My Working Notes – Version 1.0 (24th September 2013)

Contents

[1 The Two Projects 1](#_Toc367862869)

[2 A Really, Really Important Thing About Where to Add Your Code 1](#_Toc367862870)

[3 The Classes in the (non-GUI) PigWorld Project 2](#_Toc367862871)

[3.1 Empty World 2](#_Toc367862872)

[3.2 Things in PigWorld 4](#_Toc367862873)

[4 The Classes in the PigWorldGUI Project 5](#_Toc367862874)

[4.1 The PigWorld Form and Program.cs 5](#_Toc367862875)

[4.2 View Classes for Objects that Need a Visual Image 6](#_Toc367862876)

[5 One Finer Point about Debugging 6](#_Toc367862877)

[6 Tasks 7](#_Toc367862878)

# The Two Projects

In the **Solution Explorer** window, the current Visual Studio solution contains two “projects” (or “folders”, if you like):

* The (non-GUI) **PigWorld** project contains the (incomplete) code that performs the *logical processing of the simulation*. The classes in this project know nothing about the GUI and you must keep them that way. In particular, these classes **do not** use any WinForms objects (i.e. do not reference the namespace System.Windows.Forms) and you must not change that. Similarly, these classes **do not** refer to any classes from the GUI project described below and you must not change that.
* The **PigWorldGui** project contains the (incomplete) code for *displaying the simulation on a GUI*. These classes **can use** any WinForms objects (from the namespace System.Windows.Forms). Similarly, these classes **can refer** to any classes in the non-GUI PigWorld project described above.

# A Really, Really Important Thing About Where to Add Your Code

A really important feature of the code in the **PigWorldGui** project is that it automatically displays what is going on in the non-GUI **PigWorld** project. E.g. if a pig moves from one square to another, then the pig object does so by using code in the non-GUI project, not code in the GUI project. I.e. the GUI displays what is going on inside the non-GUI PigWorld.

This is in keeping with the **Software Layers** approach illustrated on slide 43 of Lecture 9. The (commonly-called) **Business Layer** on that slide is the (non-GUI) **PigWorld** project in this program. And the **Presentation Layer** on that slide is the **PigWorldGui** project in this program.

So, when the user does something that must affect the non-GUI PigWorld – such as clicking the **Step** button – that event must be *initially handled by code in the GUI project*, but that event-handler *must call the relevant method(s) in the non-GUI code*, rather than trying to directly update what is on the screen. (As well as some examples in the provided code, see the examples of calls to the **Delete** and **SaySomething** methods in BasicPigWorld, in the Week 9 Prac.)

Of course, the above description isn’t true of every Control in the GUI. Some Controls only affect the GUI, such as the **Start** and **Stop** buttons and the **TrackBar**, which only affect the GUI’s **timer** object. And it’s not true of the **Quit** button, which simply closes the form. *But it’s true in all other cases.*

This means that, after you’ve created all the GUI items – buttons, etc. – and their event-handlers, then the changes you make to the supplied code will be in the non-GUI Project, rather than in the GUI project (unless directed otherwise in a specific part of this assignment). *A simple way to think of it is this*: update the code *in the GUI project* when you’re creating GUI items and their event-handlers; and update the code *in the non-GUI project* at all other times.

As stated above, the GUI code follows along automatically, displaying what is going on in the non-GUI PigWorld. How does it do this? It uses a number of **delegates** (see Lecture 9). Delegates are powerful but can be tricky to learn how to use. Fortunately, all of the code that uses delegates has already been implemented. You can look at that code to see how it works, but you don’t have to. You can complete this assignment either way.

# The Classes in the (non-GUI) PigWorld Project

1. Rather than going though the classes in alphabetical order – the way they appear in Visual Studio’s **Solution Explorer** – it is easier to understand what each one does by looking at them in more logical groups.
2. Note that, in most cases, you will find a longer description of each class in its class header (the comment near the start of the class’s file).
3. Also note that classes that are stated to be **complete** are ones that *need no further work*. I.e. you can use these classes but *you should not change them*.

## Empty World

1. We start with the classes that are needed to provide what we can regard as an “empty world”, i.e. a world which has cells, walls, etc., but no things inside that world, such as pigs, trees, pig-food, etc.
2. **PigWorld class** – **PigWorld.cs**
3. This class hold information about the world as a whole, as well as containing all the Cells and Gaps that are part of the world. ***This class is complete.***
4. **Cell class** – **Cell.cs**
5. This class hold information about each one of the cells in the world, such as its position in the world, and whether any object(s) is(are) currently in that cell. ***This class is complete.***
6. **Air class** – **Air.cs**
7. Air is used to transmit (virtual) sound from Cell to Cell. This affects the mating behaviour of pigs. (Air has no other use in PigWorld.) ***This class is NOT complete.***
8. **Gap class** – **Gap.cs**
9. Between each pair of Cells is a Gap. Gaps can be filled with walls which are used to block the pathways from one Cell to the next. ***This class is complete.***
10. *Directions and Positions*
11. **Direction class** – **Direction.cs**
12. A Direction object represents a direction in the 2-dimensional space of the world. Directions are mainly used by Animals as they move around the world. The angle of a Direction is measured using the same convention as a magnetic compass. North corresponds to zero degrees, and the angles increase clockwise, so east is 90 degrees, south is 180, and west is 270. Other angles are possible. ***This class is complete.***
13. **Position class** – **Position.cs**
14. This class is simply used to hold a row and column combination. Using this class makes other code clearer and more concise. E.g. compared to having to pass separate row and column values as parameters, you can use one Position parameter rather than two values. Positions are used in many places, in PigWorld. ***This class is complete.***
15. *Radars and Echos*
16. **Radar class** – **Radar.cs**
17. Radars may be used by any type of LifeForm. Radars are used to find out what objects of a certain type (e.g. pig-food) are in the environment. Radars see through walls. ***This class is complete.***
18. **Echo class** – **Echo.cs**
19. An Echo object holds information returned from methods of the Radar class. An echo contains information about the Thing detected, including its type, its direction, and its distance. ***This class is complete.***
20. *Utilities*
21. **Util class** – **Util.cs**
22. This class contains various utility methods that may be useful to other classes. ***This class is complete.***

## Things in PigWorld

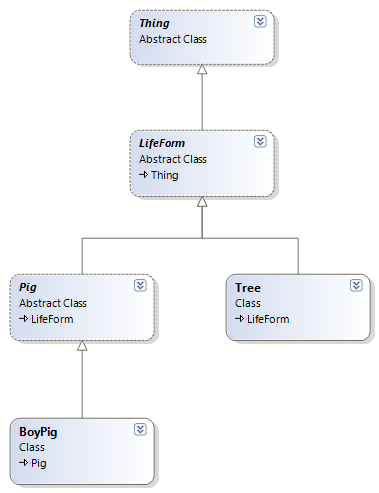
1. In the Week 9 Prac, you worked with a cutdown version of PigWorld, named BasicPigWorld. Figure 1 shows the inheritance hierarchy of the life-forms in BasicPigWorld.
2. Note: the following two figures were created by Visual Studio, which has limited formatting options. If you find them difficult to read, you may find it helpful to enlarge them either within this Word document, or by copying them into another program such as Windows Paint, where you can zoom in.
3. **

Figure 1: Things in BasicPigWorld

1. In the full version of PigWorld, we still have the five classes shown in **Figure 1**, but as **Figure 2** shows, we need extra classes in the inheritance hierarchy because we have extra kinds of objects. Although it’s a bit more complex, **Figure 2** matches what you know about objects in the real world. E.g.

* **Things** can be divided into classes (“groups”) of **LifeForms** and **NonLivingThings**.
* **LifeForms** can be divided into **Animals** and **Plants**.
* **Animals** can be divided into **Pigs** and **Wolves**, and so on.

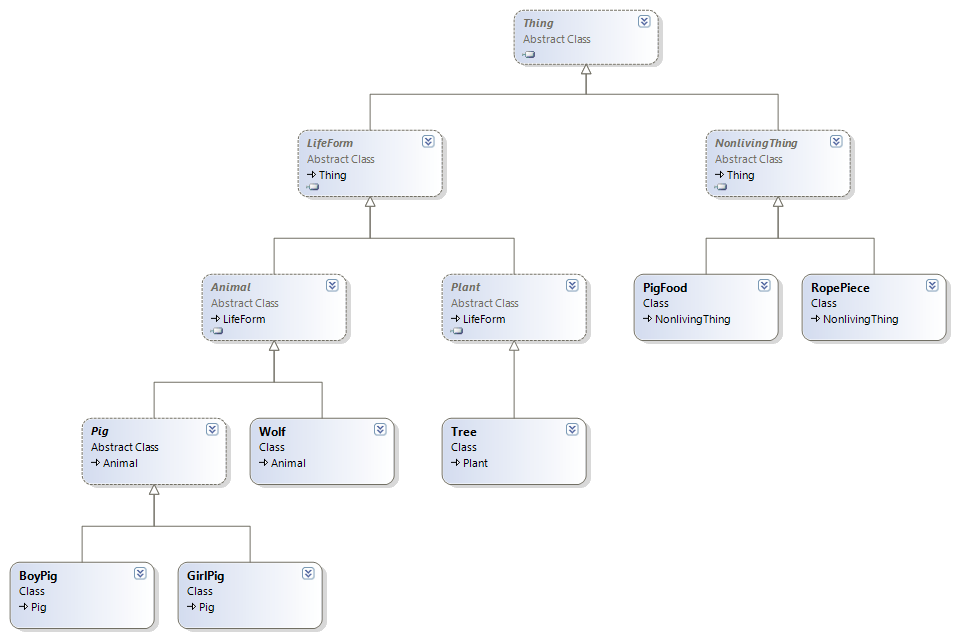
1. By the way, some of the files in BasicPigWorld omit some of the code that is needed in the assignment – so don’t try copying entire files from BasicPigWorld into your assignment. You’ll end up with a lot of problems if you do.
2. ****

Figure 2: Things in the full PigWorld

1. The purpose of each of the classes in **Figure 2** should be obvious from its name, so a further description is not given here. As stated above, in most cases you will find a longer description of each class in its class header (the comment near the start of the class’s file), if the above figure is not enough.
2. ***Most of the classes shown in Figure 2 are complete, but the following classes are NOT: Animal, Pig, GirlPig, Tree, and Wolf. (See* Tasks*, later.)***

# The Classes in the PigWorldGUI Project

## The PigWorld Form and Program.cs

1. **PigWorldForm class – PigWorldForm.cs**
2. This class is the overall form for PigWorld, containing the user controls, etc., as shown in the specification. I have started developing this form but haven’t got far. You can run this form in the code I’ve left behind, but it doesn’t do much yet. ***This class is NOT complete.***
3. A few comments about using a **TableLayoutPanel** for the PigWorld’s squares/cells. At first, I created a **TableLayoutPanel** inside this form, while using **Design View**. But later on I removed it, because I found that I needed to add extra functionality to the **TableLayoutPanel** and the best way to do that was by making the **PigWorldView** class to be a subtype (or “subclass”) of the **TableLayoutPanel** class. Anyway, subtypes of GUI controls don’t automatically appear in **Design View’s** **Toolbox**, so you can’t simple add them to a form. Instead, I create a **PigWorldView** object in the code of **PigWorldForm**, and set the **TableLayoutPanel’s** properties in **PigWorldView’s** **InitialiseTableLayoutPanel()** method. That method sets properties such as the background colour, which otherwise would have been set by using **Design View**.
4. So while you won’t add a **TableLayoutPanel** to the form in **Design View**, you’ll need to use **Design View** to add the other controls (except where they already exist), and then create event-handlers for those controls. Those event-handlers will be added into **PigWorldForm.cs**, so it’s obviously not complete yet. (Event-handlers for the context menus are added to different files.) But don’t try to add much else to this file – a lot of the functionality must be implemented in other (existing) classes, as stated in **Section 2** above. See the **Tasks** section (later) for further information about how some of those event-handlers should be implemented.
5. **Program class – Program.cs**
6. This class contains a simple **Main** method which has been generated automatically by Visual Studio. ***This class is complete.***

## View Classes for Objects that Need a Visual Image

1. Many, but not all, of the non-Gui classes (in the **PigWorld** project) need a matching GUI class to display them on the screen. All of these classes names end with the word “View”, which tells you how the classes are related, e.g. the GUI class **PigWorldView** displays information obtained from its non-GUI counterpart class **PigWorld**.
2. There are GUI classes for some of the **Empty World** classes described above: **PigWorldView**, **CellView**, and **GapView** (not every class needs a view). And there are GUI classes for most of the classes in **Figure 2**: **ThingView**, **LifeFormView**, **AnimalView**, **PigView**, **WolfView**, **PlantView**, **TreeView**, **NonlivingThingView, PigFoodView**, and **RopePieceView**. ***Most of these classes are complete, but NOT: CellView, PigView, WolfView, and TreeView. (See* Tasks*, later.)***

# One Finer Point about Debugging

Using breakpoints (followed by stepping through lines of code in the VS debugger) is a good way to see what the program code is doing, especially when it’s not doing what you think it should be doing.

But you need to be careful when you have multiple objects of the same type on the screen at the same time – e.g. several pigs – because they are using the same code, in Pig.cs for example. So when a breakpoint is hit, it can be confusing if you don’t know which pig is involved.

An obvious solution is to have *only one* pig on the screen, when you are debugging. This is a good approach in some circumstances, e.g. when you’re working on the code for the pig finding some food.

But that approach won’t work when you need both a girl and a boy pig on the screen at the same time. In such cases, it can be helpful to have a breakpoint that occurs for only one of the pigs. Remember that each life-form has an Id, and you can use its “Show Id” context menu item to display this. Once you know what that value is, you can then stop your program and add code such as the following example, wherever you’d like to have it. The “int dummy = 0;” line is included simply so that you have an easy place to set a breakpoint. That line has no other purpose.

if (Id == 3) {

int dummy = 0;

}

# Tasks

This sections describes each of the tasks that you need to do, to finish the development of PigWorld.

Note.

1. Don’t try to read this entire Tasks section in one go – that’s not what it’s here for. There’s a fair bit of information on the following pages, but you won’t need to read the details of each task until you’re actually working on that task. But if you want get to a quick overview of what all the tasks are, just skim through the rest of this document, looking at the task names, e.g. **Task 1** has the name **Creating the User Controls in the GUI**.
2. The following tasks are *not* of equally difficulty. The earlier tasks are easier than the later ones, as a general rule, but that isn’t true for every task. (The number of words describing a task is *not* an indication of its level of difficulty.)
3. Different tasks may be given different assessment weightings, to reflect differences in effort and/or skill required to complete them.
4. Many tasks require earlier tasks to be completed first (or at least seriously attempted first), because the tasks build on the work done in those earlier tasks. But you don’t have to do them *all* in the strict sequence from first to last. So that you know what your options are, each task has a comment (in parentheses) at the start, specifying which tasks must be completed (or seriously attempted), before you start that task.
5. In the descriptions of some of the tasks, upper limits are given for the number of lines of code you should write – e.g. “Don’t write more than 20 lines.” These are *upper limits* so that you don’t write lots of unnecessary code, rather than saying precisely how much code should be written. E.g. if the upper limit is 20 lines, then you may be able to write a correct answer in fewer lines, such as 5 or 10 or 15. Such line number counts do not include comment-lines or blank lines.
6. **Creating the User Controls in the GUI**

(Start here.)

Create all the user controls in the **PigWorldForm**, where these are not already part of the supplied program code. At this stage, there’s no need to create any event-handlers – although you can create empty ones to remind you of what needs to be done later, if you like.

Do not be concerned about the context-menus at this stage. Those will come later.

1. **Adding Objects to the PigWorld Squares/Cells**

(Task 1 must be completed/attempted first.)

Create event-handlers so that the user can add objects to the PigWorld squares/cells, after selecting any of the objects selected by the radio-buttons at the bottom of the form. Hint: see step 3 in the **BasicPigWorld** activity of the Week 9 Prac, and the given code in **BasicPigWorldForm.cs**. When using the code given in step 3, note that there are two differences between **BasicPigWorld** and the larger **PigWorld** being used in this assignment: (i) “worldView” is now “pigWorldView”; and (ii) “cell.World” is now “cell.PigWorld”.

**Reminder:** Do not be concerned about the context-menus at this stage, even though they were also part of the **BasicPigWorld** activity. Those will come later.

1. **Setup Demo Number 3**

(Can be done at any time.)

Based on the given code for the **setupDemoButton\_Click** method in **PigWorldForm.cs**, and the related code in **PigWorldView.cs**, create a new **SetupDemo3** method which constructs a set of objects in PigWorld, in a similar way to the code in the given **SetupDemo1** and **SetupDemo2** methods. In your new method, you must:

* create at least six objects in the PigWorld squares/cells, with at least one pig; and
* create your own set of all least six walls, *instead of* calling the given CreateDemoWalls method.

1. **“Remove Walls” and “Remove All” Buttons**

(Task 2 must be completed/attempted first, but Task 3 need not be.)

Create event-handlers so that these two buttons work. Each of those event-handlers will be in **PigWorldForm.cs**, of course, but you need to make them affect the (non-GUI) **PigWorld** object. Getting from the *form* to that *object* can be a little obscure when you first do it, so here are some hints:

* The **PigWorldForm** contains the (GUI) **PigWorldView** object.
* That **PigWorldView** object contains the (non-GUI) **PigWorld** object.
* So, a **PigWorldForm** event-handler can call any method in the **PigWorld** class, by using code like this:

pigWorldView.PigWorld.XXX();

where XXX is any (public) method in the **PigWorld** class.

* So all you need in each of these two event-handlers is a one-line call to the appropriate method in the **PigWorld** class. If you look through the names of the methods in that class, you should find that it’s not hard to find the appropriate method, in each case.

1. **Step Button**

(Task 4 must be completed/attempted first.)

Create an event-handler so that the **Step** button works. As in Task 4, this event-handler needs to affect the (non-GUI) **PigWorld** object. So you need to use a similar approach here.

Note that you won’t see movement from any pigs or wolves, when you test your **Step** button. That’s because the code for movement is developed in later tasks. But, you should see that the Age of any pigs or wolves – displayed at the bottom of each animal’s image – increases each time that you click your **Step** button.

1. **Start and Stop Buttons**

( must be completed/attempted first.)

Create a timer with a default interval of 1000 milliseconds. Then create event-handlers for the **Start** and **Stop** buttons, the timer’s **Tick** event, and the **Faster/Slower** TrackBar’s Scroll event. These should all work in the same way as the similar items that you created in the Week 9 Prac.

1. ***Radars, Echoes, and Directions***

(This material does not describe a task. It is information to help you understand the tasks that follow.)

In PigWorld, Pigs and other LifeForms have poor eyesight – they can’t simply see what’s in the entire world, unlike what we humans can see in the GUI. So they use radars to sense their surroundings. Radars are carried by their owner, so they are always in the same cell/square as the owner. Radars are penetrating – they see through walls.

Open the **Radar** class (**Radar.cs**) and have a look at its constructor. The first parameter is the owner of the Radar object. The second parameter is the type of object that we want to radar to detect. E.g., if we want a radar object that detects pig-food, we would specify **typeof(PigFood)** for this parameter. (By the way, a pig’s radar can detect other pigs if we specify **typeof(Pig)** as the second parameter.)

Radars are used to perform sweeps of PigWorld. A sweep begins in the direction NORTH, and proceeds clockwise until it returns to NORTH. In the Radar class, the **Ping** method is used to retrieve each echo from the radar. I.e. the **Ping** method returns an object of the class **Echo**.

Each **Echo** object contains information about the type of thing detected, its direction, and its distance. Read the code provided for that class. It is not a big class, less than 20 lines when you don’t count the comments and blank lines.

The **direction** inside each Echo represents a direction in the 2-dimensional space of the world. Directions are mainly used by Animals as they move around the world. The angle of a Direction is measured using the same convention as a magnetic compass. North corresponds to zero degrees, and the angles increase clockwise, so east is 90 degrees, south is 180, and west is 270. Other angles are possible.

***FindNearest***

Pigs and other LifeForms are often interested in finding an object of a particular type that is nearest to them. E.g. a pig often wants to find the nearest pig-food. A Radar object helps to do this, but its Ping method is a bit too primitive. That method tells us about objects in the PigWorld, but not which one is nearest.

To make this easier, the **LifeForm** class has a **FindNearest** method. All life-forms (pigs, trees, and wolves) can use it, as you will see in the following tasks.

Open the **LifeForm** class (**LifeForm.cs**) and have a look at this small method so you can see how it uses a Radar. In other words, all life-forms that use this **FindNearest** method are actually using a Radar.

**T**he **FindNearest** method returns an **Echo** object, as long as there is at least target object in the world, e.g. at least one pig-food object somewhere. If there aren’t any such objects in the whole world, this method returns null. *To avoid your program sometimes crashing with a* ***NullReferenceException****, your code should check for a null* ***Echo*** *being returned, after you call* ***FindNearest****.*

1. **Tree Behaviour**

( must be completed/attempted first, but Task 6 need not be.)

Trees have simple behaviour, much simpler than Pigs. Trees simply grow pig-food.

To allow this to happen, each **LifeForm** (including trees) has a **DoSomething** method which is invoked for each “step” that the simulated world takes. This is the *only* method in the **Tree** class (**Tree.cs**) that you need to complete. The given code is:

protected override void DoSomething() {

// This will detect any type of pig (both BoyPigs and GirlPigs).

Echo echo = FindNearest(typeof(Pig));

// Write your code here.

}

Every 10th time that this method is called, the Tree should attempt to deposit an item of **PigFood** onto an adjacent cell, as long as there are no pigs on *any* cell adjacent to the tree. If there are any pigs near to the tree, then it does nothing. I.e. nothing can happen until the next 10th time that this method is called. You are to write code that produces this behaviour. Note:

* See the earlier description of the **FindNearest** method.
* The Echo object has a **distance** property. If that distance is less than or equal to one, then there is a pig nearby.
* To create pig-food, call the given **DropFood** method.
* If you need to add any class variables or constants to this class, go ahead.

Hint: while your final code must drop food only every 10th time, you might like to make that happen more frequently, when you are running your program to see that your code works correctly.

The amount of code needed is quite small. E.g. if you write more than 10 lines (not counting comments and blank lines), you’re making it too complicated.

The *second part* of this task is to create the tree’s context-menu – as shown in **Figure 2** in the main assignment document. Create the context-menu in a similar way to how you created the context-menus for **TreeView** and **PigView** in **BasicPigWorld**, in the Week 9 Prac. Remember: the context-menu is part of the GUI, so you’ll be modifying **TreeView.cs** – not **Tree.cs**.

Note that the **Drop food** menu-item must work *unconditionally*, i.e. must call the tree’s **DropFood** method, not its **DoSomething** method.

Hint: for the **Show Id** menu-item, you can obtain the tree’s internal Id with this kind of code:

int id = tree.Id;

1. **Wolf Behaviour**

(Task 5 must be completed/attempted first, but Task 6 and Task 7 need not be.)

A Wolf is a type of Animal that eats Pigs, and does little else. Wolves are LifeForms (like Trees, above) so they also have a **DoSomething** method which is invoked for each “step” that the simulated world takes. This is the *only* method in the **Wolf** class (**Wolf.cs**) that you need to complete. The given code is:

protected override void DoSomething() {

// This will detect any type of pig (both BoyPigs and GirlPigs).

Echo echo = FindNearest(typeof(Pig));

// Write your code here.

}

Here’s how a wolf is supposed to behave. First the Wolf looks around for the nearest pig, using the method **FindNearest**. If there isn't a pig, the Wolf does nothing. If there is, then the Wolf either:

1. tries to moves one cell/square in the direction of the nearest pig, or
2. if a pig is in fact only one move away, the wolf eats the pig.

Note that if there is a wall between the Wolf and the nearest pig, then the Wolf won't move at all.

You are to complete the skeleton code given below, to produce this behaviour. Note:

* See the earlier description of the FindNearest method.
* Unlike a tree in Task 7, a wolf doesn’t care about the distance to the nearest pig. Instead, a wolf cares about the direction towards the nearest pig. Fortunately, the echo object has both these properties.
* The methods for moving a pig are not part of the Wolf class directly, but are *inherited* from the class Animal. That class has more than one method named Move. The one that has a targetCell parameter is best for a Wolf.

// If there are no pigs in the whole world, do nothing more.

if ...

// Otherwise, the Wolf must try to move in the direction given by the echo,

// to fulfil the behaviour described in items (1) and (2) above.

//

// The Wolf's current cell/square is specified by its Cell property.

// And the Cell class has a GetAdjacentCell method which can be used to get

// a reference to a neighbouring Cell (if any) in the specified direction.

Cell targetCell = ...

// If the targetCell is null, then there is a wall in the way, so do nothing more.

if ...

// Otherwise, see if there is a LifeForm in that targetCell,

// by setting adjacentLifeForm equal to the targetCell's LifeFormOccupant property.

// If there's no such LifeForm, then this will be null.

LifeForm adjacentLifeForm = ...

// If there's no such LifeForm, then move to that targetCell,

// using the Move method -- a method that is inherited from the Animal class,

// i.e. all animals can move.

if ...

// Else, if the adjacentLifeForm is a pig, (Hint: use an "is" test -- see Lecture 9),

// then (logically) the wolf moves to that targetCell and then eats the pig.

// But because a cell can contain ONLY ONE LifeForm at a time, this has to be

// written as program code that eats the pig first, and then moves the wolf into

// the pig's cell/square.

// All animals can eat by using the Eat method -- a method that is also inherited

// from the Animal class.

else if ...

} else {

// It's another type of LifeForm, such as a Tree or another wolf, so do nothing.

}

The amount of code needed is fairly small. If your final method contains more than 20 lines – apart from comment lines, blank lines, and lines with a single closing curly-bracket (“}”) – then you’re making it too complicated.

The *second part* of this task is to create the wolf’s context-menu – as shown in **Figure 2** in the main assignment document. Use a similar approach to that described in the second part of Task 7. Remember: the context-menu is part of the GUI, so you’ll be modifying **WolfView.cs** – not **Wolf.cs**.

1. **Pig Behaviour – Overview and RunFromWolf**

(Task 5 must be completed/attempted first, but Task 6 need not be. While Task 7 and Task 8 don’t have to be done first, it is recommended that you do at least one, because those tasks provide a simpler introduction to the **DoSomething** method.)

Pigs are the most complex objects in PigWorld, in terms of their behaviours.

Open the classes **BoyPig** (**BoyPig.cs**), **GirlPig** (**GirlPig.cs**), and **Pig** (**Pig.cs**). See that the **BoyPig** and **GirlPig** classes are both based on (or “derived from”) the **Pig** class.

Note that the **BoyPig** and **GirlPig** have a small number of lines of code. These two types of pigs have very similar behaviour, so most of the code that determines that behaviour is in the **Pig** class, so it can be shared by **BoyPigs** and **GirlPigs**. For that reason, many of the following tasks will require you to modify the **Pig** class, rather than the **BoyPig** or **GirlPig** classes. So you can close **BoyPig.cs** and **GirlPig.cs** for now.

The Pig's **DoSomething** method controls all pig behaviour in overall terms, so have a look at that method – the code is quite easy to read. Unlike the tasks above, this method is already fully provided in the given code, so you don’t have to write that method. In fact, you’re not allowed to modify the code in this method. Instead, you’ll be writing code in some of the lower-level methods, called by this **DoSomething** method.

All of the code for the various tests (**IsTired()**, etc.) is already written. Most of that code is fairly small and easy to understand. If you want to look at it, note that these parts are in the **Animal** class (**Animal.cs**) rather than the **Pig** class (which is derived from the **Animal** class): **IsTired()**, **IsHungry** and **IsInTheMoodForLove()**. Likewise, the **Rest()** method is in the **Animal** class.

So there are four main things that a pig can do:

* **Rest()** – this method is given, so you don’t have to write it.
* **RunFromWolf()** – a method for you to write, in this task.
* **LookForFood()** – a method for you to write, in a later task.
* **LookForPig()** – this method is given, so you don’t have to write it, but you will have to write some related code, in later tasks.

The **RunFromWolf()** method makes the pig (try to) run in the opposite direction of the nearest wolf. Write code to complete the skeleton method in the given code. Note:

* Once you have determined the direction of the nearest wolf, use you’ll need to work-out what is the opposite direction. Rather than doing a lot of maths, see if you can find a suitable method in the **Direction** class.
* The methods for moving a pig are not part of the **Pig** class directly, but are *inherited* from the class **Animal**. That class has more than one method named **Move**. Unlike the preceding task, the **Move** method that has a **Direction** parameter is best for a Pig, in this task.
* Each **Move** method returns a Boolean value that tells you whether the Move was successful or not. It might fail for various reasons, such as there being a wall in the way. If it fails, then call the supplied **Panic** method, which moves the pig in a random direction (perhaps even *towards* the wolf).

By the way, the **Move** method described above can be used with any **Direction**, not just the “eight named directions” given at the start of the **Direction** class. The **Move** method moves a pig to the adjacent square that is ***closest*** to the angle specified in the **Direction**. This should be the way that you most commonly move a pig in your code, using **Move** and specifying a **Direction** object as the parameter.

1. **The Pig’s Context-Menu**

( must be completed/attempted first, to give you an introduction to the **Pig** class.)

This task is to create the pig’s context-menu – as shown in **Figure 2** in the main assignment document. Use a similar approach to that described in the second part of Task 7. Remember: the context-menu is part of the GUI, so you’ll be modifying **PigView.cs** – not **Pig.cs** – except where explicitly directed otherwise, in the following.

Only one or two lines of code are needed inside each event-handler method, for each of the context-menu items. In each case, the event-handler method affects the pig directly by calling specific methods, rather than calling its **DoSomething** method (or by calling any of the methods called directly by that method).

Because these menu-items act directly on a pig, they may be useful to you while testing your code for later tasks.

Here are some hints for each of the menu-items:

* **Sleep** – A pig only sleeps when it is tired, so use its **IncreaseTiredness** method (inherited from the **Animal** class). That method has a parameter that determines how long a pig sleeps for – use the value 100 when you call that method. When you test you code, note that a sleeping pig displays “Zzz” on its image. (That display is part of the given code.)
* **Wake Up** – This wakes a pig by setting its tiredness level to zero. There is an existing method (inherited from the **Animal** class) that you must use for this.
* **Eat** – This increases a pig’s energy-level to the same extent as if the pig has just eaten a piece of pig-food. (No pig-food is actually consumed.) Use the pig’s **IncreaseEnergy** method (inherited from the **Animal** class), with PigFood.ENERGY\_IN\_PIG\_FOOD as the parameter value.
* **Put in Mood for Love** – See below.
* **Clear Rope** – See below.
* **Show Id** – Implement as in Task 7 and Task 8.
* **Delete** – Implement as in Task 7 and Task 8.

The **Put in Mood for Love** menu-item is for increasing a pig’s energy-level to a level where the pig will be in the mood for love, but there isn’t a method for doing this in the code you’ve been given. So, the first step is to add a method, called **PutInTheMoodForLove** to the **Animal** class (**Animal.cs**). The body of this new method should contain one line of code:

IncreaseEnergy(STOMACH\_FULL\_LEVEL);

Add the following XML comment before this new method’s header:

/// <summary>

/// Put this Animal in the mood for love, by setting its energy level to

/// STOMACH\_FULL\_LEVEL.

/// </summary>

To understand *why* this puts the pig in the mood for love, look at the code inside that **IncreaseEnergy** method, and in the **IsInTheMoodForLove** method. (Both are small methods in the **Animal** class.)

Once you’ve created that method, you can then call it from the event-handler for **Put in Mood for Love**. When you test you code, note that when a pig is in love it displays little hearts above its head. (That display is part of the given code.)

The **Clear Rope** menu-item is for deleting any rope that the pig might be trailing. There is an *empty* method for doing this in the code of **Pig.cs** that you’ve been given. You should call that method from the event-handler for **Clear Rope**. Although it won’t do anything yet, the pieces are now in place for a later task.

1. **Pig Behaviour – LookForFood – Basic**

(Both Task 8 and must be completed/attempted first. Task 8: so you understand how animals move and eat. : so you understand how the **LookForFood** method works as part of the Pig’s **DoSomething** method.)

In this task, you are to give the pigs a basic way to find and eat food, by adding code to the skeleton **LookForFood** method. Here’s the behaviour that you are to implement.

First the pig looks to see if there is any PigFood in its current Cell.

If there is, the pig picks it up, eats it, and does nothing more. I.e. the pig stays where it is.

Otherwise, the pig looks for the nearest PigFood, using the FindNearest method.

If there isn't any food anywhere in the world, the pig does nothing.

If there is food, then the pig (tries to) move one square in the direction of that food. If the move fails (e.g. because there's a wall, or something else, in the way), then the pig simply remains where it is.

1. The earlier tasks have shown you how to do things in PigWorld that are useful here, such as how an animal moves and eats, and how the **FindNearest** method is used. So you should be able to write code for most of this behaviour already. But here are a few notes on some things that are new or different:

* When a wolf eats a pig (in Task 8), the wolf can’t be in the same cell/square as the pig, at the same time, because a cell can contain *only one life-form* at a time. That’s not the case for nonliving-things such as pig-food. A pig *can be in the same cell* as pig-food, and *has to be*, before it can eat the pig-food.
* A pig can find out whether there is any pig-food in its Cell by using the Cell’s **Exists** method. See that method’s comment for information on how to use it.
* Before a pig can eat a piece of pig-food, the pig has to pick up the food, using the Cell’s **Pickup** method. See that method’s comment for information on how to use it.

1. Your solution should require less than 20 lines (not counting comments and blank lines). This information is given to stop you from doing something crazy and wasting time. If you get to 20 lines of code, you are probably doing something crazy.
2. When you run your program, you may find that the pig(s) get stuck in one place, because they are trying to get to food that is behind a wall. If that happens, try removing the wall (or all walls), so that you can see if the pig(s) get the food. In a later task, a different approach will be used so that pigs get stuck less often.
3. ***The Mating Habits of Pigs***

(This material does not describe a task. It is information to help you understand the tasks that follow.)

When a pig’s energy level has risen sufficiently, the pig temporarily loses interest in food. Instead, the pig becomes ”in the mood for love”. (*We can also place a pig in such a mood by using its context-menu*.) When a pig is in such a mood, little hearts appear on the pig’s image, and the pig’s tight lips give way to a smile.

While in the mood for love, a *girl pig* remains stationary, and emits a (virtual) noise. This noise is not audible to humans, but the word “oink” is displayed on the girl pig’s image (by the given code). This “oink” propagates throughout PigWorld, with the volume of the oink decreasing with distance from the girl pig.

The volume in each square can be seen by ticking the box **Show Debug Info** in the lower right of the window. In the skeleton code, propagation of sound is not implemented, so each unoccupied square will show a zero (near the bottom). One of your tasks is to add the propagation of sound.

Any *boy pig* also in the mood for love will move from his current square to the adjacent square where the “oink” is loudest. By repeatedly moving to the square with the largest volume, the boy pig will usually arrive next to the girl pig emitting the grunts. (Sometimes, however, a girl pig will lose interest and move off before a boy pig arrives.)

If a girl pig and a boy pig, both in the mood for love, should occupy adjacent squares/cells, they may mate to produce a new pig. However, the conditions have to be just right, and they don’t always mate. Adjacent pigs of the opposite sex will not produce a baby if one or the other is (1) too tired, (2) not in the mood for love, (3) they are brother-and-sister, or (4) they are parent-and-child.

If two pigs do mate, their progeny will appear on a vacant cell/square adjacent to the parents. If no adjacent square is vacant, then no new pig will be born. Also, after mating, the two parents enjoy a short nap. When sleeping, a pig displays “Zzz”.

1. **Pig Behaviour – GirlPig’s LookForPig**

(Task 9 must be completed/attempted first, so you understand how the **LookForPig** method works as part of the Pig’s **DoSomething** method.)

As you have seen in Task 9, girl pigs and boy pigs often have the same behaviour – that’s why the code is implemented in the base class **Pig.cs**. But that’s not the case when it comes to looking for love. Each of the two subclasses, **BoyPig** and **GirlPig**, has its own separate **LookForPig** method – open the **BoyPig.cs** and **GirlPig.cs** files and see. One of the virtues of OOP (Object Oriented Programming) is that the correct method is automatically called, so we don’t have to write extra code that says: “If it’s a boy pig, call this method. If it’s a girl pig, call that method.”

In this task, the focus is on the **GirlPig’s LookForPig** method. That method (in **GirlPig.cs**) is already written in the given code, but it calls a method that is incomplete – the **TransmitSound** method of the **Air** class.

To carry out this task, you will need to add code to the body of that **TransmitSound** method. Some code is already there. You are to add your code *after* that code. To propagate sound, your code will transmit sound to the **Air** objects that are within each of the 8 adjacent cells/squares (where they exist[[1]](#footnote-1)).

All of your code *must* be inside a *single loop* which cycles through the 8 different directions. (Hint: look at the XML comments describing the **GetAdjacentCellDirection** method in the **Direction** class.) Each time around the loop, your code should:

1. get a reference to the adjacent cell in that direction, using the **GetAdjacentCell** method of class **Cell**, and if there is not a wall intervening in that direction,
2. get a reference to the **Air** object in that adjacent cell, then
3. use that Air object to call its **TransmitSound** method, remembering that the volume of sound decreases by 1, as it propagates from one cell to the next.

Your code will be less than 15 lines (not counting comments and blank lines). If you find yourself writing more than 15 lines, you are probably doing something wrong.

**Reminder:** so you can check that your sound propagation code is working, click on the **Show Debug Info** checkbox to make visible the sound-level numbers in each cell.

Note that boy pigs won’t respond properly to the sounds yet. That’s the next task.

1. **Pig Behaviour – BoyPig’s LookForPig**

(Task 12 must be completed/attempted first.)

As mentioned above, when a boy pig is in the mood for love, he will move from his current square to the adjacent square with the highest volume “oink”.

In this task, the focus is on the **BoyPig’s LookForPig** method. That method (in Boy**Pig.cs**) is already written in the given code, but it calls a method that is incomplete – the **Listen** method of the **Animal** class. (This method is part of the **Animal** class because any types of animals can listen. But in the current PigWorld, only boy pigs do.)

To carry out this task, you will need to add code to the body of that **Listen** method. Your code will have to examine the **SoundLevel** in the **Air** objects that are within each of the 8 adjacent cells/squares (where they exist).

The given code in method **Listen** contains a “dummy” line:

return null; // Students should delete this "dummy" return.

While this dummy line does not do what we want the method to do, it serves the purpose of at least making the method legal so it will compile. (This method must return something, so the dummy line does that.) As the comment on that line says, you should delete that line of code, and write your own code that implements the correct functionality.

See the method comments for the **Listen** method (in **Animal.cs**) for a description of what this method has to do. Hint: because your code has to examine each of the 8 adjacent cells, there should be some similarity between your code here and the code you wrote for **TransmitSound** in the preceding task. But there are also some important differences, e.g. you don’t listen to adjacent cells whereas you do transmit sound to them.

Your code will be 20 lines or less (not counting comments and blank lines). If you find yourself writing more than 20 lines, you are probably doing something wrong.

Once you’ve implemented this method correctly, you should find that boy pigs are much better at finding girl pigs (when both are in love).

1. **Pig Behaviour – Mating – Real Audio**

(Task 13 must be completed/attempted first.)

The preceding tasks used the virtual sounds that girl pigs make. This task requires you to add some real sounds to PigWorld. While this adds another dimension to PigWorld, it doesn’t affect how the pigs behave.

The *first part* of this task is to create an event-handler for the **Enable Real Audio** checkbox, which controls whether audio files are played at certain key moments during the simulation. Hearing that audio can enhance the simulation, but not if you are using your computer’s speakers while sharing a room with other people. Hence, real audio is an option for the user.

All that event-handler needs to do is to set **ShowDebugInfo** – a Boolean property of the whole **PigWorld** (**PigWorld.cs**) – to true or false, according to whether the checkbox is checked or not.

The *second part* is to play some sounds at appropriate moments. The Pig class contains two given methods that assist with this: **Grunt** and **Shriek**.

**Grunt** is to be called whenever a girl pig starts grunting, so modify the **GirlPig’s LookForPig** method to do that.

**Shriek** is to be called whenever a new pig is born, so modify the **GirlPig’s TryToMakeBaby** method to do that.

When you test your code, you’ll need speakers or headphones to see if your code works, of course. You won’t see any difference on the screen.

1. ***Pigs and Ropes***

(This material does not describe a task. It is information to help you understand the tasks that follow.)

As you may have seen in Task 11, the basic **LookForFood** method doesn’t work very well when there is a wall between the pig and the nearest bit of food – the pig simply sits there, trying to move towards the food, but blocked by the wall. While we could change the code so that the pig moves somewhere else (at random), that wouldn’t improve things much. What is needed is a way for the pig to explore more of its world, so that it can (try to) find a path to the food.

As you know, pigs have poor eyesight so they can’t simply look at the entire world and work out what is the shortest path to food, with walls having to be gone around. Radars help a bit, but they see through walls so they can make the pig move towards a wall, rather than going around it.

A pig can still use its radar to choose the pig-food towards which it wants to move, and if the next square in the direction of that pig-food is not blocked by a wall, or occupied by another life-form, then the pig can move on to that square. But, if that square is blocked, then the pig should move somewhere else. However, we want to avoid having the pig simply going around in circles. To avoid this, the pig should have some way to remember where it has been, so that it doesn’t keep going back to squares that it has previously visited unsuccessfully.

In PigWorld, a pig doesn’t have a good memory. It doesn’t remember where it’s been. Instead, as pigs move around the world, each one trails a rope behind them. The rope serves as a reminder to a pig of squares it has already visited. Dropping rope is part of the technique that pigs use to find their way around an obstacle.

Each pig trails its *own rope*, so pigs’ ropes do not interfere with each other when more than one pig is searching for food. The presence of a piece of rope in a square does not prevent a pig from moving into that square, regardless of whether that pig owns the rope or not. Likewise, rope does not stop other types of life-forms (such as wolves) moving into a cell/square.

When a pig satisfies its goal by eating some food, it “rolls up” (clears) its rope so that it can begin a fresh search without any old rope hanging around.

1. **Pig Behaviour – LookForFood – Advanced (using Rope)**

(Task 13 must be completed/attempted first, so that you have experience with a variety of useful methods in PigWorld.)

This is the last task, and the most challenging. The requirements are specified below, but it is up to you to work out many of the coding details, based on these requirements and what you’ve learnt from the earlier tasks.

Each pig owns a rope which occupies zero or more cells/squares. Each rope consists of a number of *pieces of rope*, with each piece occupying a single cell. E.g. a rope that goes across three cells has three rope pieces. Each rope piece is represented by a **RopePiece** object – see **RopePiece.cs**. The **RopePiece** class has only a few lines of code, but it inherits some of its properties from the **Thing** class. One of those properties is the **Cell** that the **RopePiece** is in, so a **RopePiece** always “knows” where it is.

A pig knows which rope pieces it has, by keeping a (possibly empty) *collection* of **RopePieces** – see the **ropePieces** data item in **Pig.cs**. This uses a **List<RopePiece>** to hold the collection, because the number of **RopePieces** grows and shrinks as the pig moves and eats. As you may recall from Lecture 4, this **List<T>** class – where **T** is a datatype – is like an array but its size varies when we need it to.

You may like to look at the given **DropRope** method (in **Pig.cs**), to see how a piece of rope is added to a cell.

The *first part* of this task is not too hard. You are to complete the empty **ClearRope** method (in the **Pig** class) that you saw in Task 10. Your code must do two things:

* firstly, call the Delete method on each piece of rope that the pig has – so that those rope pieces are removed from PigWorld’s cells; and
* secondly, remove all those rope pieces from the pig’s collection of RopePieces (described above).

It is strongly recommended that your code does those two things in the order stated above. Why?

The s*econd part* of this task is harder. To get you started, an almost empty **LookForFoodUsingRope** method is provided. This is to be an enhanced version of the **LookForFood** method that you developed in Task 11. So that you can submit both versions in your assignment, the enhanced method has been given another name, **LookForFoodUsingRope**. So that this version gets used when you are testing your code, you should change the line in the **DoSomething** method so that:

} else if (IsHungry) {

LookForFood();

becomes:

} else if (IsHungry) {

LookForFoodUsingRope();

Your code for the **LookForFoodUsingRope** method should start out much the same as your basic **LookForFood** method: if there’s pig-food in the pig’s current cell, the pig should pick up the food and eat it. The only extra is that the pig must now clear its rope (having achieved its pig-food goal), so you need to also call **ClearRope** in this case.

Likewise, your code should use **FindNearest** to get information about the nearest pig-food. But the basic behaviour of trying move in the direction of the nearest pig-food (if any), no longer applies. Instead the pig must behave according to whichever is *the first of the following cases* that applies.

1. If there’s no pig-food anywhere in the world, the pig clears its rope and then moves by using the **WanderAround** method. That’s all it does, in this case.
2. In all the following cases, there is food somewhere.
3. If the pig can move in the direction of the food, it looks in the adjacent cell which is in the direction of the food. If there is food in that cell, or the pig does not have any rope in that cell, then the pig moves in the direction of the food.
4. Otherwise, the pig has to consider each of its 8 adjacent cells/squares (where they exist) as possible places to move. Firstly, the pig ignores any of those 8 cells that it can’t move to.
5. Of the remaining cells (if any), the pig prefers to move to a cell that doesn’t have any of this pig’s rope on it. I.e. it prefers to explore places it hasn’t already visited, in its current search for food. If there is *only one* such cell, then the pig moves to it. But when there is *more than one* such cell, then the pig moves to the cell which has the *closest direction* to the direction of the food.
6. On the other hand, if *all* of the adjacent cells already have this pig’s rope on them, then the pig moves to the cell which it visited longest ago (so that it moves away from where it’s been recently).
7. Lastly, if none of the above cases apply (e.g. the pig is completely surrounded by walls), then the pig doesn’t do anything.

In cases (2) and (3) above, *before the pig moves*, it drops a piece of rope in its current cell, so it has a history of where it’s been.

When implementing the above behaviour, you may find the pig’s **CanMove** and **GetMyRopePiece** methods useful, as well as some of the methods you used in previous tasks. You may also find the **RopePiece** class’s **GetDistanceFromOwner** method useful in case (3), for determining which of two pieces of rope is the older, because a piece that is further away from its owner pig is older than one which is closer.

If written as a single method, your **LookForFoodUsingRope** method should require less than 100 lines (not counting comments and blank lines). That’s more than the coding standard limit of 50 lines (see Lecture 6), so you should create at least one lower-level method to stay within this limit. (You can choose any name(s) you like, for any such lower-level method(s).)

1. Cells/squares at the edge of the world won’t have a full set of neighbours. [↑](#footnote-ref-1)