

O'REILLY®

Fourth
Edition

Head First

C#

A Learner's Guide to
Real-World Programming
with C# and .NET Core

Andrew Stellman
& Jennifer Greene



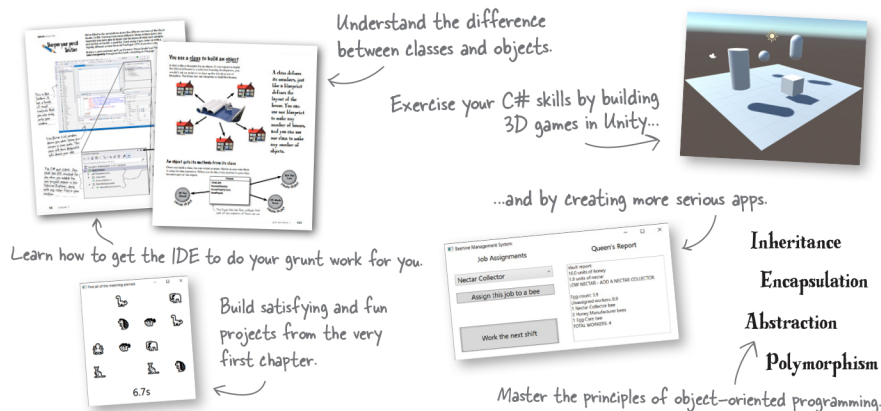
A Brain-Friendly Guide

Head First

C#

What will you learn from this book?

Dive into C# and create apps, user interfaces, games, and more using this fun and highly visual introduction to C#, .NET Core, and Visual Studio. With this completely updated guide, which covers C# 8.0 and Visual Studio 2019, beginning programmers like you will build a fully functional game in the opening chapter. Then you'll learn how to use classes and object-oriented programming, create 3D games in Unity, and query data with LINQ. And you'll do it all by solving puzzles, doing hands-on exercises, and building real-world applications. By the time you're done, you'll be a solid C# programmer—and you'll have a great time along the way!



What's so special about this book?

Based on the latest research in cognitive science and learning theory, *Head First C#* uses a visually rich format to engage your mind rather than a text-heavy approach that puts you to sleep. Why waste your time struggling with new concepts? This multisensory learning experience is designed for the way your brain really works.

"Thank you so much!
Your books have
helped me to launch
my career."

—Ryan White
Game Developer

"Andrew and Jennifer
have written a
concise, authoritative,
and most of all, fun
introduction to C#
development."

—Jon Galloway
Senior Program Manager on the
.NET Community Team
at Microsoft

"If you want to learn
C# in depth and have
fun doing it, this is THE
book for you."

—Andy Parker
Fledgling C# programmer

.NET

US \$64.99

CAN \$85.99

ISBN: 978-1-491-97670-8



O'REILLY®

Head First C#

Fourth Edition

WOULDN'T IT BE DREAMY IF
THERE WAS A C# BOOK THAT WAS
MORE FUN THAN MEMORIZING
A DICTIONARY? IT'S PROBABLY
NOTHING BUT A FANTASY...



Andrew Stellman
Jennifer Greene

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First C#

Fourth Edition

by Andrew Stellman and Jennifer Greene

Copyright © 2021 Jennifer Greene, Andrew Stellman. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Series Creators:

Kathy Sierra, Bert Bates

Cover Designer:

Ellie Volckhausen

Brain Image on Spine:

Eric Freeman

Editors:

Nicole Taché, Amanda Quinn

Proofreader:

Rachel Head

Indexer:

Potomac Indexing, LLC

Illustrator:

Jose Marzan

Page Viewers:

Greta the miniature bull terrier and Samosa the Pomeranian

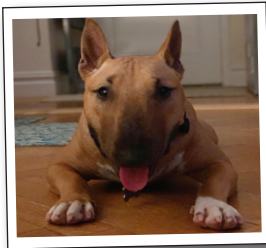
Printing History:

November 2007: First Edition.

May 2010: Second Edition.

August 2013: Third Edition.

December 2020: Fourth Edition



The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. The *Head First* series designations, *Head First C#*, and related trade dress are trademarks of O'Reilly Media, Inc.

Microsoft, Windows, Visual Studio, MSDN, the .NET logo, Visual Basic, and Visual C# are registered trademarks of Microsoft Corporation.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

No bees, space aliens, or comic book heroes were harmed in the making of this book.

ISBN: 978-1-491-97670-8

[LSI]

[2020-11-13]

6 inheritance

Visual Studio for Mac Chapter 6 Project

SO THERE I WAS RIDING MY
BICYCLE OBJECT DOWN DEAD MAN'S CURVE
WHEN I REALIZED IT INHERITED FROM **TwoWHEELER** AND
I FORGOT TO OVERRIDE THE **BRAKES** METHOD...LONG STORY
SHORT, TWENTY-SIX STITCHES AND MOM SAYS I'M
GROUNDED FOR A MONTH.

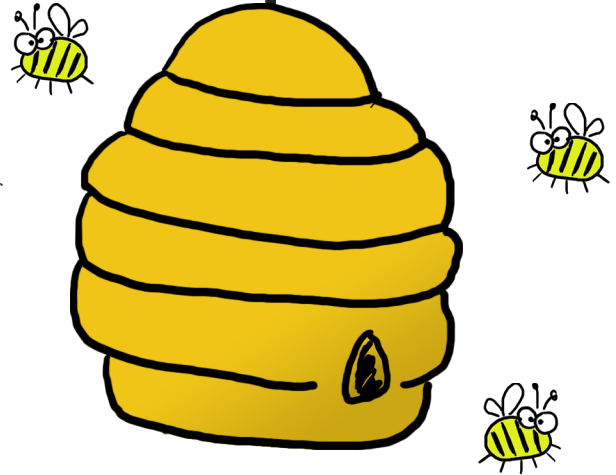


Welcome to the Visual Studio for Mac project for Chapter 6.

In Chapter 6 you learned about **inheritance**, which lets you **extend** an existing class so your new class gets all of its behavior. Inheritance is one of the most powerful concepts and techniques in the C# language, so we really wanted to give you some great practice with it—that's why we included this **extra long project** to build a **serious business application**. In this project you'll get lots of practice with inheritance building a beehive management system to help a queen bee manage her hive.

Build a beehive management system

The queen bee needs your help! Her hive is out of control, and she needs a program to help manage her honey production business. She's got a beehive full of workers, and a whole bunch of jobs that need to be done around the hive, but somehow she's lost control of which bee is doing what, and whether or not she's got the beepower to do the jobs that need to be done. It's up to you to build a **beehive management system** to help her keep track of her workers. Here's how it'll work.



① **The queen assigns jobs to her workers.**

There are three different jobs that the workers can do. **Nectar collector** bees fly out and bring nectar back to the hive. **Honey manufacturer** bees turn that nectar into honey, which bees eat to keep working. Finally, the queen is constantly laying eggs, and **egg care** bees make sure they become workers.

② **When the jobs are all assigned, it's time to work.**

Once the queen's done assigning the work, she'll tell the bees to work the next shift by clicking the "Work the next shift" button in her Beehive Management System app, which generates a shift report that tells her how many bees are assigned to each job and the status of the nectar and honey in the **honey vault**.

Job Assignments	Queen's Report
<div><div>Nectar Collector</div><div>Assign this job to a bee</div><div>Work the next shift</div></div>	<div><div>Vault report:</div><div>16.0 units of honey</div><div>1.9 units of nectar</div><div>Egg count: 3.9</div><div>Unassigned workers: 0.9</div><div>1 Nectar Collector bee</div><div>2 Honey Manufacturer bees</div><div>1 Egg Care bee</div><div>TOTAL WORKERS: 34</div></div>

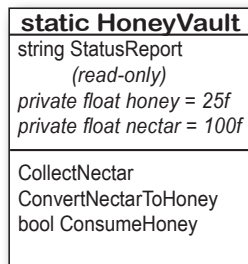
③ **Help the queen grow her hive.**

Like all business leaders, the queen is focused on **growth**. The beehive business is hard work, and she measures her hive in the total number of workers. Can you help the queen keep adding workers? How big can she grow the hive before it runs out of honey and she has to file for bee-nkrupctcy?

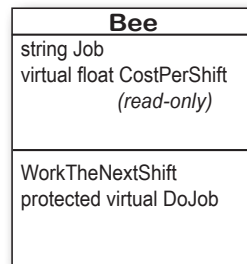
The beehive management system class model

Here are the classes that you'll build for the beehive management system. There's an inheritance model with a base class and four subclasses, a static class to manage the honey and nectar that drive the hive business, and the MainWindow class with the code-behind for the main window.

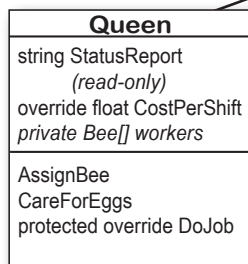
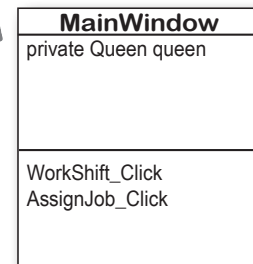
HoneyVault is a static class that keeps track of the honey and nectar in the hive. Bees use the ConsumeHoney method, which checks if there's enough honey to do their jobs, and if so subtracts the amount requested.



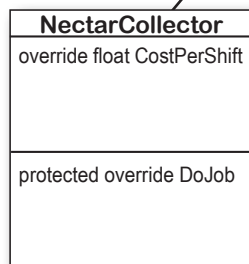
Bee is the base class for all of the bee classes. Its WorkTheNextShift method calls the Honey Vault's ConsumeHoney method, and if it returns true calls DoJob.



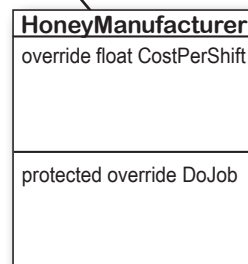
The code-behind for the main window just does a few things. It creates an instance of Queen, and has Click event handlers for the buttons to call her WorkTheNextShift and AssignBee methods and display the status report.



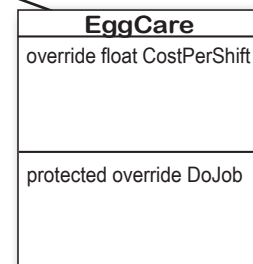
This Bee subclass uses an array to keep track of the workers and overrides DoJob to call their WorkTheNextShift methods.



This Bee subclass overrides DoJob to call the HoneyVault method to collect nectar.



This Bee subclass overrides DoJob to call the HoneyVault method to convert nectar to honey.



This Bee subclass keeps a reference to the Queen, and overrides DoJob to call the Queen's CareForEggs method.



This class model is just the start. We'll give more details so you can write the code.

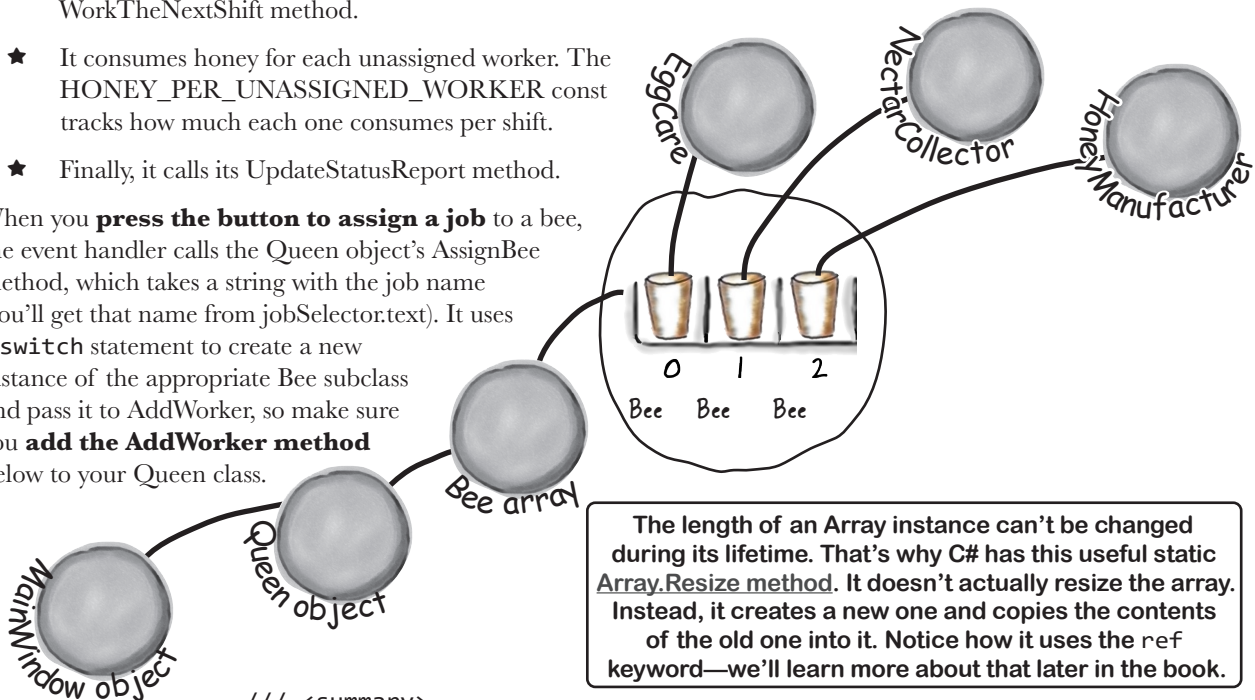
Examine this class model really carefully. It has a lot of information about the app you're about to build. Next, we'll give you all of the details you need to write the code for these classes.

The Queen class: how she manages the worker bees

When you **press the button to work the next shift**, the button's Click event handler calls the Queen object's `WorkTheNextShift` method, which is inherited from the Bee base class. Here's what happens next:

- ★ `Bee.WorkTheNextShift` calls `HoneyVault.ConsumeHoney(HoneyConsumed)`, using the `CostPerShift` property (which each subclass overrides with a different value) to determine how much honey she needs to work.
- ★ `Bee.WorkTheNextShift` then calls `DoJob`, which the Queen also overrides.
- ★ `Queen.DoJob` adds 0.45 eggs to her private eggs field (using a const called `EGGS_PER_SHIFT`). The `EggCare` bee will call her `CareForEggs` method, which decreases eggs and increases `unassignedWorkers`.
- ★ Then it uses a foreach loop to call each worker's `WorkTheNextShift` method.
- ★ It consumes honey for each unassigned worker. The `HONEY_PER_UNASSIGNED_WORKER` const tracks how much each one consumes per shift.
- ★ Finally, it calls its `UpdateStatusReport` method.

When you **press the button to assign a job** to a bee, the event handler calls the Queen object's `AssignBee` method, which takes a string with the job name (you'll get that name from `jobSelector.text`). It uses a **switch** statement to create a new instance of the appropriate Bee subclass and pass it to `AddWorker`, so make sure you **add the `AddWorker` method** below to your Queen class.



The length of an Array instance can't be changed during its lifetime. That's why C# has this useful static `Array.Resize` method. It doesn't actually resize the array. Instead, it creates a new one and copies the contents of the old one into it. Notice how it uses the `ref` keyword—we'll learn more about that later in the book.

```
/// <summary>
/// Expand the workers array by one slot and add a Bee reference.
/// </summary>
/// <param name="worker">Worker to add to the workers array.</param>
private void AddWorker(Bee worker)
```

You'll need this `AddWorker` method to add a new worker to the Queen's worker array. It calls `Array.Resize` to expand the array, then adds the new worker Bee to it.

```
{
    if (unassignedWorkers >= 1)
    {
        unassignedWorkers--;
        Array.Resize(ref workers, workers.Length + 1);
        workers[workers.Length - 1] = worker;
    }
}
```


The UI: add the HTML markup for the main window

Create a **new Blazor web app** called **BeehiveManagementSystem**. Here's the HTML markup for the main window, including its **entire @code section**.

It has three columns: one for job assignments, an empty divider column, and a column for the Queen's report.

The left column contains four rows: one for the header, one for the job dropdown, one for the job assignment button, and one for the button to work the next shift

@page "/"

```
<div class="container">
  <div class="row">
```

```
    <div class="col-4">
      <h3 class="row">Job Assignments</h3>
```

```
      <select class="row mt-1 custom-select" @bind="selectedJob">
        <option value="Nectar Collector">Nectar Collector</option>
        <option value="Honey Manufacturer">Honey Manufacturer</option>
        <option value="Egg Care">Egg Care</option>
      </select>
```

```
      <div class="row mt-1">
        <button class="col btn btn-small btn-secondary"
          @onclick="() => queen.AssignBee(selectedJob)">
          Assign this job to a bee
        </button>
      </div>
```

```
      <div class="row mt-4">
        <button class="col btn btn-lg btn-primary"
          @onclick="() => queen.WorkTheNextShift()">
          Work the next shift
        </button>
      </div>
    </div>
```

```
    <div class="col-1" />
```

Assigning the "col" Bootstrap CSS class to the buttons makes them into full-width columns so they take up the entire width of the row.

```
    <div class="col-7">
      <h3 class="row">Queen's Report</h3>
      <textarea class="row" rows="12" cols="50" value="@queen.StatusReport" readonly />
    </div>
```

```
  </div>
</div>
```

```
@code {
  Queen queen = new Queen();
  string selectedJob = "Nectar Collector";
}
```

This is a **select control**. It gives you a menu of options to choose from. You'll use it to let the user choose a job to assign and **@bind** its value to a field called **selectedJob**.

This is a **TextArea control**. It displays multiple lines of text. You'll **@bind** it to the Queen's **StatusReport** property.

The buttons call methods on the Queen object, using the same **@onclick** pattern you used in Chapters 1 and 2.

We added Bootstrap CSS classes to change the way the buttons look: **btn-small** and **btn-large** change their size, while **btn-primary** and **btn-secondary** change their styles and colors.

This is all of the C# code in the Razor page. All it does is create two fields, one with a reference to a new Queen object and one with a string to keep track of the job selected by the user.



Long Exercise

Build the **Beehive Management System**. The purpose of the system is to **maximize the number of workers assigned to jobs in the hive**, and keep the hive running as long as possible until the honey runs out.

Rules of the hive

Workers can be assigned one of three jobs: nectar collectors add nectar to the honey vault, honey manufacturers convert the nectar into honey, and egg care bees turn eggs into workers who can be assigned to jobs. During each shift, the Queen lays eggs (just under two shifts per egg). The Queen updates her status report at the end of the shift. It shows the honey vault status and the number of eggs, unassigned workers, and bees assigned to each job.

Start by building the static HoneyVault class

- The HoneyVault class is a good starting point because it has no **dependencies**—it doesn't call methods or use properties or fields from any other class. Start by creating a new class called HoneyVault. Make it **static**, then look at the class diagram and add the class members.
- HoneyVault has **two constants** (NECTAR_CONVERSION_RATIO = .19f and LOW_LEVEL_WARNING = 10f) that are used in the methods. Its private honey field is initialized to 25f and its private nectar field is initialized to 100f.
- The **ConvertNectarToHoney method** converts nectar to honey. It takes a float parameter called amount, subtracts that amount from its nectar field, and adds amount × NECTAR_CONVERSION_RATIO to the honey field. (If the amount passed to the method is greater than the nectar left in the vault, it converts all of the remaining nectar.)
- The **ConsumeHoney method** is how the bees use honey to do their jobs. It takes a parameter, amount. If honey is greater than or equal to amount, it subtracts amount from honey and returns true; otherwise it returns false.
- The **CollectNectar method** is called by the NectarCollector bee each shift. It takes a parameter, amount. If amount is greater than zero, it adds it to the nectar field.
- The **StatusReport property** only has a get accessor that returns a string with separate lines with the amount of honey and the amount of nectar in the vault. If the honey is below LOW_LEVEL_WARNING, it adds a warning ("LOW HONEY - ADD A HONEY MANUFACTURER"). It does the same for the nectar field.

Create the Bee class and start building the Queen, HoneyManufacturer, NectarCollector, and EggCare classes

- Create the Bee base class. Its **constructor** takes a string, which it uses to set the **read-only Job property**. Each Bee subclass passes a string to the base constructor—"Queen", "Nectar Collector", "Honey Manufacturer", or "Egg Care"—so the Queen class has this code: **public Queen() : base("Queen")**
- The virtual read-only **CostPerShift property** lets each Bee subclass define the amount of honey it consumes each shift. The **WorkTheNextShift method** passes HoneyConsumed to the HoneyVault.ConsumeHoney method. If ConsumeHoney returns true there's enough honey left in the hive, so WorkTheNextShift then calls DoJob.
- **Create empty** HoneyManufacturer, NectarCollector, and EggCare classes that just extend Bee—you'll need them to build the Queen class. You'll **finish the Queen class first**, then come back and finish the other Bee subclasses.
- Each Bee subclass **overrides the DoJob method** with code to do its job, and **overrides the CostPerShift property** with the amount of honey it consumes each shift.
- Here are all of the **values for the read-only Bee.CostPerShift property** for each Bee subclass: Queen.CostPerShift returns 2.15f, NectarCollector.CostPerShift returns 1.95f, HoneyManufacturer.CostPerShift returns 1.7f, and EggCare.CostPerShift returns 1.35f.



LONG Exercise

This is a long exercise, *but that's okay!* Just build it class by class. Finish building the Queen class first. Once you're done, you'll go back to the other Bee subclasses.

- The Queen class has a **private Bee[] field** called workers. It starts off as an empty array. We gave you the AddWorker method to add Bee references to it.
- Her **AssignBee method** takes a parameter with a job name (like "Egg Care"). It has switch (job) with cases that call AddWorker. For example, if job is "Egg Care" then it calls AddWorker(new EggCare(this)).
- There are two **private float fields** called eggs and unassignedWorkers to keep track of the number of eggs (which she adds to each shift) and the number of workers waiting to be assigned.
- She overrides the **DoJob method** to add eggs, tell the worker bees to work, and feed honey to the unassigned workers waiting for work. The EGGS_PER_SHIFT constant (set to 0.45f) is added to the eggs field. She uses a foreach loop to call each worker's WorkTheNextShift method. Then she calls HoneyVault.ConsumeHoney, passing it the constant HONEY_PER_UNASSIGNED_WORKER (set to 0.5f) × unassignedWorkers.
- She starts off with 3 unassigned workers—her **constructor** calls the AssignBee method three times to create three worker bees, one of each type.
- The EggCare bees call the Queen's **CareForEggs method**. It takes a float parameter called eggsToConvert. If the eggs field is >= eggsToConvert, it subtracts eggsToConvert from eggs and adds it to unassignedWorkers.
- Look carefully at the status reports in the screenshot—her private **UpdateStatusReport method** generates it (using HoneyVault.StatusReport). She calls UpdateStatusReport at the end of her DoJob and AssignBee methods.

Finish building the other Bee subclasses

- The **NectarCollector class** has a const NECTAR_COLLECTED_PER_SHIFT = 33.25f. Its **DoJob method** passes that const to HoneyVault.CollectNectar.
- The **HoneyManufacturer class** has a const NECTAR_PROCESSED_PER_SHIFT = 33.15f, and its DoJob method passes that const to HoneyVault.ConvertNectarToHoney.
- The **EggCare class** has a const CARE_PROGRESS_PER_SHIFT = 0.15f, and its DoJob method passes that const to queen.CareForEggs, using a private Queen reference that's **initialized in the EggCare constructor**.

Some more details about how the Beehive Management System works

- The goal is to get the TOTAL WORKERS line in the status report (which lists the total number of assigned workers) to go as high as possible—and that all depends on **which workers you add and when you add them**. Workers drain honey: if you've got too many of one kind of worker, the honey starts to go down. As you run the program, watch the honey and nectar numbers. After the first few shifts, you'll get a low honey warning (so add a honey manufacturer); after a few more, you'll get a low nectar warning (so add a nectar collector)—after that, you need to figure out how to staff the hive. How high can you get TOTAL WORKERS to go before the honey runs out?

Don't get overwhelmed or intimidated by the length of this exercise! Just break it down into small steps. Once you start working on it, you'll see it's all review of things you've learned.

Every single part of this exercise is something you've seen before. You CAN do this! And remember—it's not cheating to peek at our solution, and it's okay if you come up with your own solution that's different from ours.

you are here ▶



Long Exercise Solution

This project is big, and it has **a lot of different parts**. If you run into trouble, just take it piece by piece. None of it is magic—you already have the tools to understand every part of it.

Here's the code for the **static HoneyVault class**:

```
static class HoneyVault
{
    public const float NECTAR_CONVERSION_RATIO = .19f;
    public const float LOW_LEVEL_WARNING = 10f;
    private static float honey = 25f;
    private static float nectar = 100f;

    public static void CollectNectar(float amount)
    {
        if (amount > 0f) nectar += amount;
    }

    public static void ConvertNectarToHoney(float amount)
    {
        float nectarToConvert = amount;
        if (nectarToConvert > nectar) nectarToConvert = nectar;
        nectar -= nectarToConvert;
        honey += nectarToConvert * NECTAR_CONVERSION_RATIO;
    }

    public static bool ConsumeHoney(float amount)
    {
        if (honey >= amount)
        {
            honey -= amount;
            return true;
        }
        return false;
    }

    public static string StatusReport
    {
        get
        {
            string status = $"{honey:0.0} units of honey\n" +
                            $"{nectar:0.0} units of nectar";
            string warnings = "";
            if (honey < LOW_LEVEL_WARNING) warnings +=
                "\nLOW HONEY - ADD A HONEY MANUFACTURER";
            if (nectar < LOW_LEVEL_WARNING) warnings +=
                "\nLOW NECTAR - ADD A NECTAR COLLECTOR";
            return status + warnings;
        }
    }
}
```

The constants in the HoneyVault class are really important. Try making the nectar conversion ratio bigger—that adds lots of honey to the vault each shift. Try making it smaller—now the honey disappears almost immediately.

The NectarCollector bees do their jobs by calling the CollectNectar method to add nectar to the hive.

The HoneyManufacturer bees do their jobs by calling ConvertNectarToHoney, which reduces the nectar and increases the honey in the vault.

Every bee tries to consume a specific amount of honey during each shift. The ConsumeHoney method only returns true if there's enough honey for the bee to do its job.

It's okay if your code doesn't exactly match our code!

There are many different ways that you can solve this problem—and the bigger the program is, the more ways there are to write it. If your code works, then you did the exercise correctly! But take a few minutes to compare your solution with ours, and take the time to try and figure out why we made the decisions that we did.

Try using the View menu to show the Class View in the IDE (it will be docked in the Solution Explorer window). It's a useful tool for exploring your class hierarchy. Try expanding a class in the Class View window, then expand the Base Types folder to see its hierarchy. Use the tabs at the bottom of the window to switch between the Class View and Solution Explorer.



The behavior of this program is driven by the way the different classes interact with each other—especially the ones in the Bee class hierarchy. And at the top of that hierarchy is the **Bee superclass** that all of the other Bee classes extend:

```
class Bee
{
    public virtual float CostPerShift { get; }

    public string Job { get; private set; }

    public Bee(string job) ← The Bee constructor takes a single parameter, which
    {                                     it uses to set its read-only Job property. The Queen
        Job = job;                               uses that property when she generates the status
    }                                             report to figure out what subclass a specific bee is.

    public void WorkTheNextShift()
    {
        if (HoneyVault.ConsumeHoney(CostPerShift))
        {
            DoJob();
        }
    }

    protected virtual void DoJob() { /* the subclass overrides this */ }
}
```

The **NectarCollector class** collects nectar each shift and adds it to the vault:

```
class NectarCollector : Bee
{
    public const float NECTAR_COLLECTED_PER_SHIFT = 33.25f;
    public override float CostPerShift { get { return 1.95f; } }
    public NectarCollector() : base("Nectar Collector") { }

    protected override void DoJob()
    {
        HoneyVault.CollectNectar(NECTAR_COLLECTED_PER_SHIFT);
    }
}
```

← The NectarCollector and HoneyManufacturer classes have constants that determine how much nectar is collected and how much of it is converted to honey during each shift. Try changing them—the program is a lot less sensitive to changes to these constants than it is when you change the HoneyVault conversion ratio.

The **HoneyManufacturer class** converts the nectar in the honey vault into honey:

```
class HoneyManufacturer : Bee
{
    public const float NECTAR_PROCESSED_PER_SHIFT = 33.15f;
    public override float CostPerShift { get { return 1.7f; } }
    public HoneyManufacturer() : base("Honey Manufacturer") { }

    protected override void DoJob()
    {
        HoneyVault.ConvertNectarToHoney(NECTAR_PROCESSED_PER_SHIFT);
    }
}
```

←



Long Exercise Solution

Each of the Bee subclasses has a different job, but they have **shared behaviors**—even the Queen. They all work during each shift, but only do their jobs if there's enough honey.

The **Queen class** manages the workers and generates the status reports:

```
class Queen : Bee
{
    public const float EGGS_PER_SHIFT = 0.45f;
    public const float HONEY_PER_UNASSIGNED_WORKER = 0.5f;

    private Bee[] workers = new Bee[0];
    private float eggs = 0;
    private float unassignedWorkers = 3;

    public string StatusReport { get; private set; }
    public override float CostPerShift { get { return 2.15f; } }

    public Queen() : base("Queen") {
        AssignBee("Nectar Collector");
        AssignBee("Honey Manufacturer");
        AssignBee("Egg Care");
    }

    private void AddWorker(Bee worker)
    {
        if (unassignedWorkers >= 1)
        {
            unassignedWorkers--;
            Array.Resize(ref workers, workers.Length + 1);
            workers[workers.Length - 1] = worker;
        }
    }

    private void UpdateStatusReport()
    {
        StatusReport = $"Vault report:\n{HoneyVault.StatusReport}\n" +
            $"Egg count: {eggs:0.0}\nUnassigned workers: {unassignedWorkers:0.0}\n" +
            $"{WorkerStatus("Nectar Collector")}\n{WorkerStatus("Honey Manufacturer")}" +
            $"Egg Care\nTOTAL WORKERS: {workers.Length}";
    }

    public void CareForEggs(float eggsToConvert)
    {
        if (eggs >= eggsToConvert)
        {
            eggs -= eggsToConvert;
            unassignedWorkers += eggsToConvert;
        }
    }
}
```

The constants in the Queen class are really important because they determine how the program behaves over the course of many shifts. If she lays too many eggs, they eat more honey, but also speed up progress. If unassigned workers consume more honey, it adds more pressure to assign workers quickly.

The Queen starts things off by assigning one bee of each type in her constructor.

We gave you this AddWorker method. It resizes the array and adds a Bee object to the end. Have you noticed that sometimes the status report lists the unassigned workers as 1.0 but you can't add a worker? Add a breakpoint to the first line of AddWorker—you'll see unassignedWorkers is equal to 0.9999999999.... Can you think of how to fix that?

You had to look really closely at the status report in the screenshot to figure out what to include here.

The EggCare bees call the CareForEggs method to convert eggs into unassigned workers.



Long Exercise Solution

The **Queen** class drives all of the work in the program—she keeps track of the instances of the worker **Bee** objects, creates new ones when they need to be assigned to their jobs, and tells them to start working their shifts:

```
private string WorkerStatus(string job)
{
    int count = 0;
    foreach (Bee worker in workers)
        if (worker.Job == job) count++;
    string s = "s";
    if (count == 1) s = "";
    return $"{count} {job} bee{s}";
}

public void AssignBee(string job)
{
    switch (job)
    {
        case "Nectar Collector":
            AddWorker(new NectarCollector());
            break;
        case "Honey Manufacturer":
            AddWorker(new HoneyManufacturer());
            break;
        case "Egg Care":
            AddWorker(new EggCare(this));
            break;
    }
    UpdateStatusReport();
}

protected override void DoJob()
{
    eggs += EGGS_PER_SHIFT;
    foreach (Bee worker in workers)
    {
        worker.WorkTheNextShift();
    }
    HoneyVault.ConsumeHoney(unassignedWorkers * HONEY_PER_UNASSIGNED_WORKER);
    UpdateStatusReport();
}
}
```

The private `WorkerStatus` method uses a `foreach` loop to count the number of bees in the `workers` array that match a specific job. Notice how it uses the `"s"` variable to use the plural `"bees"` unless there's just one bee.

The `AssignBee` method uses a `switch` statement to determine which type of worker to add. The strings in the case statements need to match the `Content` properties of each `ListBoxItem` in the `ComboBox` exactly, otherwise none of the cases will match.

The Queen does her job by adding eggs, telling each worker to work the next shift, and then making sure each of the unassigned workers consumes honey. She updates the status report after every bee assignment and shift to make sure it's always up to date.

The Queen is not a micromanager. She lets the worker Bee objects do their jobs and consume their own honey.

That's a good example of separation of concerns: queen-related behavior is encapsulated in the `Queen` class, and the `Bee` class contains only the behavior common to all bees.



Long Exercise Solution

The **constants at the top of each of the Bee subclasses** are really important. We came up with the values for those constants through trial and error: we tweaked one of the numbers, then ran the program to see what effect it had. We tried to come up with a good balance between the classes. Do you think we did a good job? *Can you do better? We bet you can!*

The **EggCare class** uses a reference to the Queen object to call her CareForEggs method to turn eggs into workers:

```
class EggCare : Bee
{
    public const float CARE_PROGRESS_PER_SHIFT = 0.15f;
    public override float CostPerShift { get { return 1.35f; } }

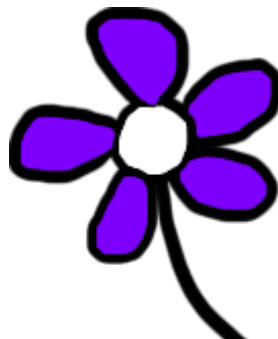
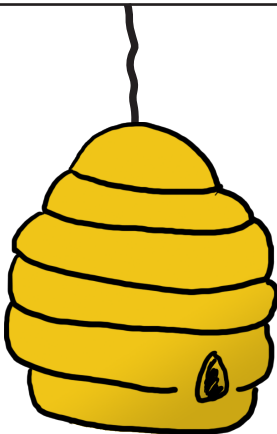
    private Queen queen;

    public EggCare(Queen queen) : base("Egg Care")
    {
        this.queen = queen;
    }

    protected override void DoJob()
    {
        queen.CareForEggs(CARE_PROGRESS_PER_SHIFT);
    }
}
```

The EggCare bee's constant determines how rapidly the eggs are turned into unassigned workers. More workers can be good for the hive, but they also consume more honey. The challenge is getting the right balance of different worker types.

If you run into trouble writing the code, it's absolutely okay to look at the solution!





HEY, WAIT A MINUTE. THIS...THIS ISN'T
A SERIOUS BUSINESS APPLICATION.
IT'S A GAME!

YOU GUYS REALLY SUCK.

Okay, you got us. Yes, you're right. This is a game.

Specifically, it's a **resource management game**, or a game where the mechanics are focused on collecting, monitoring, and using resources. If you've played a simulation game like SimCity or strategy game like Civilization, you'll recognize resource management as a big part of the game, where you need resources like money, metal, fuel, wood, or water to run a city or build an empire.

Resource management games are a great way to experiment with the relationship between **mechanics, dynamics, and aesthetics**:

- ★ The **mechanics** are simple: the player assigns workers and then initiates the next shift. Then each bee either adds nectar, reduces nectar/increases honey, or reduces eggs/increases workers. The egg count increases, and the report is displayed.
- ★ The **aesthetics** are more complex. Players feel stress as the honey or nectar levels fall and the low level warning is displayed. They feel excitement when they make a choice, and satisfaction when it affects the game—and then stress again, as the numbers stop increasing and start decreasing again.
- ★ The game is driven by the **dynamics**. There's nothing in the code that makes the honey or nectar scarce—they're just consumed by the bees and eggs.

Really take a minute and think about this, because it gets to the heart of what dynamics are about. Do you see any way to use some of these ideas in other kinds of programs, and not just games?



A small change in HoneyVault.NECTAR_CONVERSION_RATIO can make the game much easier or much harder by making the honey drain slowly or quickly. What other numbers affect gameplay? What do you think is driving those relationships?

Feedback drives your Beehive Management game

Let's take a few minutes and really understand how this game works. The nectar conversion ratio has a big impact on your game. If you change the constants, it can make really big differences in gameplay. If it takes just a little honey to convert an egg to a worker, the game gets really easy. If it takes a lot, the game gets much harder. But if you go through the classes, you won't find a difficulty setting. There's no Difficulty field on any of them. Your Queen doesn't get special power-ups to help make the game easier, or tough enemies or boss battles to make it more difficult. In other words, there's **no code that explicitly creates a relationship** between the number of eggs or workers and the difficulty of the game. So what's going on?

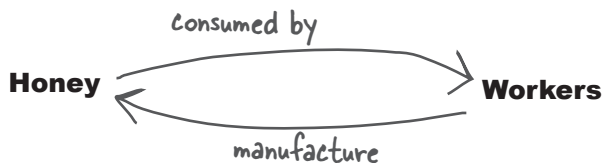
When you point a camera at a screen displaying its video output, you create a feedback loop that can cause these weird patterns.



You've probably played with **feedback** before. Start a video call between your phone and your computer. Hold the phone near the computer speaker and you'll hear noisy echoes. Point the camera at the computer screen and you'll see a picture of the screen inside the picture of the screen inside the picture of the screen, and it will turn into a crazy pattern if you tilt the phone. This is feedback: you're taking the live video or audio output and *feeding* it right *back* into the input. There's nothing in the code of the video call app that specifically generates those crazy sounds or images. Instead, they **emerge** from the feedback.

Workers and honey are in a feedback loop

Your Beehive Management game is based on a series of **feedback loops**: lots of little cycles where parts of the game interact with each other. For example, honey manufacturers add honey to the vault, which is consumed by honey manufacturers, who make more honey.

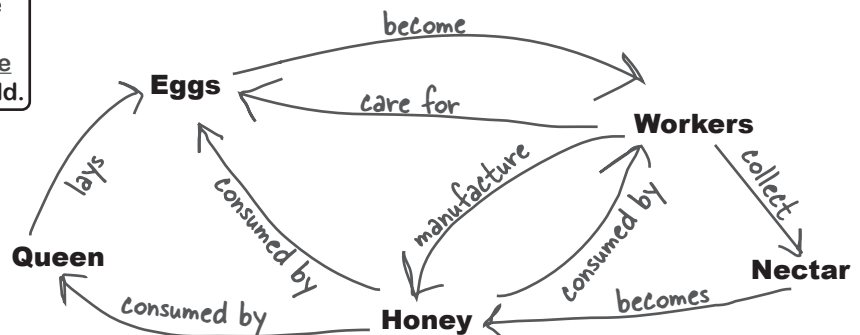


The feedback loop between the workers and honey is just one small part of the whole system that drives the game. See if you can spot it in the bigger picture below.

And that's just one feedback loop. There are many different feedback loops in your game, and they make the whole game more complex, more interesting, and (hopefully!) more fun.

A series of **feedback loops** drive the dynamics of your game. The code you build won't explicitly manage these feedback loops. They **emerge** out of the mechanics that you'll build.

And this same concept is actually really important in a lot of real-world business applications, not just games. Everything you're learning here, you can use on the job as a professional software developer.



Mechanics, Aesthetics, and Dynamics Up Close



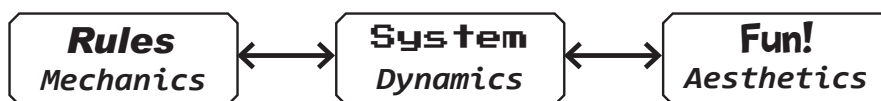
Feedback loops... equilibrium... making your code do something indirectly by creating a system... does all this have your head spinning a little? Here's another opportunity to **use game design to explore a larger programming concept**.

You've learned about mechanics, dynamics, and aesthetics—now it's time to bring them all together. The **Mechanics-Dynamics-Aesthetics framework**, or **MDA framework**, is a formal tool ("formal" just means it's written down) that's used by researchers and academics to analyze and understand games. It defines the relationship between mechanics, dynamics, and aesthetics, and gives us a way to talk about how they create feedback loops to influence each other.

The MDA framework was developed by Robin Hunicke, Marc LeBlanc, and Robert Zubek, and published in a 2004 paper called "MDA: A Formal Approach to Game Design and Game Research"—it's quite readable, without a ton of academic jargon. (Remember back in Chapter 5, when we talked about how aesthetics includes challenge, narrative, sensation, fantasy, and expression? That came from this paper.) Take a few minutes and look through it, it's actually a great read: <http://bit.ly/mda-paper>.

The goal of the MDA framework is to give us a formal way to think about and analyze video games. This may sound like something that's only important in an academic setting, like a college course on game design. But it's actually really valuable to us as everyday game developers, because it can help us understand how people perceive the games we create, and give us a deeper insight into **what makes those games fun**.

Game designers had been using the terms mechanics, dynamics, and aesthetics informally, but the paper really gave them a solid definition, and established the relationship between them.



One thing that the MDA framework tackles is the **difference in perspective** between gamers and game designers. Players, first and foremost, want the game to be fun—but we've already seen how "fun" can differ wildly from player to player. Game designers, on the other hand, typically see a game through the lens of its mechanics, because they spend their time writing code, designing levels, creating graphics, and tinkering with the mechanical aspects of the game.

All developers (not just game developers!) can use the MDA framework to get a handle on feedback loops

Let's use the MDA framework to analyze a classic game, Space Invaders, so we can better understand feedback loops.

- Start with mechanics of the game: the player's ship moves left and right and fires shots up; the invaders march in formation and fire shots down; the shields block shots. The fewer enemies there are on screen, the faster they go.
- Players figure out strategies: shoot where the invaders will be, pick off enemies on the sides of the formation, hide behind the shields. The code for the game doesn't have an `if/else` or `switch` statement for these strategies; they emerge as the player figures out the game. Players learn the rules, then start to understand the system, which helps them better take advantage of the rules. In other words, **the mechanics and dynamics form a feedback loop**.
- The invaders get faster, the marching sounds speed up, and the player gets a rush of adrenaline. The game gets more exciting—and in turn, the player has to make decisions more quickly, makes mistakes, and changes strategy, which has an effect on the system. **The dynamics and aesthetics form another feedback loop**.
- None of this happened by accident. The speed of the invaders, the rate at which they increase, the sounds, the graphics... these were all carefully balanced by the game's creator, Tomohiro Nishikado, who spent over a year designing it, drawing inspiration from earlier games, H. G. Wells, even his own dreams to create a classic game.

The Beehive Management System is turn-based... now let's convert it to real-time

A **turn-based game** is a game where the flow is broken down into parts—in the case of the Beehive Management System, into shifts. The next shift doesn't start until you click a button, so you can take all the time you want to assign workers. We can use a `DispatcherTimer` (like the one you used in Chapter 1) to **convert it to a real-time game** where time progresses continuously—and we can do it with just a few lines of code.

1 Add a `@using` line to the top of your `Index.razor` file.

We'll be using a `Timer` to force the game to work the next shift every second and a half. `Timer` is in the `System.Timers` namespace, so you'll need to add this `@using` line to the top of your `Index.razor` file:

```
@page "/"
@using System.Timers
```

You used a `Timer` in Chapter 1 to add a timer to your animal matching game. This code is very similar to the code you used in Chapter 1. Take a few minutes and flip back to that project to remind yourself how the `Timer` works.

2 Add a field to hold the `Timer` reference.

A `Timer` is an object—add field to the top of the `@code` section to hold a reference to it:

```
@code {
    Queen queen = new Queen();
    string selectedJob = "Nectar Collector";
    Timer timer;
```

3 Create the new `Timer` and add its `Tick` event handler.

We want the timer to keep the game moving forward, so if the player doesn't click the button quickly enough it will automatically trigger the next shift. Add an `OnInitialized` method to create the new timer object and hook up its event handler method, then add the event handler that calls the `Queen` object's `WorkTheNextShift` method:

```
protected override void OnInitialized()
{
    timer = new Timer(1500);
    timer.Elapsed += Timer_Tick;
    timer.Start();
}
```

← You used `OnInitialized` methods in Chapters 1 and 4. Take a closer look at the declaration—now you know what the `protected` and `override` keywords mean.

```
private void Timer_Tick(Object source, ElapsedEventArgs e)
{
    InvokeAsync(() => {
        queen.WorkTheNextShift();
        StateHasChanged();
    });
}
```

The `() =>` code is a lambda expression. You'll learn more about how that works in Chapter 9.

This is very similar to code that you used in Chapter 1 in your animal matching game. `InvokeAsync` tells the Blazor page to run a block of code without causing the page to become unresponsive. `StateHasChanged` tells the Blazor page to refresh all of its bindings.

IT TOOK **JUST A FEW LINES OF CODE** TO ADD THE TIMER, BUT THAT COMPLETELY CHANGED THE GAME. IS THAT BECAUSE IT HAD A BIG IMPACT ON THE **RELATIONSHIP** BETWEEN MECHANICS, DYNAMICS, AND AESTHETICS?

Yes! The timer changed the mechanics, which altered the dynamics, which in turn impacted the aesthetics.

Let's take a minute and think about that feedback loop. The change in mechanics (a timer that automatically clicks the "Work the next shift" button every 1.5 seconds) creates a totally new dynamic: a window when players must make decisions, or else the game makes the decision for them. That increases the pressure, which gives some players a satisfying shot of adrenaline, but just causes stress in other players—the aesthetics changed, which makes the game more fun for some people but less fun for others.

But you only added half a dozen lines of code to your game, and none of them included "make this decision or else" logic. That's an example of behavior that **emerged** from how the timer and the button work together.

← There's a feedback loop here, too. As players feel more stress, they make worse decisions, changing the game... aesthetics feeds back into mechanics.

THIS WHOLE DISCUSSION OF FEEDBACK LOOPS SEEMS PRETTY IMPORTANT—ESPECIALLY THE PART ABOUT **HOW BEHAVIOR EMERGES**.

Feedback loops and emergence are important programming concepts.

We designed this project to give you practice with inheritance, but *also* to let you explore and experiment with **emergent** behavior. That's behavior that comes not just from what your objects do individually, but also out of ***the way objects interact with each other***. The constants in the game (like the nectar conversion ratio) are an important part of that emergent interaction. When we created this exercise, we started out by setting those constants to some initial values, then we tweaked them by making tiny adjustments until we ended up with a system that's not quite in **equilibrium**—a state where everything is perfectly balanced—so the player needs to keep making decisions in order to make the game last as long as possible. That's all driven by the feedback loops between the eggs, workers, nectar, honey, and queen.

Try experimenting with these feedback loops. Add more eggs per shift or start the hive with more honey, for example, and the game gets easier. Go ahead, give it a try! You can change the entire feel of the game just by making small changes to a few constants.

Nice work! You can head back to Chapter 6—resume at the "Some classes should never be instantiated" section.