
pictureBox1.Image =
    Properties.Resources.Bee_animation_1;
```

This sets the image used for a particular PictureBox's image (and for our starting image).

These images are stored as public properties of the Properties.Resources class.

You'll need to add a "using System.Windows.Forms" line for the PictureBox and Timer.

4 **Now add your BeeControl!** Just add this `BeeControl` class to your project:

```
class BeeControl : PictureBox {
  private Timer animationTimer = new Timer();
  public BeeControl() {
    animationTimer.Tick += new EventHandler(animationTimer_Tick);
    animationTimer.Interval = 150;
    animationTimer.Start();
    BackColor = System.Drawing.Color.Transparent;
    BackgroundImageLayout = ImageLayout.Stretch;
  }

  private int cell = 0;
  void animationTimer_Tick(object sender, EventArgs e) {
    cell++;
    switch (cell) {
       case 1: BackgroundImage = Properties.Resources.Bee_animation_1; break;
       case 2: BackgroundImage = Properties.Resources.Bee_animation_2; break;
       case 3: BackgroundImage = Properties.Resources.Bee_animation_3; break;
       case 4: BackgroundImage = Properties.Resources.Bee_animation_4; break;
       case 5: BackgroundImage = Properties.Resources.Bee_animation_3; break;
       default: BackgroundImage = Properties.Resources.Bee_animation_2;
       cell = 0; break;
    }
  }
}
```

Make sure you add "using System.Windows.Forms" to the top of the class file.

Here's where you initialize the timer by instantiating it, setting its Interval property, and then adding its tick event handler.

Once we get back to frame #1, we'll reset cell back to 0.

Each time the timer's tick event fires, it increments cell, and then does a switch based on it to assign the right picture to the Image property (inherited from PictureBox).

When you change the code for a control, you need to rebuild your program to make your changes show up in the designer.

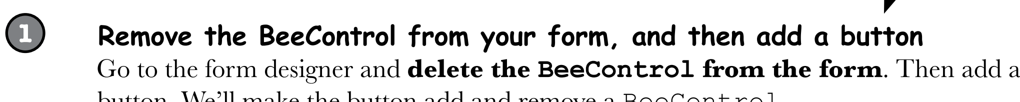
Then **rebuild your program**. Go back to the form designer and look in the toolbox, and the `BeeControl` is there. Drag it onto your form—you get an **animated** bee!

www.itbook.store/books/9781449343507

Create a button to add the BeeControl to your form

It's easy to add a control to a form—just add it to the `Controls` collection. And it's just as easy to remove it from the form by removing it from `Controls`. But controls implement `IDisposable`, so make sure you **always dispose your control** after you remove it.

Now do this



1. **Remove the BeeControl from your form, and then add a button**
   Go to the form designer and **delete the `BeeControl` from the form**. Then add a button. We'll make the button add and remove a `BeeControl`.

2. **Add a button to add and remove the bee control**
   Here's the event handler for it:

You can use an object initializer to set the BeeControl properties after it's instantiated.

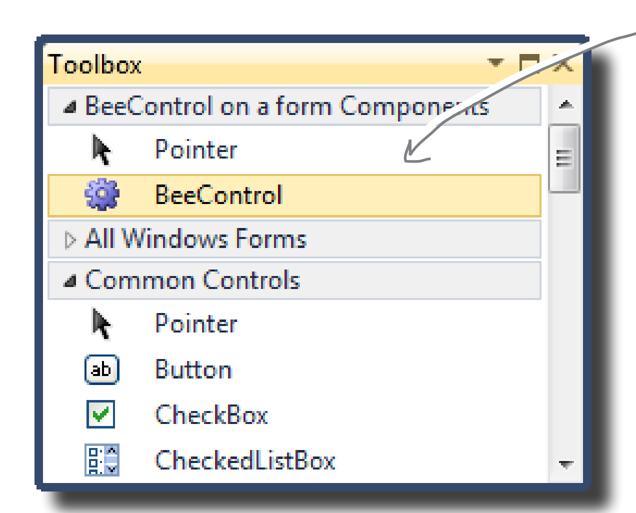
When you add a control to the Controls collection, it appears on the form immediately.

```
BeeControl control = null;
private void button1_Click(object sender, EventArgs e) {
    if (control == null) {
        control = new BeeControl() { Location = new Point(100, 100) };
        Controls.Add(control);
    } else {
        using (control) {
          Controls.Remove(control);
        }
        control = null;
    }
}
```

We're taking advantage of a using statement to make sure the control is disposed after it's removed from the Controls collection.

Now when you run your program, if you click the button once it'll add a new `BeeControl` to the form. Click it again and it'll delete it. It uses the private `control` field to hold the reference to the control. (It sets the reference to null when there's no control on the form.)

Behind the Scenes



You can add your own control to the toolbox just by creating a class that inherits from Control.

Every visual control in your toolbox inherits from `System.Windows.Forms.Control`. That class has members that should be pretty familiar by now: `Visible`, `Width`, `Height`, `Text`, `Location`, `BackColor`, `BackgroundImage`... all of those familiar properties you see in the Properties window for any control.

www.itbook.store/books/9781449343507

Your controls need to dispose their controls, too!

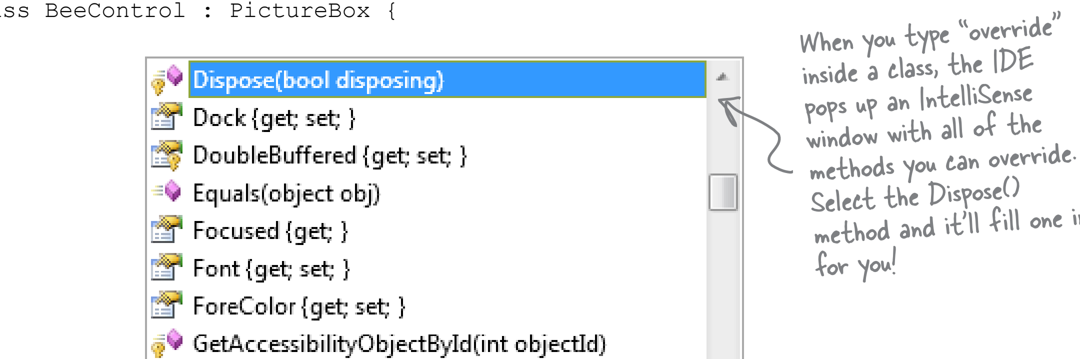
There's a problem with the BeeControl. Controls need to be disposed after they're done. But the BeeControl creates a new instance of Timer, which is a control that shows up in the toolbox…and it never gets disposed! That's a problem. Luckily, it's easy to fix—just override the Dispose() method.

The control class implements IDisposable, so you need to make sure every control you use gets disposed.

Override the Dispose() method and dispose of the timer

Since BeeControl inherits from a control, then that control must have a Dispose() method. So we can just override and extend that method to dispose our timer. Just go into the control and type override:

```
class BeeControl : PictureBox {
```



When you type "override" inside a class, the IDE pops up an IntelliSense window with all of the methods you can override. Select the Dispose() method and it'll fill one in for you!

As soon as you click on Dispose(), the IDE will fill in the method with a call to base.Dispose():

```
protected override void Dispose(bool disposing) {
    base.Dispose(disposing);
}
```

Add the code to dispose the timer

Add code to the end of the new Dispose() method that the IDE added for you so that it calls animationTimer.Dispose() if the disposing argument is true.

```
protected override void Dispose(bool disposing) {
    base.Dispose(disposing);
    if (disposing) {
        animationTimer.Dispose();
    }
}
```

Here we're overriding a protected Dispose() method that's called by the control's implementation of IDisposable.Dispose(). It should only dispose the timer if the disposing argument is true.

Now the BeeControl will dispose of its timer as part of its own Dispose() method. It cleans up after itself!

But don't take our word for it—**set a breakpoint** on the line you added and run your program. Every time a BeeControl object is removed from the form's Controls collection, its Dispose() method is called.

Any control that you write from scratch is responsible for disposing any other controls (or disposable objects) that it creates.

We won't go into any more detail about this particular disposal pattern. But if you plan on building custom controls, you definitely should read this: http://msdn.microsoft.com/en-us/library/system.idisposable.aspx

www.itbook.store/books/9781449343507

A UserControl is an easy way to build a control

There's an easier way to build your own toolbox controls. Instead of creating a class that inherits from an existing control, all you need to do is **use the IDE to add a `UserControl` to your project**. You work with a `UserControl` just like a form. You can drag other controls out of the toolbox and onto it—it uses the normal form designer in the IDE. And you can use its events just like you do with a form. So let's rebuild the `BeeControl` using a `UserControl`.

1. Create a brand-new Windows Forms Application project. Add the four bee images to its resources. Drag a button to the form and give it exactly the same code as to add and remove a `BeeControl`.

2. Right-click on the project in the Solution Explorer and select "Add >> User Control…". Have the IDE **add a user control called `BeeControl`**. The IDE will open up the new control in the form designer.

Use the animationTimer_Tick() method and the cell field from the old bee control.

3. Drag a Timer control onto your user control. It'll show up at the bottom of the designer, just like with a form. Use the Properties window to **name it `animationTimer` and set its `Interval` to 150 and its `Enabled` to `true`**. Then double-click on it—the IDE will add its Tick event handler. Just use the same Tick event handler that you used earlier to animate the first bee control.

4. Now update the `BeeControl`'s constructor:

```
public BeeControl() {
    InitializeComponent();
    BackColor = System.Drawing.Color.Transparent;
    BackgroundImageLayout = ImageLayout.Stretch;
}
```

You can also do this from the Properties page in the IDE, instead of using code.

5. Now **run your program—**the button code should still work exactly the same as before, except now it's creating your new `UserControl`-based `BeeControl`. The button now adds and removes your `UserControl`-based `BeeControl`.

A UserControl is an easy way to add a control to the toolbox. Edit a UserControl just like a form—you can drag other controls out of the toolbox onto it, and you can use its events exactly like a form's events.

www.itbook.store/books/9781449343507

You didn't dispose your controls because your forms did it for you.

But don't take our word for it. Use the IDE's search function to search your project for the word "Dispose", and you'll find that the IDE added a method in `Form1.Designer.cs` to override the `Dispose()` method that calls its own `base.Dispose()`. When the form is disposed, **it automatically disposes everything in its `Controls` collection** so you don't have to worry about it. But once you start removing controls from that collection or creating new instances of controls (like the `Timer` in the `BeeControl`) outside of the `Controls` collection, then you need to do the disposal yourself.

there are no Dumb Questions

Q: Why does the form code for the `PictureBox`-based `BeeControl` work exactly the same with the `UserControl`-based `BeeControl`?

A: Because the code doesn't care how the `BeeControl` object is implemented. It just cares that it can add the object to the form's `Controls` method.

Q: I double-clicked on my `OldBeeControl` class in the Solution Explorer, and it had a message about adding components to my class. What's that about?

A: When you create a control by adding a class to your project that inherits from `PictureBox` or another control, the IDE does some clever things. One of the things it does is let you work with **components**, those non-visual controls like `Timer` and `OpenFileDialog` that show up in the space beneath your form when you work with them.

Give it a try—create an empty class that inherits from `PictureBox`. Then rebuild your project and double-click on it in the IDE. You'll get this message:

To add components to your class, drag them from the Toolbox and use the Properties window to set their properties.

Drag an `OpenFileDialog` out of the toolbox and onto your new class. It'll appear as an icon. You can click on it and set its properties. Set a few of them. Now go back to the code for your class. Check out the constructor—the IDE added code to instantiate the `OpenFileDialog` object and set its properties.

Q: When I changed the properties in the `OpenFileDialog`, I noticed an error message in the IDE: "You must rebuild your project for the changes to show up in any open designers." Why did I get this error?

A: Because the designer runs your control, and until you rebuild your code it's not running the latest version of the control.

Remember how the wings of the bee were flapping when you first created your `BeeControl`, even when you dragged it out of the toolbox and into the designer? You weren't running your program yet, but the code that you wrote was being executed. The timer was firing its `Tick` event, and your event handler was changing the picture. The only way the IDE can make that happen is if the code were actually compiled and running in memory somewhere. So it's reminding you to update your code so it can display your controls properly.

www.itbook.store/books/9781449343507

Your simulator's renderer will use your BeeControl to draw animated bees on your forms

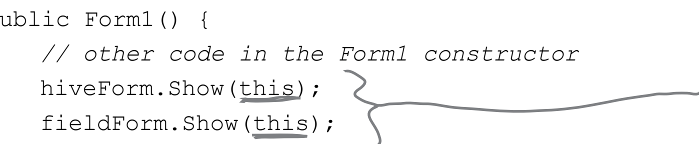
Now you've got the tools to start adding animation to your simulator. With a `BeeControl` class and two forms, you just need a way to position bees, move them from one form to the other, and keep up with the bees. You'll also need to position flowers on the `FieldForm`, although since flowers don't move, that's pretty simple. All of this is code that we can **put into a new class, `Renderer`**. Here's what that class will do:

You'll want the hive and field forms "linked" to the stats form—that does useful things like minimizing the hive and field forms when you minimize the stats form. You can do this by telling Windows that the stats form is their owner.

We'll build the renderer in a minute. But before we jump in and start coding, let's take a minute and come up with a plan for how the Renderer class will work...

The stats form will be the parent of the hive and field forms

The first step in adding graphics to the beehive simulator will be adding two forms to the project. You'll add one called `HiveForm` (to show the inside of the hive) and one called `FieldForm` (which will show the field of flowers). Then you'll add lines to the main form's constructor to show its two child forms. Pass a reference to the main form to tell Windows that the stats form is their **owner**:



```
public Form1() {
    // other code in the Form1 constructor
    hiveForm.Show(this);
    fieldForm.Show(this);
```

Every form object has a Show() method. If you want to set another form as its owner, just pass a reference to that form to Show().

The renderer keeps a reference to the world and each child form

At the very top of the `Renderer` class you'll need a few important fields. The class has to know the location of each bee and flower, so it needs a reference to the `World`. And it'll need to add, move, and remove controls in the two forms, so it needs a reference to each of those forms:

```
class Renderer {
    private World world;
    private HiveForm hiveForm;
    private FieldForm fieldForm;
```

Start your Renderer class with these lines. We'll add to this class throughout the chapter.

3 **The renderer uses dictionaries to keep track of the controls**

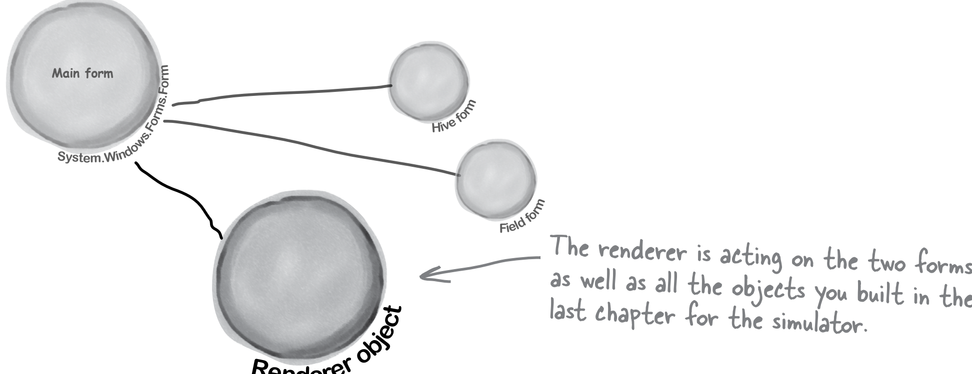
`World` keeps track of its `Bee` objects using a `List<Bee>` and a `List<Flower>` to store its flowers. The renderer needs to be able to look at each of those `Bee` and `Flower` objects and figure out what `BeeControl` and `PictureBox` they correspond to—or, if it can't find a corresponding control, it needs to create one. So here's a perfect opportunity to use dictionaries. We'll need two more private fields in `Renderer`:

```
    private Dictionary<Flower, PictureBox> flowerLookup =
        new Dictionary<Flower, PictureBox>();
    private Dictionary<Bee, BeeControl> beeLookup =
        new Dictionary<Bee, BeeControl>();
```

These dictionaries become one-to-one mappings between a bee or flower and the control for that bee or flower.

These two dictionary collections let the renderer store exactly one control for each bee or flower in the world.

www.itbook.store/books/9781449343507



4 The bees and flowers already know their locations

There's a reason we stored each bee and flower location using a `Point`. Once we have a `Bee` object, we can easily look up its `BeeControl` and set its location.

```
beeControl = beeLookup[bee];
beeControl.Location = bee.Location;
```

For each bee or flower, we can look up the matching control. Then, set that control's location to match the location of the bee or flower object.

5 If a bee doesn't have a control, the renderer adds it to the hive form

It's easy enough for the renderer to figure out if a particular bee or flower has a control. If the dictionary's `ContainsKey()` method returns `false` for a particular `Bee` object, that means there's no control on the form for that bee. So `Renderer` needs to create a `BeeControl`, add it to the dictionary, and then add the control to the form. (It also calls the control's `BringToFront()` method, to make sure the control doesn't get hidden behind the flower `PictureBoxes`.)

```
if (!beeLookup.ContainsKey(bee)) {
   beeControl = new BeeControl() { Width = 40, Height = 40 };
   beeLookup.Add(bee, beeControl);
   hiveForm.Controls.Add(beeControl);
   beeControl.BringToFront();
} else
   beeControl = beeLookup[bee];
```

ContainsKey() tells us if the bee exists in the dictionary. If not, then we need to add that bee, along with a corresponding control.

BringToFront() ensures the bee appears "on top of" any flowers on the FieldForm, and on top of the background of the HiveForm.

Remember how a dictionary can use anything as a key? Well, this one uses a Bee object as a key. The renderer needs to know which BeeControl on the form belongs to a particular bee. So it looks up that bee's object in the dictionary, which spits out the correct control. Now the renderer can move it around.

www.itbook.store/books/9781449343507

Add the hive and field forms to the project

This is a PictureBox control with its BackgroundImage set to the outside hive picture and BackgroundImageLayout set to Stretch. When you load the hive pictures into the Resource Designer, they'll show up in the list of resources when you click the "..." button next to BackgroundImage in the Properties window.

Now you need forms to put bees on. So **start with your existing beehive simulator project**, and use "Add >> Existing Item..." to **add your new `BeeControl` user control**. The `UserControl` has a `.cs` file, a `.designer.cs` file, and a `.resx` file—you'll need to add all three. Then open up the code for both the `.cs` and `.designer.cs` files, and change the namespace lines so they match the namespace of your new project. Rebuild your project; the `BeeControl` should now show up in the toolbox. You'll also need to add the graphics to the new project's resources. Then add two more Windows forms to the project by **right-clicking on the project** in the Solution Explorer and choosing "Windows Form..." from the Add menu. If you name the files `HiveForm.cs` and `FieldForm.cs`, the IDE will automatically set their `Name` properties to `HiveForm` and `FieldForm`. You already know that forms are just objects, so `HiveForm` and `FieldForm` are really just two more classes.

The Hive

Honey Factory

Nursery

Exit

Make sure you resize both forms so they look like these screenshots.

The Field

You'll need the inside and outside hive images—"Hive (inside).png" and "Hive (outside).png"—loaded into your resources. Then add these two forms. Set each form's FormBorderStyle property to FixedSingle (so the user can't resize it), the ControlBox property to false (to take away its minimize and maximize controls), and StartPosition to Manual (so its Location property is settable).

Set the form's BackgroundImage property to the inside hive picture, and its BackgroundImageLayout property to Stretch.

Remember, go to the Properties window, click on the lightning-bolt icon to bring up the Events window, scroll down to the MouseClick row and double-click on it. The IDE will add the event handler for you.

Figure out where your locations are

You need to figure out where the hive is on your `FieldForm`. Using the Properties window, create a handler for the **`MouseClick` event for the `Hive` form**, and add this code:

```
private void HiveForm_MouseClick(object sender, MouseEventArgs e) {
    MessageBox.Show(e.Location.ToString());
}
```

Exit

{X=77,Y=162}

OK

We'll get your form running on the next few pages. Once it's running, click on the exit of the hive in the picture. The event handler will show you the exact coordinates of the spot that you clicked.

Add the same handler to the `Field` form, too. Then, by clicking, get the coordinates of the exit, the nursery, and the honey factory. Using all these locations, you'll be able to update the `InitializeLocations()` method you wrote in the `Hive` class in the last chapter:

```
private void InitializeLocations()
{
    locations = new Dictionary<string, Point>();
    locations.Add("Entrance", new Point(626, 110));
    locations.Add("Nursery", new Point(77, 162));
    locations.Add("HoneyFactory", new Point(157, 78));
    locations.Add("Exit", new Point(175, 180));}
```

Once you get your simulator running, you can use this to tweak the Hive's locations collection.

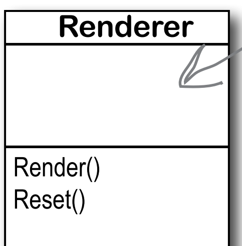
Remove the mouse click handler when you're done... you just needed it to get the locations on your forms.

These are the coordinates that worked for us, but if your form is a little bigger or smaller, your coordinates will be different.

www.itbook.store/books/9781449343507

Build the renderer

Here's the complete class. The main form calls this class's method right after it calls to draw the bees and flowers on the forms. You'll need to make sure that the flower graphic () is loaded into the project, just like the animated bee images.



All fields in the renderer are private because no other class needs to update any of its properties. It's fully encapsulated. The world just calls Render() to draw the world to the forms, and Reset() to clear the controls on the forms if it needs to reset.

```
class Renderer {
    private World world;
    private HiveForm hiveForm;
    private FieldForm fieldForm;

    private Dictionary<Flower, PictureBox> flowerLookup =
         new Dictionary<Flower, PictureBox>();
    private List<Flower> deadFlowers = new List<Flower>();

    private Dictionary<Bee, BeeControl> beeLookup =
         new Dictionary<Bee, BeeControl>();
    private List<Bee> retiredBees = new List<Bee>();

    public Renderer(World world, HiveForm hiveForm, FieldForm fieldForm) {
        this.world = world;
        this.hiveForm = hiveForm;
        this.fieldForm = fieldForm;
    }

    public void Render() {
        DrawBees();
        DrawFlowers();
        RemoveRetiredBeesAndDeadFlowers();
    }

    public void Reset() {
        foreach (PictureBox flower in flowerLookup.Values) {
            fieldForm.Controls.Remove(flower);
            flower.Dispose();
        }
        foreach (BeeControl bee in beeLookup.Values) {
            hiveForm.Controls.Remove(bee);
            fieldForm.Controls.Remove(bee);
            bee.Dispose();
        }
        flowerLookup.Clear();
        beeLookup.Clear();
    }
```

The renderer keeps references to the world and the two forms it draws the bees on.

The world uses Bee and Flower objects to keep track of every bee and flower in the world. The forms use a PictureBox to display each flower and a BeeControl to display each bee. The renderer uses these dictionaries to connect each bee and flower to its own BeeControl or PictureBox.

When a flower dies or a bee retires, it uses the deadFlowers and retiredBees lists to clean out the dictionaries.

The timer on the main form that runs the animation calls the Render() method, which updates the bees and the flowers, and then cleans out its dictionaries.

If the simulator is reset, it calls each form's Controls.Remove() method to completely clear out the controls on the two forms. It finds all of the controls in each of its two dictionaries and removes them from the forms, calling Dispose() on each of them. Then it clears the two dictionaries.

www.itbook.store/books/9781449343507

It takes two foreach loops to draw the flowers. The first looks for new flowers and adds their PictureBoxes. The second looks for dead flowers and removes their PictureBoxes.

The first foreach loop uses the flowerLookup dictionary to check each flower to see if it's got a control on the form. If it doesn't, it creates a new PictureBox using an object initializer, adds it to the form, and then adds it to the flowerLookup dictionary.

DrawFlowers() uses the Location property in the Flower object to set the PictureBox's location on the form.

```
private void DrawFlowers() {
    foreach (Flower flower in world.Flowers)
        if (!flowerLookup.ContainsKey(flower)) {
            PictureBox flowerControl = new PictureBox() {
                Width = 45,
                Height = 55,
                Image = Properties.Resources.Flower,
                SizeMode = PictureBoxSizeMode.StretchImage,
                Location = flower.Location
            };
            flowerLookup.Add(flower, flowerControl);
            fieldForm.Controls.Add(flowerControl);
        }

    foreach (Flower flower in flowerLookup.Keys) {
        if (!world.Flowers.Contains(flower)) {
            PictureBox flowerControlToRemove = flowerLookup[flower];
            fieldForm.Controls.Remove(flowerControlToRemove);
            flowerControlToRemove.Dispose();
            deadFlowers.Add(flower);
        }
    }
}
```

The second foreach loop looks for any PictureBox in the flowerLookup dictionary that's no longer on the form and removes it.

After it removes the PictureBox, it calls its Dispose() method. Then it adds the Flower object to deadFlowers so it'll get cleared later.

DrawBees() also uses two foreach loops, and it does the same basic things as DrawFlowers(). But it's a little more complex, so we split some of its behavior out into separate methods to make it easier to understand.

```
private void DrawBees() {
    BeeControl beeControl;
    foreach (Bee bee in world.Bees) {
        beeControl = GetBeeControl(bee);
        if (bee.InsideHive) {
            if (fieldForm.Controls.Contains(beeControl))
                MoveBeeFromFieldToHive(beeControl);
        } else if (hiveForm.Controls.Contains(beeControl))
            MoveBeeFromHiveToField(beeControl);
        beeControl.Location = bee.Location;
    }

    foreach (Bee bee in beeLookup.Keys) {
        if (!world.Bees.Contains(bee)) {
            beeControl = beeLookup[bee];
            if (fieldForm.Controls.Contains(beeControl))
                fieldForm.Controls.Remove(beeControl);
            if (hiveForm.Controls.Contains(beeControl))
                hiveForm.Controls.Remove(beeControl);
            beeControl.Dispose();
            retiredBees.Add(bee);
        }
    }
}
```

DrawBees() checks if a bee is in the hive but its control is on the FieldForm, or vice versa. It uses two extra methods to move the BeeControls between the forms.

Once the BeeControl is removed, we need to call its Dispose() method—the user control will dispose of its timer for us.

The second foreach loop works just like in DrawFlowers(), except it needs to remove the BeeControl from the right form.

www.itbook.store/books/9781449343507

You'll need to make sure you've got using System.Drawing and using System.Windows.Forms at the top of the Renderer class file.

GetBeeControl() looks up a bee in the beeLookup dictionary and returns it. If it's not there, it creates a new 40 x 40 BeeControl and adds it to the hive form (since that's where bees are born).

```
private BeeControl GetBeeControl(Bee bee) {
    BeeControl beeControl;
    if (!beeLookup.ContainsKey(bee)) {
        beeControl = new BeeControl() { Width = 40, Height = 40 };
        beeLookup.Add(bee, beeControl);
        hiveForm.Controls.Add(beeControl);
        beeControl.BringToFront();
    }
    else
        beeControl = beeLookup[bee];
    return beeControl;
}
```

Don't forget that the ! means NOT.

MoveBeeFromHiveToField() takes a specific BeeControl out of the hive form's Controls collection and adds it to the field form's Controls collection.

```
private void MoveBeeFromHiveToField(BeeControl beeControl) {
    hiveForm.Controls.Remove(beeControl);
    beeControl.Size = new Size(20, 20);
    fieldForm.Controls.Add(beeControl);
    beeControl.BringToFront();
}
```

The bees on the field form are smaller than the ones on the hive form, so the method needs to change BeeControl's Size property.

```
private void MoveBeeFromFieldToHive(BeeControl beeControl) {
    fieldForm.Controls.Remove(beeControl);
    beeControl.Size = new Size(40, 40);
    hiveForm.Controls.Add(beeControl);
    beeControl.BringToFront();
}
```

MoveBeeFromFieldToHive() moves a BeeControl back to the hive form. It has to make it bigger again.

```
private void RemoveRetiredBeesAndDeadFlowers() {
    foreach (Bee bee in retiredBees)
        beeLookup.Remove(bee);
    retiredBees.Clear();
    foreach (Flower flower in deadFlowers)
        flowerLookup.Remove(flower);
    deadFlowers.Clear();
}
}
```

Whenever DrawBees() and DrawFlowers() found that a flower or bee was no longer in the world, it added them to the deadFlowers and retiredBees lists to be removed at the end of the frame.

After all the controls are moved around, the renderer calls this method to clear any dead flowers and retired bees out of the two dictionaries.

www.itbook.store/books/9781449343507

Now connect the main form to your two new forms, HiveForm and FieldForm

It's great to have a renderer, but so far, there aren't any forms to render onto. We can fix that by going back to the main Form class (probably called Form1) and making some code changes:

```
public partial class Form1 : Form {
  private HiveForm hiveForm = new HiveForm();
  private FieldForm fieldForm = new FieldForm();
  private Renderer renderer;

  // the rest of the fields

  public Form1() {
    InitializeComponent();

    MoveChildForms();
    hiveForm.Show(this);
    fieldForm.Show(this);
    ResetSimulator();

    timer1.Interval = 50;
    timer1.Tick += new EventHandler(RunFrame);
    timer1.Enabled = false;
    UpdateStats(new TimeSpan());
  }

  private void MoveChildForms() {
    hiveForm.Location = new Point(Location.X + Width + 10, Location.Y);
    fieldForm.Location = new Point(Location.X,
      Location.Y + Math.Max(Height, hiveForm.Height) + 10);
  }

  public void RunFrame(object sender, EventArgs e) {
    framesRun++;
    world.Go(random);
    renderer.Render();
    // previous code
  }

  private void Form1_Move(object sender, EventArgs e) {
    MoveChildForms();
  }
```

When the main form loads, it creates an instance of each of the other two forms. They're just objects in the heap for now—they won't be displayed until their Show() methods are called.

The code to reset the world moved to the ResetSimulator() method.

Move the code to instantiate the World into the ResetSimulator() method.

The form passes a reference to itself into Form.Show() so it becomes the parent form.

The main form's constructor moves the two child forms in place, then displays them. Then it calls ResetSimulator(), which instantiates Renderer.

Since both child forms have StartPosition set to Manual, the main form can move them using the Location property.

This code moves the two forms so that the hive form is next to the main stats form and the field form is below both of them.

Adding this one line to RunFrame makes the simulator update the graphics each time the world's Go() method is called.

The Move event is fired every time the main form is moved. Calling MoveChildForms() makes sure the child forms always move along with the main form.

Use the Events button in the Properties window to add the Move event handler.

Make sure you've set the field and hive forms' StartPosition property to Manual, or else MoveChildForms() won't work.

www.itbook.store/books/9781449343507

Here's where we **create new instances** of the World and Renderer classes, which resets the simulator.

```
private void ResetSimulator() {
  framesRun = 0;
  world = new World(new BeeMessage(SendMessage));
  renderer = new Renderer(world, hiveForm, fieldForm);
}

private void reset_Click(object sender, EventArgs e) {
   renderer.Reset();
   ResetSimulator();
   if (!timer1.Enabled)
       toolStrip1.Items[0].Text = "Start simulation";
}

private void openToolStripButton_Click(object sender, EventArgs e) {
  // The rest of the code in this button stays exactly the same.

  renderer.Reset();
  renderer = new Renderer(world, hiveForm, fieldForm);
}
}
```

The Reset button needs to call Reset() to clear out all the BeeControls and flower PictureBoxes, and then reset the simulator.

Finally, you'll need to add code to the Open button on the ToolStrip to use the Reset() method to remove the bees and flowers from the two forms' Controls collections, and then create a new renderer using the newly loaded world.

there are no Dumb Questions

Q: I saw that you showed the form using a `Show()` method, but I don't quite get what was going on with passing `this` as a parameter.

A: This all comes down to the idea that a form is just another class. When you display a form, you're just instantiating that class and calling its `Show()` method. There's an overloaded version of `Show()` that takes one parameter, a parent window. When one form is a parent of another, it causes Windows to set up a special relationship between them—for example, when you minimize the parent window, it automatically minimizes all of that form's child windows, too.

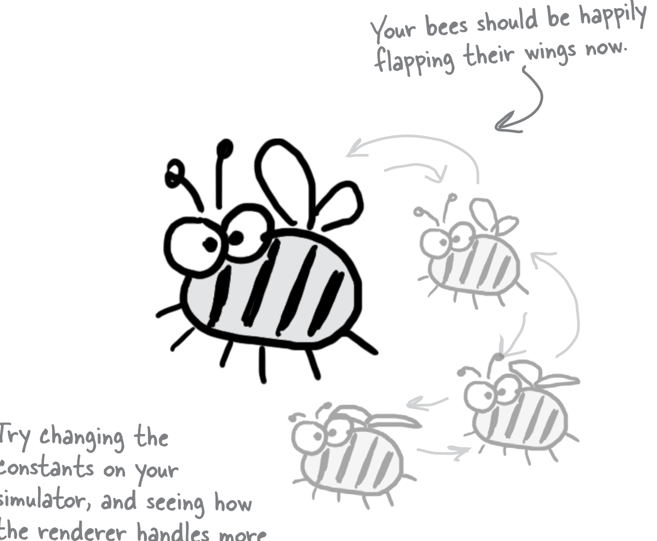
Q: Can you alter the preexisting controls and muck around with their code?

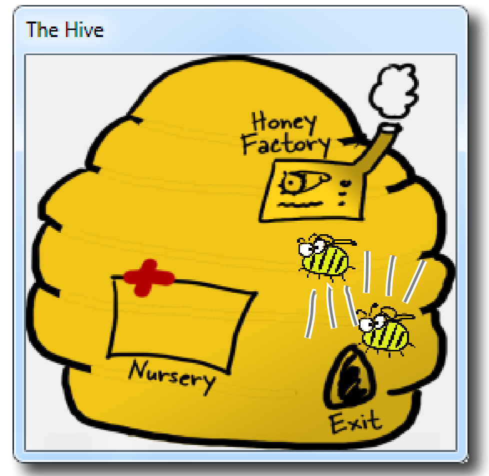
A: No, you can't actually access the code inside the controls that ship with Visual Studio. However, every single one of those controls is a class that you can inherit, just like you inherited from `PictureBox` to create your `BeeControl`. If you want to add or change behavior in any of those controls, you add your own methods and properties that manipulate the ones in the base class.

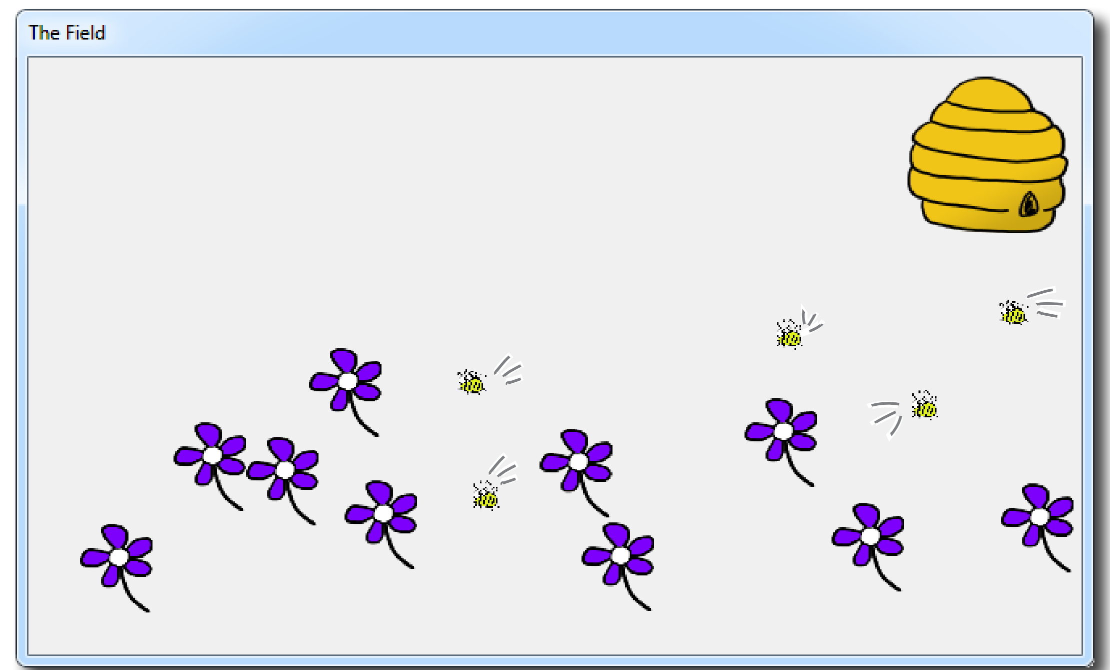
www.itbook.store/books/9781449343507

Test drive...ahem...buzz

Compile all your code, chase down any errors you're getting, and run your simulator.







www.itbook.store/books/9781449343507

Looks great, but something's not quite right...

Look closely at the bees buzzing around the hive and the flowers, and you'll notice some problems with the way they're being rendered. Remember how you set each `BeeControl`'s `BackColor` property to `Color.Transparent`? Unfortunately, that wasn't enough to keep the simulator from having some problems that are actually pretty typical of graphics programs.

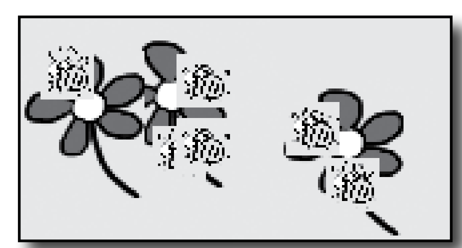
1. **There are some serious performance issues**
   Did you notice how the whole simulator slows down when all the bees are inside the hive? If not, try adding more bees by increasing the constants in the `Hive` class. Keep your eye on the frame rate—add more bees, and it starts to drop significantly.

2. **The flowers' "transparent" backgrounds aren't really transparent**
   And there's another, completely ***separate*** problem. When we saved the graphics files for the flowers, we gave them transparent backgrounds. But while that made sure that each flower's background matched the background of the form, it doesn't look so nice when flowers overlap each other.

When you set a PictureBox's background color to Transparent, it draws any transparent pixels in the image so they match the background of the form...which isn't always the right thing to do.

When one PictureBox overlaps another, C# draws the transparent pixels so they match the form, not the other control that it overlaps, causing weird rectangular "cut-outs" any time two flowers overlap.

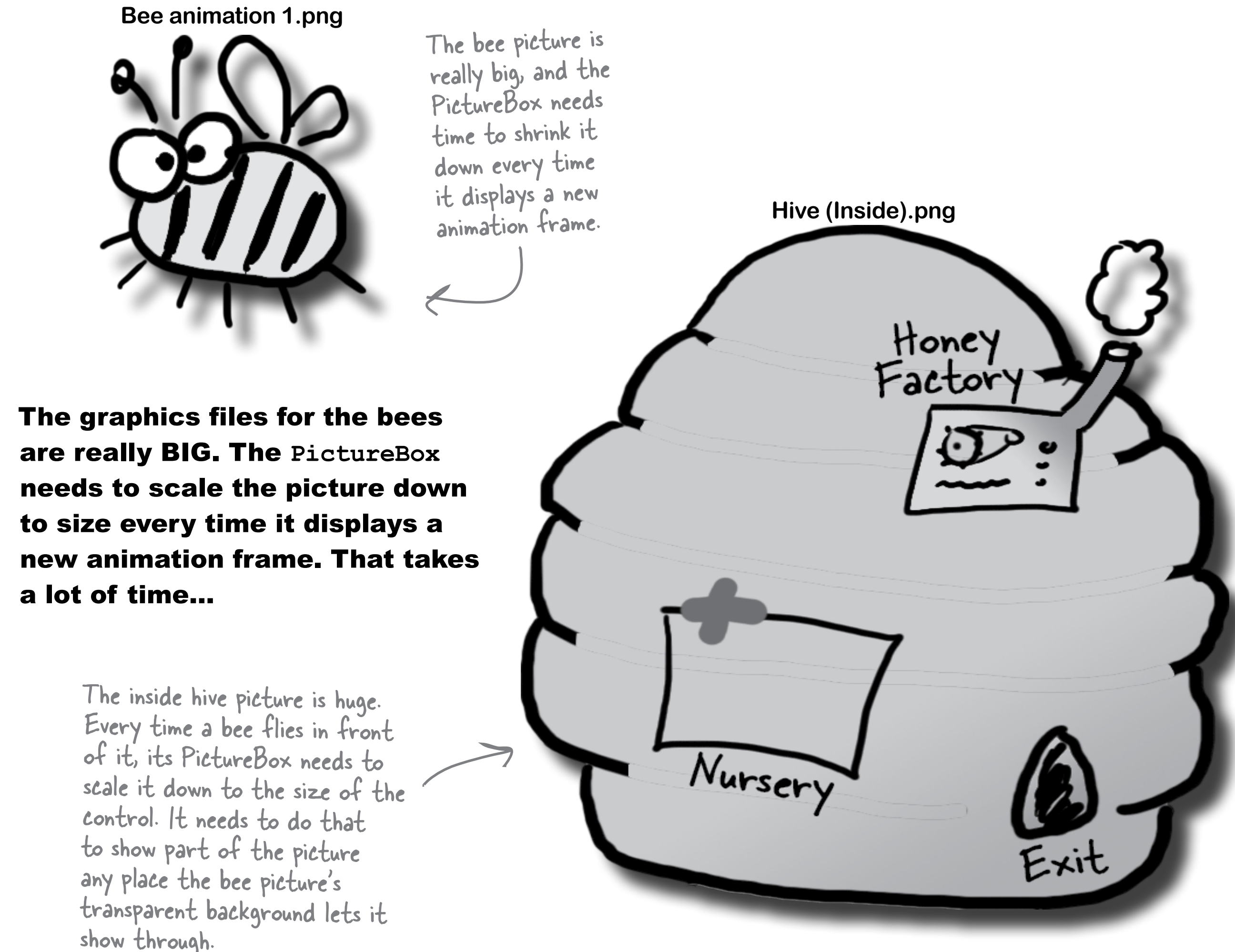
3. **The bees' backgrounds aren't transparent, either**
   It turns out that `Color.Transparent` really does have some limitations. When the bees are hovering over the flowers, the same "cut-out" glitch happens. Transparency works a little better with the hive form, where the form's background image does show through the transparent areas of the bee graphics. But when the bees overlap, the same problems occur. And if you watch closely as the bees move around the hive, you'll see some glitches where the bee images are sometimes distorted when they move.



www.itbook.store/books/9781449343507

Let's take a closer look at those performance issues

Each bee picture you downloaded is big. Really big. Pop one of them open in Windows Picture Viewer and see for yourself. That means the `PictureBox` needs to shrink it down every time it changes the image, and scaling an image up or down takes time. The reason the bees move a lot slower when there's a lot of them flying around inside the hive is that the inside hive picture is HUGE. And when you made the background for the `BeeControl` transparent, it needs to do double work: first it has to shrink the bee picture down, and then it needs to shrink a portion of the form's background down so that it can draw it in the transparent area behind the bee.



...so all we need to do to speed up the simulator's performance is to shrink down all the pictures *before* we try to display them.

www.itbook.store/books/9781449343507

All we need to do to speed up the graphics performance is add a method to the renderer that scales any image to a different size. Then we can **resize each picture once when it's loaded**, and only use the scaled-down version in the bee control and for the hive form's background.

Add the `ResizeImage` method to the renderer

All of the pictures in your project (like `Properties.Resources.Flower`) are stored as `Bitmap` objects. Here's a static method that resizes bitmaps—add it to the `Renderer` class:

```
public static Bitmap ResizeImage(Bitmap picture, int width, int height) {
    Bitmap resizedPicture = new Bitmap(width, height);
    using (Graphics graphics = Graphics.FromImage(resizedPicture)) {
        graphics.DrawImage(picture, 0, 0, width, height);
    }
    return resizedPicture;
}
```

We'll take a closer look at what this Graphics object is and how this method works in the next few pages

Add this `ResizeCells` method to your `BeeControl`

Your `BeeControl` can store its own `Bitmap` objects—in this case, an array of four of them. Here's a control that'll populate that array, resizing each one so that it's exactly the right size for the control:

```
private Bitmap[] cells = new Bitmap[4];
private void ResizeCells() {
  cells[0] = Renderer.ResizeImage(Properties.Resources.Bee_animation_1, Width, Height);
  cells[1] = Renderer.ResizeImage(Properties.Resources.Bee_animation_2, Width, Height);
  cells[2] = Renderer.ResizeImage(Properties.Resources.Bee_animation_3, Width, Height);
  cells[3] = Renderer.ResizeImage(Properties.Resources.Bee_animation_4, Width, Height);
}
```

These lines take each of the Bitmap objects that store the bee pictures and shrink them down using the ResizeImage() method we wrote.

Change the `switch` statement so that it uses the cells array, not the resources

The `BeeControl`'s `Tick` event handler has a switch statement that sets its `BackgroundImage`:

```
BackgroundImage = Properties.Resources.Bee_animation_1;
```

Replace `Properties.Resources.Bee_animation_1` with `cells[0]`. Now replace the rest of the **`case`** lines, so that case 2 uses `cells[1]`, case 3 uses `cells[2]`, case 4 uses `cells[3]`, case 5 uses `cells[2]`, and the default case uses `cells[1]`. That way only the resized image is displayed.

Add calls to `ResizeCells()` to the `BeeControl`

You'll need to add two calls to the new `ResizeCells()` method. First, **add it** to the bottom of the constructor. Then go back to the IDE designer by double-clicking on the `BeeControl` in the Properties window. Go over to the Events page in the Properties window (by clicking on the lightning-bolt icon), scroll down to Resize, and double-click on it to **add a `Resize` event handler**. Make the new `Resize` event handler call `ResizeCells()`, too—that way it'll resize its animation pictures every time the form is resized.

5 Set the form's background image manually

Go to the Properties window and set the hive form's background image to `(none)`. Then go to its constructor and set the image to one that's sized properly.

```
public partial class HiveForm : Form {
    public HiveForm() {
        InitializeComponent();
        BackgroundImage = Renderer.ResizeImage(
                Properties.Resources.Hive__inside_,
                ClientRectangle.Width, ClientRectangle.Height);
    }
}
```

Your form has a ClientRectangle property that contains a Rectangle that has the dimensions of its display area.

Now run the simulator—it's much faster!

www.itbook.store/books/9781449343507

You resized your Bitmaps using a Graphics object

Forms and controls have a CreateGraphics() method that returns a new Graphics object. You'll see a lot more about that shortly.

Let's take a closer look at that `ResizeImage()` method you added to the renderer. The first thing it does is create a new `Bitmap` object that's the size that the picture will be resized to. Then it uses `Graphics.FromImage()` to **create a new Graphics object**. It uses that `Graphics` object's `DrawImage()` method to draw the picture onto the `Bitmap`. Notice how you passed the `width` and `height` parameters to `DrawImage()`—that's how you tell it to scale the image down to the new size. Finally you returned the new `Bitmap` you created, so it can be used as the form's background image or one of the four animation cells.

You pass a picture into the method, along with a new width and height that it'll be resized to.

```
public static Bitmap ResizeImage(Bitmap picture, int width, int height) {
    Bitmap resizedPicture = new Bitmap(width, height);
    using (Graphics graphics = Graphics.FromImage(resizedPicture)) {
        graphics.DrawImage(picture, 0, 0, width, height);
    }
    return resizedPicture;
}
```

The FromImage() method returns a new Graphics object that lets you draw graphics onto that image. Take a minute and use the IDE's IntelliSense to look at the methods in the Graphics class. When you call DrawImage(), it copies the image into the resizedPicture bitmap at the location (0, 0) and scaled to the width and height parameters.

Let's see image resizing in action

Just do this temporarily. Delete the button and code when you're done.

Drag a button onto the `Field` form and add this code. It creates a new `PictureBox` control that's 100 ×100 pixels, setting its border to a black line so you can see how big it is. Then it uses `ResizeImage()` to make a bee picture that's squished down to 80×40 pixels and assigns that new picture to its `Image` property. Once the `PictureBox` is added to the form, the bee is displayed.

```
private void button1_Click(object sender, EventArgs e)
{
    PictureBox beePicture = new PictureBox();
    beePicture.Location = new Point(10, 10);
    beePicture.Size = new Size(100, 100);
    beePicture.BorderStyle = BorderStyle.FixedSingle;
    beePicture.Image = Renderer.ResizeImage(
         Properties.Resources.Bee_animation_1, 80, 40);
    Controls.Add(beePicture);
}
```

You can see the image resizing in action—the squished bee image is much smaller than the PictureBox. ResizeImage() squished it down.

The Field

button1

The ResizeImage() method creates a Graphics object to draw on an invisible Bitmap object. It returns that Bitmap so it can be displayed on a form or in a PictureBox.

www.itbook.store/books/9781449343507

Your image resources are stored in Bitmap objects

If you don't see any performance problems, keep adding bees until the program slows down!

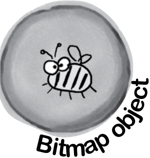
When you import graphics files into your project's resources, what happens to them? You already know that you can access them using `Properties.Resources`. But what, exactly, is your program doing with them once they're imported?

.NET turns your image into a new `Bitmap` object:

The Bitmap class has several overloaded constructors. This one loads a graphics file from disk. You can also pass it integers for width and height—that'll create a new Bitmap with no picture.

Bee animation 1.png

```
Bitmap bee = new Bitmap("Bee animation 1.png")
```



Then each Bitmap is drawn to the screen

Once your images are in objects, your form draws them to the screen with a call like this:

This call gets a Graphics object to draw on the form. We use a using statment to make sure the Graphics object is disposed.

```
using (Graphics g = CreateGraphics()) {
    g.DrawImage(myBitmap, 30, 30, 150, 150);
}
```

DrawImage() takes a Bitmap, the image to draw...

...a starting X, Y coordinate...

...and a size, 150x150 pixels.

The bigger they are...

Did you notice those last two parameters to `DrawImage()`? What if the image in the `Bitmap` is 175 by 175? The graphics library must then resize the image to fit 150 by 150. What if the `Bitmap` contains an image that's 1,500 by 2,025? Then the scaling becomes even slower....

This image, which is 300x300 pixels...

150

150

...gets shrunk to this size, which is (for example) 150x150 pixels. And that slows your simulator down!

Resizing images takes a lot of processing power! If you do it once, it's no big deal. But if you do it EVERY FRAME, your program will slow down. We gave you REALLY BIG images for the bees and the hive. When the renderer moves the bees around (especially in front of the inside hive picture), it has to resize them over and over again. And that was causing the performance problems!

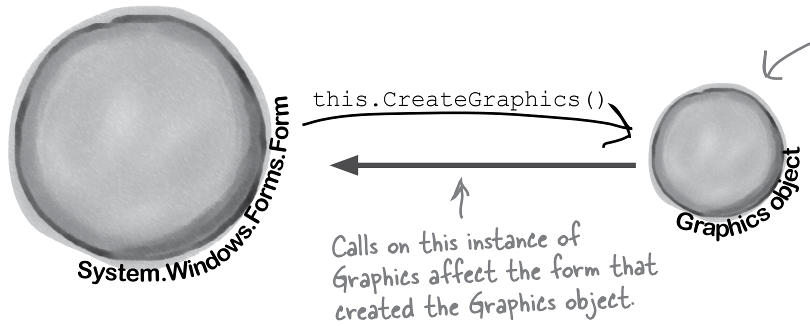
www.itbook.store/books/9781449343507

Use System.Drawing to TAKE CONTROL of graphics yourself

The Graphics object is part of the System.Drawing namespace. The .NET Framework comes with some pretty powerful graphics tools that go a lot further than the simple PictureBox control that's in the toolbox. You can draw shapes, use fonts, and do all sorts of complex graphics…and it all starts with a Graphics object. Any time you want to add or modify any object's graphics or images, you'll create a Graphics object that's **linked to the object you want to draw on**, and then use the Graphics object's methods to draw on your target.

> **System.Drawing**
>
> The graphics methods in the System.Drawing namespace are sometimes referred to as GDI+, which stands for Graphics Device Interface. When you draw graphics with GDI+, you start with a Graphics object that's hooked up to a Bitmap, form, control, or another object that you want to draw on using the Graphics object's methods.

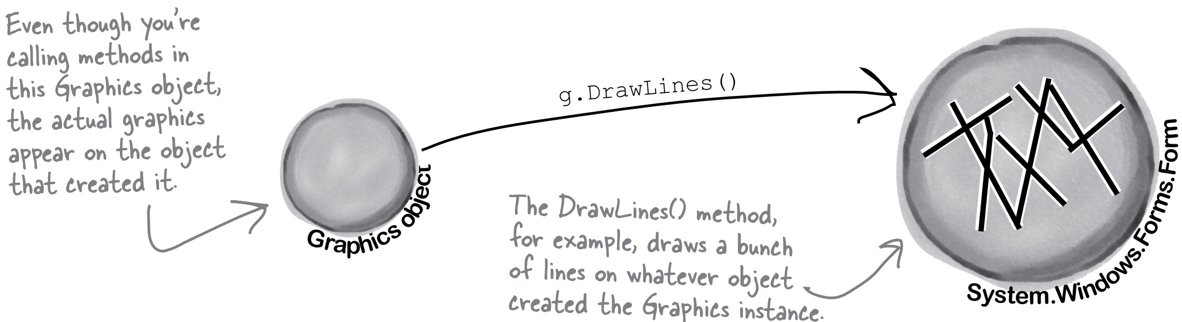
1. **Start with the object you want to draw on**
For instance, think about a form. When you call the form's CreateGraphics() method, it returns an instance of Graphics that's set up to draw on itself.



The form can call its own CreateGraphics() method, or another object can call it. Either way, the method returns a reference to a Graphics object whose methods will draw on it.

You don't draw on the Graphics object itself. You only use it to draw on other objects.

2. **Use the Graphics object's methods to draw on your object**
Every Graphics object has methods that let you draw on the object that created it. When you call methods in the Graphics object to draw lines, circles, rectangles, text, and images, they appear on the form.



www.itbook.store/books/9781449343507

A 30-second tour of GDI+ graphics

There are all sorts of shapes and pictures that you can draw once you've created a `Graphics` object. All you need to do is call its methods, and it'll draw directly onto the object that created it.

You'll need to make sure you've got a using System.Drawing; line at the top of your class to use these methods. Or, when you add a form to your project, the IDE adds that line to your form class automatically.

1. The first step is always to grab yourself a `Graphics` object. Use a form's `CreateGraphics()` method, or have a `Graphics` object passed in. Remember, `Graphics` implements the `IDisposable()` interface, so if you create a new one, use a `using` statement:

```
using (Graphics g = this.CreateGraphics()) {
```

Remember, this draws on the object that created this instance.

2. If you want to draw a line, call `DrawLine()` with a starting point and ending point, each represented by X and Y coordinates:

```
g.DrawLine(Pens.Blue, 30, 10, 100, 45);
```

The start coordinate... ...and the end coordinate.

or you can do it using a couple of `Points`:

```
g.DrawLine(Pens.Blue, new Point(30, 45), new Point(100, 10));
```

3. Here's code that draws a filled slate gray rectangle, and then gives it a sky blue border. It uses a `Rectangle` to define the dimensions—in this case, the upper left-hand corner is at (150, 15), and it's 140 pixels wide and 90 pixels high.

```
g.FillRectangle(Brushes.SlateGray, new Rectangle(150, 15, 140, 90));
g.DrawRectangle(Pens.SkyBlue, new Rectangle(150, 15, 140, 90));
```

There are a whole lot of colors you can use—just type "Color", "Pens", or "Brushes" followed by a dot, and the IntelliSense window will display them.

4. You can draw an ellipse or a circle using the `DrawCircle()` or `FillCircle()` methods, which also use a `Rectangle` to specify how big the shape should be. This code draws two ellipses that are slightly offset to give a shadow effect:

```
g.FillEllipse(Brushes.DarkGray, new Rectangle(45, 65, 200, 100));
g.FillEllipse(Brushes.Silver, new Rectangle(40, 60, 200, 100));
```

5. Use the `DrawString()` method to draw text in any font and color. To do that, you'll need to create a `Font` object. It implements `IDisposable`, so use a `using` statement:

```
using (Font arial24Bold = new Font("Arial", 24, FontStyle.Bold)) {
   g.DrawString("Hi there!", arial24Bold, Brushes.Red, 50, 75);
}
```

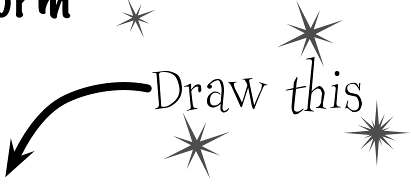
If the above statements are executed in order, this is what will end up on the form. Each of the statements above matches up with the numbers here. The upper left-hand corner is coordinate (0, 0).

There's no step 1 on this picture, since that was creating the actual Graphics object.

www.itbook.store/books/9781449343507

Use graphics to draw a picture on a form

Let's create a new Windows application that draws a picture on a form **when you click on it**.



1 Start by adding a `Click` event to the form

Go to the **Events page in the Properties window** (by clicking on the lightning-bolt icon), scroll down to the Click event, and double-click on it.

Start the event handler with a `using` line to create the `Graphics` object. When you work with GDI+, you use a lot of objects that implement `IDisposable`. If you don't dispose of them, they'll slowly suck up your computer's resources until you quit the program. So you'll end up using a lot of `using` statements:

```
using (Graphics g = CreateGraphics()) {
```

Here's the first line in your Form1_Click() event handler method. We'll give you all the lines for the event handler—put them together to draw the picture.

2 Pay attention to the order you draw things on our form

We want a sky blue background for this picture, so you'll draw a big blue rectangle first—then anything else you draw afterward will be drawn **on top of it**. You'll take advantage of one of the form's properties called `ClientRectangle`. It's a `Rectangle` that defines the boundaries of the form's drawing area. `Rectangles` are really useful—you can create a `new` rectangle by specifying a `Point` for its upper left-hand corner, and its width and height. Once you do that, it'll automatically calculate its `Top`, `Left`, `Right`, and `Bottom` properties for you. And it's got **useful methods like `Contains()`, which will return true if a given point is inside it**.

```
g.FillRectangle(Brushes.SkyBlue, ClientRectangle);
```

This will come in really handy later on in the book! What do you think you'll be doing with Contains()?

3 Draw the bee and the flower

You already know how the `DrawImage()` method works. Make sure you add the image resources.

```
g.DrawImage(Properties.Resources.Bee_animation_1, 50, 20, 75, 75);
g.DrawImage(Properties.Resources.Flower, 10, 130, 100, 150);
```

Pens are for drawing lines, and they have a width. If you want to draw a filled shape or some text, you'll need a Brush.

4 Add a pen that you can draw with

Every time you draw a line, you use a `Pen` object to determine its color and thickness. There's a built-in `Pens` class that gives you plenty of pens (`Pens.Red` is a thin red pen, for example). But you can create your own pen using the `Pen` class constructor, which takes a `Brush` object and a thickness (it's a float, so make sure it ends with `F`). Brushes are how you draw filled graphics (like filled rectangles and ellipses), and there's a `Brushes` class that gives you brushes in various colors.

```
using (Pen thickBlackPen = new Pen(Brushes.Black, 3.0F)) {
```


www.itbook.store/books/9781449343507

This goes inside the inner using statement that created the Pen.

5 Add an arrow that points to the flower

There are some `Graphics` methods that take an array of `Points`, and connect them using a series of lines or curves. We'll use the `DrawLines()` method to draw the arrow head, and the `DrawCurve()` method to draw its shaft. There are other methods that take point arrays, too (like `DrawPolygon()`, which draws a closed shape, and `FillPolygon()`, which fills it in).

```
 g.DrawLines(thickBlackPen, new Point[] {
    new Point(130, 110), new Point(120, 160), new Point(155, 163)});
 g.DrawCurve(thickBlackPen, new Point[] {
    new Point(120, 160), new Point(175, 120), new Point(215, 70) });
}
```

Here's where the using block ends—we don't need the thickBlackPen any more, so it'll get disposed.

When you pass an array of points to DrawCurve(), it draws a smooth curve that connects them all in order.

6 Add a font to draw the text

Whenever you work with drawing text, the first thing you need to do is create a `Font` object. Again, use a `using` statement because `Font` implements `IDisposable`. Creating a font is straightforward. There are several overloaded constructors—the simplest one takes a font name, font size, and `FontStyle` enum.

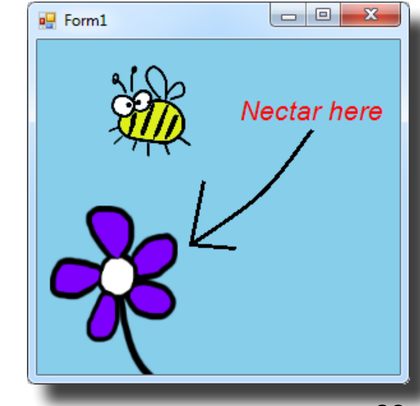
```
    using (Font font = new Font("Arial", 16, FontStyle.Italic)) {
```

7 Add some text that says "Nectar here"

Now that you've got a font, you can figure out where to put the string by measuring how big it will be when it's drawn. The `MeasureString()` method returns a `SizeF` that defines its size. (`SizeF` is just the `float` version of `Size`—and both of them just define a width and height.) Since we know where the arrow ends, we'll use the string measurements to position its center just above the arrow.

```
        SizeF size = g.MeasureString("Nectar here", font);
        g.DrawString("Nectar here", font, Brushes.Red, new Point(
             215 - (int)size.Width / 2, 70 - (int)size.Height));
    }
}
```

Make sure you close out both using blocks.



You can create a Rectangle by giving it a point and a Size (or width and height). Once you've got it, you can find its boundaries and check its Contains() method to see if it contains a Point.

www.itbook.store/books/9781449343507

Sharpen your pencil

1. Most of your work with **`Graphics`** will involve thinking about your forms as a grid of X, Y coordinates. Here’s the code to build the grid shown below; your job is to fill in the missing parts.

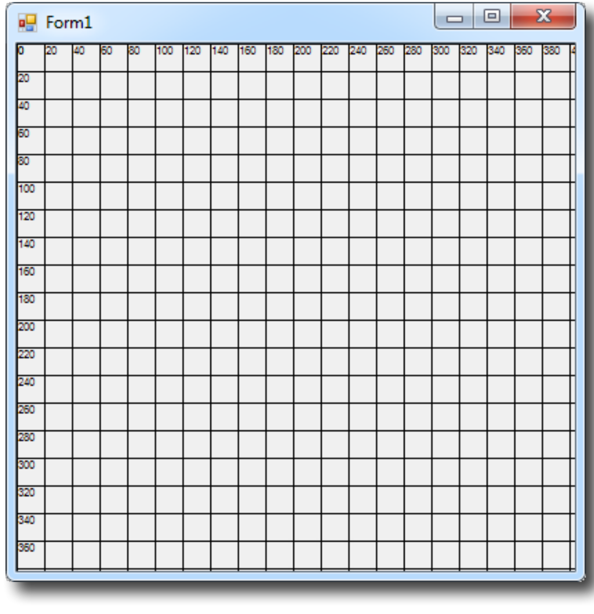
```
using (Graphics g = this.CreateGraphics())
using (Font f = new Font(“Arial”, 6, FontStyle.Regular)) {
  for (int x = 0; x < this.Width; x += 20) {

    ..............................................................................

  } ..............................................................................
  for (int y = 0; y < this.Height; y += 20) {

    ..............................................................................

  } ..............................................................................
}
```



2. Can you figure out what happens when you run the code below? Draw the output onto the form, using the grid you just rendered for locating specific points.

```
using (Pen pen =
         new Pen(Brushes.Black, 3.0F)) {
   g.DrawCurve(pen, new Point[] {
      new Point(80, 60),
      new Point(200,40),
      new Point(180, 60),
      new Point(300,40),
   });
   g.DrawCurve(pen, new Point[] {
      new Point(300,180), new Point(180, 200),
      new Point(200,180), new Point(80, 200),
   });
   g.DrawLine(pen, 300, 40, 300, 180);
   g.DrawLine(pen, 80, 60, 80, 200);
   g.DrawEllipse(pen, 40, 40, 20, 20);
   g.DrawRectangle(pen, 40, 60, 20, 300);
   g.DrawLine(pen, 60, 60, 80, 60);
   g.DrawLine(pen, 60, 200, 80, 200);
}
```

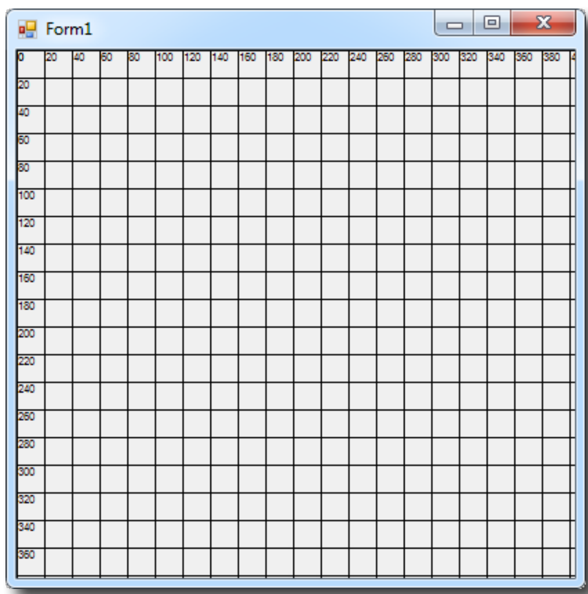

www.itbook.store/books/9781449343507

FillPolygon(), DrawLines(), and a few other graphics methods have a constructor that takes an array of Points that define the vertices of a series of connected lines.

3. Here's some more graphics code, dealing with irregular shapes. Figure out what's drawn using the grid we've given you below.

```
g.FillPolygon(Brushes.Black, new Point[] {
        new Point(60,40), new Point(140,80), new Point(200,40),
        new Point(300,80), new Point(380,60), new Point(340,140),
        new Point(320,180), new Point(380,240), new Point(320,300),
        new Point(340,340), new Point(240,320), new Point(180,340),
        new Point(20,320), new Point(60, 280), new Point(100, 240),
        new Point(40, 220), new Point(80,160),
    });

using (Font big = new Font("Times New Roman", 24, FontStyle.Italic)) {
    g.DrawString("Pow!", big, Brushes.White, new Point(80, 80));
    g.DrawString("Pow!", big, Brushes.White, new Point(120, 120));
    g.DrawString("Pow!", big, Brushes.White, new Point(160, 160));
    g.DrawString("Pow!", big, Brushes.White, new Point(200, 200));
    g.DrawString("Pow!", big, Brushes.White, new Point(240, 240));
}
```



www.itbook.store/books/9781449343507

Sharpen your pencil Solution

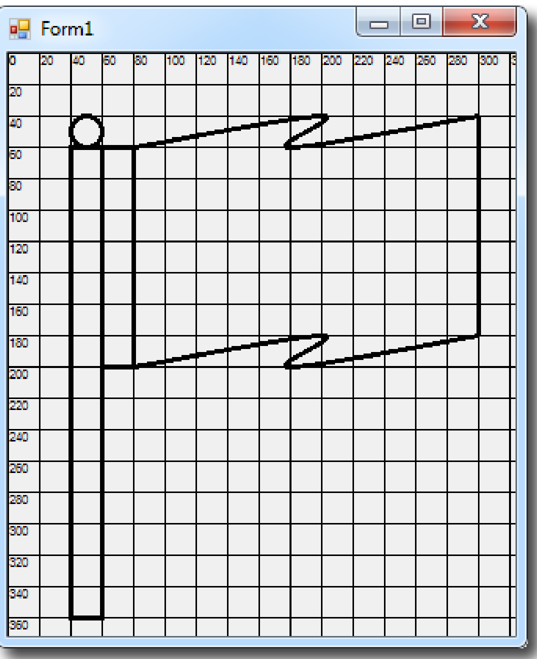
Your job was to fill in the missing code to draw a grid, and plot two chunks of code on the grids.

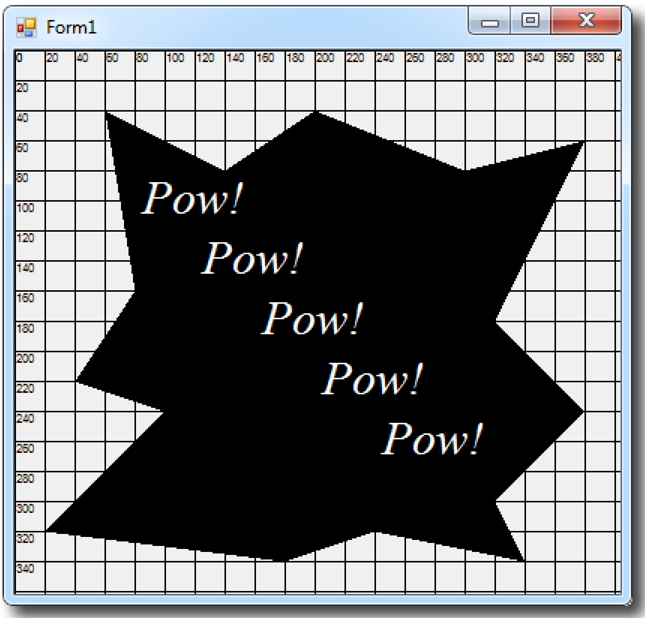
```
using (Graphics g = this.CreateGraphics())
using (Font f = new Font("Arial", 6, FontStyle.Regular)) {
    for (int x = 0; x < this.Width; x += 20) {
        g.DrawLine(Pens.Black, x, 0, x, this.Height);
        g.DrawString(x.ToString(), f, Brushes.Black, x, 0);
    }
    for (int y = 0; y < this.Height; y += 20) {
        g.DrawLine(Pens.Black, 0, y, this.Width, y);
        g.DrawString(y.ToString(), f, Brushes.Black, 0, y);
    }
}
```

First we draw the vertical lines and the numbers along the Y axis. There's a vertical line every 20 pixels along the X axis.

We used using statements to make sure the Graphics and Font objects get disposed after the form's drawn.

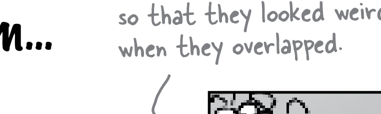
Next we draw the horizontal lines and X axis numbers. To draw a horizontal line, you choose a Y value and draw a line from (0, y) on the left-hand side of the form to (0, this.Width) on the right-hand side of the form.





www.itbook.store/books/9781449343507

Graphics can fix our transparency problem...



The renderer drew the bees so that they looked weird when they overlapped.

Remember those pesky graphics glitches? Let's tackle them! `DrawImage()` is the key to fixing the problem in the renderer where the images were drawing those boxes around the bees and flowers that caused the overlap issues. We'll start out by going back to our Windows application with the picture and changing it to draw a bunch of bees that overlap each other without any graphics glitches.

1. Add a `DrawBee()` method that draws a bee on any `Graphics` object. It uses the overloaded `DrawImage()` constructor that takes a `Rectangle` to determine where to draw the image, and how big to draw it.

```
public void DrawBee(Graphics g, Rectangle rect) {
  g.DrawImage(Properties.Resources.Bee_animation_1, rect);
}
```

Much better—click on the form and the bees overlap just fine.

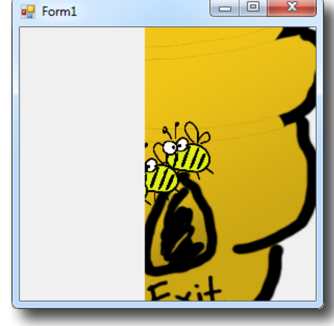
2. Here's the new **`Click` event handler for the form**. Take a close look at how it works—it draws the hive so that its upper left-hand corner is way off the form, at location `(-Width, -Height)`, and it draws it at twice the width and height of the form—so you can resize the form and it'll still draw OK. Then it draws four bees using the `DrawBee()` method.

```
private void Form1_Click(object sender, EventArgs e) {
    using (Graphics g = CreateGraphics()) {
        g.DrawImage(Properties.Resources.Hive__inside_,
            -Width, -Height, Width * 2, Height * 2);
            Size size = new Size(Width / 5, Height / 5);
            DrawBee(g, new Rectangle(
                    new Point(Width / 2 - 50, Height / 2 - 40), size));
            DrawBee(g, new Rectangle(
                    new Point(Width / 2 - 20, Height / 2 - 60), size));
            DrawBee(g, new Rectangle(
                    new Point(Width / 2 - 80, Height / 2 - 30), size));
            DrawBee(g, new Rectangle(
                    new Point(Width / 2 - 90, Height / 2 - 80), size));
    }
}
```

First we'll draw the hive background, with its corner far off the page so we only see a small piece of it. Then we'll draw four bees so that they overlap—if they don't, make your form bigger and then click on it again so they do.

...but there's a catch

But look what happens if you drag it off the side of the screen and back! Oh no!



3. Run your program and click on the form, and watch it draw the bees! But something's wrong. When you drag the form off the side of the screen and back again, **the picture disappears!** Now go back and check the "Nectar here" program you wrote a few pages ago—**it's got the same problem!**

What do you think happened?

www.itbook.store/books/9781449343507

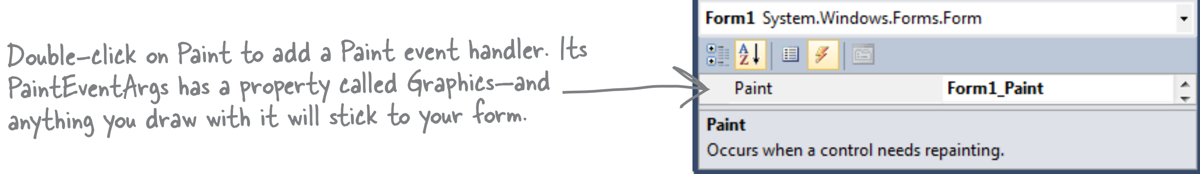
Use the Paint event to make your graphics stick

Forms and controls have a Paint event that gives you a Graphics object. Anything you draw on it is repainted automatically.

What good are graphics if they disappear from your form as soon as part of your form gets covered up? They're no good at all. Luckily, there's an easy way to make sure your graphics stay on your form: just **write a `Paint` event handler**. Your form fires a `Paint` event every time it needs to redraw itself—like when it's dragged off the screen. One of the properties of its `PaintEventArgs` parameter is a `Graphics` object called `Graphics`, and anything that you draw with it will "stick."

Add a Paint event handler

Double-click on "Paint" in the Events page in the Properties window to add a `Paint` event handler. The `Paint` event is fired any time the image on your form gets "dirty." So drawing your graphics inside of it will make your image stick around.



2 Use the Graphics object from the Paint event's EventArgs

Instead of starting with a `using` statement, make your event handler start like this:

```
private void Form1_Paint(object sender, PaintEventArgs e) {
    Graphics g = e.Graphics;
```

You **don't** have to use a `using` statement—since you didn't create it, **you don't have to dispose it**.

3 Copy the code that draws the overlapping bees and hive

Add the new `DrawBee()` method from the previous page into your new user control. Then copy the code from the `Click` event into your new `Paint` event—**except for the first line with the `using` statement**, since you already have a `Graphics` object called `g`. (Since you don't have the `using` statement anymore, make sure you take out its closing curly bracket.) Now run your program. **The graphics stick!**

Do the same with your "Nectar here" drawing to make it stick, too.

Forms and controls redraw themselves all the time

It may not look like it, but your forms have to redraw themselves all the time. Any time you have controls on a form, they're displaying graphics—labels display text, buttons display a picture of a button, checkboxes draw a little box with an X in it. You work with them as controls that you drag around, but each control actually draws its own image. Any time you drag a form off the screen or under another form and then drag it back or uncover it, the part of the form that was covered up is now invalid, which means that it no longer shows the image that it's supposed to. That's when .NET sends a message to the form telling it to redraw itself. The form fires off a Paint event any time it's "dirty" and needs to be redrawn. If you ever want your form or user control to redraw itself, you can tell .NET to make it "dirty" by calling its Invalidate() method.

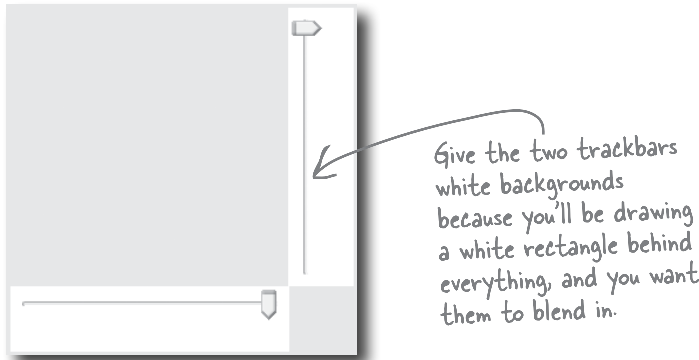
www.itbook.store/books/9781449343507

See if you can combine your knowledge of forms and user controls—and get a little more practice using `Bitmap` objects and the `DrawImage()` method—by building a user control that uses `TrackBars` to zoom an image in and out.

Add two `TrackBar` controls to a new user control

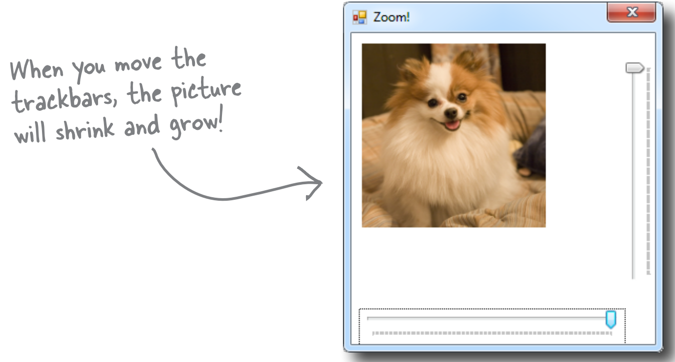
Create a new Windows Application project. **Add a `UserControl`**—call it Zoomer—and set its `Size` property to (300, 300). Drag two `TrackBar` controls out of the toolbox and onto it. Drag `trackBar1` to the bottom of the control. Then drag `trackBar2` to the right-hand side of the control and set its `Orientation` property to `Vertical`. Both should have the `Minimum` property set to 1, `Maximum` set to 175, `Value` set to 175, and `TickStyle` set to `None`. Set each `TrackBar`'s background color to white. Finally, double-click on each `TrackBar` to add a `Scroll` event handler. Make both event handlers call the control's `Invalidate()` method.

Your user control has a `Paint` event, and it works just like the one you just used in the form. Just use its `PaintEventArgs` parameter `e`. It has a property called `Graphics`, and anything that you draw with that `Graphics` object will be painted onto any instance of the user control you drag out of the toolbox.



Load a picture into a `Bitmap` object and draw it on the control

Add a private `Bitmap` field called `photo` to your Zoomer user control. When you create the instance of `Bitmap`, use its constructor to load your favorite image file—we used a picture of a fluffy dog. Then add a `Paint` event to the control. The event handler should create a graphics object to draw on the control, draw a white filled rectangle over the entire control, and then use `DrawImage()` to draw the contents of your `photo` field onto your control so its upper left-hand corner is at (10, 10), its width is `trackBar1.Value`, and its height is `trackBar2.Value`. Then drag your control onto the form—make sure to resize the form so the trackbars are at the edges.



Whenever the user scrolls one of the `TrackBars`, they call the user control's `Invalidate()` method. That will cause the user control to fire its `Paint` event and resize the photo. Remember, since you didn't create the `Graphics` object—it was passed to you in `PaintEventArgs`—you don't need to dispose it. So you don't have to use a `using` statement with it. Just draw the image inside the `Paint` event handler.

www.itbook.store/books/9781449343507

Exercise Solution

Get a little more practice using `Bitmap` objects and the `DrawImage()` method by building a form that uses them to load a picture from a file and zoom it in and out.

```
public partial class Zoomer : UserControl {

  Bitmap photo = new Bitmap(@"c:\Graphics\fluffy_dog.jpg");

  public Zoomer() {
     InitializeComponent();
  }

  private void Zoomer_Paint(object sender, PaintEventArgs e) {
      Graphics g = e.Graphics;
      g.FillRectangle(Brushes.White, 0, 0, Width, Height);
      g.DrawImage(photo, 10, 10, trackBar1.Value, trackBar2.Value);
  }

  private void trackBar1_Scroll(object sender, EventArgs e) {
     Invalidate();
  }
  private void trackBar2_Scroll(object sender, EventArgs e) {
     Invalidate();
  }
}
```

This particular Bitmap constructor loads its picture from a file. It's got other overloaded constructors, including one that lets you specify a width and height—that one creates an empty bitmap.

Substitute your own file—the Bitmap constructor can take many file formats. Even better, see if you can use an OpenFileDialog to zoom any image you want!

First we draw a big white rectangle so it fills up the whole control, then we draw the photo on top of it. The last two parameters determine the size of the image being drawn—trackBar1 sets the width, trackBar2 sets the height.

Every time the user slides one of the trackbar controls, it fires off a Scroll event. By making the event handlers call the control's Invalidate() method, we cause the form to repaint itself...and when it does, it draws a new copy of the image with a different size.

Zoom!

Each drag here is causing another image resize from DrawImage().

```
g.DrawImage(myBitmap, 30, 30, 150, 150);
```

??? ???

www.itbook.store/books/9781449343507

A closer look at how forms and controls repaint themselves

Earlier, we said that when you start working with `Graphics` objects, you're really taking control of graphics. It's like you tell .NET, "Hey, I know what I'm doing, I can handle the extra responsibility." In the case of drawing and redrawing, you may not want to redraw when a form is minimized and maximized…or you may want to redraw ***more often***. Once you know what's going on behind the scenes with your form or control, you can take control of redrawing yourself:

1 Every form has a Paint event that draws the graphics on the form

Go to the event list for any form and find the event called **`Paint`**. Whenever the form has to repaint itself, this event is fired. Every form and control uses a `Paint` event internally to decide when to redraw itself. But what fires that event? It's called by a method called **`OnPaint`** that the form or user control inherits from the `Control` class. (That method follows the pattern you saw in Chapter 11, where methods that fire an event are named "On" followed by the event name.) Go to any form and override `OnPaint`:

Do this just like you did earlier with Dispose()

Override OnPaint on any form and add this line.

```
protected override void OnPaint(PaintEventArgs e) {
    Console.WriteLine("OnPaint {0} {1}", DateTime.Now, e.ClipRectangle);
    base.OnPaint(e);
}
```

Drag your form around—drag it halfway off the screen, minimize it, hide it behind other windows. Look closely at the output that it writes. You'll see that your `OnPaint` method fires off a `Paint` event any time part of it is "dirty"—or **invalid**—and needs to be redrawn. And if you look closely at the `ClipRectangle`, you'll see that it's a rectangle that describes the part of the form that needs to be repainted. That gets passed to the `Paint` event's `PaintEventArgs` so it can improve performance by only redrawing the portion that's invalid.

2 `Invalidate()` controls when to redraw, and WHAT to redraw

.NET fires the `Paint` event when something on a form is interfered with, covered up, or moved offscreen, and then shown again. It calls `Invalidate()`, and passes the method a `Rectangle`. The `Rectangle` tells the `Invalidate()` method what part of the form needs to be redrawn…i.e., what part of the form is "dirty." Then .NET calls `OnPaint` to tell your form to fire a `Paint` event and repaint the dirty area.

Invalidate() essentially says that some part of the form might be "invalid," so redraw that part to make sure it's got the right things showing.

So when you call it yourself, you're telling .NET that your whole form or control is invalid, and the whole thing needs to be redrawn. You can pass it your own clip rectangle if you want—that'll get passed along to the Paint event's PaintEventArgs.

3 The `Update()` method gives your `Invalidate` request top priority

You may not realize it, but your form is getting messages all the time. The same system that tells it that it's been covered up and calls `OnPaint` has all sorts of other messages it needs to send. See for yourself: type `override` and scroll through all the methods that start with "On"—every one of them is a message your form responds to. The `Update()` method moves the Invalidate message to the top of the message list.

4 The form's `Refresh()` method is `Invalidate()` plus `Update()`

Forms and controls give you a shortcut. They have a `Refresh()` method that first calls `Invalidate()` to invalidate the whole client area (the area of the form where graphics appear), and then calls `Update()` to make sure that message moves to the top of the list.

www.itbook.store/books/9781449343507

there are no Dumb Questions

Q: It still seems like just resizing the graphics in a program like Paint or PhotoShop would be better. Why can't I do that?

A: You can, if you're in control of the images you work with in your applications, and if they'll always stay the same size. But that's not often the case. Lots of times, you'll get images from another source, whether it's online or a co-worker in the design group. Or, you may be pulling an image from a read-only source, and you'll have to size it in code.

Q: But if I can resize it outside of .NET, that's better, right?

A: If you're sure you'll never need a larger size, it could be. But if your program might need to display the image in multiple sizes during the program, you'll have to resize at some point anyway. Plus, if your image ever needs to be displayed larger than the resize, you'll end up in real trouble. It's much easier to size down than it is to size up.

More often than not, it's better to be able to resize an image programmatically, than to be limited by an external program or constraints like read-only files.

Q: I get that `CreateGraphics()` gets the `Graphics` object for drawing on a form, but what was that `FromImage()` call in the `ResizeImage()` method about?

A: `FromImage()` retrieves the `Graphics` object for a `Bitmap` object. And just as `CreateGraphics()` called on a form returns the `Graphics` object for drawing on that form, `FromImage()` retrieves a `Graphics` object for drawing on the `Bitmap` the method was called on.

Q: So a `Graphics` object isn't just for drawing on a form?

A: Actually, a `Graphics` object is for drawing on, well, anything that gives you a `Graphics` object. The `Bitmap` gives you a `Graphics` object that you can use to draw onto an invisible image that you can use later. And you'll find `Graphics` objects on a lot more than forms. Drag a button onto a form, then go into your code and type its name followed by a period. Check out the IntelliSense window that popped up—it's got a `CreateGraphics()` method that returns a `Graphics` object. Anything you draw on it will show up on the button! Same goes for `Label`, `PictureBox`, `StatusStrip`...almost every toolbox control has a `Graphics` object.

Q: Wait, I thought `using` was just something I used with streams. Why am I using `using` with graphics?

A: The `using` keyword comes in handy with streams, but it's something that you use with *any* class that implements the `IDisposable` interface. When you instantiate a class that implements `IDisposable`, you should always call its `Dispose()` method when you're done with the object. That way it knows to clean up after itself. With streams, the `Dispose()` method makes sure that any file that was opened gets closed.

`Graphics`, `Pen`, and `Brush` objects are also disposable. When you create any of them, they take up some small amount of memory and other resources, and they don't always give them back immediately. If you're just drawing something once, you won't notice a difference. But most of the time, your graphics code will be called over and over and over again—like in a `Paint` event handler, which could get called many times a second for a particularly busy form. That's why you should always `Dispose()` of your graphics-related objects. And the easiest way to make sure that you do is to use a `using` line, and let .NET worry about disposal. Any object you create with `using` will automatically have its `Dispose()` method called at the end of the block following the `using` statement. That will guarantee that your program won't slowly take up more and more memory if it runs for a long time.

Q: If I'm creating a new control, should I use a `UserControl` or should I create a class that inherits from one of the toolbox controls?

A: That depends on what you want your new control to do. If you're building a control that's really similar to one that's already in the toolbox, then you'll probably find it easiest to inherit from that control. But most of the time, when programmers create new controls in C#, they use user controls. One advantage of a user control is that you can **drag toolbox controls onto it**. It works a lot like a `GroupBox` or another container control—you can drag a button or checkbox onto your user control, and work with them just like you'd work with controls on a form. The IDE's form designer becomes a powerful tool to help you design user controls.

A user control can host other controls. The IDE's form designer lets you drag controls out of the toolbox and onto your new user control.

www.itbook.store/books/9781449343507

I noticed a whole lot of flickering in my Zoomer control. With all this talk of taking control of graphics, I'll bet there's something we can do about that! But why does it happen?

Even without resizing, it takes time to draw an image onto a form.

Suppose you've got every image in the simulator resized. It still takes time to draw all those bees and flowers and the hive. And right now, we're drawing right to the `Graphics` object on the form. So if your eye catches the tail end of a render, you're going to perceive it as a little flicker.

The problem is that a lot of drawing is happening, so there's a good chance that some flickering will occur, even with our resizing. And that's why you run into problems with some amateur computer games, for example: the human eye catches the end of a rendering cycle, and perceives it as a little bit of flickering on the screen.

How could you get rid of this flicker? If drawing lots of images onto the form causes flickering, and you have to draw lots of images, how do you think you might be able to avoid all the flickering?

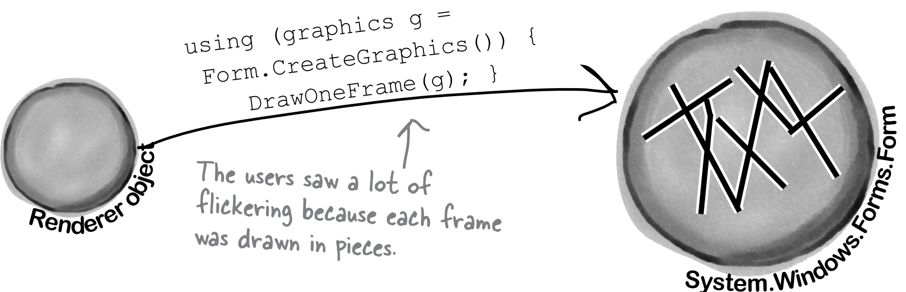
Here's a quick tip to make your resized graphics look better. Before you call a Graphics object's `DrawImage()` method, try setting its InterpolationMode property to InterpolationMode.HighQualityBicubic. (You'll need to add "`using System.Drawing.2D;`" to the top of your code.) You can learn more about how InterpolationMode works here: http://msdn.microsoft.com/en-us/library/k0fsyd4e.aspx

www.itbook.store/books/9781449343507

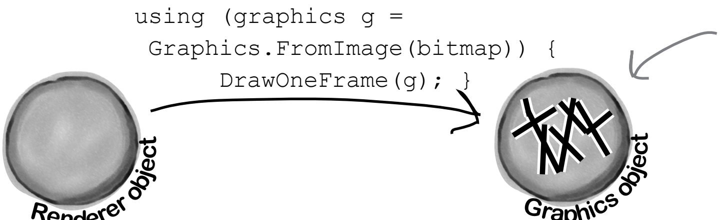
Double buffering makes animation look a lot smoother

Go back to your image zoomer and fiddle with the trackbars. Notice how there's a whole lot of flickering when you move the bars? That's because the `Paint` event handler first has to draw the white rectangle and then draw the image every time the trackbar moves a tiny little bit. When your eyes see alternating white rectangles and images many times a second, they interpret that as a flicker. It's irritating…and it's avoidable using a technique called **double buffering**. That means drawing each frame or cell of animation to an invisible bitmap (a "buffer"), and only displaying the new frame once it's been drawn entirely. Here's how it would work with a `Bitmap`:

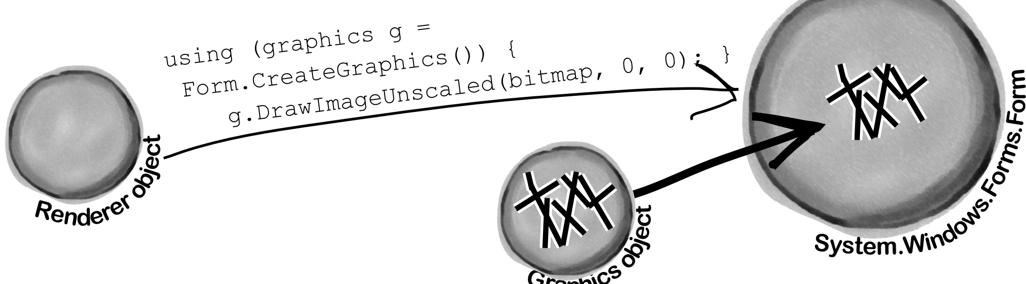
1. Here's a typical program that draws some graphics on a form using its `Graphics` object.



2. To do double buffering, we can add a object to the program to act as a buffer. Every time our form or control needs to be repainted, instead of drawing the graphics directly on the form, we draw on the buffer instead.



3. Now that the frame is completely drawn out to the invisible `Bitmap` object, we can use `DrawImageUnscaled()` to copy the object back to the form's `Graphics`. It all gets copied at once, and that eliminates the flicker.



www.itbook.store/books/9781449343507

Double buffering is built into forms and controls

You can do double buffering yourself using a `Bitmap`, but C# and .NET make it even easier with built-in support for double buffering. **All you need to do is set its `DoubleBuffered` property to true.** Try it out on your Zoomer user control—go to its Properties window, set `DoubleBuffered` to true, and your control will stop flickering! Now **go back to your `BeeControl`** and do the same. That won't fix all of the graphics problems—we'll do that in a minute—but it *will* make a difference.

Now you're ready to fix the graphics problems in the simulator!

When you use the Paint event for all your graphics, you can turn on double buffered painting simply by changing one property.

Overhaul the beehive simulator

In the next exercise, you'll take your beehive simulator and completely overhaul it. You'll probably want to create a whole new project and use "Add >> Existing Item…" to add the current files to it so you have a backup of your current simulator. (Don't forget to change their namespace to match your new project.)

Here's what you're going to do:

1. **You'll start by removing the `BeeControl` user control**
   There won't be any controls on the hive and field at all. No `BeeControls`, no `PictureBoxes`, nothing. The bees, flowers, and hive pictures will all be drawn using GDI+ graphics. So right-click on `BeeControl.cs` in the Solution Explorer and click Delete—they'll be removed from the project and permanently deleted.

2. **You'll need a timer to handle the bee wing flapping**
   The bees flap their wings much more slowly than the simulator's frame rate, so you'll need a second, slower timer. This shouldn't be too surprising, since the `BeeControl` had its own timer to do the same thing.

3. **The big step: overhaul the renderer**
   You'll need to throw out the current renderer entirely, because it does everything with controls. You won't need those lookup dictionaries, because there won't be any `PictureBoxes` or `BeeControls` to look up. Instead, it'll have two important methods: `DrawHive(g)` will draw a `Hive` form on a graphics object, and `DrawField(g)` will draw a `Field` form.

4. **Last of all, you'll hook up the new renderer**
   The `Hive` and `Field` forms will need `Paint` event handlers. Each of them will call the `Renderer` object's `DrawField(g)` or `DrawHive(g)` methods. The two timers—one for telling the simulator to draw the next frame, and the other to flap the bees' wings—will call the two forms' `Invalidate()` methods to repaint themselves. When they do, their `Paint` event handlers will render the frame.

Let's get started! →

www.itbook.store/books/9781449343507

Exercise

It's time to get rid of the graphics glitches in the beehive simulator. Use graphics and double buffering to make the simulator look polished.

1 Change the main form's `RunFrame()` method

You'll need to remove the call to `Renderer.Render()` and add two `Invalidate()` statements.

```
public void RunFrame(object sender, EventArgs e) {
    framesRun++;
    world.Go(random);
    end = DateTime.Now;
    TimeSpan frameDuration = end - start;
    start = end;
    UpdateStats(frameDuration);
    hiveForm.Invalidate();
    fieldForm.Invalidate();
}
```

You'll need to remove the call to renderer.Render(), since that method will go away.

As long as you keep the world up to date and both forms have a reference to the renderer object, all you need to do to animate them is call their Invalidate() methods. Their Paint event handlers will take care of the rest.

2 Add a second timer to the main form to make the bees' wings flap

Drag a new timer onto the main form and set its `Interval` to 150ms and `Enabled` to true. Then double-click on it and add this event handler:

```
private void timer2_Tick(object sender, EventArgs e) {
    renderer.AnimateBees();
}
```

Then add this `AnimateBees()` method to the renderer to make the bees' wings flap:

```
private int cell = 0;
private int frame = 0;
public void AnimateBees() {
    frame++;
    if (frame >= 6)
    frame = 0;
    switch (frame) {
        case 0: cell = 0; break;
        case 1: cell = 1; break;
        case 2: cell = 2; break;
        case 3: cell = 3; break;
        case 4: cell = 2; break;
        case 5: cell = 1; break;
        default: cell = 0; break;
    }
    hiveForm.Invalidate();
    fieldForm.Invalidate();
}
```

The whole idea here is to set a field called Cell that you can use when you're drawing the bees in the renderer. Make sure you're always drawing BeeAnimationLarge[Cell] in the hive form and BeeAnimationSmall[Cell] in the field form. The timer will constantly call the AnimateBees() method, which will cause the cell field to keep changing, which will cause your bees to flap their wings.

If your bees are flying to the wrong places, make sure your locations are correct! Use the event trick from earlier in the chapter to find the right coordinates.

www.itbook.store/books/9781449343507

The hive form and field form both need a public property

Add a public `Renderer` property to the hive form and the field form:

```
public Renderer Renderer { get; set; }
```

Add this to both forms.

Don't forget to add these access modifiers!

To make this work, you'll need to **change the declaration** of your `Renderer` to add the `public` modifier: `public class Renderer`. You'll also need to **do the same** for the `World`, `Hive`, `Bee`, and `Flower` classes and the `BeeState` enum—add the `public` access modifier to each of their declarations. *(See **Leftover #2 in the Appendix** to understand why!)*

There are two places where you create a `new Renderer()`: in the open button (underneath a call to `renderer.Reset()` and in the `ResetSimulator()` method. **Remove all calls to .** Then update your `Renderer`'s constructor to set each form's `Renderer` property:

```
    hiveForm.Renderer = this;
    fieldForm.Renderer = this;
```

All the Reset() method did was remove the controls from the forms, and there won't be any controls to remove.

Set up the hive and field forms for double-buffered animation

Remove the code from the hive form's constructor that sets the background image. Then remove all controls from both forms and **set their `DoubleBuffered` properties to `true`**. Finally, add a `Paint` event handler to each of them. Here's the handler for the hive form—the field form's `Paint` event handler is identical, except that it calls `Renderer.PaintField()` instead of `Renderer.PaintHive()`:

```
private void HiveForm_Paint(object sender, PaintEventArgs e) {
    Renderer.PaintHive(e.Graphics);
}
```

Make sure you turn on double buffering, or your forms will flicker!

Overhaul the renderer by removing control-based code and adding graphics

Here's what you need to do to fix the renderer:

- Remove the two dictionaries, since there aren't any more controls. And while you're at it, you don't need the `BeeControl` anymore, or the `Render()`, `DrawBees()`, or `DrawFlowers()` methods.
- Add some `Bitmap` fields called `HiveInside`, `HiveOutside`, and `Flower` to store the images. Then create two `Bitmap[]` arrays called `BeeAnimationLarge` and `BeeAnimationSmall`. Each of them will hold four bee pictures—the large ones are 40×40 and the small are 20x20. Create a method called `InitializeImages()` to resize the resources and store them in these fields, and call it from the `Renderer` class constructor.
- Add the `PaintHive()` method that takes a `Graphics` object as a parameter and paints the hive form onto it. First draw a sky blue rectangle, then use `DrawImageUnscaled()` to draw the inside hive picture, then use `DrawImageUnscaled()` to draw each bee that is inside the hive.
- Finally, add the `PaintField()` method. It should draw a sky blue rectangle on the top half of the form, and a green rectangle on the bottom half. You'll find two form properties helpful for this: `ClientSize` and `ClientRectangle` tell you how big the drawing area is, so you can find half of its height using `ClientSize.Height / 2`. Then use `FillEllipse()` to draw a yellow sun in the sky, `DrawLine()` to draw a thick line for a branch the hive can hang from, and `DrawImageUnscaled()` to draw the outside hive picture. Then draw each flower onto the form. Finally, draw each bee (using the small bee pictures)—draw them last so they're in front of the flowers.
- When you're drawing the bees, remember that `AnimateBees()` sets the `cell` field.

www.itbook.store/books/9781449343507

Exercise Solution

It's time to get rid of the graphics glitches in the beehive simulator. Use graphics and double buffering to make the simulator look polished.

Here's the complete Renderer class, including the AnimateBees() method that we gave you. Make sure you make all the modifications to the three forms—especially the Paint event handlers in the hive and field forms. Those event handlers call the renderer's PaintHive() and PaintField() methods, which do all of the animation.

```
using System.Drawing;

public class Renderer {
    private World world;
    private HiveForm hiveForm;
    private FieldForm fieldForm;

    public Renderer(World TheWorld, HiveForm hiveForm, FieldForm fieldForm) {
        this.world = TheWorld;
        this.hiveForm = hiveForm;
        this.fieldForm = fieldForm;
        fieldForm.Renderer = this;
        hiveForm.Renderer = this;
        InitializeImages();
    }

    public static Bitmap ResizeImage(Image ImageToResize, int Width, int Height) {
        Bitmap bitmap = new Bitmap(Width, Height);
        using (Graphics graphics = Graphics.FromImage(bitmap)) {
            graphics.DrawImage(ImageToResize, 0, 0, Width, Height);
        }
        return bitmap;
    }

    Bitmap HiveInside;
    Bitmap HiveOutside;
    Bitmap Flower;
    Bitmap[] BeeAnimationSmall;
    Bitmap[] BeeAnimationLarge;
    private void InitializeImages() {
        HiveOutside = ResizeImage(Properties.Resources.Hive__outside_, 85, 100);
        Flower = ResizeImage(Properties.Resources.Flower, 75, 75);
        HiveInside = ResizeImage(Properties.Resources.Hive__inside_,
            hiveForm.ClientRectangle.Width, hiveForm.ClientRectangle.Height);
        BeeAnimationLarge = new Bitmap[4];
        BeeAnimationLarge[0] = ResizeImage(Properties.Resources.Bee_animation_1, 40, 40);
        BeeAnimationLarge[1] = ResizeImage(Properties.Resources.Bee_animation_2, 40, 40);
        BeeAnimationLarge[2] = ResizeImage(Properties.Resources.Bee_animation_3, 40, 40);
        BeeAnimationLarge[3] = ResizeImage(Properties.Resources.Bee_animation_4, 40, 40);
        BeeAnimationSmall = new Bitmap[4];
        BeeAnimationSmall[0] = ResizeImage(Properties.Resources.Bee_animation_1, 20, 20);
        BeeAnimationSmall[1] = ResizeImage(Properties.Resources.Bee_animation_2, 20, 20);
        BeeAnimationSmall[2] = ResizeImage(Properties.Resources.Bee_animation_3, 20, 20);
        BeeAnimationSmall[3] = ResizeImage(Properties.Resources.Bee_animation_4, 20, 20);
    }
```

* Don't forget to change the class declaration in Renderer.cs from class Renderer to public class Renderer, and then do the same for World, Hive, Flower, and Bee; otherwise, you'll get a **build error** about field and type accessibility. Flip to Leftover #2 in the Appendix to learn about why you need to do this.

The InitializeImages() method resizes all of the image resources and stores them in Bitmap fields inside the Renderer object. That way the PaintHive() and PaintForm() methods can draw the images unscaled using the forms' Graphics objects' DrawImageUnscaled() methods.

www.itbook.store/books/9781449343507

```
    public void PaintHive(Graphics g) {
        g.FillRectangle(Brushes.SkyBlue, hiveForm.ClientRectangle);
        g.DrawImageUnscaled(HiveInside, 0, 0);
        foreach (Bee bee in world.Bees) {
            if (bee.InsideHive)
                g.DrawImageUnscaled(BeeAnimationLarge[cell],
                                    bee.Location.X, bee.Location.Y);
        }
    }

    public void PaintField(Graphics g) {
        using (Pen brownPen = new Pen(Color.Brown, 6.0F)) {
            g.FillRectangle(Brushes.SkyBlue, 0, 0,
                     fieldForm.ClientSize.Width, fieldForm.ClientSize.Height / 2);
            g.FillEllipse(Brushes.Yellow, new RectangleF(50, 15, 70, 70));
            g.FillRectangle(Brushes.Green, 0, fieldForm.ClientSize.Height / 2,
                     fieldForm.ClientSize.Width, fieldForm.ClientSize.Height / 2);
            g.DrawLine(brownPen, new Point(593, 0), new Point(593, 30));
            g.DrawImageUnscaled(HiveOutside, 550, 20);
            foreach (Flower flower in world.Flowers) {
                g.DrawImageUnscaled(Flower, flower.Location.X, flower.Location.Y);
            }
            foreach (Bee bee in world.Bees) {
                if (!bee.InsideHive)
                    g.DrawImageUnscaled(BeeAnimationSmall[cell],
                                        bee.Location.X, bee.Location.Y);
            }
        }
    }

    private int cell = 0;
    private int frame = 0;
    public void AnimateBees() {
        frame++;
        if (frame >= 6)
        frame = 0;
        switch (frame) {
            case 0: cell = 0; break;
            case 1: cell = 1; break;
            case 2: cell = 2; break;
            case 3: cell = 3; break;
            case 4: cell = 2; break;
            case 5: cell = 1; break;
            default: cell = 0; break;
        }
        hiveForm.Invalidate();
        fieldForm.Invalidate();
    }

}
```

A form's ClientSize property is a Rectangle that tells you how big its drawing area is.

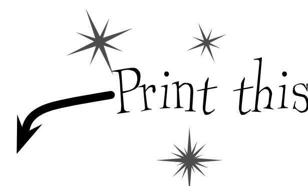
The PaintField() method looks at the bees and flowers in the world and draws a field using their locations. First it draws the sky and the ground, then it draws the sun, and then the beehive. After that, it draws the flowers and the bees. It's important that everything is drawn in the right order—if it were to draw the flowers before the bees, then the bees would look like they were flying behind the flowers.

Here's the same AnimateBees() method from the exercise. It cycles through the animations using the Frame field—first it shows cell 0, then cell 1, then 2, then 3, and then back to 2, then 1 again. That way the wing flapping animation is smooth.

www.itbook.store/books/9781449343507

Use a Graphics object and an event handler for printing

The `Graphics` methods you've been using to draw on your forms are **the same ones you use to print**. .NET's printing objects in `System.Drawing.Printing` make it really easy to add printing and print preview to your applications. All you need to do is **create a `PrintDocument` object**. It's got an event called `PrintPage`, which you can use exactly like you use a timer's `Tick` event. Then call the `PrintDocument` object's `Print()` method, and it prints the document. And remember, the IDE makes it especially easy to add the event handler. Here's how:



1. **Start a new Windows application** and add a button to the form. Go to the form code and add a **`using System.Drawing.Printing;`** line to the top. Double-click on the button and add the event handler. Watch what happens as soon as you type **+=**:

```
private void button1_Click(object sender, EventArgs e) {
    PrintDocument document = new PrintDocument();
    document.PrintPage +=
```

`new PrintPageEventHandler(document_PrintPage);     (Press TAB to insert)`

2. Press Tab and the IDE automatically fills in the rest of the line. This is just like how you added event handlers in Chapter 11:

```
private void button1_Click(object sender, EventArgs e) {
    PrintDocument document = new PrintDocument();
    document.PrintPage += new PrintPageEventHandler(document_PrintPage);
```

`Press TAB to generate handler 'document_PrintPage' in this class`

3. As soon as you press Tab, the IDE generates an event handler method and adds it to the form.

```
void document_PrintPage(object sender, PrintPageEventArgs e) {
    throw new NotImplementedException();
}
```

Now you can put ANY graphics code here—just replace the throw line and use e.Graphics for all of the drawing. We'll show you how in a minute....

The `PrintPageEventArgs` parameter `e` has a `Graphics` property. Just replace the `throw` statement with code that calls the `e.Graphics` object's drawing methods.

4. Now finish off the `button1_Click` event handler by calling **`document.Print()`**. When that method is called, the `PrintDocument` object creates a `Graphics` object and then fires off a `PrintPage` event with the `Graphics` object as a parameter. Anything that the event handler draws onto the `Graphics` object will get sent to the printer.

```
private void button1_Click(object sender, EventArgs e) {
    PrintDocument document = new PrintDocument();
    document.PrintPage += new PrintPageEventHandler(document_PrintPage);
    document.Print();
}
```


www.itbook.store/books/9781449343507

PrintDocument works with the print dialog and print preview window objects

Adding a print preview window or a print dialog box is a lot like adding an open or save dialog box. All you need to do is create a `PrintDialog` or `PrintPreviewDialog` object, set its `Document` property to your `Document` object, and then call the dialog's `Show()` method. The dialog will take care of sending the document to the printer—no need to call its `Print()` method. So let's add this to the button you created in Step 1:

Once you've got a PrintDocument and an event handler to print the page, you can pop up a print preview window just by creating a new PrintPreviewDialog object.

5

```
private void button1_Click(object sender, EventArgs e) {
    PrintDocument document = new PrintDocument();
    document.PrintPage += new PrintPageEventHandler(document_PrintPage);
    PrintPreviewDialog preview = new PrintPreviewDialog();
    preview.Document = document;
    preview.ShowDialog(this);
}

void document_PrintPage(object sender,
                                   PrintPageEventArgs e) {
    DrawBee(e.Graphics, new Rectangle(0, 0, 300, 300));
}
```

We'll reuse our DrawBee() method from a few pages ago.

Print preview — Close — Page 1

Use e.HasMorePages to print multipage documents

If you need to print more than one page, all you need to do is have your `PrintPage` event handler set `e.HasMorePages` to true. That tells the `Document` that you've got another page to print. It'll call the event handler over and over again, once per page, as long as the event handler keeps setting `e.HasMorePages` to true. So modify your `Document`'s event handler to print two pages:

6

```
bool firstPage = true;
void document_PrintPage(object sender, PrintPageEventArgs e) {
    DrawBee(e.Graphics, new Rectangle(0, 0, 300, 300));
    using (Font font = new Font("Arial", 36, FontStyle.Bold)) {
        if (firstPage) {
            e.Graphics.DrawString("First page", font, Brushes.Black, 0, 0);
            e.HasMorePages = true;
            firstPage = false;
        } else {
            e.Graphics.DrawString("Second page", font, Brushes.Black, 0, 0);
            firstPage = true;
        }
    }
}
```

If you set e.HasMorePages to true, the Document object will call the event handler again to print the next page.

Now run your program again, and make sure it's displaying two pages in the print preview.

www.itbook.store/books/9781449343507

Write the code for the `Print` button in the simulator so that it pops up a print preview window showing the bee stats and pictures of the hive and the field.

1 Make the button pop up a print preview window

Add an event handler for the button's click event that pauses the simulator, pops up the print preview dialog, and then resumes the simulator when it's done. (If the simulator is paused when the button is clicked, make sure it stays paused after the preview is shown.)

2 Create the document's `PrintPage` event handler

It should create a page that looks exactly like the one on the facing page. We'll start you off:

```
private void document_PrintPage(object sender, PrintPageEventArgs e) {
    Graphics g = e.Graphics;
    Size stringSize;
    using (Font arial24bold = new Font("Arial", 24, FontStyle.Bold)) {
        stringSize = Size.Ceiling(
              g.MeasureString("Bee Simulator", arial24bold));
        g.FillEllipse(Brushes.Gray,
            new Rectangle(e.MarginBounds.X + 2, e.MarginBounds.Y + 2,
            stringSize.Width + 30, stringSize.Height + 30));
        g.FillEllipse(Brushes.Black,
            new Rectangle(e.MarginBounds.X, e.MarginBounds.Y,
            stringSize.Width + 30, stringSize.Height + 30));
        g.DrawString("Bee Simulator", arial24bold,
            Brushes.Gray, e.MarginBounds.X + 17, e.MarginBounds.Y + 17);
        g.DrawString("Bee Simulator", arial24bold,
            Brushes.White, e.MarginBounds.X + 15, e.MarginBounds.Y + 15);
    }
    int tableX = e.MarginBounds.X + (int)stringSize.Width + 50;
    int tableWidth = e.MarginBounds.X + e.MarginBounds.Width - tableX - 20;
    int firstColumnX = tableX + 2;
    int secondColumnX = tableX + (tableWidth / 2) + 5;
    int tableY = e.MarginBounds.Y;
    // Your job: fill in the rest of the method to make it print this
```

We created the oval with text in it using the MeasureString() method, which returns a Size that contains the size of a string. We drew the oval and text twice to give it a shadow effect.

You'll need these to build the table.

3 This `PrintTableRow()` method will come in handy

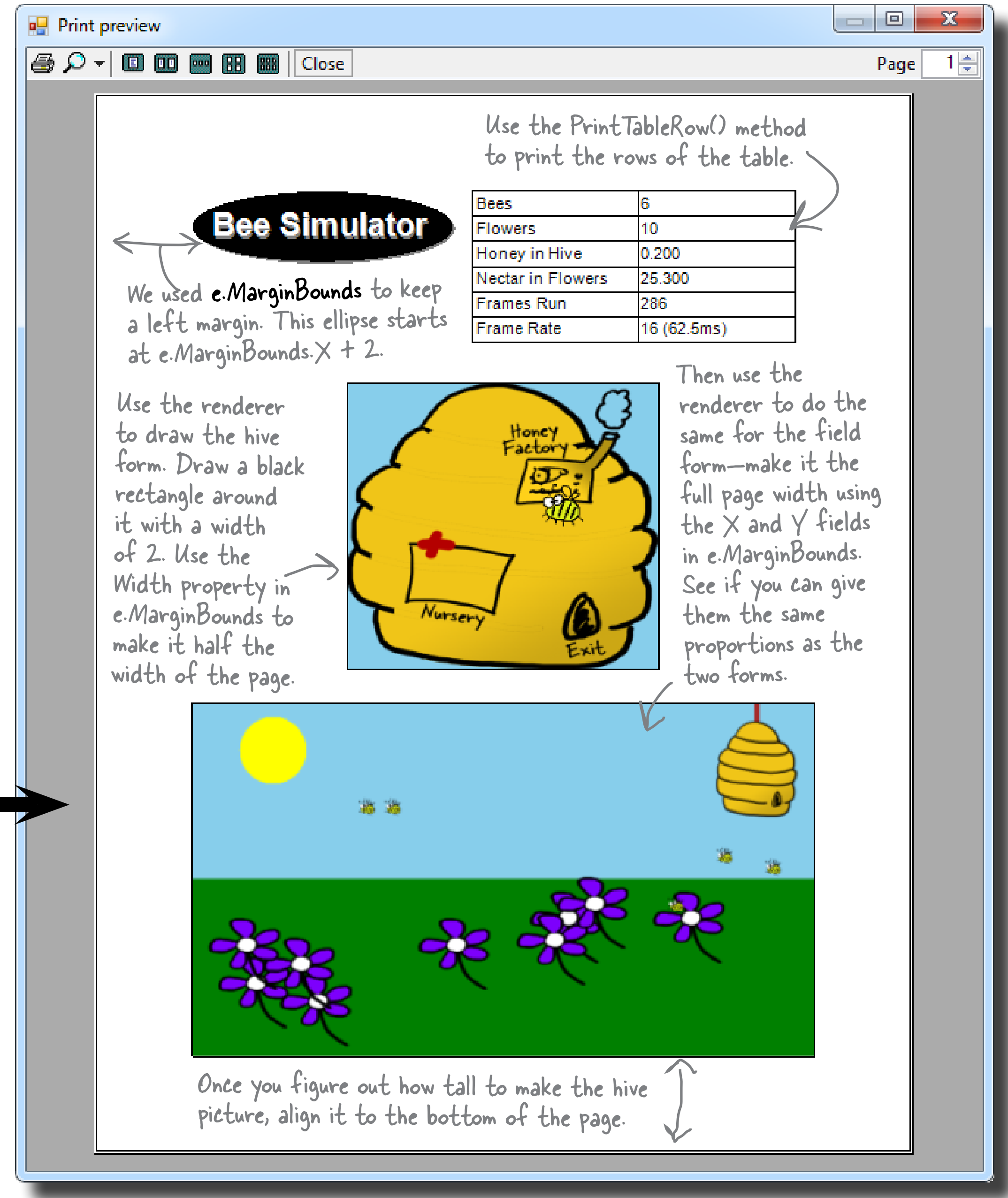
You'll find this method useful when you create the table of bee stats at the top of the page.

```
private int PrintTableRow(Graphics printGraphics, int tableX,
                 int tableWidth, int firstColumnX, int secondColumnX,
                 int tableY, string firstColumn, string secondColumn) {
  Font arial12 = new Font("Arial", 12);
  Size stringSize = Size.Ceiling(printGraphics.MeasureString(firstColumn, arial12));
  tableY += 2;
  printGraphics.DrawString(firstColumn, arial12, Brushes.Black,
                 firstColumnX, tableY);
  printGraphics.DrawString(secondColumn, arial12, Brushes.Black,
                 secondColumnX, tableY);
  tableY += (int)stringSize.Height + 2;
  printGraphics.DrawLine(Pens.Black, tableX, tableY, tableX + tableWidth, tableY);
  arial12.Dispose();
  return tableY;
}
```

Each time you call PrintTableRow(), it adds the height of the row it printed to tableY and returns the new value.

www.itbook.store/books/9781449343507

Take a close look at the notes we wrote on the printout. This is a little complex—take your time!



Here's a hint: To find the height of each form, find the ratio of its height divided by its width and multiply that by the final width. You can locate the top of the field form by subtracting its height from the bottom margin of the page: (e.MarginBounds.Y + e.MarginBounds.Height – fieldHeight).

www.itbook.store/books/9781449343507

Exercise Solution

Write the code for the `Print` button in the simulator so that it pops up a print preview window showing the bee stats and pictures of the hive and the field.

```
using System.Drawing.Printing;

private void document_PrintPage(object sender, PrintPageEventArgs e) {
   Graphics g = e.Graphics;

   Size stringSize;
   using (Font arial24bold = new Font("Arial", 24, FontStyle.Bold)) {
       stringSize = Size.Ceiling(
             g.MeasureString("Bee Simulator", arial24bold));
       g.FillEllipse(Brushes.Gray,
           new Rectangle(e.MarginBounds.X + 2, e.MarginBounds.Y + 2,
           stringSize.Width + 30, stringSize.Height + 30));
       g.FillEllipse(Brushes.Black,
           new Rectangle(e.MarginBounds.X, e.MarginBounds.Y,
           stringSize.Width + 30, stringSize.Height + 30));
       g.DrawString("Bee Simulator", arial24bold,
           Brushes.Gray, e.MarginBounds.X + 17, e.MarginBounds.Y + 17);
       g.DrawString("Bee Simulator", arial24bold,
           Brushes.White, e.MarginBounds.X + 15, e.MarginBounds.Y + 15);
   }

   int tableX = e.MarginBounds.X + (int)stringSize.Width + 50;
   int tableWidth = e.MarginBounds.X + e.MarginBounds.Width - tableX - 20;
   int firstColumnX = tableX + 2;
   int secondColumnX = tableX + (tableWidth / 2) + 5;
   int tableY = e.MarginBounds.Y;

   tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,
        secondColumnX, tableY, "Bees", Bees.Text);
   tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,
        secondColumnX, tableY, "Flowers", Flowers.Text);
   tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,
         secondColumnX, tableY, "Honey in Hive", HoneyInHive.Text);
   tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,
        secondColumnX, tableY, "Nectar in Flowers", NectarInFlowers.Text);
   tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,
        secondColumnX, tableY, "Frames Run", FramesRun.Text);
   tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,
        secondColumnX, tableY, "Frame Rate", FrameRate.Text);

   g.DrawRectangle(Pens.Black, tableX, e.MarginBounds.Y,
        tableWidth, tableY - e.MarginBounds.Y);
   g.DrawLine(Pens.Black, secondColumnX, e.MarginBounds.Y,
        secondColumnX, tableY);
```

Here's the event handler for the Document's PrintPage event. It goes in the form.

We gave you this part already. It draws the oval header, and sets up variables that you'll use to draw the table of bee stats.

Did you figure out how the PrintTableRow() method works? All you need to do is call it once per row, and it prints whatever text you want in the two columns. The trick is that it returns the new tableY value for the next row.

Don't forget to draw the rectangle around the table and the line between the columns.

www.itbook.store/books/9781449343507

```
        using (Pen blackPen = new Pen(Brushes.Black, 2))
        using (Bitmap hiveBitmap = new Bitmap(hiveForm.ClientSize.Width,
                                              hiveForm.ClientSize.Height))
        using (Bitmap fieldBitmap = new Bitmap(fieldForm.ClientSize.Width,
                                               fieldForm.ClientSize.Height))
        {
            using (Graphics hiveGraphics = Graphics.FromImage(hiveBitmap))
            {
                renderer.PaintHive(hiveGraphics);
            }

            int hiveWidth = e.MarginBounds.Width / 2;
            float ratio = (float)hiveBitmap.Height / (float)hiveBitmap.Width;
            int hiveHeight = (int)(hiveWidth * ratio);
            int hiveX = e.MarginBounds.X + (e.MarginBounds.Width - hiveWidth) / 2;
            int hiveY = e.MarginBounds.Height / 3;
            g.DrawImage(hiveBitmap, hiveX, hiveY, hiveWidth, hiveHeight);
            g.DrawRectangle(blackPen, hiveX, hiveY, hiveWidth, hiveHeight);

            using (Graphics fieldGraphics = Graphics.FromImage(fieldBitmap))
            {
                renderer.PaintField(fieldGraphics);
            }
            int fieldWidth = e.MarginBounds.Width;
            ratio = (float)fieldBitmap.Height / (float)fieldBitmap.Width;
            int fieldHeight = (int)(fieldWidth * ratio);
            int fieldX = e.MarginBounds.X;
            int fieldY = e.MarginBounds.Y + e.MarginBounds.Height - fieldHeight;
            g.DrawImage(fieldBitmap, fieldX, fieldY, fieldWidth, fieldHeight);
            g.DrawRectangle(blackPen, fieldX, fieldY, fieldWidth, fieldHeight);
        }
    }


    private void printToolStripButton1_Click(object sender, EventArgs e) {
        bool stoppedTimer = false;
        if (timer1.Enabled) {
            timer1.Stop();
            stoppedTimer = true;
        }
        PrintPreviewDialog preview = new PrintPreviewDialog();
        PrintDocument document = new PrintDocument();
        preview.Document = document;
        document.PrintPage += new PrintPageEventHandler(document_PrintPage);
        preview.ShowDialog(this);
        if (stoppedTimer)
            timer1.Start();
    }
```

You'll need a black pen that's 2 pixels wide to draw the lines around the screenshots.

The bitmaps need to be the same size as the form's drawing area, so ClientSize comes in handy.

Since the pen and the two bitmaps need to be disposed, we put them all in one big using block.

The PaintHive() method needs a Graphics object to draw on, so this code creates an empty Bitmap object and passes it to PaintHive().

e.MarginBounds.Width has the width of the printable area of the page. That's how wide the field screenshot should be drawn.

Here's where the height of the screenshot is calculated using the form's height–width ratio.

Here's the code for the print button. It pauses the simulator (if it's running), creates a PrintDocument, hooks it up to the PrintPage event handler, shows the dialog, and then restarts the simulator.

www.itbook.store/books/9781449343507

There's so much more to be done...

You've built a pretty neat little simulator, but why stop now? There's a whole lot more that you can do on your own. Here are some ideas—see if you can implement some of them.

Add a control panel

Convert the constants in the World and Hive classes to properties. Then add a new form with a control panel that has sliders to control them.

Add enemies

Add enemies that attack the hive. The more flowers there are, the more enemies are attracted to the hive. Then add Sting Patrol bees to defend against the enemies, and Hive Maintenance bees to defend and repair the hive. Those bees take extra honey.

A good simulation will have lots of tradeoffs, and will give the user ways to decide which tradeoffs to make to influence the progress of the hive.

Add hive upgrades

If the hive gets enough honey, it gets bigger. A bigger hive can hold more bees, but takes more honey and attracts more enemies. If enemies cause too much damage, the hive gets smaller again.

Add a queen bee who lays eggs

The eggs need Baby Bee Care worker bees to take care of them. More honey in the hive causes the queen to lay more eggs, which need more workers to care for them, who consume more honey.

Add animation

Animate the background of the Hive form so the sun slowly travels across the sky. Make it get dark at night, and draw stars and a moon. Add some perspective–make the bees get smaller the further they get from the hive in the field of flowers.

Use your imagination!

Try to think of other ways you can make the simulation more interesting or more interactive.

Did you come up with a cool modification to the simulator? Show off your skills—upload your project's source code to the Head First C# forums at www.headfirstlabs.com/books/hfcsharp/.

www.itbook.store/books/9781449343507

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.

www.itbook.store/books/9781449343507

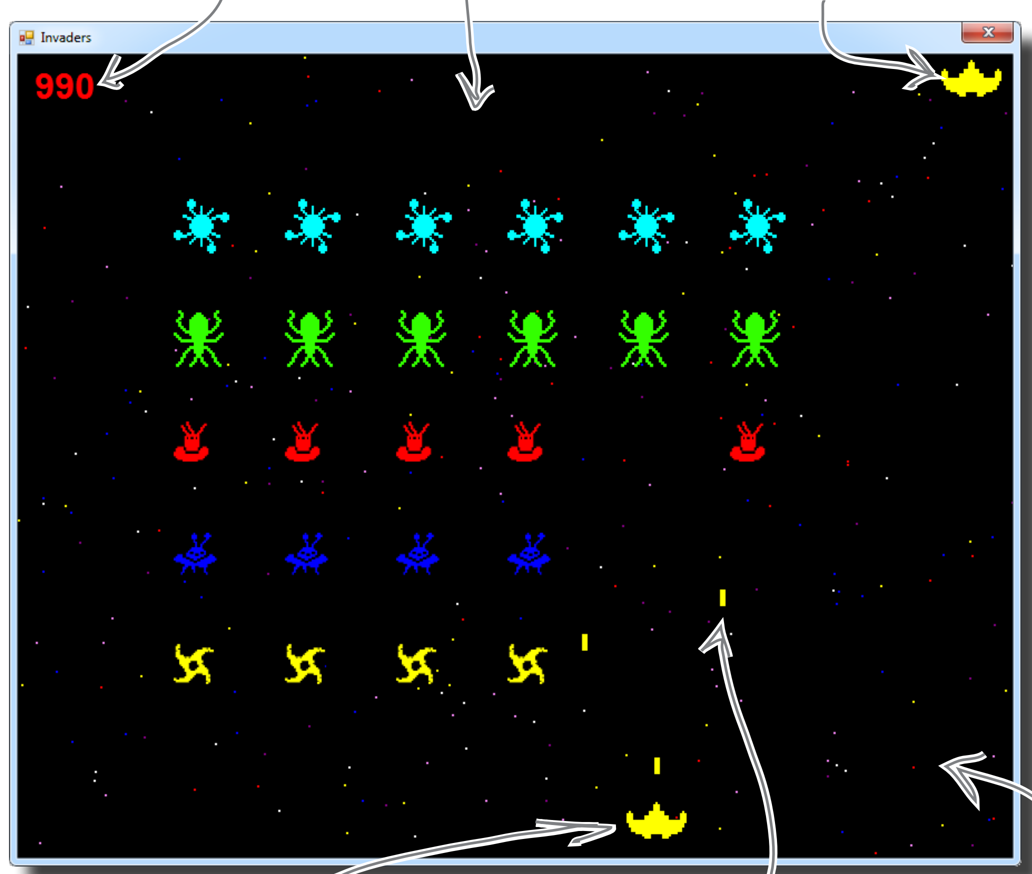
The grandfather of video games

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.

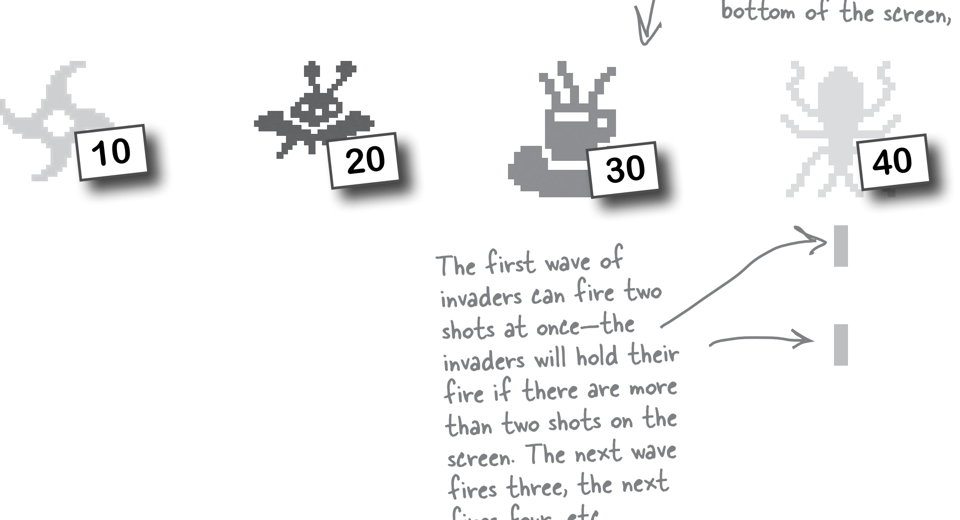
The multicolored stars in the background twinkle on and off, but don't affect gameplay at all.

www.itbook.store/books/9781449343507

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.



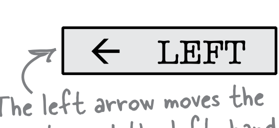
The spacebar shoots, but there can only be two player shots on the screen at once. As soon as a shot hits something or disappears, another shot can be fired.

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).

SPACE

Fire!



RIGHT →

The left arrow moves the ship toward the left-hand edge of the screen.

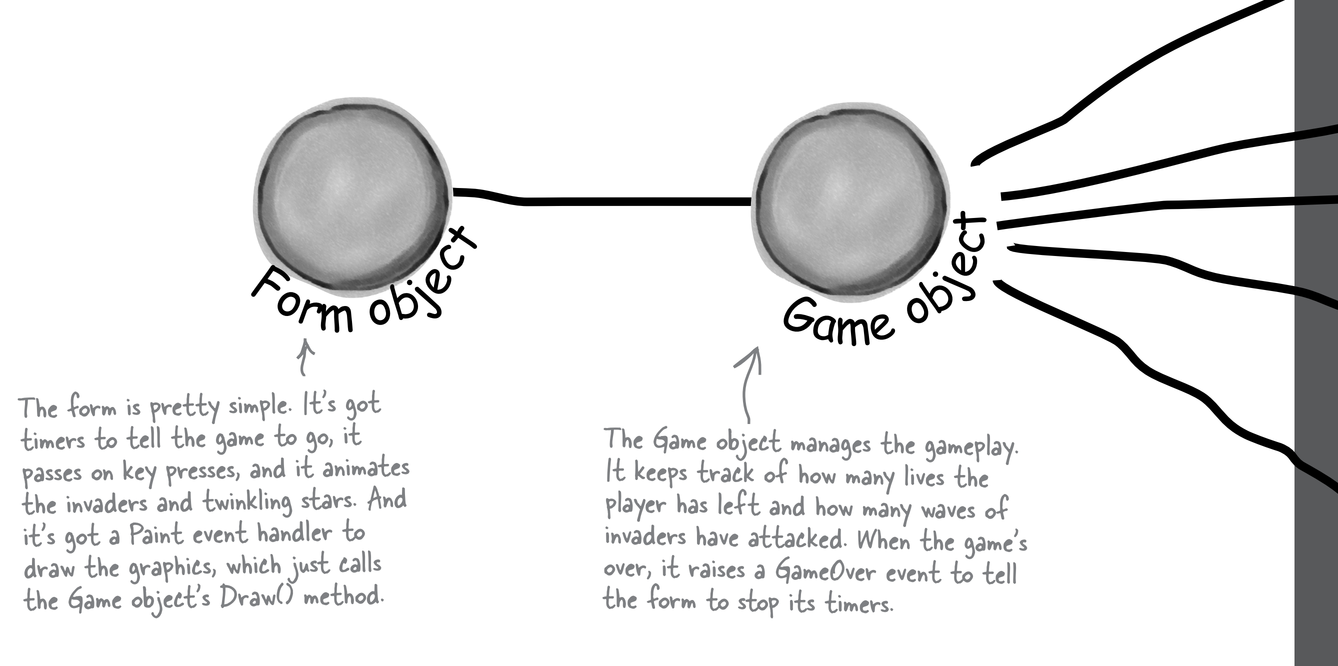
The right arrow key moves the ship to the right.

www.itbook.store/books/9781449343507

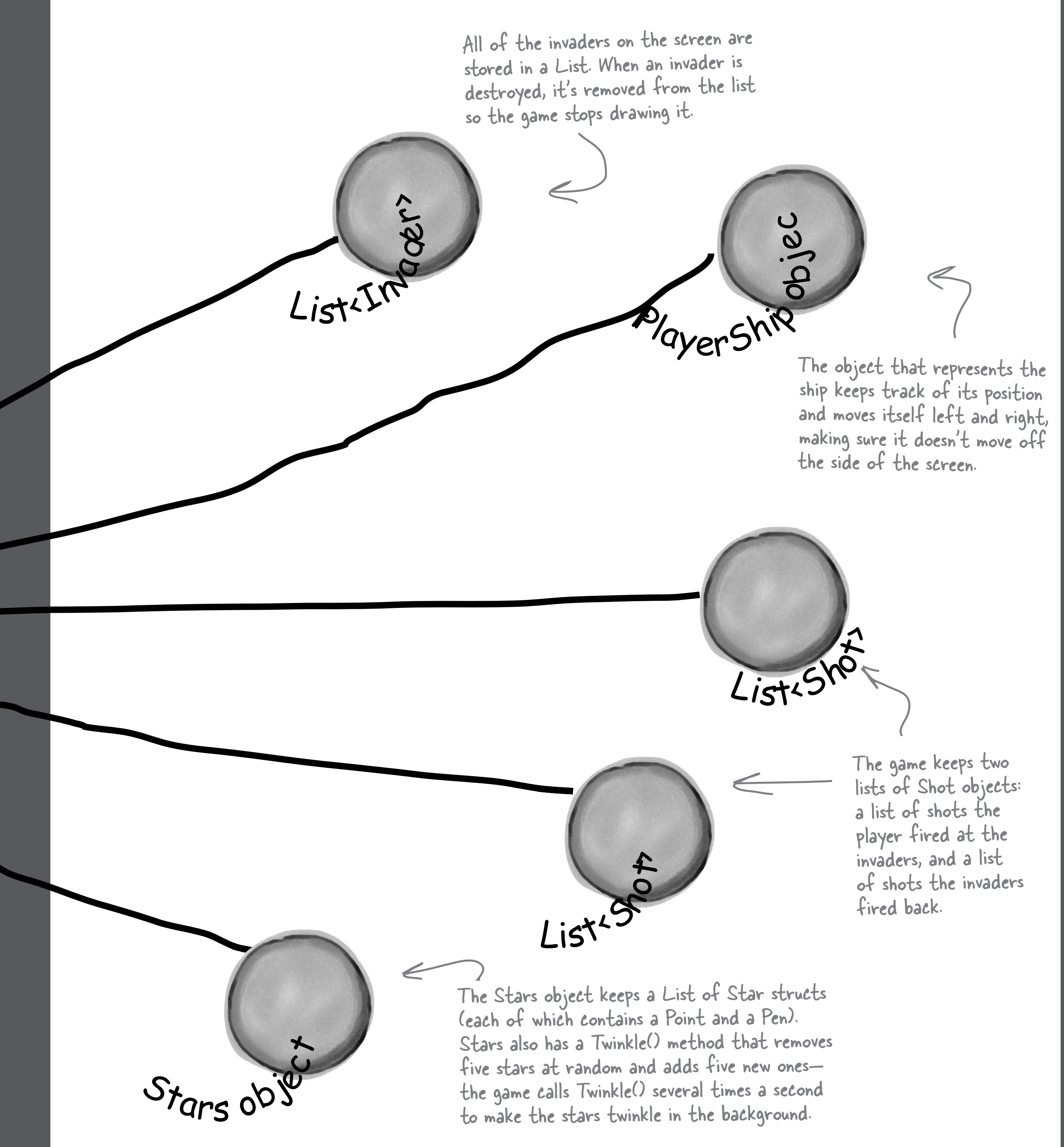
The architecture of Invaders

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:



www.itbook.store/books/9781449343507



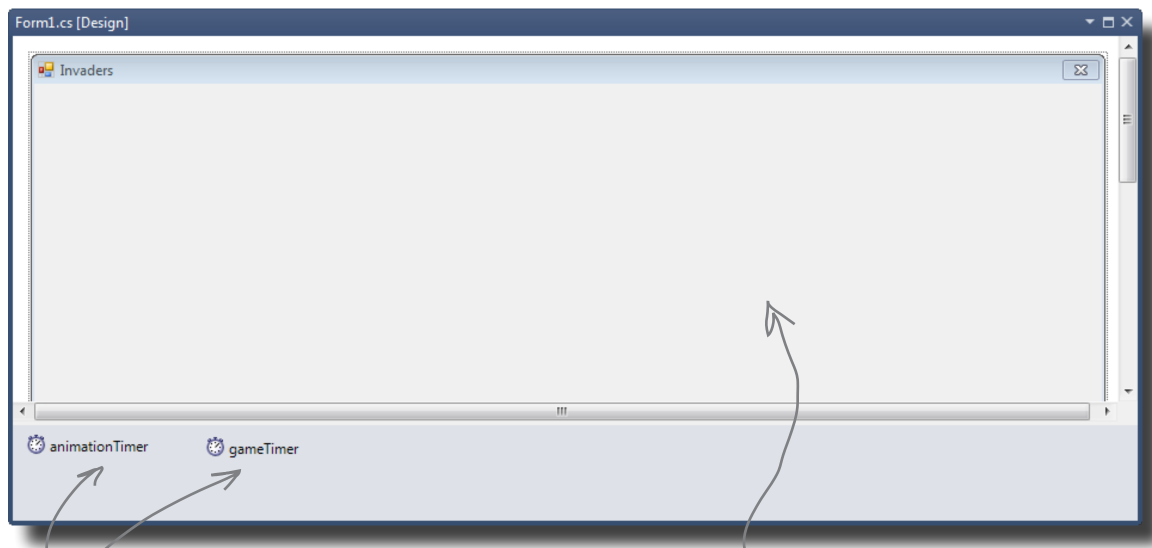
www.itbook.store/books/9781449343507

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.

www.itbook.store/books/9781449343507

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.

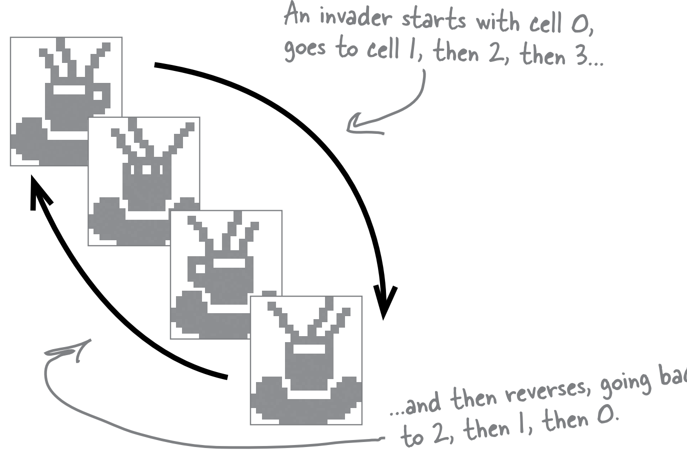
Adjust the timers for smooth animation

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.



www.itbook.store/books/9781449343507

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.

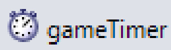
We want the most current key pressed to be at the very top of the list, so that if the player mashes a few keys at the same time, the game responds to the one that was hit most recently. Then, when he lets up one key, the game responds to the next one in the list.

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!

www.itbook.store/books/9781449343507

The game timer handles movement and gameplay



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.

www.itbook.store/books/9781449343507

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

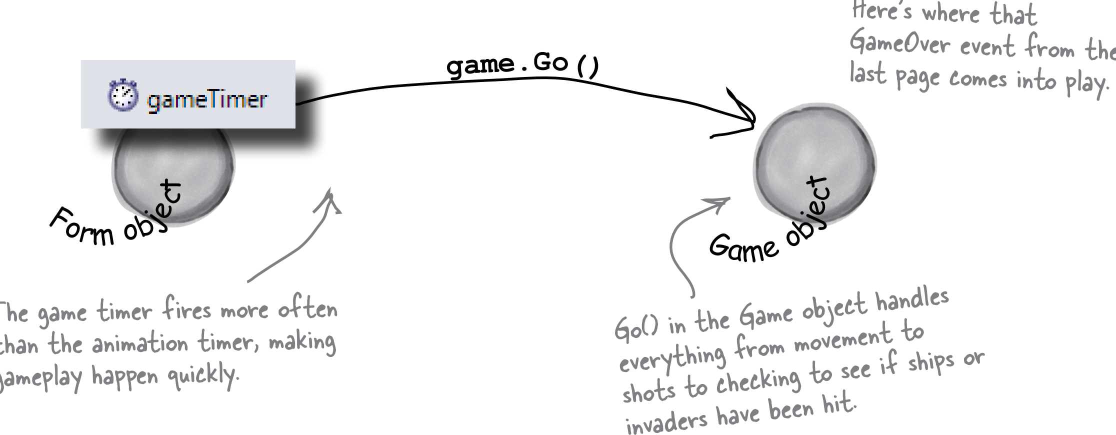
In addition to handling movement left and right, the main job of the game timer is to call the `Game` object's `Go()` method. That's where all of the gameplay is managed. The `Game` object keeps track of the state of the game, and its `Go()` method advances the game by one frame. That involves:

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.

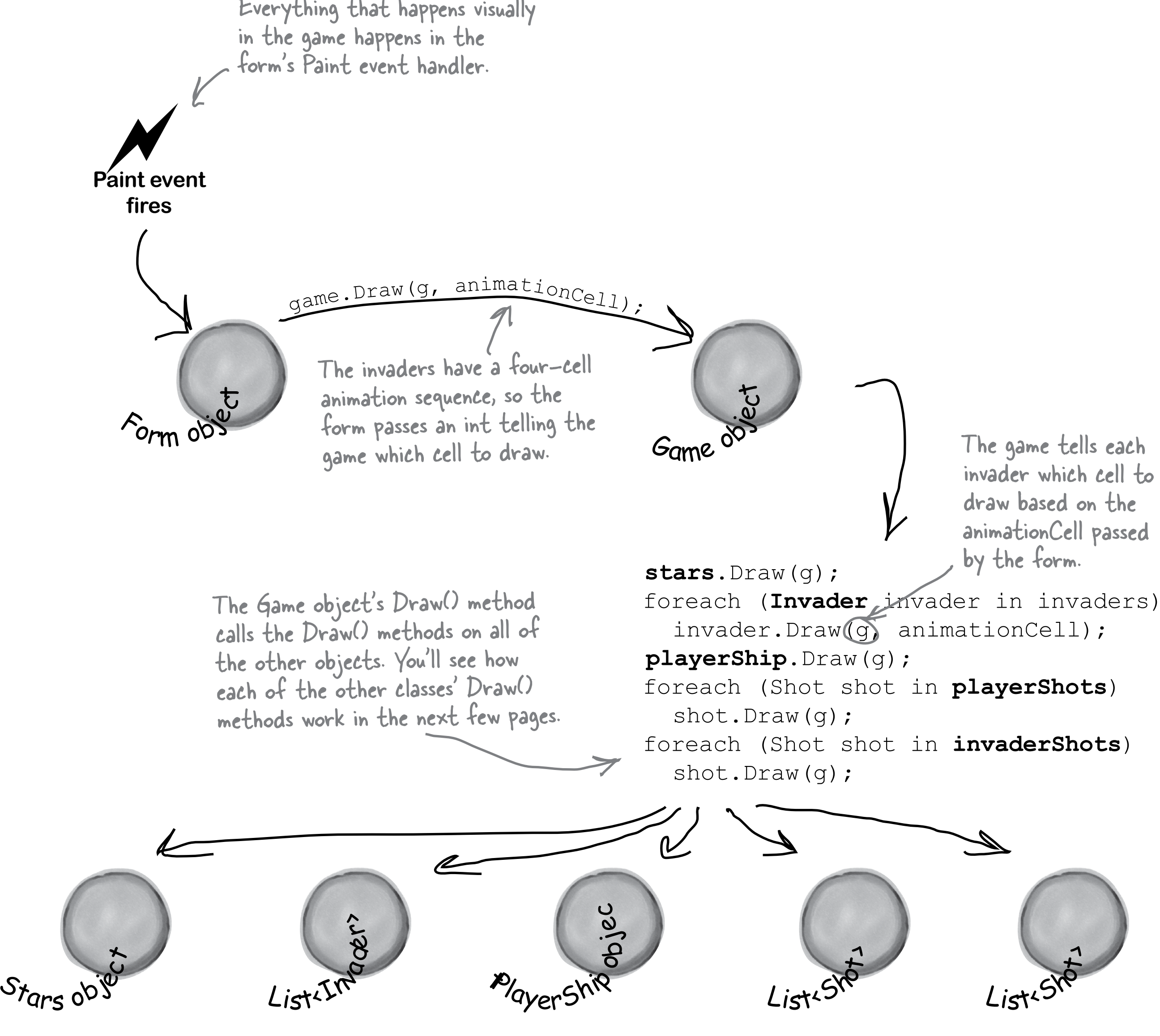


www.itbook.store/books/9781449343507

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.



www.itbook.store/books/9781449343507

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.

www.itbook.store/books/9781449343507

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.

www.itbook.store/books/9781449343507

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 119 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()`)

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?

www.itbook.store/books/9781449343507

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!

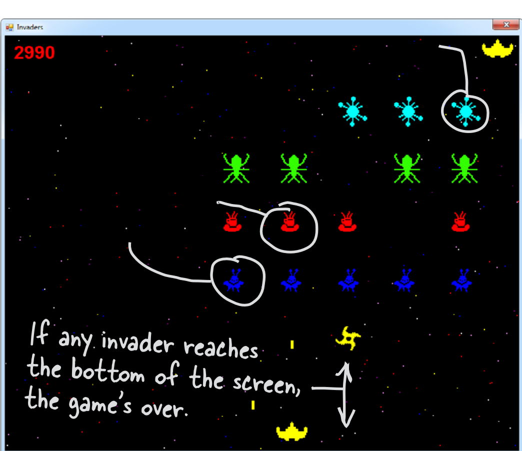
You've got collections of invaders and shots, and you need to search through those collections to find certain invaders and shots. Any time you hear collections and searching in the same sentence, you should think LINQ. Here's what you need to do:

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.

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).



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.)

www.itbook.store/books/9781449343507

Crafting the Invader class

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:

Invader
Location: Point InvaderType: ShipType Area: Rectangle Score: int
Draw(g: Graphics, animationCell: int) Move(direction: Direction)

```
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.

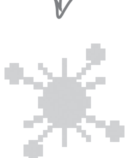
```
enum ShipType {
    Bug,
    Saucer,
    Satellite,
    Spaceship,
    Star,
  }
```

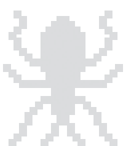

www.itbook.store/books/9781449343507

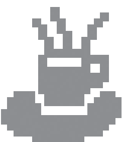
Build the Invaders' methods

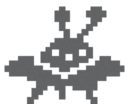
The three core methods for `Invader` are `Move()`, `Draw()`, and `InvaderImage()`. Let's look at each in turn.

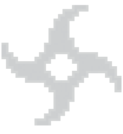
There are five types of invaders, and each of them has four different animation cell pictures.











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
}
```

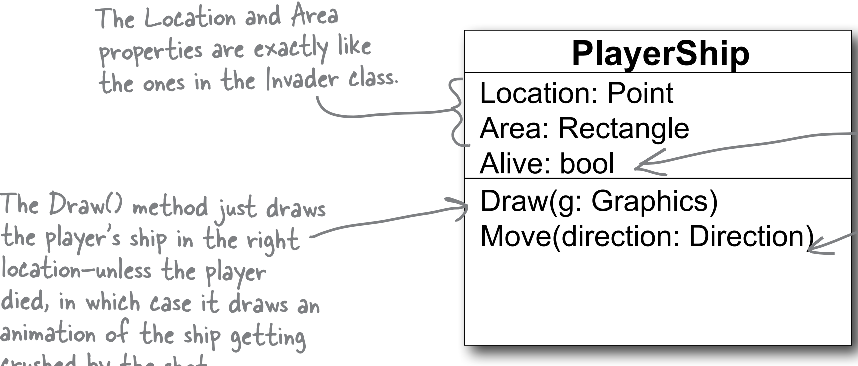
Each invader knows its type. So if you give its InvaderImage() method a number for its animation cell, it can return a Bitmap that's got the right graphic in it.

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

www.itbook.store/books/9781449343507

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.



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.

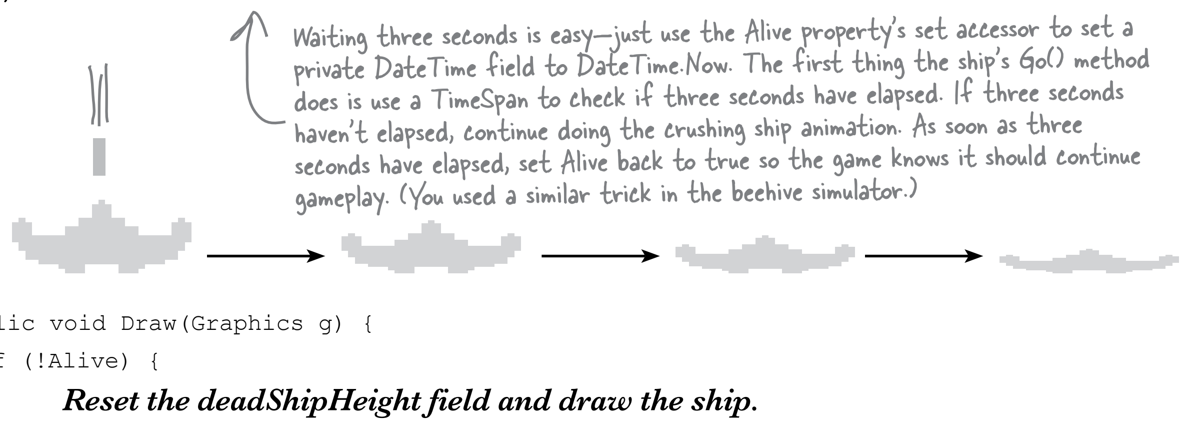
The Move() method takes one parameter, a Direction enum, and moves the player in that direction.

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.)

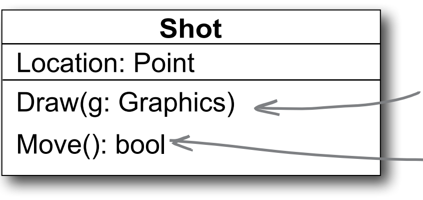


```
public void Draw(Graphics g) {
  if (!Alive) {
        Reset the deadShipHeight field and draw the ship.
    } else {
        Check the deadShipHeight field. If it's greater than zero, decrease it by 1
        and use DrawImage() to draw the ship a little flatter.
    }
}
```


www.itbook.store/books/9781449343507

"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
}
```

ou 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.

www.itbook.store/books/9781449343507

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;
  }
}
```

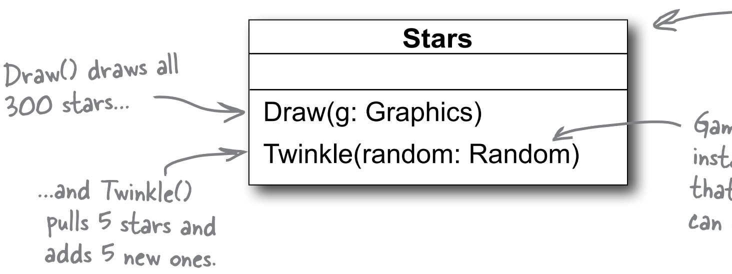
Each star has a point (its location) and a pen (for its color).

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()` 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.

www.itbook.store/books/9781449343507

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? Upload it to CodePlex and claim your bragging rights: www.headfirstlabs.com/books/hfcsharp/

www.itbook.store/books/9781449343507