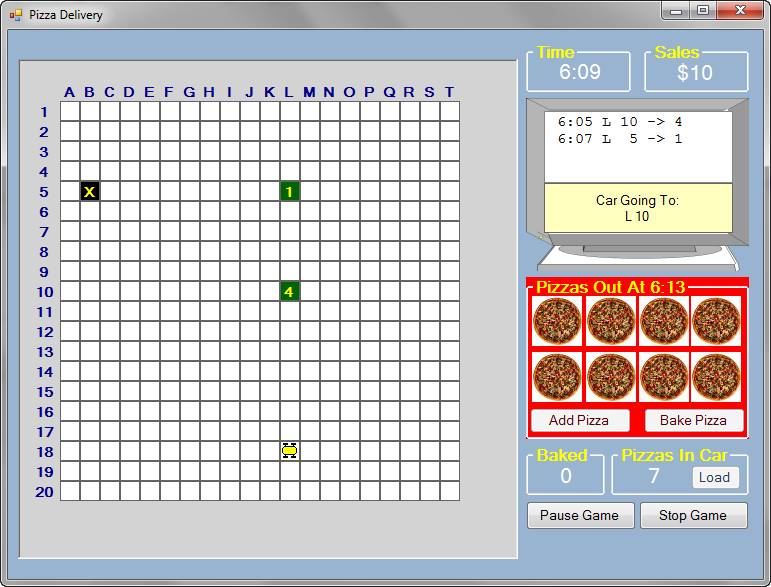
**Pizza Delivery**

The next project we will build a **Pizza Delivery** game. In this simulation game, lots of decisions need to be made. The basic idea is to read the incoming phone orders and tell the delivery car where to go. You also need to make sure you always have fresh-baked pizzas ready to go out the door. The delivery area is a grid of 20 by 20 squares. The more pizzas you sell, the more money you make!

The program teaches mental math, logical thinking skills and good business practices. We’ll see how to use multiple timers, use the list box control, develop a multiple form application and add controls at run-time.



The panel control on the left of the **Pizza Delivery** form holds the delivery grid. At the top right are group boxes with a single label control to display current time and current sales. The computer monitor (in a picture box) displays orders and status using a list box and label control. Another group box holds the pizza oven where pizzas are displayed using eight picture box controls. Two button controls in the group box control oven operation. The group boxes below the oven show how many pizzas are ready for delivery and how many are in the car (a button control loads pizzas in the car).

Notice, we said the panel control holds the delivery grid. But, where is that grid? The grid consists of 400 label controls in 20 rows (marked by a number) of 20 columns (marked by a letter). Can you imagine trying to place that many controls on a form? In **Pizza Delivery**, we will learn how to place controls on a form (or panel control in this case) using code. This saves us lots of design time.

Before running the game, let’s describe its operation. You sell and deliverpizzas from 6:00 to 11:00. A clock in the upper right corner shows the time. It doesn't take 5 hours to use the program - the five hours go by in about 15 minutes, so think fast. The phone rings when an order comes in. The order box in the monitor shows what time the order was called in, the coordinates for delivery, and the number of pizzas ordered. The order is also displayed on the

neighborhood grid.

In the monitor message area, you are told the current location of the car in the grid. At the beginning of the program, the car is at the pizza parlor (marked by an **X** on the grid). Group boxes at the bottom tell you how many pizzas are baked and ready and how many are in the car. To load pizzas in the car, click the **Load** button. To tell the car where to go, click on the grid position. The car will travel to the desired position. When it arrives, the car’s horn will sound and tell you the result of the delivery. There are four possible results: (1) on-time delivery, (2) late delivery, (3) not enough pizzas in the car to make the delivery, or (4) no pizza ordered at the given location. If you make a successful delivery, the amount of sales (displayed at the top of the screen) is updated. When you need to return to the pizza parlor to load more pizzas, click the location marked by **X**.

You also need to control the pizza oven to make sure there are always pizzas available for delivery. To bake pizzas, click **Add Pizza** for each pizza you want to bake (pizzas can’t be taken out once they’re in). When the oven is loaded, click **Bake Pizza**. You will be told when the pizzas are ready. When the pizzas are done, a bell will ring and the pizzas are moved to the ready area. The **Baked** box always displays how many pizzas are available for loading into the car. At any time during the program, you may pause the action by clicking **Pause Game**. To restart, click **Restart Game**. To stop the program before 11:00, click **Stop Game**. Once the program stops, a summary screen will show your sales and costs results and compute your profits (if any). After reading the results, click **Return to Pizza Delivery** to play again.

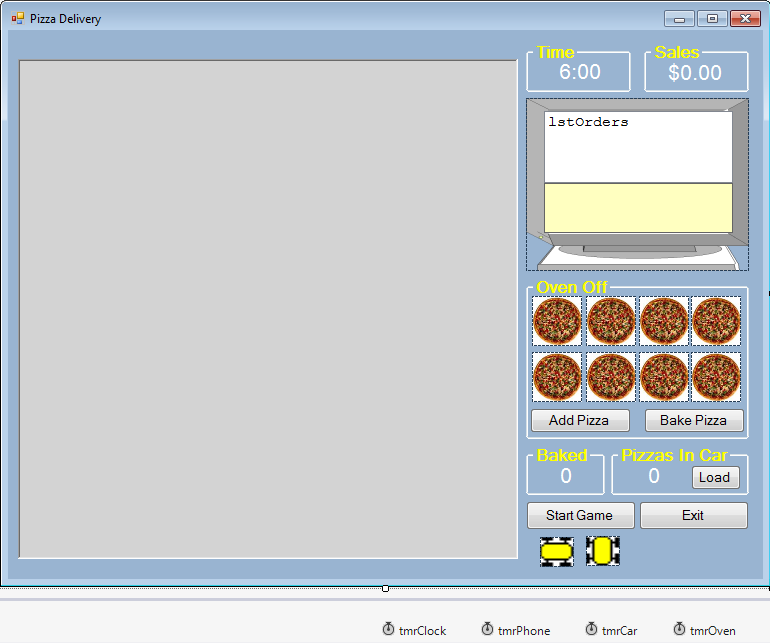
This may seem like lots of instructions but the process becomes fairly obvious as you play the game. Run the project (press <**F5**>). The game will appear in its ‘stopped’ state and the delivery grid will appear. At this point, you simply click **Start Game** to get things going. There are no options to select. The randomness inherent in the **Pizza Delivery** game makes each new game a different experience. Once the game starts, the first thing you notice is the phone starts ringing with orders.

**Pizza Delivery Form Design**

We begin building the **Pizza Delivery** project. Let’s build the first form – the one with the delivery grid, status monitor, pizza oven and car control. The results form will be built later.

Start a new project and place a panel control on the left side of the form. It should be square. Mine is 500 pixels by 500 pixels. On the right side, add 2 group boxes at the top (each with a label control) and a picture box under the group boxes. Put a list box control and a label control in the picture box. Place another group box under the picture box control. In this group box, place 8 square picture box controls (to display baking pizzas) and two button controls. Add two more group boxes and two button controls. Add labels in the two group boxes and in the second, add a button control. Add two small picture box controls under the two control buttons at the bottom (these will not be visible in the running project). Lastly, add four timer controls to the form. (Clock, Phone, Car and Oven) and their intervals to 100.

When done setting properties, my form looks like this:



This completes the basic form design (note the hidden picture boxes with the car graphics). You should be able to identify all the controls in the game. But, we still need the grid. This grid will use over 400 label controls added in the form **Load** event method. To do this, we need to discuss how to add controls at runtime and how to connect such controls to event methods.

**Adding Controls at Run-Time**

In building Visual C# Express applications, the process of adding controls and setting properties at design time is straightforward and simple. There are times, however, when it might be advantageous to add controls at run-time. For example, in our **Pizza Delivery** project, we need a 21 x 21 grid of label controls. That’s a lot of controls! With just a bit of code, we can automate the task of **adding controls** to a form at **run-time**. We can also remove controls if desired.

To add a control at run-time, we need to follow five steps:

1. declare thecontrol
2. create the control
3. set control properties
4. add control to form
5. connect event handlers.

We look at each step separately for a generic control named **myControl** of type **ControlType**.

The first step is to declare a control using the usual statement:

**ControlType myControl;**

Quite often we declare an array of controls. The control is created using the respective constructor:

**myControl = new ControlType();**

At this point, **myControl** is established with a default set of properties. You can overwrite any properties you choose. In particular, you must set values for the **Left** and **Top** properties. If you don’t, all your new controls will be stacked in the upper left corner of the form (Left = 0, Top = 0). You probably also want to change the **Width** and **Height** properties. Once the properties are set, the control is added to the form using the **Add** method of the **Controls** object:

**this.Controls.Add(myControl);**

If you are adding the control to a group box or panel control, you would replace **this** (referring to the form) in the above statement with the container control’s name.

So, now the control is created and on the form, but recognizes no events. Decide what events you want your control to respond to. If you want event **MyEvent** to be handled by a method **MyMethod**, an event handler is created using:

**myControl.MyEvent += new**

**System.EventHandler(this.MyMethod);**

Before using this statement, the method **MyMethod** should exist in the code window. It could be a method corresponding to an existing control or a new method you create. If you create it, the format is:

**private void MyMethod(object sender, EventArgs e)**

**{**

**.**

**}**

You would write code in this method, assuming event handlers are added at runtime. As an aside, you can add event methods in code for existing controls also. This sometimes saves a little typing at design time. For example, say you have a method that handles clicking on 20 button controls. Rather than assign the event method 20 times in the properties window, you could add it in code.

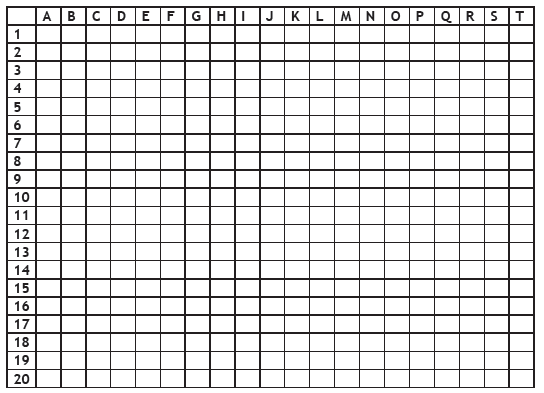
You can also remove controls from your application. To remove a control (named **myControl**) from the form, use:

**this.Controls.Remove(myControl);**

If you are removing a control from a group box or panel control, replace the keyword **this** with the container control’s name. When the control is removed, all event handlers for this control are modified to no longer include the removed control. This also happens when a control is deleted at design time.

Let’s use this new information to build the delivery grid for the **Pizza Delivery** game.

**Form Design – Building Delivery Grid**



Notice we have labeling information for the columns and rows and a 20 x 20 grid used to display orders. Individual elements in the grid are referred to by a column letter and a row number (like Excel spreadsheets).

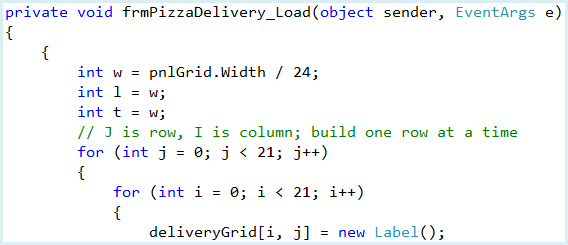
We will build this grid in the **pnlGrid** control using 441 (21 x 21) square label controls. The process is fairly straightforward. For each of the 441 label controls, we create the control, set properties (particularly **Left**, **Top**, **Width**, **Height** for proper positioning) and place the control in the panel control. The first row in the grid will be used to label the columns; the first column will be used to label the rows. We will ignore the label control in the upper left corner. The remaining 20 x 20 grid of label controls will be used to display orders and car motion.

First, define a form level two-dimensional array of label controls:

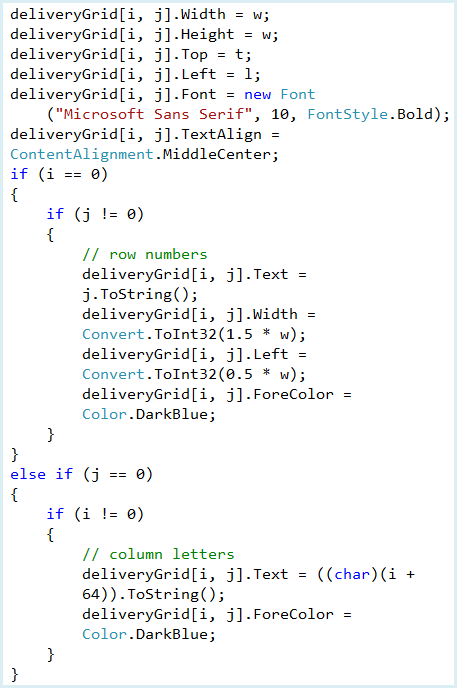
****

We will use the 0th elements of **deliveryGrid** to represent the column and row labels. We ignore **deliveryGrid[0, 0]**, the upper left corner in the grid.

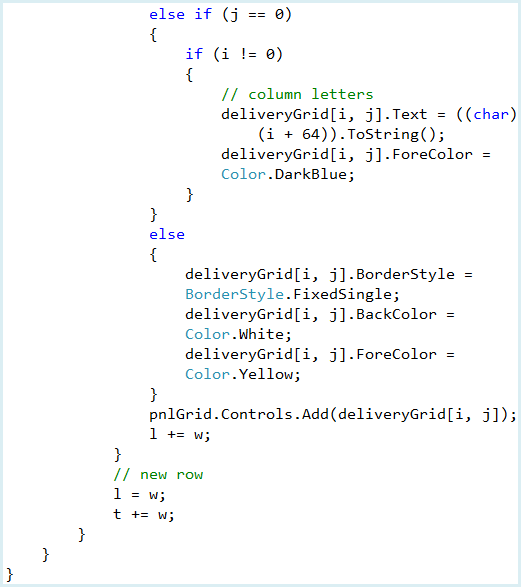
The code to establish this grid and display it in the panel control (**pnlGrid**) goes in the **frmPizzaDelivery Load** event method:



Code continued below

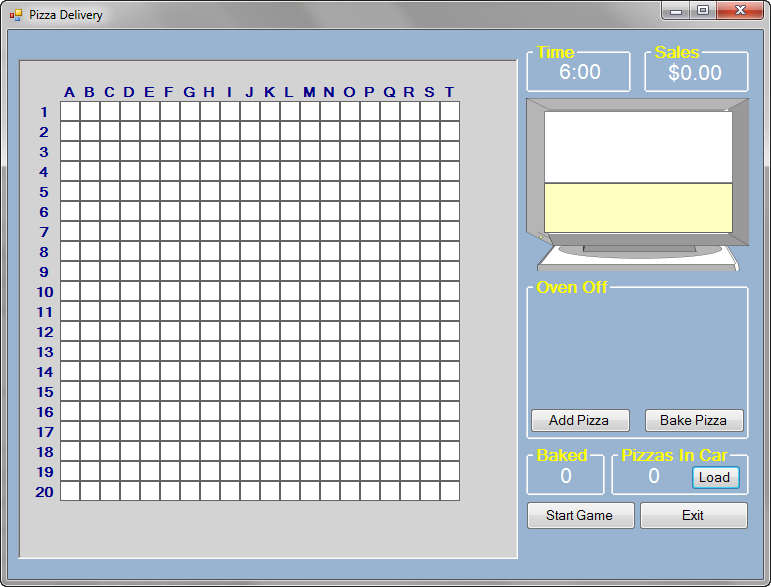
****

Code continued below

****

This code builds the label control grid, one row at a time. Each control is first declared, and assigned a **Top**, **Left**, **Width** and **Height** property. We assume square controls, with the size based on the width of the panel – (1/24th of the panel, which allows for some margin space). We start at the upper left corner, working our way to the right. Once a row is finished, the **Top** property is incremented down one row and the **Left** property returned to the left “margin.”

Other properties are assigned based on locations. Controls with column labels (**j = 0**) and row labels (**i = 0**) have no border and are given an appropriate letter or number. Notice how the row labels are shifted a bit to the left to allow 2 digit numbers. Internal labels (**i = 1 to 20**, **j = 1 to 20**) have a **FixedSingle** border and white background to give a grid effect. After setting properties, controls are added to the panel control. Now, let’s give it a try. Save and run the project. Like magic, you should see the grid:



I think you see the power of adding controls at run-time. Make sure you understand how the code to add these controls work. Such code is very useful.

**Code Design – Initializing Stopped State**

All initializations are done in the form’s **Load** event method. When initialized, we need to make sure the only thing a user can do is click on **Start Game** or **Exit**. They should not be able to operate the pizza oven or load the car. Add these two lines at the end of the **frmPizzaDelivery Load** event method:

****

Save and run the project making sure the oven and car loading are disabled. We now begin adding other elements to our simulation, one at a time. We add code for running the clock, accepting phone orders, running the pizza oven, loading the car, moving the car and evaluating the deliveries. Each element will require certain “tuning” parameters – values that define how the simulation works.

Every simulation has such parameters. These are constants you adjust as you define the program. In the code here, we give you values of such constants we use. Feel free to change them as you choose.

**Code Design – Clock**

The first element we add to the **Pizza Delivery** is the clock that keeps track of time. Recall the clock starts at 6:00 and stops at 11:00. We don’t want the game to actually last five hours, so we speed things up a bit. We need to decide how fast we want the five hours to really go by and still be able to play the game.

The clock is controlled by the timer named **tmrClock**. With each **Tick** event for this control, we will add one minute to the clock. We want to define a constant (**mSecPerMinute**) which tells us how many milliseconds (the **Interval** property of **tmrClock**) should elapse for each minute on the clock. We choose to take approximately 15 minutes to play a complete game, or to simulate 300 minutes (5 hours x 60 minutes/hour) of pizza deliveries. So, we need invoke the **Tick** event 300 times in 900000 milliseconds (15 minutes x 60 seconds/minute x 1000 milliseconds/second).

The constant we use is thus:

**MSecPerMinute** = 900000 / 300 = **3000** milliseconds/minute

Add this constant to the form level declarations area:

****

And, add this single line at the end of the **frmPizzaDelivery Load** event method to set up the timer control:

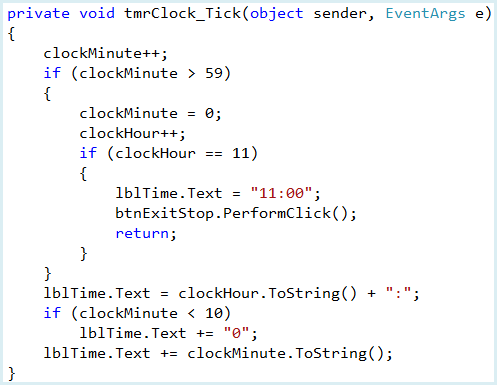
****

As mentioned, with each **Tick** event, we add one minute to the clock display.

Add these form level declarations to keep track of the time:

****

The corresponding code in the **tmrClock Tick** event is:

****

Note how the displayed time is formatted. Also, notice once 11:00 is reached, the game stops by clicking **btnExitStop**.

Now, we’ll write the code that starts, pauses, restarts and stops the clock. The **btnStartPause** control can have three possible **Text** properties. If the **Text** property is **Start Game** when clicked, the game is initialized and put in playing mode using:

* Change **Text** to **Pause Game**.
* Initialize time to 6:00.
* Start **timClock**.
* Change **Text** of **btnExitStop** to **Stop Game**.

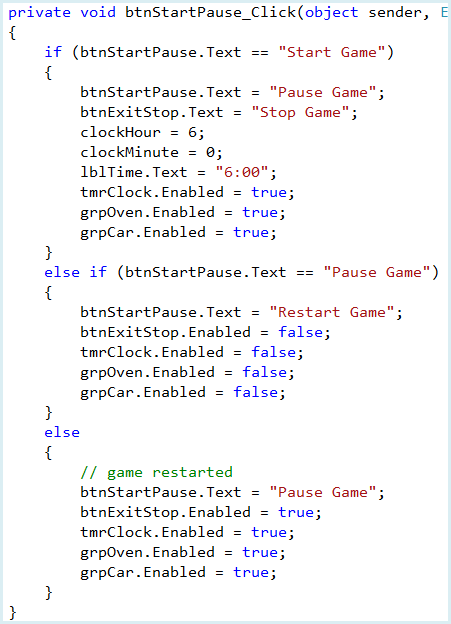
When **Text** property is **Pause Game** and the button is clicked, we enter pause mode:

* Change **Text** to **Restart Game**.
* Stop **timClock**.
* Disable **btnExitStop**.

When **Text** property is **Restart Game** and the button is clicked, we enter playing mode:

* Change **Text** to **Pause Game**.
* Start **timClock**.
* Enable **btnExitStop**.

The code that covers the above steps is in the **btnStartPause Click** event method:

****

Notice in pause and playing mode, we also disable/enable **grpOven** and **grpCar** appropriately.

Save and run the project. Click **Start Game** and the clock should begin counting up the minutes. Try the pause and restart feature. How long does it take one hour to go by? It should take about 3 minutes based on our “tuning” parameter (**mSecPerMinute**).

The **Stop Game** button doesn’t work yet – let’s fix that.

When a player clicks **Stop Game** (**btnExitStop**), the following should happen:

* Play **Game Over** sound
* Change **Text** to **Exit**.
* Change **Text** of **btnStartPause** to **Start Game**.
* Stop **timClock**.
* Disable **grpOven** and **grpCar**.

If **btnExitStop** is clicked when the caption is **Exit**, the program stops.

Add this form level declaration for a **Game Over** sound:

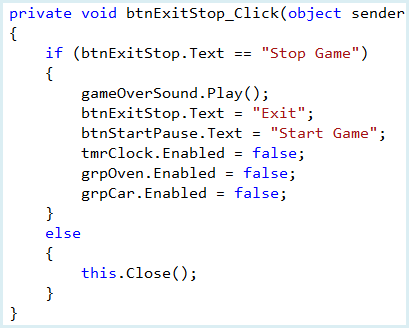


We use the **tada.wav** file for this sound. Copy all the sound files attached to the project file from the website to your project’s **Bin\Debug** folder.

Add this single line to the **frmPizzaDelivery Load** event to load the sound.



Now, the **btnExitStop Click** event method is:



Save and run **Pizza Delivery**. You should now be able to start the game, pause the game, restart the game and stop the game. You can also stop the program. Make sure you hear a ‘tada’ sound when the game is ended.

You also want to make sure the program automatically stops once the clock reaches 11:00. But, you don’t really want to sit around for 15 minutes to see that happen. To speed things up, temporarily set the **mSecPerMinutes** to something like 100. Then, run the program, click **Start Game** and watch time fly by! Once the clock hits 11:00, a sound will be played and the program should return to stopped state.

We often do this in programming – change parameters (even if the changes are not realistic) to test out different elements of code. Just remember, after changing the constant and making sure the clock stops at 11:00, to change the constant back to its original value of 3000. Next, let’s write the code to get phone orders coming in.

**Code Design – Phone Orders**

In **Pizza Delivery**, orders are phoned in randomly. The orders (time, location and number of pizzas) are displayed in the list box control (time sorted). The number of pizzas ordered is displayed at the proper location on the delivery grid (a green background is used). You then use this information to send a car to deliver the pizzas.

Here’s another place where I just made up some rules for the orders – feel free to change them if you like. The rules implemented for pizza phone orders are:

* Orders arrive at random times, in 1 to 7 minute increments.
* Orders come from random locations on the delivery grid.
* Each order is for 1 to 5 pizzas, using these probabilities:
  + 1 pizza, 30 percent
  + 2 pizzas, 20 percent
  + 3 pizzas, 20 percent
  + 4 pizzas, 15 percent
  + 5 pizzas, 15 percent
* No orders are accepted after 10:30 to allow car to finish deliveries.

As mentioned, the order information will be displayed in the list box control. We use a specific format for that listing.

As an example, say an order arrives at **6:20**, from grid location **G 12**, for **2 pizzas**. The order listing will appear as:

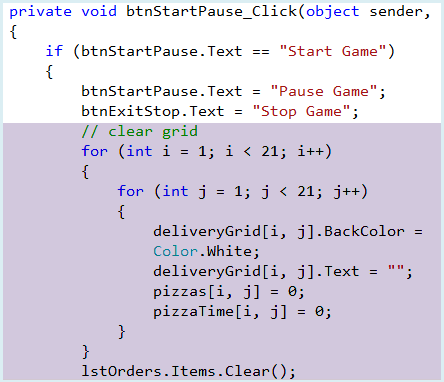
**6:20 G 12 -> 2**

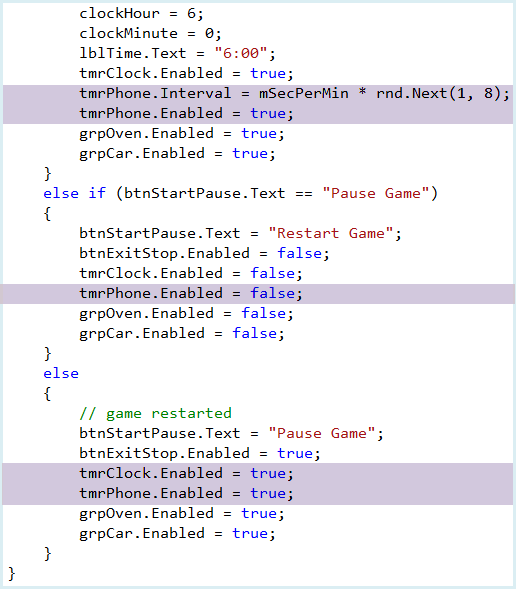
Two arrays will be used to keep track of order information. **pizzas** is a two-dimensional array that says how many pizzas were ordered at a grid location and **pizzaTime** is a two-dimensional array holding the corresponding order time (in minutes). Add these two declarations in the form level area:

****

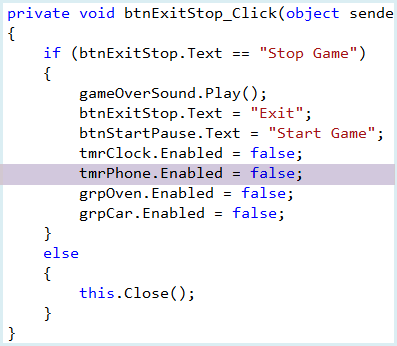
The timer control **tmrPhone** will govern how often orders arrive. The **Interval** property will be set randomly. Add this form level declaration to create a random object (**myRandom**):

****  
  
We need to make certain initializations for the order process. Make the shaded modifications to the **btnStartPause Click** event to set the initial **Interval** value (1 to 7 minutes), to clear any past orders from the grid and to start/stop the timer at the proper times:





We make a similar shaded change to the **btnExitStop Click** event to turn off the timer when the game is stopped:

****

When an order comes in, we will play a phone sound. Add this form level declaration for this sound:

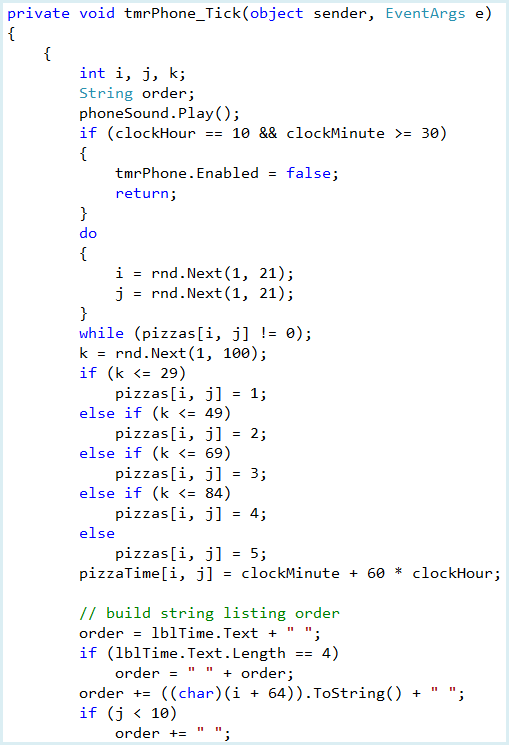
****

We use the **phone.wav** file for this sound.

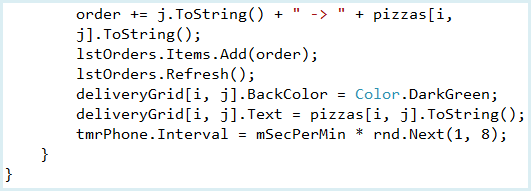
Add this single line to the **frmPizzaDelivery Load** event to load the sound.

****

The **tmrPhone Tick** event that encodes the previously listed rules for phone orders and plays the sound is:

****

Code continued below

****

In this code, we play the phone sound. If it is past 10:30, the routine is exited. Next, we select a random unselected column (**i**) and a random row (**j**) for the order. Using the specified percentages, the number of pizzas ordered is stored in the **pizzas** array. Likewise, the order time is stored in the **pizzaTime** array. Next, the text string to put in the list box is formed and added to the list box. The corresponding grid location is marked with a dark green background and the number of pizzas ordered. Before leaving the method, an new random interval

value is computed. This gives us the delay (between 1 and 7 minutes) until the next order is phoned in.

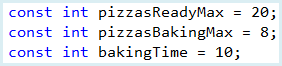
Save and run the project. Click **Start Game** and the phone should start ringing,with orders in the list box and the delivery grid area.

Next, we look at how to get the pizzas ready for delivery.

**Code Design – Pizza Oven**

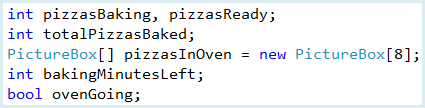
Now that we know where the orders are coming from, we need to get some pizzas baked. The **grpOven** group box holds the oven. As you can see, we can bake, at most, eight pizzas at a time, depicted by picture box controls. This is another one of those rules we chose – you can change it if you like. Once the pizzas are done baking (taking 10 minutes), they are placed in the **grpPizzas** group box for storage. Up to 20 pizzas can be stored. Any pizzas baked that cannot fit in the ready area are thrown out, costing you money.

Define three constants to embody these rules:

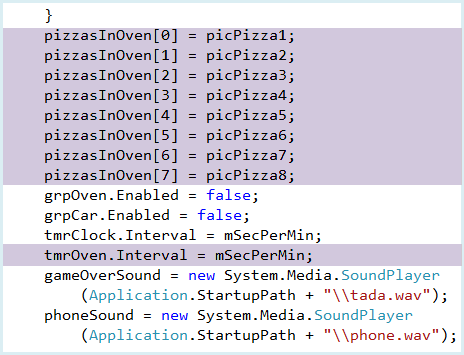
****

We use one variable (**pizzasBaking**) to keep track of pizzas in the oven and another (**pizzasReady**) to specify the number of pizzas baked and stored. **totalPizzasBaked** will keep track of all pizzas baked during a single game. The picture box controls are placed in a control array (**pizzasInOven**) for easier use. **bakingMinutesLeft** is used to tell us how many minutes are left in the baking process. And, we use a Boolean variable (**ovenGoing**) to know whether the oven is on or not – it is used for the paused state.

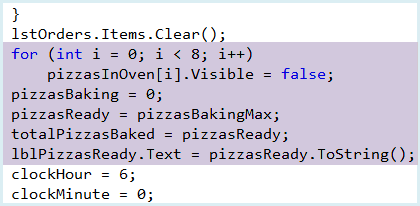
Add these declarations in the form level area:

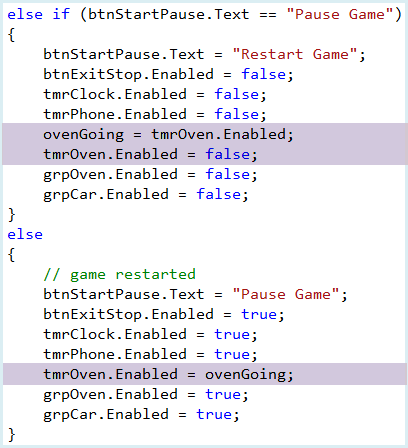
****

The timer control **tmrOven** will be used to time the baking process. Make the shaded changes to **frmPizzaDelivery Load** event to create the picture box control array and set the timer’s **Interval** property. We set the interval to **mSecPerMinute**, counting down the baking process one minute at a time. This allows proper pausing of the simulation:

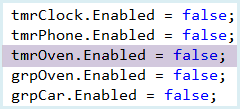


Make the shaded modifications to the **btnStartPause Click** event to clear the pizza oven, initialize variables (we assume we have baked one oven load prior to opening the store) and to start/stop the timer at the proper times:





Note how we used the **ovenGoing** variable to pause and restart the simulation. We make a similar shaded change to the **btnExitStop Click** event to turn off the oven timer when the game is stopped:

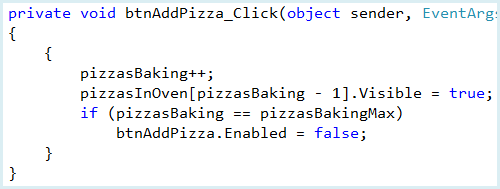


Pizzas are added to the oven by clicking the **Add Pizza** button (**btnAddPizza**) in the **grpOven** control. With each click of the button, we follow these steps:

* Increment **pizzasBaking**.
* Make the corresponding **pizzasInOven** picture box visible.
* If maximum number of pizzas reached, disable **btnAddPizza**.

Note once a pizza is placed in the oven, it cannot be removed.

The **btnAddPizza Click** event that implement these steps is:

****

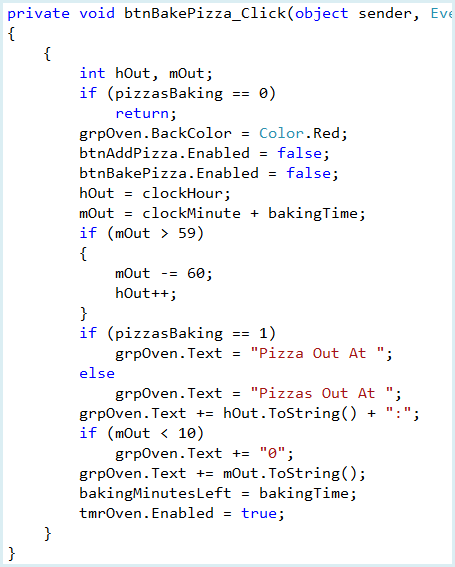
Save and run the project. Click **Start Game**, then click **Add Pizza** a few times. You should now be able to load the oven.

Once the oven is loaded, baking begins by clicking the **Bake Pizza** button.

The steps to follow once this button is clicked are:

* Change group box background color to red.
* Disable button controls.
* Display time pizzas will be done baking.
* Initialize **bakingMinutesLeft**.
* Enable **timOven** to start the baking.

The corresponding **btnBakePizza Click** event method is:

****

Save and run the project again. You should now be able load the oven and start the baking process. Always remember, a red oven is a baking oven; a blue oven is a cold oven.

Sometimes when playing the game, you forget to start the baking process.

With each **Tick** event on the oven timer (**tmrOven**), we will do these steps:

* If **bakingMinutesLeft != 0**, decrement **bakingMinutesLeft** and exit; else continue with next step.
* Play ding sound.
* Turn oven off.
* Increment **pizzasReady** and **totalPizzasBaked**, check limit against **pizzasReadyMax**.
* Set **grpOven Text** to **Oven Off**; change background color to blue.
* Remove all pizza picture boxes.
* Set **pizzasBaking** to 0.
* Enable **Add Pizza** and **Bake Pizza** buttons.

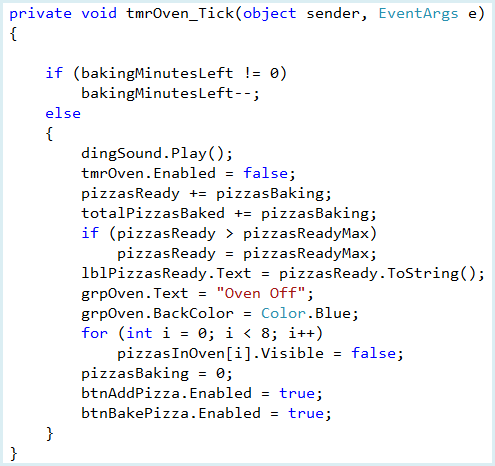
Notice, when the baking is done, we will play a ding sound. Add this form level declaration for the sound:

****

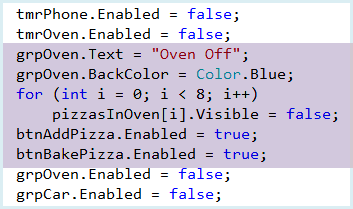
Add this single line to the **frmPizzaDelivery Load** event to load the sound.

****

The **tmrOven Tick** event method that increments the needed steps is:

****

We are almost ready to test everything, but need one more change. If we stop the game while the oven is running, the control buttons will be disabled, the oven will still be red with pizzas in it. Make a few more shaded changes in the **btnExitStop Click** event to handle this possibility:

****

Save and run the project. Click **Start Game**, then click **Add Pizza** some number of times. Click **Bake Pizza** and wait. After 10 minutes elapse (may be 11 minutes depending on just when you click **Bake Pizza**), you will hear a ding sound and the pizzas you baked will be moved to the **Baked** area. Continue baking pizzas if you like – remember you can only store 20 pizzas. Make sure you can pause, restart and stop the game.

Next, we see how to move pizzas into the car and start delivering them.

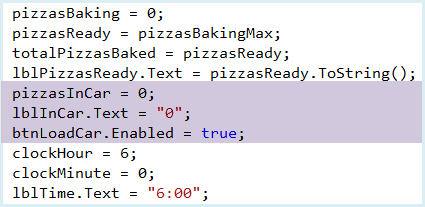
**Code Design – Load Car**

A delivery car is available to take pizzas from the pizza parlor to the order locations on the grid. A first step in the delivery process is to load pizzas from the **Baked** area into the car. This is done by clicking the **Load** button in the **grpCar** group box control. We use a variable **pizzasInCar** to keep track of how many pizzas are in the car. We assume a maximum of 10 pizzas can fit in the car. Add the variable declaration and another constant (**PizzasInCarMax**) to the form level declarations area:

****

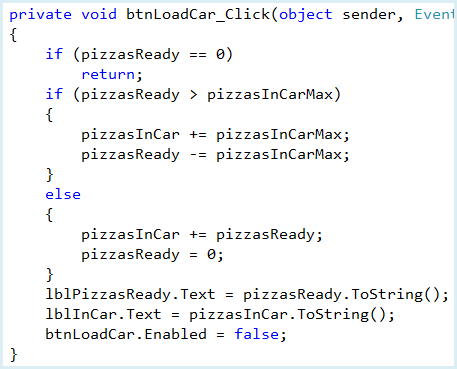
Add the shaded code to the **btnStartPause Click** event to initialize the car

variable and make sure the **Load** button is enabled:

****

As stated, to load the car, you click the **Load** button in the group box labelled **Pizzas In Car**. The steps followed (assuming there are pizzas to load) are simple. If the maximum number of pizzas to load are available, load that number in the car, otherwise load all the ready pizzas.

The **btnLoadCar Click** event that handles this task (and associated labeling) is:

****

Notice we disable the **Load** button once the pizzas are loaded. In the motion code, we use the **Enabled** status of this button to tell us whether the car has pizzas available for delivery.

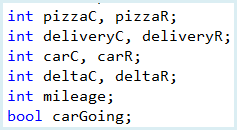
Save and run the project. Start the game, bake some pizzas and load the car. In this screen, I baked six pizzas, giving me 14 baked pizzas. I then clicked the **Load** button. The car now has the maximum of 10 pizzas and there are 4 pizzas remaining in the **Baked** area:

Next, let’s start delivering pizzas by driving the car around the delivery grid.

**Code Design – Move Car**

Once there are pizzas in the car, we’re ready to start deliveries. Click on a location on the delivery grid and the car (represented by a little yellow image) travels from the pizza parlor to that location. You follow the same process to have the car return to the pizza parlor when you need more pizzas. There are lots of things to do here: define the pizza parlor location (randomly set), detect clicks on the delivery grid, identify the clicked label control, and then move the car to the clicked location.

We need several variables to keep track of the car. Add these form level declarations to the code window:

****

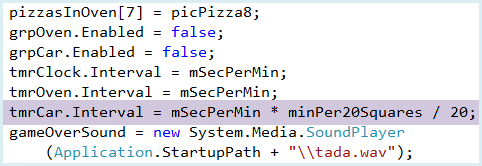
**pizzaC** and **pizzaR** are the column and row numbers for the pizza parlor location in the delivery grid (marked by an **X**). **deliveryC** and **deliveryR** are the column and row numbers for the delivery (the clicked label control). **carC** and **carR** are the column and row numbers for the current car location. **deltaC** and **deltaR** are the difference between the car and delivery columns and rows. Lastly, **mileage** is the number of grid locations the car has moved through and **carGoing** indicates if the car is currently moving (used for pause mode).

The timer control **timCar** will be used to make the car move. We define one constant to set car speed:

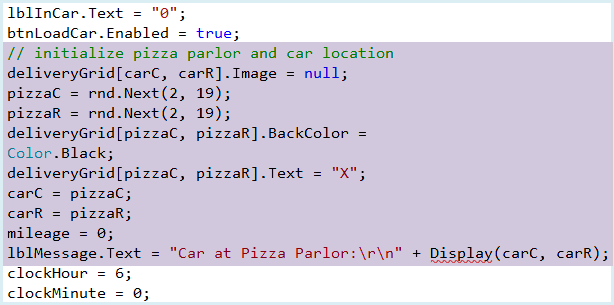
****

This tells us how many minutes (of clock time) we want to take for the car to travel across the delivery grid (20 squares).

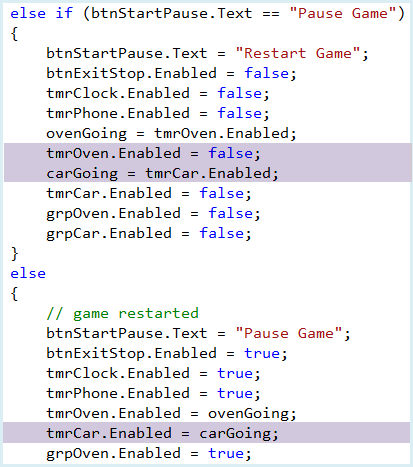
Make the shaded change to **frmPizzaDelivery Load** event to set the timer’s **Interval** property:



Make the shaded modifications to the **btnStartPause Click** event to initialize the pizza parlor location (also the initial car location) and to start/stop the timer at the proper times:

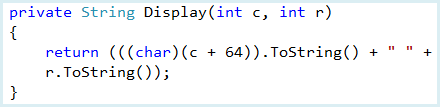


Code continued below



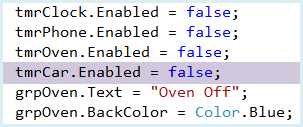
We first erase any previous car on the grid. Then, the pizza parlor is placed somewhere on the interior of the grid and marked by an **X**. The car is positioned at this location and the mileage initialized. A message is posted announcing the car is at the pizza parlor. Note how we used the **carGoing** variable to pause and restart the simulation.

In the messaging in the above code, we use a general method **Display** that converts column and row numbers to column letter and row number. Add this general method to the code window:



Note how **char** is used to convert the column number **c** to a letter (based on its Unicode).

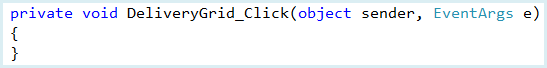
We make a shaded change to the **btnExitStop Click** event to turn off the car timer when the game is stopped:

****

Save and run the project. Click **Start Game** to make sure the pizza parlor is marked and the proper message appears. The parlor is at location **M 4**. That is also the car’s initial location. Let’s see how to indicate where we want the car to go to and how to move the car.

Once orders come in and there are pizzas in the car, we can start making deliveries. To move the car to a desired location on the delivery grid, we simply click that location. We will have one event method handle the **Click** event for all 400 (20 x 20) of the label controls in the grid. That method will have the name **DeliveryGrid\_Click**.

Add this empty method to the code window:

****

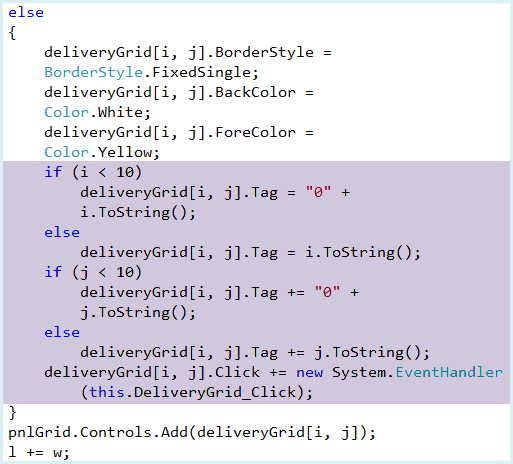
Once a grid location is clicked, we follow these steps:

* Deliveries can only be made if the car is not already moving (**timCar** is disabled), **grpCar** is enabled (the game is in play mode) and **btnLoad** is disabled (pizzas have been loaded in the car). If this is not the case, exit; else continue.
* Determine column (**deliveryC**) and row (**deliveryR**) of clicked label.
* Compute **deltaC** and **deltaR**. If **deltaC = 0** and **deltaR = 0**, exit.
* Display message saying where the car is going.
* Enable **timCar**.

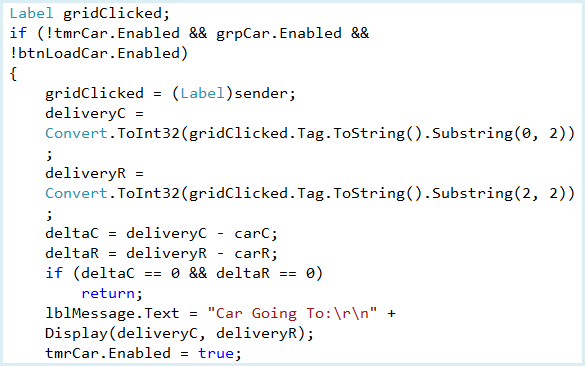
Each step should be clear except how to determine **deliveryC** and **deliveryR**.

We need some way to determine the row and column of the clicked label in the **deliveryGrid** control array. We will use the **Tag** property of the **deliveryGrid** elements. The **Tag** property (**String** type) is a user-defined property available for whatever use is needed. For each element in the **deliveryGrid** array, the **Tag** property will have four numbers, the first two indicating the column number, the second two indicating the row number. The **Tag** property will be set in the form’s **Load** event method. In that method, we also ‘connect’ each element of the control array to the **DeliveryGrid\_Click** event method.

Make the shaded changes to the **frmPizzaDelivery Load** code:



With this new **Tag** property, we code the needed steps in the **DeliveryGrid Click** event method we created earlier:

****

Save and run the game. Click **Start Game**, click **Load**. You should now be able to click the grid and have the clicked location appear in the message area.

Let’s see how to make the car travel to a selected location. Remember those little hidden picture boxes with car graphics (**picHCar** and **picVCar**)? We’ll use those to show the car moving around.

Clicking a grid location starts **tmrCar**. With each **Tick** event on this timer, we move the car one square. We do horizontal movements first, then vertical movements. The steps to accomplish this are:

* Erase car image in current position.
* If **deltaC != 0**:
  + Increment **mileage** by 1.
  + Set car image to **picHCar.Image.**
  + Recompute **carC.**
  + Recompute **deltaC.**
* If **deltaR != 0**:
  + Increment **mileage** by 1.
  + Set car image to **picVCar.Image.**
  + Recompute **carR.**
  + Recompute **deltaR.**
* Draw car at new location.
* If **carC = deliveryC** and **carR = deliveryR**:
  + Play beep sound.
  + Disable **timCar**.
  + If **carC = pizzaC** and **carR = pizzaR**, we’re back at the pizza parlor, so:
    - Establish message.
    - Erase car image.
    - Set **pizzasInCar** to **0** (must throw extras out due to health department rules).
    - Enable **btnLoad**.
  + If not at pizza parlor, we’re at the clicked location:
    - Establish message.
    - Check delivery status.

This may seem like a lot of steps, but they are straightforward. Notice, in particular, we re-enable **btnLoad** when we return to the pizza parlor to allow reloading of the car.

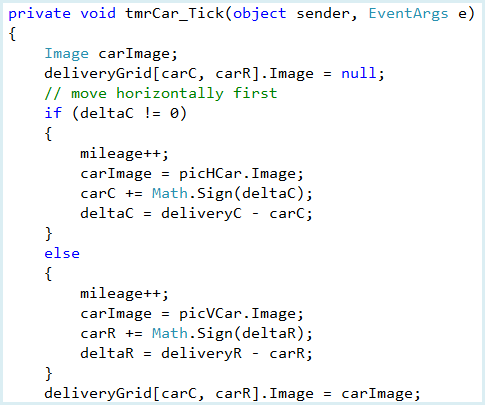
When the car stops, we will play a beep sound. Add this form level declaration for the sound:

****

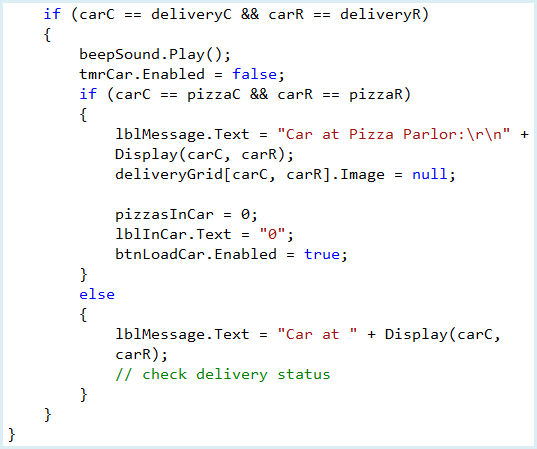
We use the **carbeep.wav** file for this sound. Add this single line to the **frmPizzaDelivery Load** event to load the sound.

****

The **tmrCar Tick** event method that increments the above steps is:

****

Code Continued Below.



Notice the use of the **Math.Sign** method to impart the proper direction on car movement. Also, we have left a “hook” for checking delivery status (a comment statement). Also notice once the car starts moving, it doesn’t stop until it reaches the clicked location.

Save and run the project. Click **Start Game**, then click **Load** to put some pizzas in the car. Now, click a location on the grid and watch the car go there. And, once the car arrives, you should hear a beep. You need to click the pizza parlor (marked by an **X**) when you need more pizzas to deliver. The car will return and disappear from the screen. The message will indicate the car is at the pizza parlor. The **Load** button will be enabled and you will have zero pizzas in the car. As mentioned, if the car returns with any pizzas, they must be thrown away due to health department rules. At this point, you need to bake more pizzas and reload the car if you want to go back on the road.

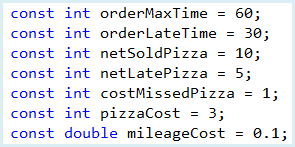
We’re getting close to being done. Just two more steps. We need to check the status of any deliveries made and, lastly, we need to present the final sales results when a game is stopped.

**Code Design - Deliveries**

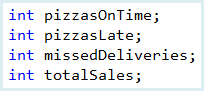
Once the car arrives at a selected location, we need check the status of the delivery at that location. We’ll also determine how the delivery affects our earnings. The rules used for deliveries are:

* An on-time delivery is within 30 minutes. On-time deliveries net $10 for each pizza delivered. On-time deliveries will have green backgrounds in the grid.
* A late delivery is between 30 and 60 minutes. Late deliveries net $5 for each pizza delivered. Late deliveries will have red backgrounds in the grid.
* After 60 minutes, the order is cancelled and there is a $1 penalty for each pizza not delivered.
* It costs $3 to bake each pizza.
* It costs $0.10 for each square the car travels through as it moves around.

The above rules are implemented in several constants. Add these to the form level declarations area:

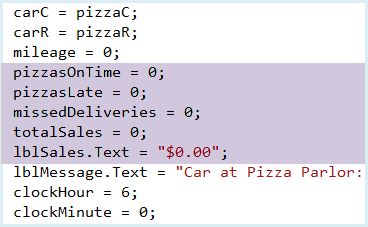
****

We need variables to keep track of the various delivery quantities. Add these form level declarations:

****

**pizzasOnTime** is the number of on-time deliveries, **pizzasLate** is the number of late deliveries, **missedDeliveries** is the number of missed deliveries and **totalSales** is the sum of all moneys taken in during deliveries.

Make the shaded changes to the **btnStartPause Click** event method to initialize the new variables:

****

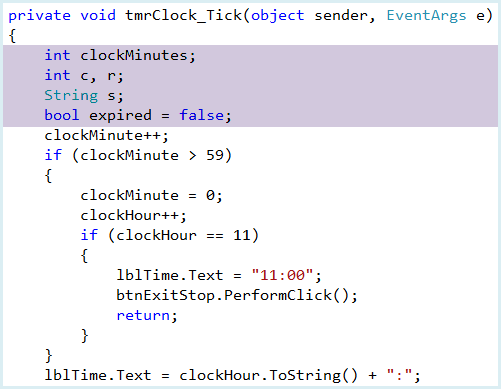
Notice we indicate on-time deliveries by green backgrounds in the grid and late deliveries will have red backgrounds. The backgrounds are initialized at green. Once 30 minutes elapse after an order is placed, we need to change the corresponding display grid background to red. And, once 60 minutes elapse for an order, it is removed from the grid with penalty costs incurred. All of this is done in the **tmrClock Tick** event method. At each **Tick** event, we follow these

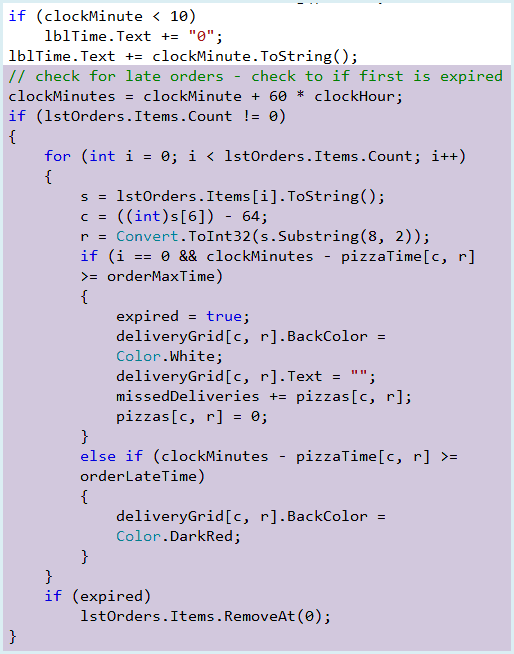
steps:

* Check to see if 60 minutes have elapsed for the first item in the list box. This is the oldest order in the list and the only candidate for removal. If 60 minutes have elapsed, increment **missedDeliveries** by the proper number of pizzas and remove the order from the delivery grid and list box.
* For all remaining list box items, check to see if 30 minutes have elapsed. If so, and the background isn’t already red, change it to red.

Make the shaded changes to the **tmrClock Tick** event code to implement these

steps:





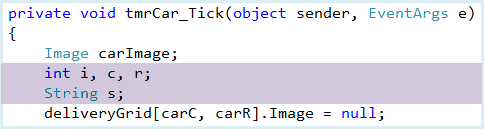
Note how we extract the column (**c**) and row (**r**) information from the list box items. In the code, we use the previously defined constants for on-time and late delivery times.

Save and run the game. Click **Start Game**. Watch as orders start out green. After 30 minutes, they should turn red. After 60 minutes, they should disappear from the grid. Make sure the program is working as desired. You might like to change the **mSecPerMinute** constant (temporarily) to a smaller number to speed things up.

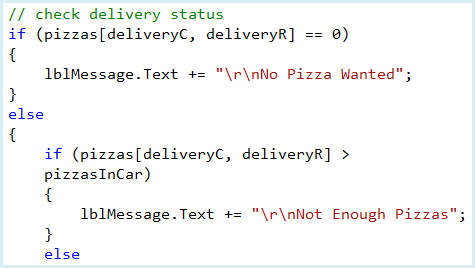
As seen previously, once the car reaches a delivery location, a check on the delivery status is done. This check is completed in the **tmrCar Click** event (we left a “hook” for this code). Using our rules, the steps followed are:

* If the delivery location did not order any pizzas, post message and exit.
* If the delivery location needs more pizza than is available, post message and exit.
* If on-time delivery, increment **pizzasOnTime** by number of pizzas ordered. Increment **totalSales** accordingly.
* If late delivery, increment **pizzasLate** by number of pizzas ordered. Increment **totalSales** accordingly.
* If successful delivery (whether on-time or late), decrement **pizzasInCar** and remove order from delivery grid and list box control.

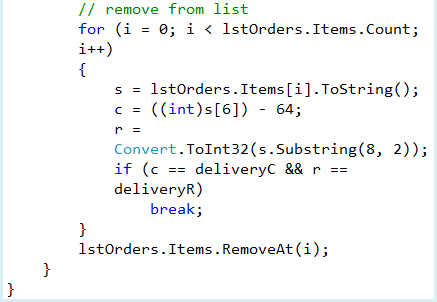
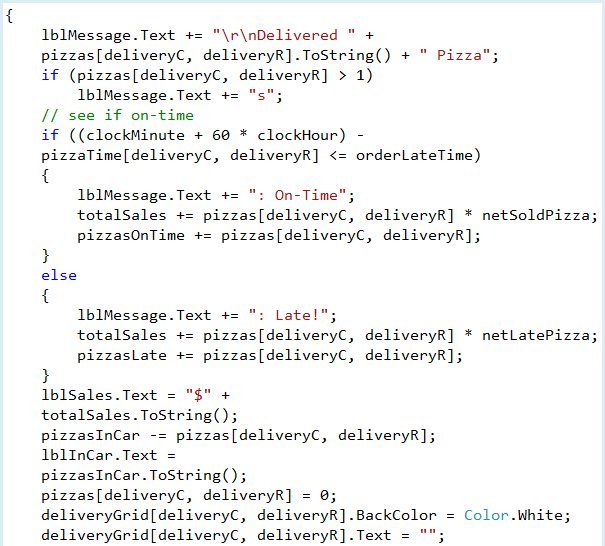
Add the shaded code to the **tmrCar Click** event method to implement these steps:

****

At the bottom of the **tmrCar Click** event method you should have a comment // check delivery status. It’s not time to add in that code below it.



Code continued below.



Notice how we search through the list box to find the item corresponding (matching column and row) to the order to remove. Also, notice many of the code lines take up more than one line due to margin restrictions.

Run and save the project. At long last, you can run the entire simulation. You can accept orders, bake pizzas, load the car and direct the car to orders. You can return to the pizza parlor when you need more pizzas. Try playing for a while.

Stop the game and program when you’re ready to go on. When you stop the game, the only results currently shown are the total sales. A total financial analysis should include costs to determine net profits. We’ll use a separate form to present this analysis. So let’s see how to use multiple forms in a Visual C# Express project.

**Multiple Form Visual C# Express Applications**

We want to learn how to manage multiple forms in our projects. There are four major considerations in using multiple forms:

1. Adding forms to a project
2. Deciding which form appears when the project begins
3. How to make forms appear and disappear using code
4. Transferring information from one form to another

To add a new, blank form to an application, click the **Add New Item** button on the toolbar and select **Add Windows Form** or select **Add Windows Form** under the **Projects** menu. A dialog box will appear asking you to name the new form. Choose a name and click **Add**. The newly added form will be listed in the **Solution Explorer** window. To have the new form appear in the Design window, highlights its name and click the **View Designer** button in the Solution Explorer toolbar. Or, just double-click its name in the Solution Explorer. Each form is designed using exactly the same method we always use: draw the controls, assign properties, and write code.

It is also possible to add an existing form to an application. You can use such a form, as is, or modify it for the new application. To add an existing form to a project, choose the **Project** menu item and select **Add Existing Item**. Browse the directories until the desired form file is located. Select it and click **Open**. If you want to delete a form from a project, simply right-click the form name in the Solution Explorer window. A menu will appear. Select **Delete**. Make sure you really want to delete the form – you will not be asked if you know what you’re doing!

Display of the different forms is handled by code you write. You need to decide when and how you want particular forms to be displayed. The user always interacts with the ‘active’ form.

The first decision to make is to determine which form will be the **startup form**. This is the form that appears when your application first begins. Up to now, the startup form has been the one, single form in our project. Now, we must choose among multiple forms. To do this, in **Solution Explorer**, open the file **Program.cs**. In that file is a short construction (**Main**) with this code:

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new Form1());

}

The startup form is specified by the final line in this method:

Application.Run(new Form1());

By changing this form **Name** to another form in your project, that newly identified form becomes the startup form.

The startup form automatically loads when your application is run. Any other form in the project will not display itself by default. When you want another form to appear, you write code to load and display it (using the **Show** or **ShowDialog** methods). Similarly, when you want a form to disappear, you write code to unload or hide it (using **Close** and **Hide** methods).

There are two ways to display a form (here, named **frmExample**). First, you must create the form you are displaying. This is done by declaring and constructing a new ‘instance’ of the form:

Form myForm = new frmExample();

It can then be displayed as a modal form:

myForm.ShowDialog(); // Modal display

or as a modeless form:

myForm.Show();

In **modal** display, no other open window in the application may be accessed as long as the modal window is open (works like a message box). In **modeless** display, any other open application window may be clicked and made active. Modeless forms are harder to program, because users can access them in an unpredictable order.

To make a form disappear, we can use the **Close** method (removes form from memory) or the **Hide** method (sets the **Visible** property to **False**). To close the current form, use:

this.Close();

To hide the form, use:

this.Hide();

The last consideration in a multiple form application is to decide how (if needed) to transfer information from one form to another. There are several ways to do this. We will look at transferring information using the **form constructor**. Above we said to construct an instance of **frmExample**, we could use:

Form myForm = new frmExample();

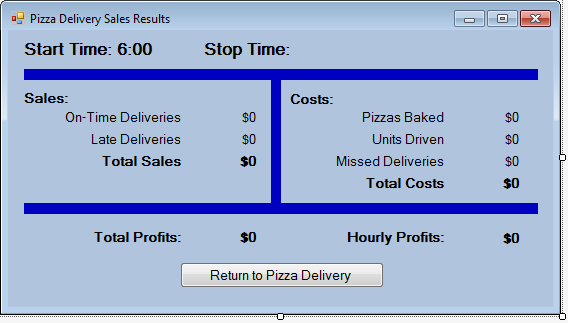
To transfer information to this new form, we will not use this code, but replace it with an **overloaded** constructor for **frmExample**:

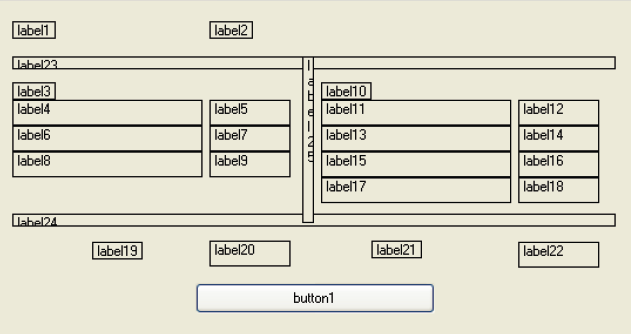
Form myForm = new frmExample(variables);

where **variables** is a comma-delimited list of variables to transfer to **frmExample**. An overloaded constructor gives us an alternate way to create an object. Once created, **myForm** can access all the information provided by **variables**. We’ll see how to do this with the **Pizza Delivery** results. You’ll see it’s a pretty easy modification.

**Form Design – Sales Results**

To add a new, blank form to our project, select **Add Windows Form** under the **Projects** menu. Name the file **SalesResults**. We do not have to change the startup form (it will remain as **frmPizzaDelivery**). The form will hold several label controls for display and a single button control to close the form when done viewing the results.





Add 25 label controls and one button control to the form.

For the label controls, set **AutoSize** to **False.**

The names I used are as follows:

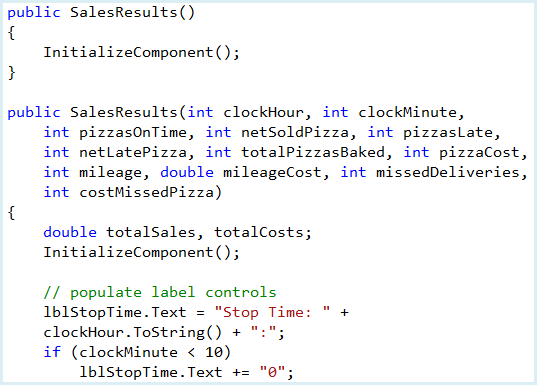
|  |  |
| --- | --- |
| **label1** Label: Name lblStartTime  **label2** Label: Name lblStopTime  **label3** Label: Name lblSalesHeader  **label4** Label: Name lblOnTime  **label5** Label: Name lblOnTimeSales  **label6** Label: Name lblLate  **label7** Label: Name lblLateSales  **label8** Label: Name lblTotalSales  **label9** Label: Name lblSales  **label10** Label: Name lblCostsHeader  **label11** Label: Name lblBaked | **label12** Label: Name lblBakedCosts  **label13** Label: Name lblMiles  **label14** Label: Name lblMilesCosts  **label15** Label: Name lblMissed  **label16** Label: Name lblMissedCosts  **label17** Label: Name lblTotalCosts  **label18** Label: Name lblCosts  **label19** Label: Name lblTotalProfits  **label20** Label: Name lblProfits  **label21** Label: Name lblHourlyProfits  **label22** Label: Name lblHourly |

The code to use this form is relatively simple. We will build an overloaded constructor to pass in the information needed to populate the various label controls. And, we’ll add one line of code to the results form’s **btnReturn Click** event to close it. Lastly, we will place code in the main form (**frmPizzaDelivery**) **btnExitStop Click** event to display **frmDeliveryResults** (and pass it the needed information).

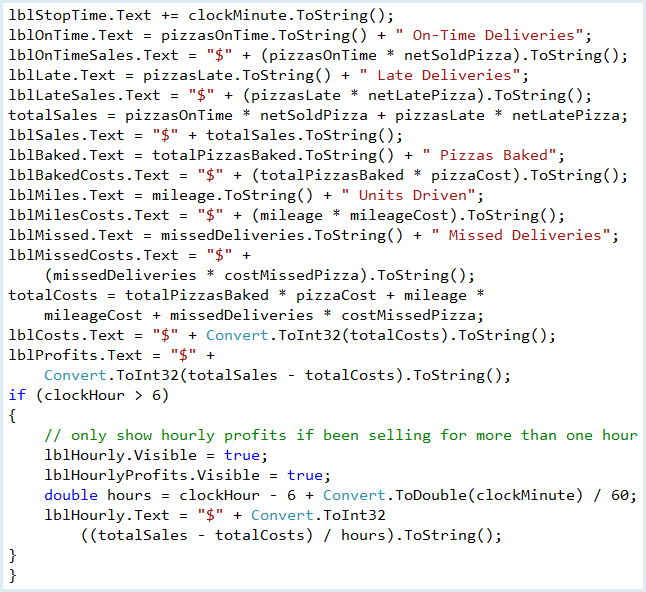
**Code Design – Sales Results**

To fill all the label controls on the sales results form, we need a bunch of variables from **frmPizzaDelivery**:

We will build an overloadedconstructor that passes this information into the form and displays it in the proper label controls. Open the code window for **frmSalesResults** and add this constructor under the default constructor:

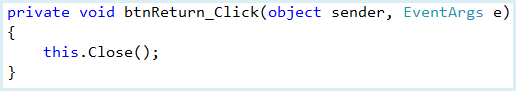


Code continued below

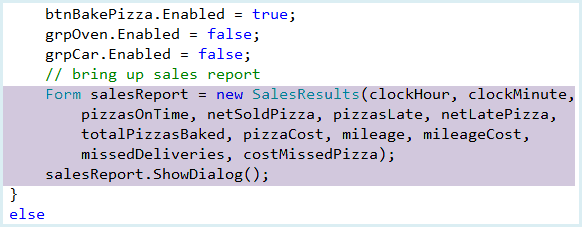
****

The code in this constructor is fairly simple. It sets the **Text** properties for the various label controls on **frmSalesResults**. You may wonder if we now have two constructors (the **default** constructor or the new **overloaded** constructor) for **frmSalesResults**, how does Visual C# know which constructor to use? With overloaded constructors (or overloaded methods), the argument list indicates which form to use. For **frmSalesResults**, if the argument list is empty, the default constructor. Otherwise, if the argument list contains all of the variables needed to fill the label controls, the overloaded version is used.

While the sales results form is displayed, add the single line to the **btnReturn Click** event method to close the form:

****

Now, return to the code window for the **Pizza Delivery** form. Add the shaded code to the **btnExitStop Click** event method:



The new code creates an instance of the results form (using our new constructor). It then displays the results form.

Save and run the project one last time. It is now complete.

You should be able to play a full game of **Pizza Delivery** and receive a sales report.