O'REILLY®

~~Fourth~~ Fifth Edition

# Head First C#

This is part of an early release preview of the 5th edition of Head First C# by Andrew Stellman and Jenny Greene. We'll release the final version of this PDF when the book is published in late 2023.

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

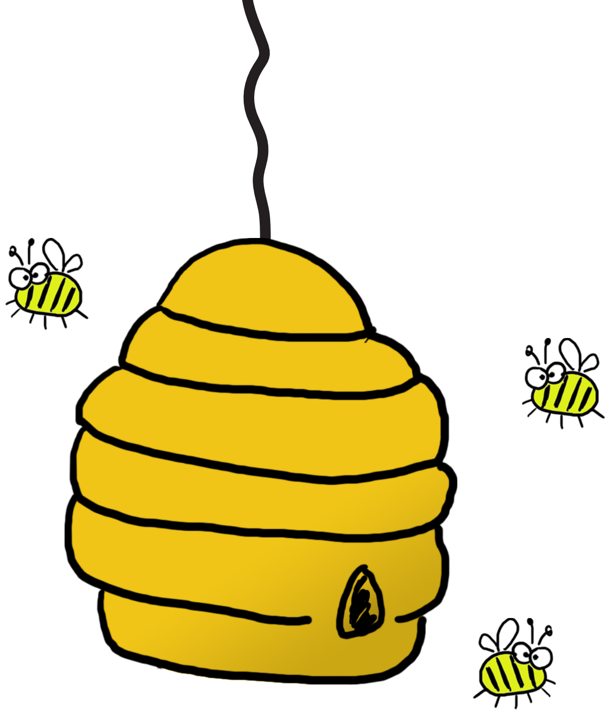
Andrew Stellman & Jennifer Greene

This is the .NET MAUI project from Chapter 6. You'll build an app called Beehive Management System where you help a queen bee manage her workers and keep her hive up and running.

A Brain-Friendly Guide

## Build a beehive management system

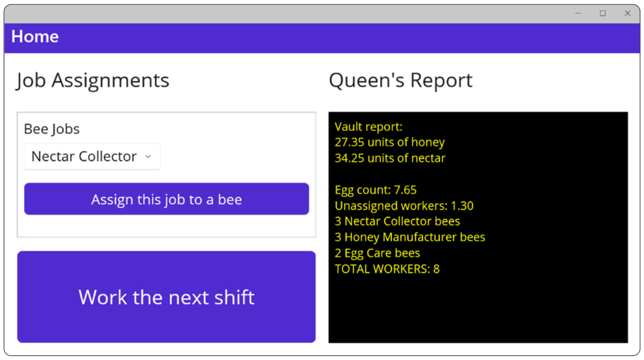
***The queen bee needs your help!*** Her hive is out of control, and she needs a program to help manage her honey production business. She's got a beehive full of workers, and a whole bunch of jobs that need to be done around the hive, but somehow she's lost control of which bee is doing what, and whether 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. At the end of the shift she gets 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**.

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-nkruptcy?



```
Vault report:
27.35 units of honey
34.25 units of nectar

Egg count: 7.65
Unassigned workers: 1.30
3 Nectar Collector bees
3 Honey Manufacturer bees
2 Egg Care bees
TOTAL WORKERS: 8
```

**This is a <u>bigger project</u> than the ones in the last few chapters.**

This is a big project. You <u>can</u> do this!

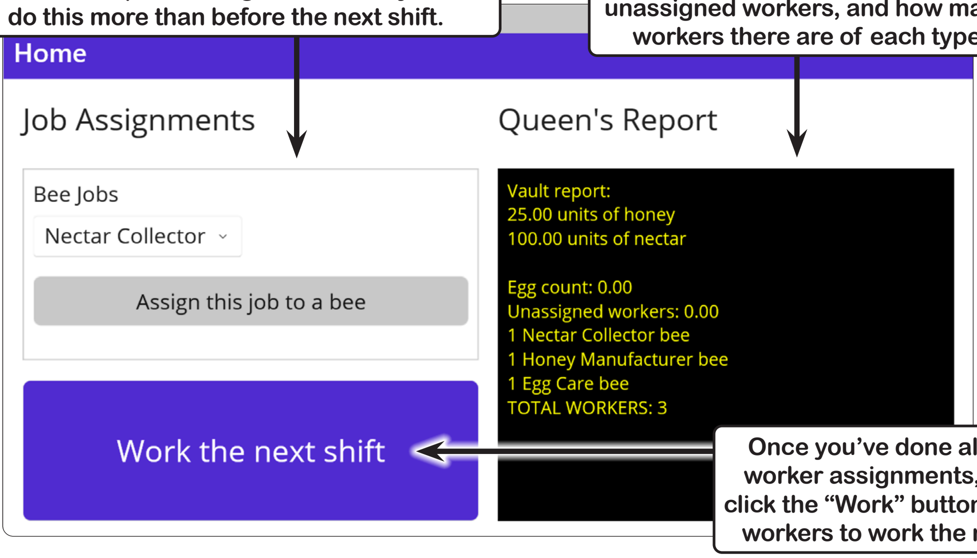
The main goal of this book is to help you learn C#. But we'll also teach important skills that can help you ***become a great developer***. One way to do that is to help show you how to work on—and finish!—larger projects. When you did the Animal Matching Game project in Chapter 1, you broke it down into smaller pieces. You'll do the same for the Beehive Management System project. **First** you'll create the XAML for the main page, **then** you'll do a "Sharpen Your Pencil" exercise to complete the code for several of the classes, and **finally** you'll do an exercise to finish the rest of the code for the project.

## How the Beehive Management System app works

When the app starts, the honey vault has 25 units of honey and 100 units of nectar, and the hive has three workers: a nectar collector bee, a honey manufacturer bee, and an egg care bee. The first shift report delivered is displayed on the right hand side of the app.

**If there are any unassigned workers at the start of the shift, you can use the Bee Jobs picker to choose a job, then chlick the "Assign" button to give a worker that job (it's disabled if there aren't enough unassigned workers). If there are multiple unassigned wokers, you can do this more than before the next shift.**

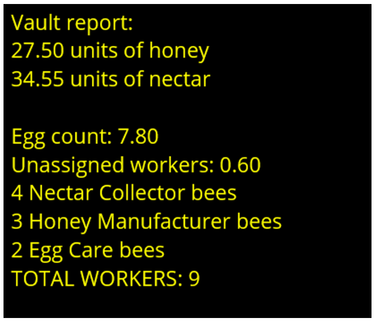
**This is the shift report that the queen generates at the end of every shift. It shows the status of the honey valut, followed by the number of eggs, unassigned workers, and how many workers there are of each type.**



```
Vault report:
25.00 units of honey
100.00 units of nectar

Egg count: 0.00
Unassigned workers: 0.00
1 Nectar Collector bee
1 Honey Manufacturer bee
1 Egg Care bee
TOTAL WORKERS: 3
```

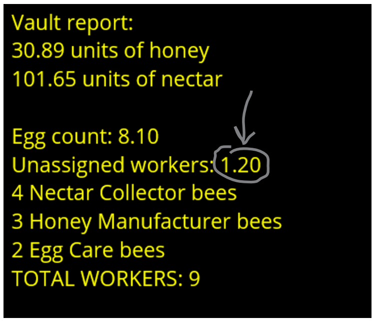
**Once you've done all of your worker assignments, you can click the "Work" button to tell the workers to work the next shift.**

**Each worker consumes honey to do a job. The numbers change at the end of each shift to show what they did.**



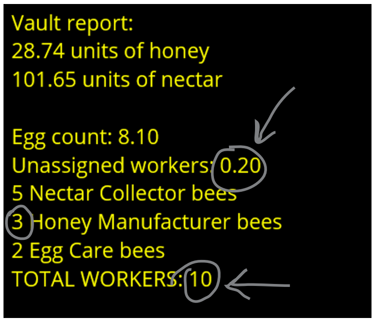
```
Vault report:
27.50 units of honey
34.55 units of nectar

Egg count: 7.80
Unassigned workers: 0.60
4 Nectar Collector bees
3 Honey Manufacturer bees
2 Egg Care bees
TOTAL WORKERS: 9
```

**If the Unassigned Workers count is at least 1, clicking the Assign button assigns a worker to the selected job.**



```
Vault report:
30.89 units of honey
101.65 units of nectar

Egg count: 8.10
Unassigned workers: 1.20
4 Nectar Collector bees
3 Honey Manufacturer bees
2 Egg Care bees
TOTAL WORKERS: 9
```

**Assigning a worker makes the unassigned workers go down by 1 and the total workers increase by 1.**

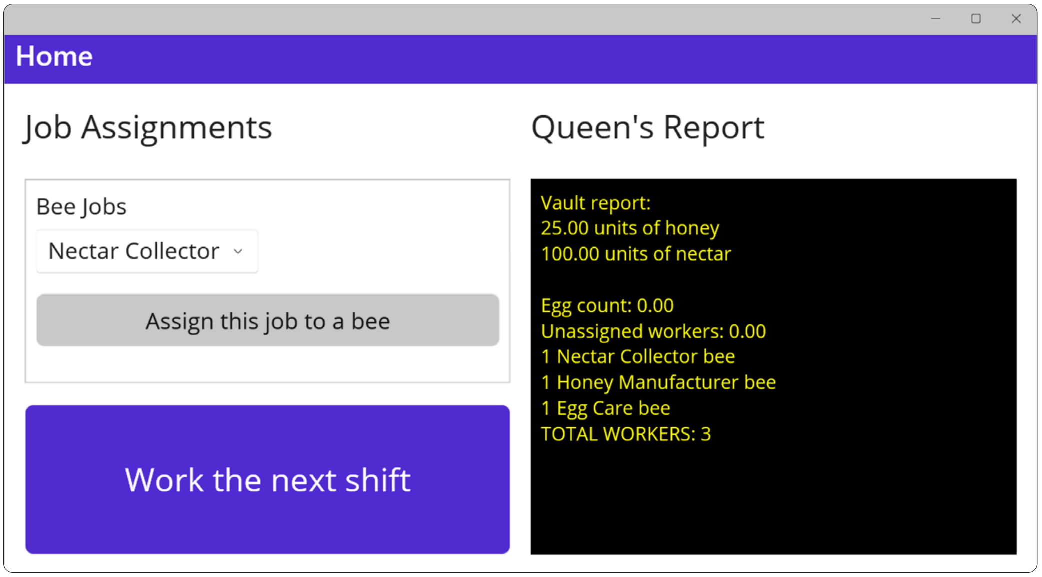


```
Vault report:
28.74 units of honey
101.65 units of nectar

Egg count: 8.10
Unassigned workers: 0.20
5 Nectar Collector bees
3 Honey Manufacturer bees
2 Egg Care bees
TOTAL WORKERS: 10
```

## The page uses a grid to lay out the controls for the UI

The Beehive Management System app is a .NET MAUI app. The main page uses a Grid control to lay out the Labels, Buttons, Picker, and other controls.

Take a closer look at it to see how it works:



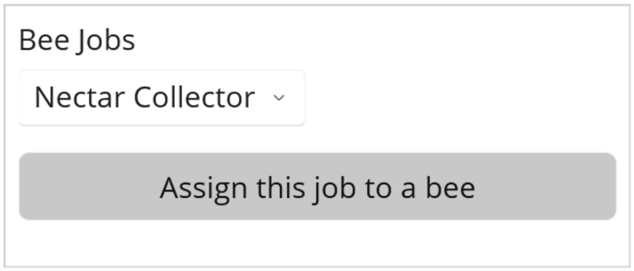
```
Vault report:
25.00 units of honey
100.00 units of nectar

Egg count: 0.00
Unassigned workers: 0.00
1 Nectar Collector bee
1 Honey Manufacturer bee
1 Egg Care bee
TOTAL WORKERS: 3
```

The Grid has three properties:

- `Margin="20"` adds 20 pixles of space between the controls in the grid and the window frame.
- `ColumnSpacing="20"` adds 20 pixels of space between the two columns.
- `MinimumHeightRequest="400"` keeps the grid from getting smaller than 400 pixels high. If you resize the window smaller than that, the ScrollView will scroll the grid for you.

### The Bee Jobs box uses a Border with a VerticalStackLayout

The middle row of the left column has a box that contains a picker and a button. You'll use a Border to draw the box. You need to put two controls inside the Border, but Border controls can only contain one other control, so you'll put a VerticalStackLayout inside it to contain the Picker and Button controls.

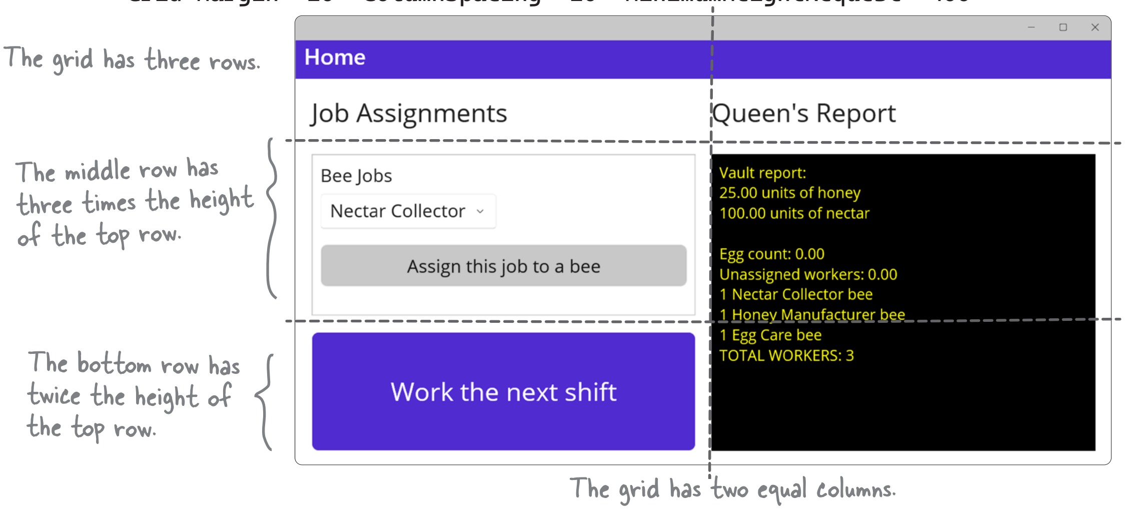


```
<Border Padding="10" Margin="0,0,0,20" >
  <VerticalStackLayout>

     <Picker Title="Bee Jobs" ... />

     <Button x:Name="AssignJobButton" ... />

  </VerticalStackLayout>
</Border>
```

## Exercise

Add the XAML for tne main page to your MainPage.xaml.cs file. It uses a grid—here's how it's laid out:

```
<ContentPage ... >
    <ScrollView>
        <Grid Margin="20" ColumnSpacing="20" MinimumHeightRequest="400">
```


The grid has three rows.
The middle row has three times the height of the top row.
The bottom row has twice the height of the top row.
The grid has two equal columns.
Home
Job Assignments
Queen's Report
Bee Jobs
Nectar Collector
Assign this job to a bee
Work the next shift

```
Vault report:
25.00 units of honey
100.00 units of nectar

Egg count: 0.00
Unassigned workers: 0.00
1 Nectar Collector bee
1 Honey Manufacturer bee
1 Egg Care bee
TOTAL WORKERS: 3
```

```
        </Grid>
    </ScrollView>
</ContentPage>
```

The left column of the grid has the following controls:

- The top row has a Label control. Its FontSize property is set to Large.
- The middle row has a Border control. Itcontains a VerticalStackLayout, which contains a Picker and a Button. Use x:Name to name the Picker JobPicker and name the button AssignJobButton.
- The Picker has a 20 pixel bottom margin. They both have FontSize set to Medium. Look at the screenshot to figure out the Picker's Title and the Button's Text properties. Add a SemanticProperties.Hint to each of them.
- Modify the MainPage constructor in MainPage.xaml.cs to set JobPicker.ItemsSource to a new array that contains the following strings: `"Nectar Collector", "Honey Manufacturer", "Egg Care"`
- The bottom row has a Button control. Use x:Name to name it WorkShiftButton. Its font size is large.
- The bottom row has a ***second*** **Button control**. You can't see it because it has the `IsVisible="False"` property. Set its BackgroundColor property to Red, its FontSize to Large, and use x:Name to name it OutOfHoneyButton. It should display the following text: `"The hive is out of honey"`
- Add an event handler method for all three of of the Button controls. Make sure they all semantic hints.

The right column of the grid has the following controls:

- The top row has a Label control. Its FontSize property is set to large.
- The bottom two rows have another Label control with RowSpan="2" so it spans both rows.Its background color is set to black, and text color is set to yellow. It has the Padding="10" property to give space between the edge and the text, and VerticalOptions="FillAndExpand" so it filles the cell. Use x:Name to name it StatusReport.

**WARNING! At the time we're writing this, there's a bug in the Windows version of .NET MAUI when you put a Border inside a Grid that causes it to display incorrectly. To fix it, drag the lower right corner of the window a few pixels—the Grid will adjust its contents when you resize it. 😿 😿**

## Exercise Solution

Here's the XAML for the main page. Did yours come out a little different than ours? That's okay! There are a lot of different ways to design this page—if it still works and you like the way it looks, then you got it right.

```
<ScrollView>
  <Grid Margin="20" ColumnSpacing="20" MinimumHeightRequest="400">
    <Grid.RowDefinitions>
        <RowDefinition Height="1*"/>
        <RowDefinition Height="3*"/>
        <RowDefinition Height="2*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Label Text="Job Assignments" FontSize="Large" />

    <Border Grid.Row="1" Padding="10" Margin="0,0,0,20" >
        <VerticalStackLayout>

            <Picker x:Name="JobPicker" Title="Bee Jobs"
                 FontSize="Medium" Margin="0,0,0,20"
                SemanticProperties.Hint="Lets the user pick a job to assign" />

            <Button x:Name="AssignJobButton" Text="Assign this job to a bee"
                FontSize="Medium" Clicked="AssignJobButton_Clicked"
                SemanticProperties.Hint="Assigns the job to a worker bee" />

        </VerticalStackLayout>
    </Border>

    <Button x:Name="WorkShiftButton" Grid.Row="2" Text="Work the next shift"
            FontSize="Large" Clicked="WorkShiftButton_Clicked"
            SemanticProperties.Hint="Makes the bees work the next shift" />

    <Button x:Name="OutOfHoneyButton" Grid.Row="2" Text="The hive is out of honey"
            BackgroundColor="Red" FontSize="Large" IsVisible="False"
            Clicked="OutOfHoneyButton_Clicked"
            SemanticProperties.Hint="Restarts the Beehive Management System" />

    <Label Text="Queen's Report" Grid.Column="1" FontSize="Large"  />

    <Label x:Name="StatusReport" Grid.Column="1" Grid.Row="1" Grid.RowSpan="2"
           VerticalOptions="FillAndExpand" FontSize="Small"
           BackgroundColor="Black" TextColor="Yellow" Padding="10" />
  </Grid>
</ScrollView>
```

The middle row is three times as tall as the top row, and the bottom row is twice as tall.

The combination of the Border and VerticalStackLayout makes both the Picker and Button appear inside a box.

The "Work the next shift" button will be visible while the hive is working. If the hive runs out of honey, the app will make it invisble and show the other button to let the user restart the hive.

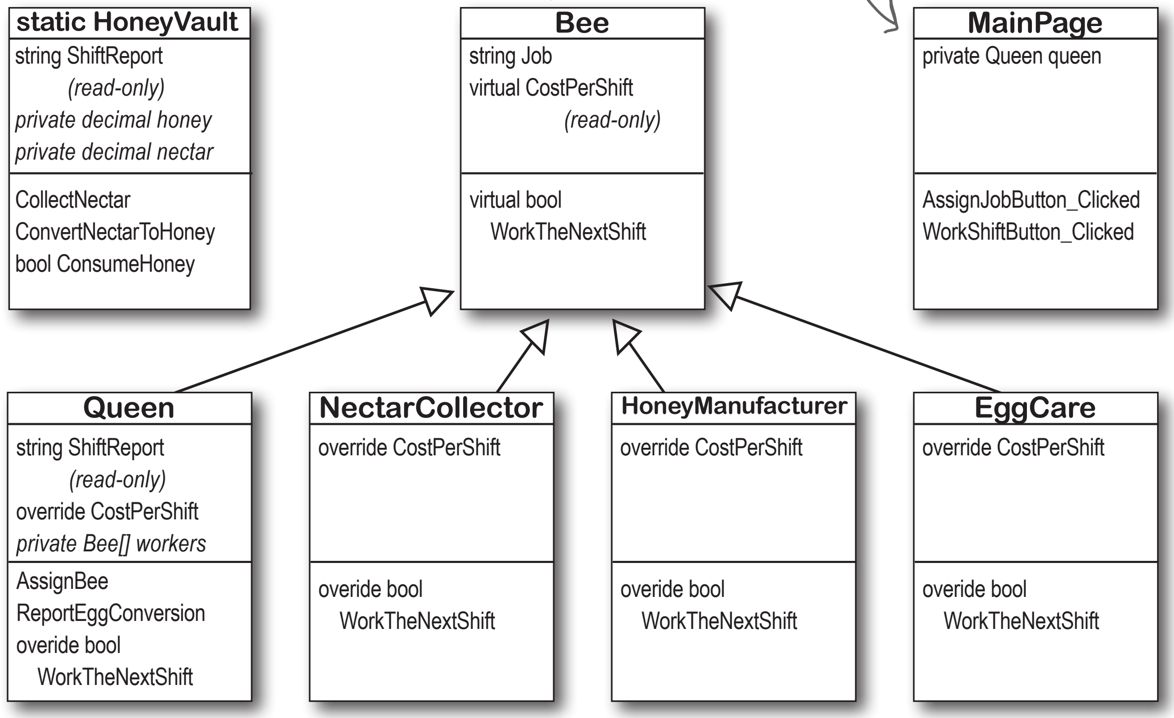
## The beehive management system class model

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 MainPage class with the code-behind for the main page.

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 to make sure each bee consumes the honey they need to do their jobs. If there's not enough honey, it returns false.

The code-behind for the main page 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 shift report.



This Bee subclass uses an array to keep track of the workers and calls each of their WorkTheNextShift methods to make them do their jobs. Queen also has private methods, which we'll talk about later.

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

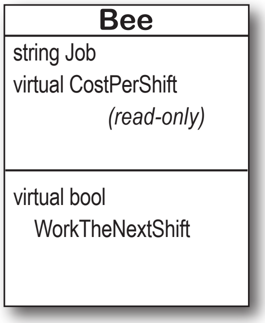
This Bee subclass overrides WorkTheNextShift to call the HoneyVault method to convert nectar to honey.

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

**You'll use decimal fields to keep track of honey and nectar, for the same reasons you'd use decimal for money.**

## All bees in the system extend the Bee class

This is the Bee class. It’s the **superclass of all the other bee classes in the system**. Each shift, every bee—the queen and every one of her wokers—consumes honey and does their job. The system does this by calling the WorkTheNextShift method, which consumes the honey from the vault using the CostPerShift property. The WorkTheNextShift method is marked **`virtual`** because each subclass will extend it to do their specific job.



```
class Bee
{
    /// <summary>
    /// The units of honey this bee consumes per shift
    /// </summary>
    public virtual decimal CostPerShift { get; }

    /// <summary>
    /// The job assigned to this bee
    /// </summary>
    public string Job { get; private set; }

    /// <summary>
    /// The Bee class constructor
    /// </summary>
    /// <param name="job">The job to assign to this bee</param>
    public Bee(string job)
    {
        Job = job;
    }

    /// <summary>
    /// If there's enough honey for this bee to work the next shift, consume the honey
    /// </summary>
    /// <returns>True if there was enough honey to do the job, false otherwise</returns>
    public virtual bool WorkTheNextShift()
    {
        if (HoneyVault.ConsumeHoney(CostPerShift))
            return true;
        else
            return false;
    }
}
```

**Each subclass will override CostPerShift to return a different value. Egg care work is just as important as collecting honey, but flying around and collecting nectar takes more energy than caring for eggs, so the NectarCollector’s CostPerShift property will return a higher value than the EggCare’s CostPerShift property.**

**Every bee needs to consume honey, so the WorkTheNextShift method consumes the amount of honey that particular bee needs. Every bee class does a different job, so they all consume different amounts of honey—and since C# always calls the most specific member, WorkTheNextShift will use the CostPerShift property to figure out how much honey to consume, and each subclass’s CostPerShift propery will return the amount of honey consumed for that specifc type of bee.**

## All the constants are in their own static class

Each of the bees consumes honey and does a job. These constants determine how both of those things work.

```
static class Constants
{
    /// <summary>
    /// The amount of honey the queen consumes each shift
    /// </summary>
    public const decimal QUEEN_COST_PER_SHIFT = 2.15m;

    /// <summary>
    /// Number of eggs an egg the queen lays per shift
    /// </summary>
    public const decimal EGGS_PER_SHIFT = 0.45m;

    /// <summary>
    /// The shift report shows a warning if the honey level drops below this
    /// </summary>
    public const decimal LOW_LEVEL_WARNING = 10m;

    /// <summary>
    /// How much honey each unassigned worker consumes
    /// </summary>
    public const decimal HONEY_PER_UNASSIGNED_WORKER = 0.5m;

    /// <summary>
    /// Cost of a nectar collector per shift
    /// </summary>
    public const decimal NECTAR_COLLECTOR_COST = 1.95m;

    /// <summary>
    /// How much nectar a nectar collector can collect per shift
    /// </summary>
    public const decimal NECTAR_COLLECTED_PER_SHIFT = 33.25m;

    /// <summary>
    /// Cost of an honey manufacturer per shift
    /// </summary>
    public const decimal HONEY_MANUFACTURER_COST = 1.7m;

    /// <summary>
    /// How much nectar the honey manufacturer processes per shift
    /// </summary>
    public const decimal NECTAR_PROCESSED_PER_SHIFT = 33.15m;

    /// <summary>
    /// How many units of honey gets converted from one unit of nectar
    /// </summary>
    public const decimal NECTAR_CONVERSION_RATIO = .19m;

   /// <summary>
    /// Cost of an egg care worker per shift
    /// </summary>
    public const decimal EGG_CARE_COST = 1.35m;

    /// <summary>
    /// Number of eggs an egg care worker converts to unassigned workers per shift
    /// </summary>
    public const decimal CARE_PROGRESS_PER_SHIFT = 0.15m;
}
```

Every shift the queen consumes a certain amount of honey. During the shift she lays a ceratin number of eggs. These constants determine exactly how much honey she consumes and how many eggs she lays each shift.

Every unassigned worker consumes honey per shift, but less than the rest of the workers because they're sitting around waiting to do a job.

The nectar collector class will use these constants to determine how much each worker costs and how much nectar they collect during the shift.

These constants determine how much honey a honey manufacturer processes each shift, and how much they consume.

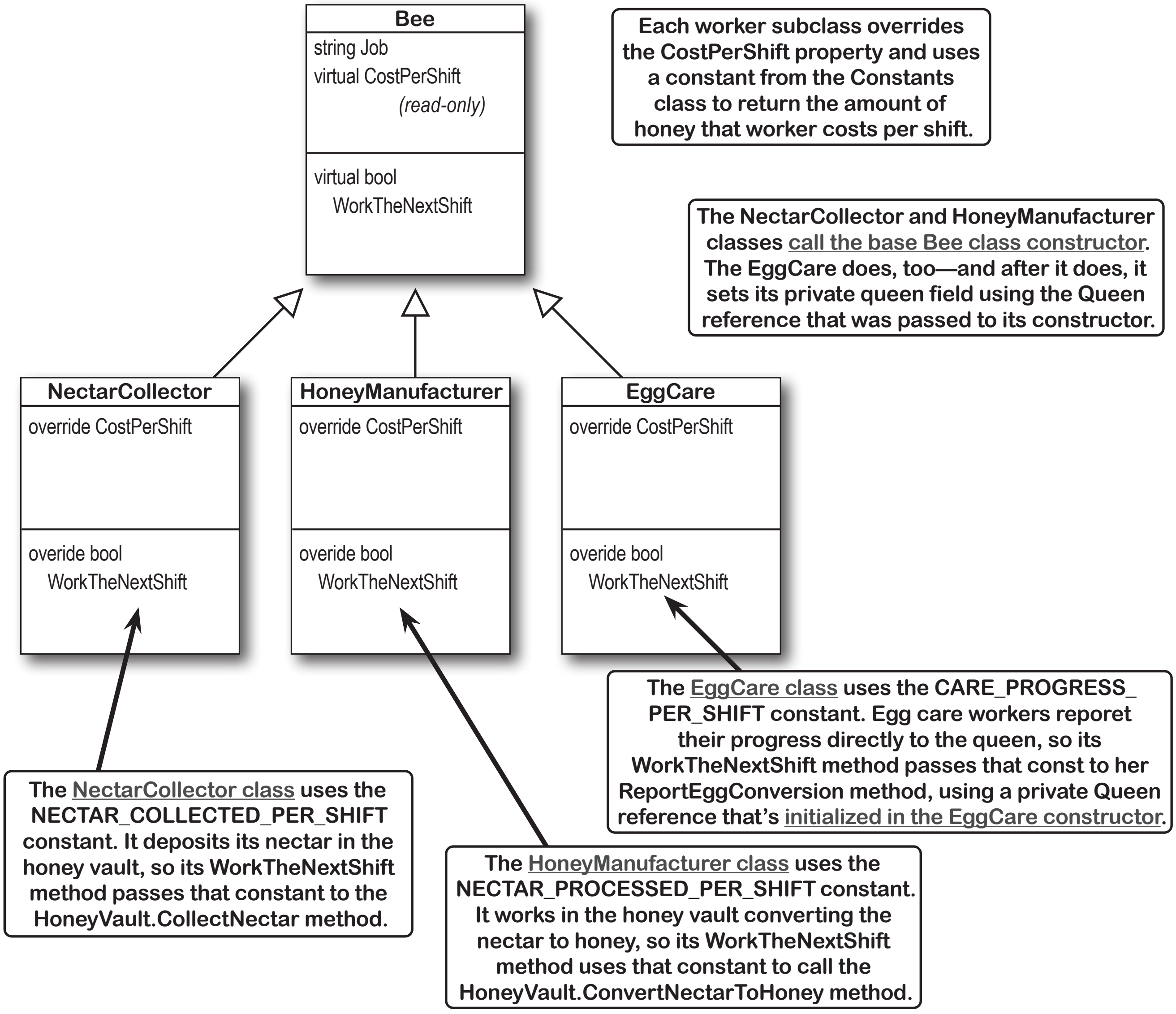
This is used by the honey vault in its method to convert nectar to honey.

This is used by the honey vault in its method to convert nectar to honey.

## The worker bees extend the Bee class

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 shift 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.

Here's a closer look at how the NectarCollector, HoneyManufacturer, and EggCare classes work. They're all subclasses of the Bee class. They all use constants defined in the static Constants class. Each bee does a different job, so they all have different WorkTheNextShift methods that override Bee.WorkTheNextShift to do their jobs.



**Each worker subclass overrides the CostPerShift property and uses a constant from the Constants class to return the amount of honey that worker costs per shift.**

**The NectarCollector and HoneyManufacturer classes call the base Bee class constructor. The EggCare does, too—and after it does, it sets its private queen field using the Queen reference that was passed to its constructor.**

**The EggCare class uses the CARE_PROGRESS_ PER_SHIFT constant. Egg care workers reporet their progress directly to the queen, so its WorkTheNextShift method passes that const to her ReportEggConversion method, using a private Queen reference that's initialized in the EggCare constructor.**

**The NectarCollector class uses the NECTAR_COLLECTED_PER_SHIFT constant. It deposits its nectar in the honey vault, so its WorkTheNextShift method passes that constant to the HoneyVault.CollectNectar method.**

**The HoneyManufacturer class uses the NECTAR_PROCESSED_PER_SHIFT constant. It works in the honey vault converting the nectar to honey, so its WorkTheNextShift method uses that constant to call the HoneyVault.ConvertNectarToHoney method.**

## Sharpen your pencil

Fill in the missing parts of the Bee subclasses based on what we've told you about how they work.

```
class NectarCollector : Bee
{
    public NectarCollector() : _______________________

    public override decimal CostPerShift {
        get { return _________________________________ }
    }

    public override bool WorkTheNextShift()
    {
        ____________________(____________________________________);
        return base.WorkTheNextShift();
    }
}

class HoneyManufacturer : Bee
{
    public HoneyManufacturer() : _________________________

    public override decimal CostPerShift {
      get { return ___________________________________ }
    }

    public override bool WorkTheNextShift()
    {
        __________________________(____________________________________);
        return base.WorkTheNextShift();
    }
}

class EggCare : Bee
{
    private Queen queen;

    public EggCare(Queen queen) : _______________
    {
        this.queen = queen;
    }

    public override decimal CostPerShift {
        get { return ________________________; }
    }

    public override bool WorkTheNextShift()
    {
        _____________________(__________________________________);
        return base.WorkTheNextShift();
    }
}
```

Each Bee subclass overrides WorkTheNextShift so it first does its job, then consumes its honey – so this statement is what makes the NectarCollector bee do its job.

**The only difference between these bees is the job they do and how much noney they consume, so it makes sense that they all extend the Bee class and override the CostPerShift property and the WorkTheNextShift method.**

## Sharpen your pencil Solution

Fill in the missing parts of the Bee subclasses based on what we've told you about how they work.

```
class NectarCollector : Bee
{
    public NectarCollector() : base("Nectar Collector") { }

    public override decimal CostPerShift {
        get { return Constants.NECTAR_COLLECTOR_COST; }
    }

    public override bool WorkTheNextShift()
    {
        HoneyVault.CollectNectar(Constants.NECTAR_COLLECTED_PER_SHIFT);
        return base.WorkTheNextShift();
    }
}

class HoneyManufacturer : Bee
{
    public HoneyManufacturer() : base("Honey Manufacturer") { }

    public override decimal CostPerShift {
      get { return Constants.HONEY_MANUFACTURER_COST; }
    }

    public override bool WorkTheNextShift()
    {
        HoneyVault.ConvertNectarToHoney(Constants.NECTAR_PROCESSED_PER_SHIFT);
        return base.WorkTheNextShift();
    }
}

class EggCare : Bee
{
    private Queen queen;

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

    public override decimal CostPerShift {
        get { return Constants.EGG_CARE_COST; }
    }

    public override bool WorkTheNextShift()
    {
        queen.ReportEggConversion(Constants.CARE_PROGRESS_PER_SHIFT);
        return base.WorkTheNextShift();
    }
}
```

All three classes call the base constructor, passing it a string that's used to figure out which job the bee is doing.

The NectarCollector's WorkTheNextShift method passes the NECTAR_COLLECTED_PER_SHIFT constant to the HoneyVault.CollectNectar method.

The HoneyManufacturer's WorkTheNextShift method passes the NECTAR_PROCESSED_PER_SHIFT constantto the HoneyVault.ConvertNectarToHoney method.

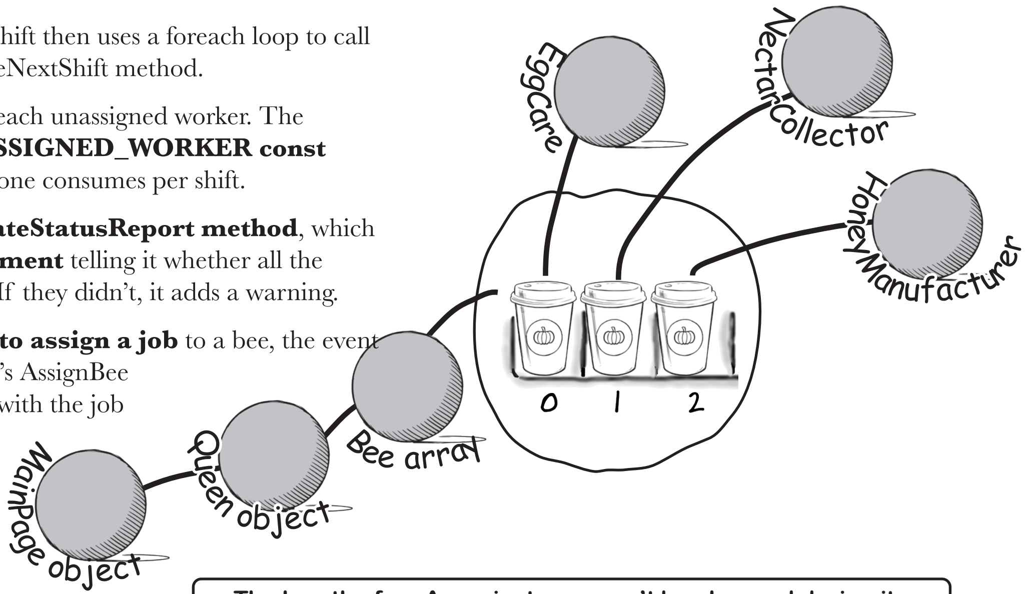
All three classes override the CostPerShift property and have it return a constant from the static Constants class.

Each EggCare object calls the queen's ReportEggConversion method to tell the queen how many eggs they converted into unassigned worker bees.

## The Queen class: how she manages the worker bees

When you **press the button to work the next shift**, the button's 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.
- The Queen overrides **overrides WorkTheNextShift** so that each shift she add 0.45 eggs to her private eggs field (using the EGGS_PER_SHIFT constant). The EggCare bee will call her ReportEggConversion method to tell the Queen to update the egg and worker counts. She'll then decrease eggs and increase unassignedWorkers.
- Queen.WorkTheNextShift then 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**, which takes **a boolean argument** telling it whether all the workers did their jobs. If they didn't, it adds a warning.



When you **press the button to assign a job** to a bee, the event handler calls the Queen object's AssignBee methoad, 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)
{
    if (unassignedWorkers >= 1)
    {
        unassignedWorkers--;
        Array.Resize(ref workers, workers.Length + 1);
        workers[workers.Length - 1] = 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.

## Here's the code-behind for MainPage.xaml.cs

This code will help you figure out how to write the code for the HoneyVault and Queen classes.

```
public partial class MainPage : ContentPage
{
    private Queen queen = new Queen();

    public MainPage()
    {
        InitializeComponent();

        JobPicker.ItemsSource = new string[]
        {
            "Nectar Collector",
            "Honey Manufacturer",
            "Egg Care"
        };
        JobPicker.SelectedIndex = 0;

        UpdateStatusAndEnableAssignButton();
    }

    private void UpdateStatusAndEnableAssignButton()
    {
        StatusReport.Text = queen.StatusReport;
        AssignJobButton.IsEnabled = queen.CanAssignWorkers;
    }

    private void WorkShiftButton_Clicked(object sender, EventArgs e)
    {
        if (!queen.WorkTheNextShift())
        {
            WorkShiftButton.IsVisible = false;
            OutOfHoneyButton.IsVisible = true;
            SemanticScreenReader.Default.Announce(OutOfHoneyButton.Text);
        }

        UpdateStatusAndEnableAssignButton();
    }

    private void AssignJobButton_Clicked(object sender, EventArgs e)
    {
        queen.AssignBee(JobPicker.SelectedItem.ToString());
        UpdateStatusAndEnableAssignButton();
    }

    private void OutOfHoneyButton_Clicked(object sender, EventArgs e)
    {
        HoneyVault.Reset();
        queen = new Queen();
        WorkShiftButton.IsVisible = true;
        OutOfHoneyButton.IsVisible = false;
        UpdateStatusAndEnableAssignButton();
    }
}
```

**The UpdateStatusAndEnableAssignButton method does two things. It uses the Queen object's StatusReport property to set the text of the label that displays the status report. Then it uses the CanAssignWorkers property to enable or disable the button that assigns the job—which means that property needs to return True if there's at least one unassigned worker, or False if there isn't.**

If the honey vault runs out of honey, the Queen object's WorkTheNextShift method returns false. If that happens, the bees can't work the next shift, so it hides the button to work the next shift and shows the button to resset the Queen and the Honey Vault.

The "assign job" button passes the text from the selected Picker item directly to Queen. AssignBee, so it's really important that the cases in the switch statement match the Picker items exactly.

This button's event handler resets the Queen and the honey vault, and sets the controls back to the way they were when the app started.

## Exercise

This is the biggest exercise we've given you so far. You can do this! Just take it one step at a time.

**Add the Constants class we gave you earlier, then add a static class called HoneyVault to manage the honey**

- Look back at the "Sharpen Your Pencil" solution to see how the Bee subclasses call methods in HoneyVault.
- **Add two constants** to the static Constants class (`INITIAL_HONEY = 25f` and `INITIAL_NECTAR = 100f`) and use them to initialze the two private fields called `honey` and `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 **ConvertNectarToHoney method** converts nectar to honey. It takes a decimal 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 the remaining nectar.)
- 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. Then use the SemanticScreenReader.Default.Announce method to **announce the warnings to improve accessibility**.

**Add the HoneyManufacturer, NectarCollector, and BeeCare classes, then create the Queen class**

- The "Sharpen Your Pencil" solution has the code for the first three Bee subclasses. The Queen class is the other subclass of Bee. We showed you how the Queen class works (and gave you an AddWorker method that adds workers to an array of Bee objects). Also look carefully at the code-behind to see how it's used.
- 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 to it—you'll create the AssignBee method that instantiates Bee objects and calls it.
- There are two **private decimal 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.
- The queen starts off with three unassigned workers (so set unassignedWorkers to 3) and zero eggs. The Queen's **constructor** calls the AssignBee method three times to create three worker bees, one of each type.
- The **AssignBee method** takes a parameter with a job name (like `"Egg Care"`). It has `switch (job)` with cases that call the AddWorker method that we gave you when we described the queen class. For example, if `job` is `"Egg Care"` then it calls AddWorker(new EggCare(this)).
- Go back to the "Sharpen Your Pencil" solution and look at how the EggCare bees call the Queen's **ReportEggConversion method**. The method takes a decimal parameter called eggsToConvert. It checks if the eggs field is >= eggsToConvert—if it is, it subtracts eggsToConvert from eggs and adds it to unassignedWorkers.
- She overrides the **WorkTheNextShift 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. The loop keeps track of whether all of the workers did their jobs, and passes that to UpdateStatusReport.
- Look carefully at the shift reports in the screenshots. The private **UpdateStatusReport method** generates it (using HoneyVault.StatusReport). She calls UpdateShiftReport at the end of her WorkTheNextShift and AssignBee methods. UpdateShiftReport calls a **private WorkerStatus method** with a string parameter called job ("Nectar Collector") and returns a string with the number of workers doing that job ("3 Nectar Collector bees"). It also takes a bool parameter—if it's true, it adds `"WARNING: NOT ALL WORKERS DID THEIR JOBS"` to the end of the report.

If you get stuck, it's not cheating to look at our solution. That's a great way to learn, too!

## Exercise Solution

Here are the constants that were added to the static Constants class. They're used to initialize and reset HoneyVault.

```
static class Constants
{
    /// <summary>
    /// The amount of honey the hive starts with
    /// </summary>
    public const decimal INITIAL_HONEY = 25m;

    /// <summary>
    /// The amount of nectar the hive starts with
    /// </summary>
    public const decimal INITIAL_NECTAR = 100m;
```

Here's the Queen class, a Bee subclass that assigns workers, tells them to do their jobs, and makes status reports:

```
class Queen : Bee
{
    private Bee[] workers = new Bee[0];
    private decimal eggs = 0;
    private decimal unassignedWorkers = 3;

    public bool CanAssignWorkers { get { return unassignedWorkers >= 1; } }

    public string StatusReport { get; private set; }

    public override decimal CostPerShift {
        get { return Constants.QUEEN_COST_PER_SHIFT; } }

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

    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(true);
    }
```

The Queen's private workers is a Bee array that starts empty. It will get initialized with Bee objects in the constructor.

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

The AssignBee method takes a job string as a parameter. It uses that string with a switch statement to figure out what kind of worker to create, then it instantiates the right Bee subclass and calls the AddWorker method that we gave you to add it to the workers array.

**Take a few minutes and use the debugger to really understand how this works. Add a <u>breakpoint to each Bee subclass's WorkTheNextShift method</u> and step through them.**

## Exercise Solution

We gave you this AddWorker method. It resizes the array and adds a Bee to the end.

Try doing a global search and replace to change 'decimal' to 'float', then add a breakpoint to the first line of AddWorker. You'll see unassignedWorkers is sometimes equal to 0.999999... – this is just like the 0.30000000000000004 problem we talked about in Chapter 4. That's why we asked you to use decimal for number values.

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

    private void UpdateStatusReport(bool allWorkersDidTheirJobs)
    {
        StatusReport = $"Vault report:\n{HoneyVault.StatusReport}\n" +
        $"\nEgg count: {eggs:0.00}\nUnassigned workers: {unassignedWorkers:0.00}\n" +
        $"{WorkerStatus("Nectar Collector")}\n{WorkerStatus("Honey Manufacturer")}" +
        $"\n{WorkerStatus("Egg Care")}\nTOTAL WORKERS: {workers.Length}";

        if (!allWorkersDidTheirJobs)
            StatusReport += "\nWARNING: NOT ALL WORKERS DID THEIR JOBS";
    }
```

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

```
    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}";
    }
```

The instructions to add the UpdateStatusReport method didn't tell you to create this WorkerStatus method. We added it because we felt it made the code easier to unedstand. Your UpdateStatusReport might look really different from ours—and that's okay!

The "Sharpen" soliution showed you how the EggCare bees call the ReportEggConversion method to tell the queen how many eggs have been converted into unassigned workers.

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

    public override bool WorkTheNextShift()
    {
        eggs += Constants.EGGS_PER_SHIFT;
        bool allWorkersDidTheirJobs = true;
        foreach (Bee worker in workers)
        {
            if (!worker.WorkTheNextShift())
                allWorkersDidTheirJobs = false;
        }
        HoneyVault.ConsumeHoney(unassignedWorkers * Constants.HONEY_PER_UNASSIGNED_WORKER);
        UpdateStatusReport(allWorkersDidTheirJobs);
        return base.WorkTheNextShift();
    }
}
```

The constants used 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.

WorkTheNextShift tells each of the workers to do their jobs, then records how much honey gets consumed by the unassigned workers, and finally calls the UpdateStatusReport method.

## Exercise Solution

```
static class HoneyVault
{
    private static decimal honey = Constants.INITIAL_HONEY;
    private static decimal nectar = Constants.INITIAL_NECTAR;

    internal static void Reset()
    {
        honey = Constants.INITIAL_HONEY;
        nectar = Constants.INITIAL_NECTAR;
    }

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

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

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

    public static string StatusReport
    {
        get
        {
            string status = $"{honey:0.00} units of honey\n" +
                            $"{nectar:0.00} units of nectar";
            string warnings = "";
            if (honey < Constants.LOW_LEVEL_WARNING) warnings +=
                                "\nLOW HONEY - ADD A HONEY MANUFACTURER";

            if (nectar < Constants.LOW_LEVEL_WARNING) warnings +=
                                "\nLOW NECTAR - ADD A NECTAR COLLECTOR";

            SemanticScreenReader.Default.Announce(warnings);
            return status + warnings;
        }
    }
}
```

Here are the two constants that were added to the static Constants. They're used to initialze the private honey and nectar fields.

The ConsumeHoney method checks the amount parameter—if there's enough honey to do the job, it consumes it and returns true, otherwise it returns false.

The NectarCollector bees use the CollectNectar method to add nectar to the hive. The if check is important so the bees don't accidentally add negative values and reduce the nectar.

The HoneyManufacturer bees use the ConvertNectarToHoney method when they're making honey. It reduces the nectar and adds the honey—but if they try to convert too much, it will only convert as much as is left in the vault.

The StatusReport property generates the honey vault portion of the status report, which the Queen uses to generate the full status report. It also tells the screen reader to announce any warnings.

## Abstract properties work just like abstract methods

Let's go back to the Bee class from our earlier example. We already know that we don't want the class to be instantiated, so let's modify it to turn it into an abstract class. We can do that just by adding the `abstract` modifier to the class declaration:

```
abstract class Bee
{
   /* the rest of the class stays the same */
}
```

But there's one other virtual member—and it's not a method. It's the CostPerShift property, which the Bee.WorkTheNextShift method calls to figure out how much honey the bee will require this shift:

```
   public virtual decimal CostPerShift { get; }
```

We learned in Chapter 5 that properties are really just methods that are called like fields. Use the **abstract keyword to create an abstract property** just like you do with a method:

```
   public abstract decimal CostPerShift { get; }
```

Abstract properties can have a get accessor, a set accessor, or both get and set accessors. Setters and getters in abstract properties **can't have method bodies**. Their declarations look just like automatic properties—but they're not, because they don't have any implementation at all. Like abstract methods, abstract properties are placeholders for properties that must be implemented by any subclass that extends their class.

Here's the whole abstract Bee class, complete with abstract method and property (but not the comments):

```
abstract class Bee
{
    public abstract decimal CostPerShift { get; }

    public string Job { get; private set; }

    public Bee(string job)
    {
        Job = job;
    }

    public virtual bool WorkTheNextShift()
    {
        if (HoneyVault.ConsumeHoney(CostPerShift))
            return true;
        else
            return false;
    }
}
```

Replace this!

**Replace the Bee class** in your Beehive Management System app with this new abstract one. It will still work! But now if you try to instantiate the Bee class with `new Bee()`; you'll get a compiler error. Even more importantly, ***you'll get an error if you extend Bee but forget to implement CostPerShift***.