Name: Date:

C# Lab

The Quest

This lab gives you a spec that describes a program for you to build, using the knowledge you've gained over the last few chapters.

This project is bigger than the ones you've seen so far. So read the whole thing before you get started, and give yourself a little time. And don't worry if you get stuck—there's nothing new in here, so you can move on in the book and come back to the lab later.

We've filled in a few design details for you, and we've made sure you've got all the pieces you need...and nothing else.

It's up to you to finish the job. You can download an executable for this lab from the website...but we won't give you the code for the answer.

The spec: build an adventure game

Your job is to build an adventure game where a mighty adventurer is on a quest to defeat level after level of deadly enemies. You'll build a **turn-based system**, which means the player makes one move and then the enemies make one move. The player can move ***or*** attack, and then each enemy gets a chance to move ***and*** attack. The game keeps going until the player either defeats all the enemies on all seven levels or dies.

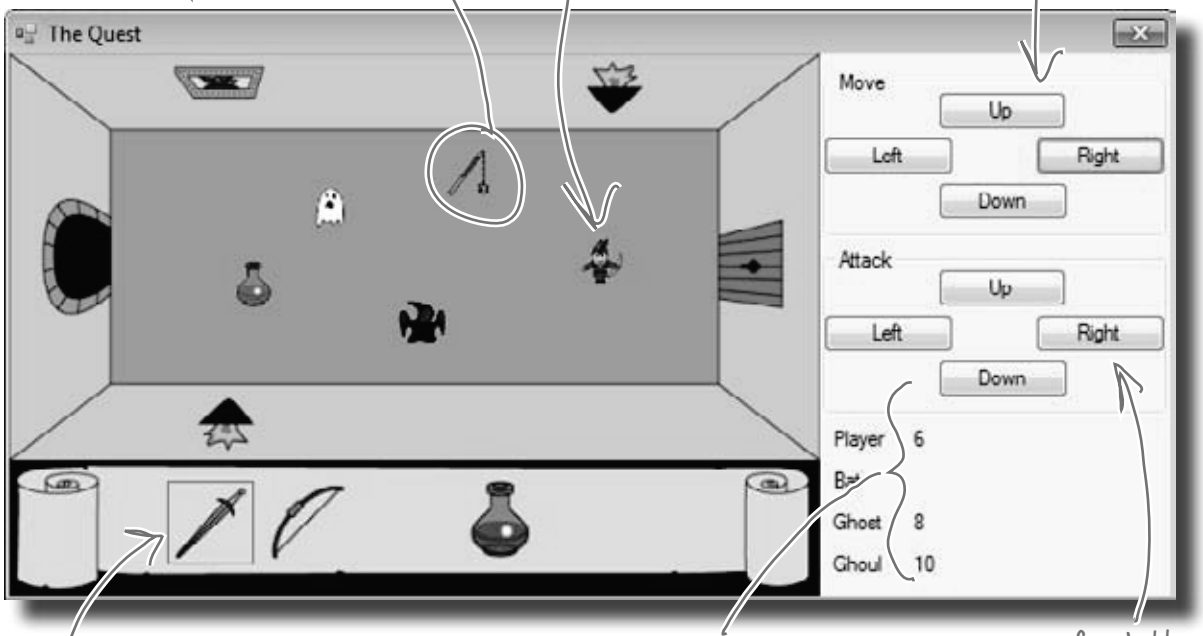
The enemies get a bit of an advantage—they move every turn, and after they move they'll attack the player if he's in range.

The game window gives an overhead view of the dungeon where the player fights his enemies.

The player can pick up weapons and potions along the way.

The player and enemies move around in the dungeon.

The player moves using the four Move buttons.



Here's the player's inventory. It shows what items the player's picked up, and draws a box around the item that they're currently using. The player clicks on an item to equip it, and uses the Attack button to use the item.

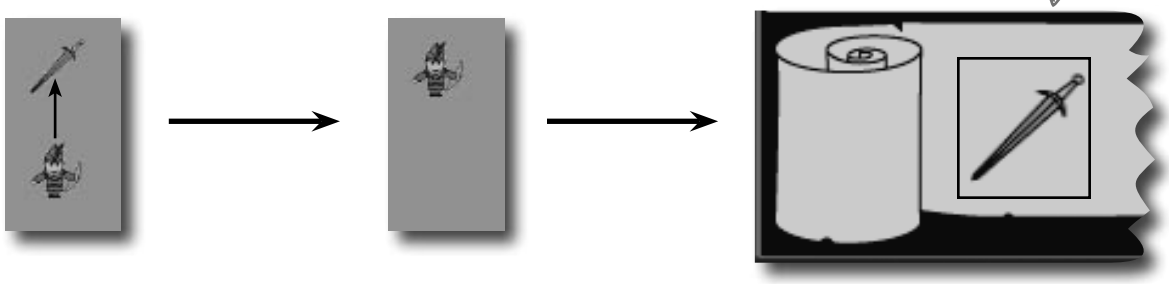
The game shows you the number of **hit points** for the player and enemies. When the player attacks an enemy, the enemy's hit points go down. Once the hit points get down to zero, the enemy or player dies.

These four buttons are used to attack enemies and drink potions. (The player can use any of the buttons to drink a potion.)

The player picks up weapons...

There are weapons and potions scattered around the dungeon that the player can pick up and use to defeat his enemies. All he has to do is move onto a weapon, and it disappears from the floor and appears in his inventory.

A black box around a weapon means it's currently **equipped**. Different weapons work differently—they have different ranges, some only attack in one direction while others have a wider range, and they cause different levels of damage to the enemies they hit.

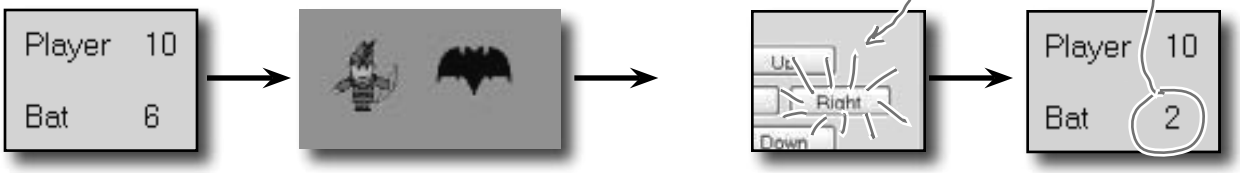


...and attacks enemies with them

Every level in the game has a weapon that the player can pick up and use to defeat his enemies. Once the weapon's picked up, it should disappear from the game floor.

The bat is to the right of the player, so he hits the Right attack button.

The attack causes the bat's hit points to drop, from 6 to 2 in this case.



Higher levels bring more enemies

There are three different kinds of enemies: a bat, a ghost, and a ghoul. The first level has only a bat. The seventh level is the last one, and it has all three enemies.

A ghoul moves quickly toward the player, and causes heavy damage when it attacks.

The bat flies around somewhat randomly. When it's near the player, it causes a small amount of damage.

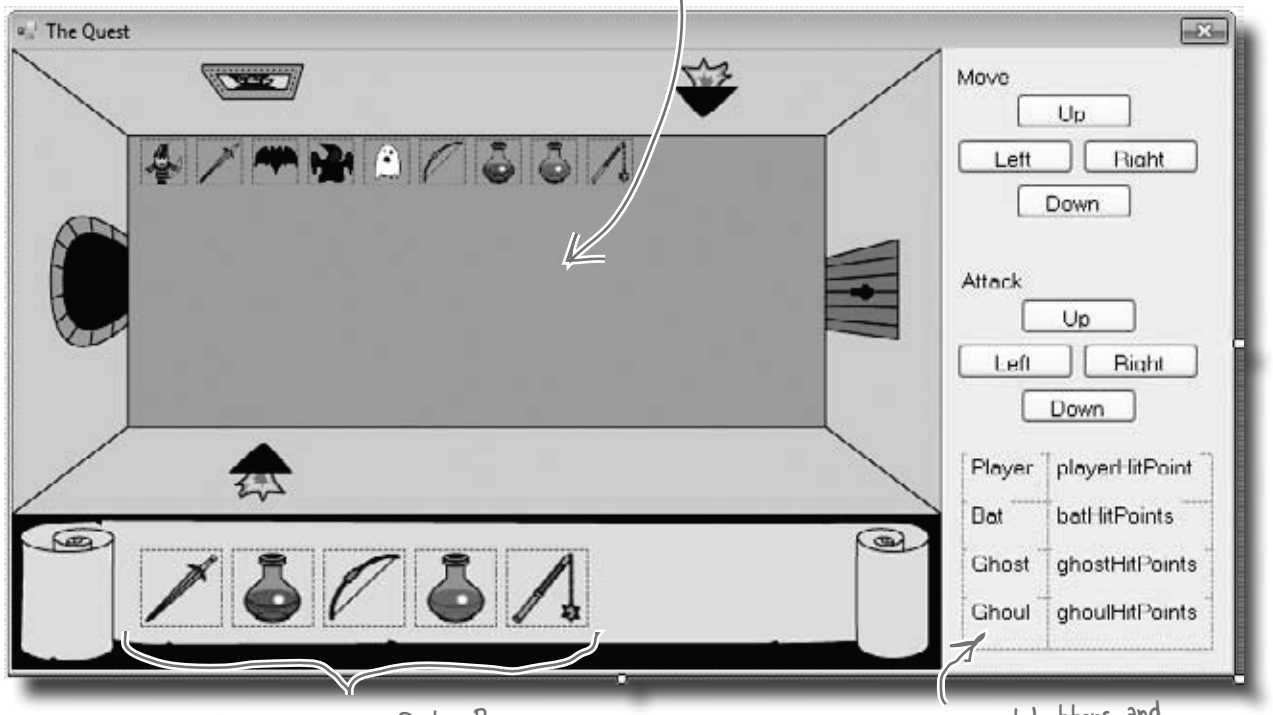
The ghost moves slowly toward the player. As soon as it's close to the player, it attacks and causes a medium amount of damage.

The design: building the form

The form gives the game its unique look. Use the form's `BackgroundImage` property to display the image of the dungeon and the inventory, and a series of `PictureBox` controls to show the player, weapons, and enemies in the dungeon. You'll use a `TableLayoutPanel` control to display the hit points for the player, bat, ghost, and ghoul as well as the buttons for moving and attacking.

The dungeon itself is a static image, displayed using the BackgroundImage property of the form.

Make sure the BackgroundImageLayout property is set to None.



Each of these icons is a PictureBox.

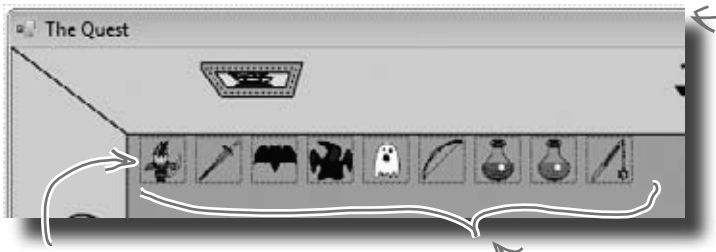
Hit points, movement buttons, and attack buttons are all displayed in a TableLayoutPanel.

Download the background image and the graphics for the weapons, enemies, and player from the Head First Labs website: www.headfirstlabs.com/books/hfcsharp

Everything in the dungeon is a PictureBox

Players, weapons, and enemies should all be represented by icons. Add nine `PictureBox` controls, and set their `Visible` properties to `False`. Then, your game can move around the controls, and toggle their `Visible` properties as needed.

You can set a PictureBox's BackColor property to Color.Transparent to let the form's background picture or color show through any transparent pixels in the picture.



Add nine PictureBox controls to the dungeon. Use the Size property to make each one 30x30. It doesn't matter where you place them—the form will move them around. Use the little black arrow that shows up when you click on the PictureBox to set each to one of the images from the Head First Labs website.

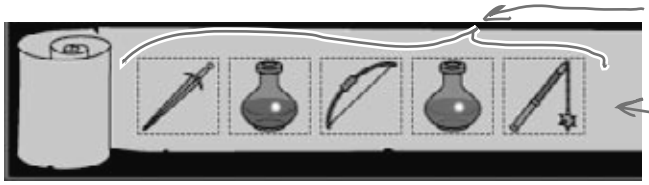
After you've added the nine PictureBox controls, right-click on the player's icon and select "Bring to Front", then send the three weapon icons to the back. That ensures player icons stay "above" any items that are picked up.

Controls overlap each other in the IDE, so the form needs to know which ones are in front, and which are in back. That's what the "Bring to Front" and "Send to Back" form designer commands do.

The inventory contains PictureBox controls, too

You can represent the inventory of the player as five 50×50 `PictureBox` controls. Set the `BackColor` property of each to **`Color.Transparent`** (if you use the Properties window to set the property, just type it into the `BackColor` row). Since the picture files have a transparent background, you'll see the scroll and dungeon behind them:

You'll need five more 50x50 PictureBoxes for the inventory.

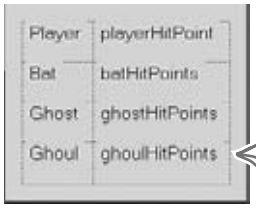


When the player equips one of the weapons, the form should set the BorderStyle of that weapon icon to FixedSingle and the rest of the icons' BorderStyle to None.

Build your stats window

The hit points are in a `TableLayoutPanel`, just like the attack and movement buttons. For the hit points, create two columns in the panel, and drag the column divider to the left a bit. Add four rows, each 25% height, and add in `Label` controls to each of the eight cells:

2 columns, 4 rows...8 cells for your hit point statistics.

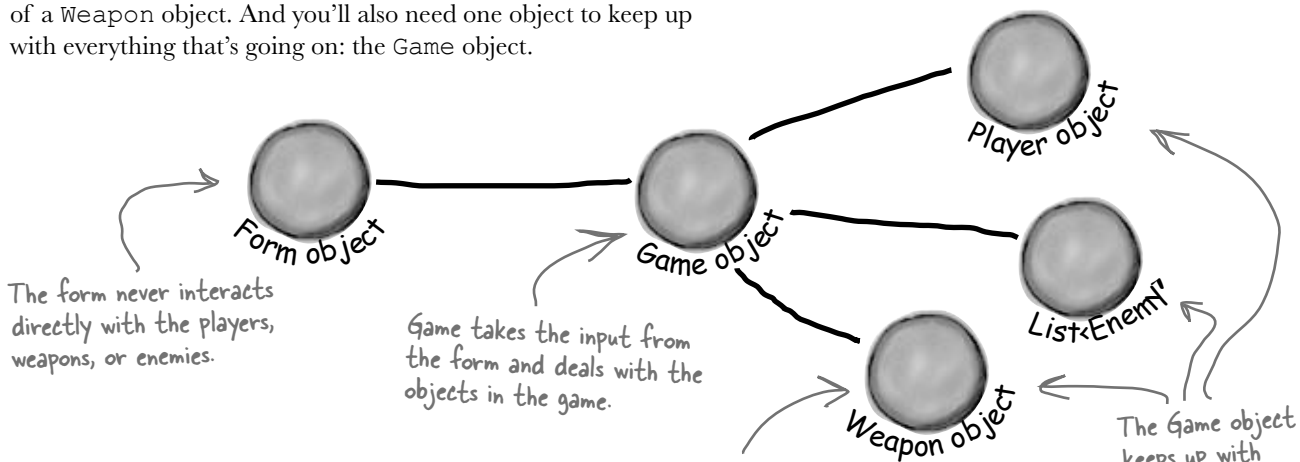


Each cell has a Label in it, and you can update those values during the game.

The architecture: using the objects

You'll need several types of objects in your game: a `Player` object, several subtypes of an `Enemy` object, and several sub-types of a `Weapon` object. And you'll also need one object to keep up with everything that's going on: the `Game` object.

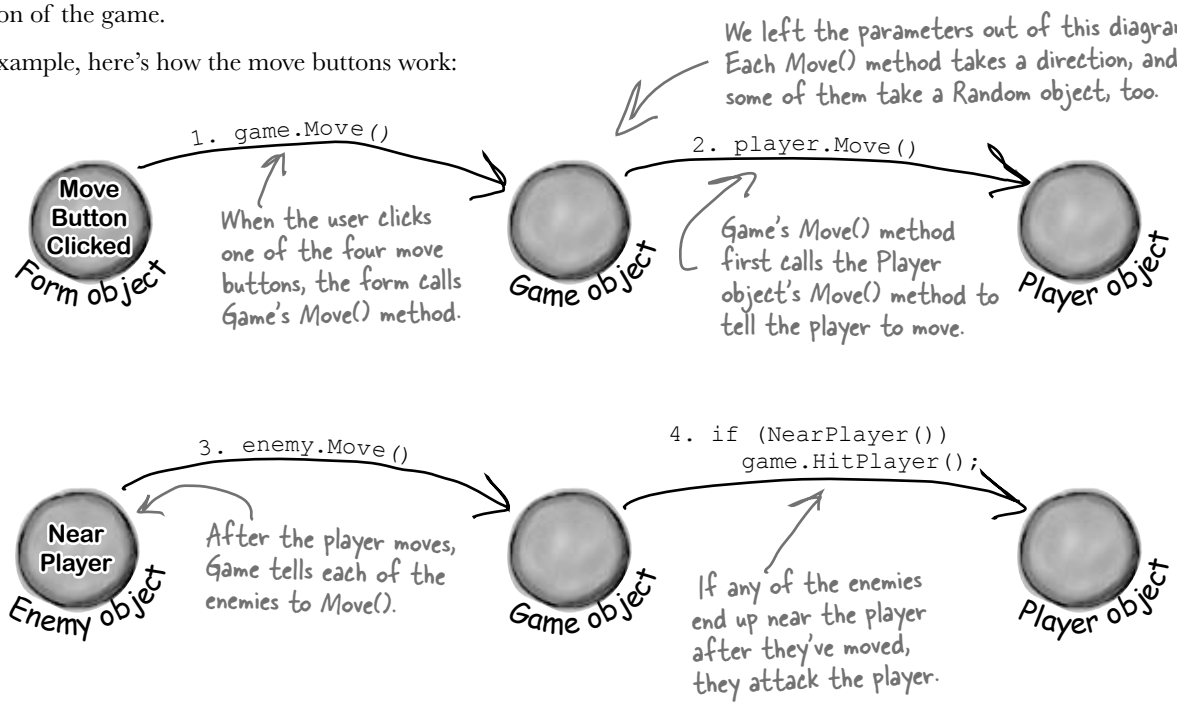
This is just the general overview. We'll give you **a lot more details** on how the player and enemies move, how the enemy figures out if it's near the player, etc.



The Game object handles turns

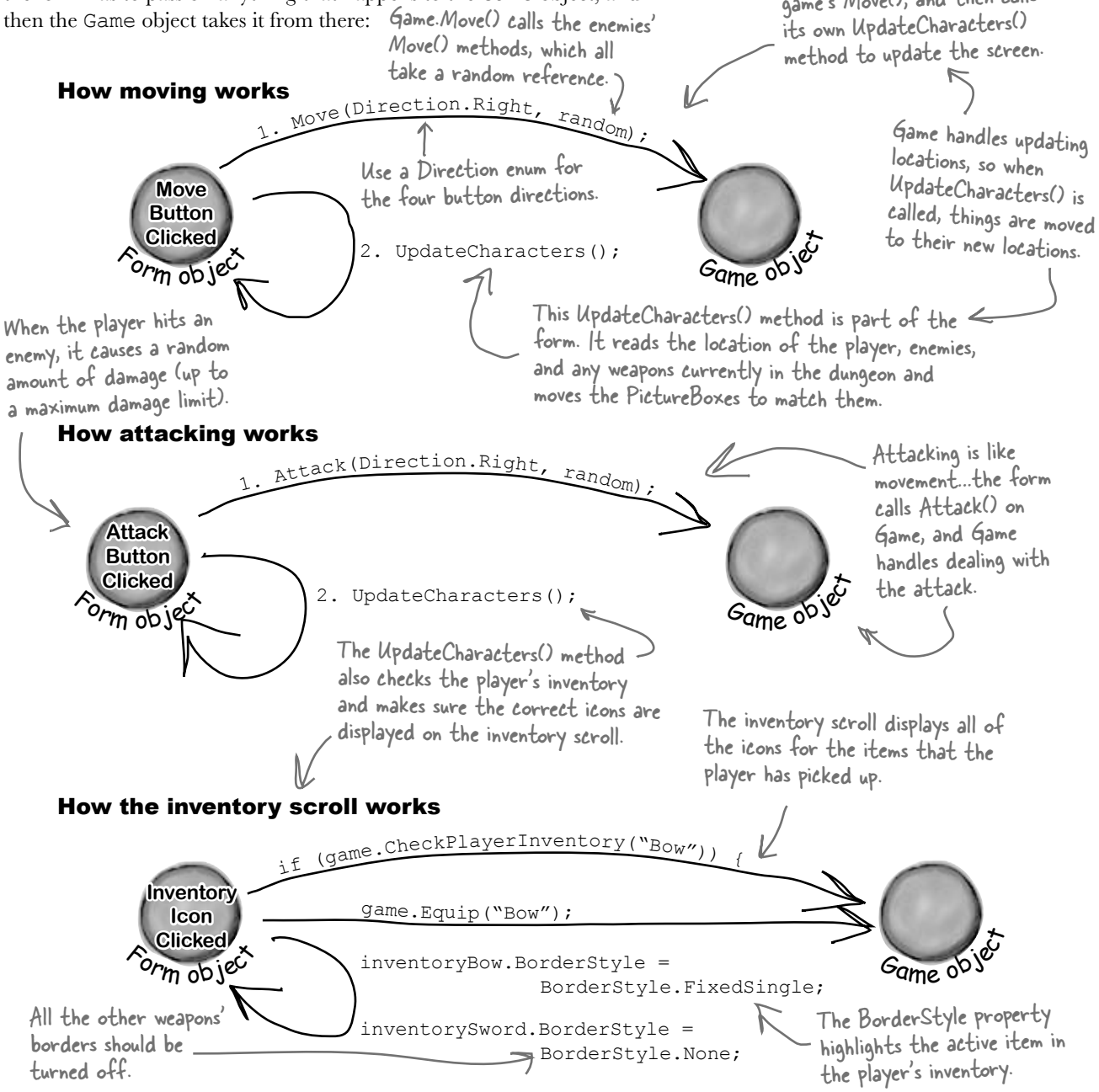
When one of your form's move buttons is clicked, the form will call the `Game` object's `Move()` method. That method will let the player take a turn, and then let all the enemies move. So it's up to `Game` to handle the turn-based movement portion of the game.

For example, here's how the move buttons work:



The form delegates activity to the Game object

Movement, attacking, and inventory all begin in the form. So clicking a movement or attack button, or an item in inventory, triggers code in your form. But it's the `Game` object that controls the objects in the game. So the form has to pass on anything that happens to the `Game` object, and then the `Game` object takes it from there:



Building the Game class

We've gotten you started with the Game class in the code below. There's a lot for you to do—so read through this code carefully, get it into the IDE, and get ready to go to work:

```
using System.Drawing;

class Game {
  public List<Enemy> Enemies;
  public Weapon WeaponInRoom;

  private Player player;
  public Point PlayerLocation { get { return player.Location; } }
  public int PlayerHitPoints { get { return player.HitPoints; } }
  public List<string> PlayerWeapons { get { return player.Weapons; } }

  private int level = 0;
  public int Level { get { return level; } }

  private Rectangle boundaries;
  public Rectangle Boundaries { get { return boundaries; } }

  public Game(Rectangle boundaries) {
    this.boundaries = boundaries;
    player = new Player(this,
      new Point(boundaries.Left + 10, boundaries.Top + 70));
  }
  public void Move(Direction direction, Random random) {
    player.Move(direction);
    foreach (Enemy enemy in Enemies)
      enemy.Move(random);
  }
  public void Equip(string weaponName) {
    player.Equip(weaponName);
  }
  public bool CheckPlayerInventory(string weaponName) {
    return player.Weapons.Contains(weaponName);
  }
  public void HitPlayer(int maxDamage, Random random) {
    player.Hit(maxDamage, random);
  }
```

You'll need Rectangle and Point from System.Drawing, so be sure to add this to the top of your class.

These are OK as public properties if Enemy and Weapon are well encapsulated...in other words, just make sure the form can't do anything inappropriate with them.

The game keeps a private Player object. The form will only interact with this through methods on Game, rather than directly.

The Rectangle object has Top, Bottom, Left, and Right fields, and works perfectly for the overall game area.

Game starts out with a bounding box for the dungeon, and creates a new Player object in the dungeon.

Movement is simple: move the player in the direction the form gives us, and move each enemy in a random direction.

These are all great examples of encapsulation.... Game doesn't know how Player handles these actions, it just passes on the needed information and lets Player do the rest.

```
    public void IncreasePlayerHealth(int health, Random random) {
      player.IncreaseHealth(health, random);
    }

    public void Attack(Direction direction, Random random) {
      player.Attack(direction, random);
      foreach (Enemy enemy in Enemies)
        enemy.Move(random);
    }

    private Point GetRandomLocation(Random random) {
      return new Point(boundaries.Left +
        random.Next(boundaries.Right / 10 - boundaries.Left / 10) * 10,
            boundaries.Top +
        random.Next(boundaries.Bottom / 10 - boundaries.Top / 10) * 10);
    }

    public void NewLevel(Random random) {
      level++;
      switch (level) {
        case 1:
          Enemies = new List<Enemy>();
          Enemies.Add(new Bat(this, GetRandomLocation(random)));
          WeaponInRoom = new Sword(this, GetRandomLocation(random));
          break;
      }
    }
}
```

Attack() is almost exactly like Move(). The player attacks, and the enemies all get a turn to move.

GetRandomLocation() will come in handy in the NewLevel() method, which will use it to determine where to place enemies and weapons.

This is just a math trick to get a random location within the rectangle that represents the dungeon area.

We only added the case for Level 1. It's your job to add cases for the other levels.

We've only got room in the inventory for one blue potion and one red potion. So if the player already has a red potion, then the game shouldn't add a red potion to the level (and the same goes for the blue potion).

Finish the rest of the levels

It's your job to finish the `NewLevel()` method. Here's the breakdown for each level:

Level	Enemies	Weapons
2	Ghost	Blue potion
3	Ghoul	Bow
4	Bat, Ghost	Bow, if not picked up on 3; otherwise, blue potion
5	Bat, Ghoul	Red potion
6	Ghost, Ghoul	Mace
7	Bat, Ghost, Ghoul	Mace, if not picked up on 6; otherwise, red potion
8	N/A	N/A – end the game with Application.Exit()

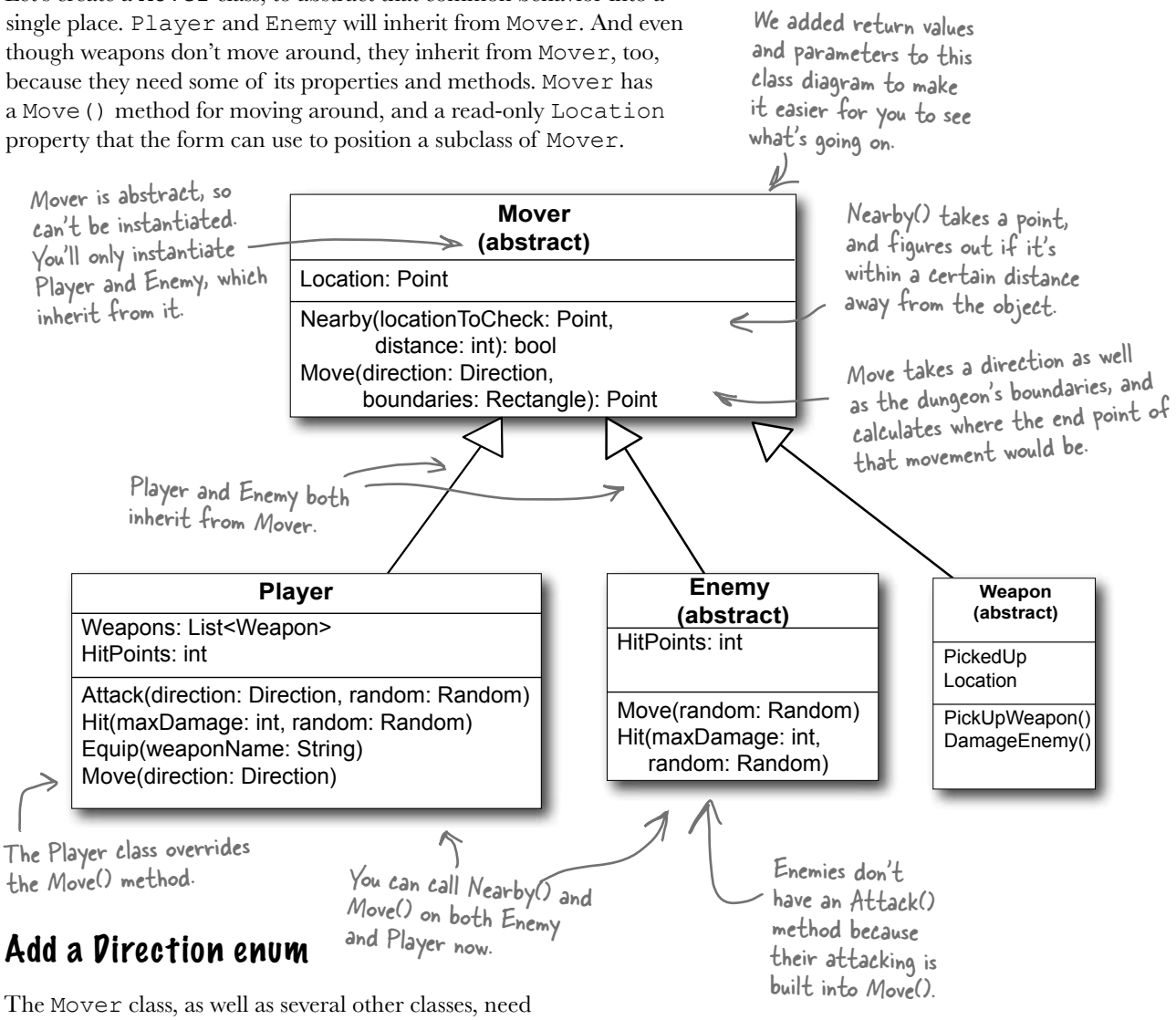
So if the blue potion is still in the player's inventory from Level 2, nothing appears on this level.

This only appears if the red potion from Level 5 has already been used up.

Finding common behavior: movement

You already know that duplicate code is bad, and duplicate code usually shows up when two or more objects share the same behavior. That's the case in the dungeon game, too…both enemies and players move.

Let's create a `Mover` class, to abstract that common behavior into a single place. `Player` and `Enemy` will inherit from `Mover`. And even though weapons don't move around, they inherit from `Mover`, too, because they need some of its properties and methods. `Mover` has a `Move()` method for moving around, and a read-only `Location` property that the form can use to position a subclass of `Mover`.



Add a Direction enum

The `Mover` class, as well as several other classes, need a `Direction` enum. Create this enum, and give it four enumerated values: `Up`, `Down`, `Left`, and `Right`.

The Mover class source code

Here's the code for Mover:

```
abstract class Mover {
  private const int MoveInterval = 10;
  protected Point location;
  public Point Location { get { return location; } }
  protected Game game;

  public Mover(Game game, Point location) {
    this.game = game;
    this.location = location;
  }

  public bool Nearby(Point locationToCheck, int distance) {
    if (Math.Abs(location.X - locationToCheck.X) < distance &&
       (Math.Abs(location.Y - locationToCheck.Y) < distance)) {
      return true;
    } else {
      return false;
    }
  }
  public Point Move(Direction direction, Rectangle boundaries) {
    Point newLocation = location;
    switch (direction) {
      case Direction.Up:
        if (newLocation.Y - MoveInterval >= boundaries.Top)
          newLocation.Y -= MoveInterval;
        break;
      case Direction.Down:
        if (newLocation.Y + MoveInterval <= boundaries.Bottom)
          newLocation.Y += MoveInterval;
        break;
      case Direction.Left:
        if (newLocation.X - MoveInterval >= boundaries.Left)
          newLocation.X -= MoveInterval;
        break;
      case Direction.Right:
        if (newLocation.X + MoveInterval <= boundaries.Right)
          newLocation.X += MoveInterval;
        break;
      default: break;
    }
    return newLocation;
  }
}
```

Since protected properties are only available to subclasses, the form object can't set the location...only read it through the public get method we define.

Instances of Mover take in the Game object and a current location.

The Nearby method checks a Point against this object's current location. If they're within distance of each other, then it returns true; otherwise, it returns false.

The Move() method tries to move one step in a direction. If it can, it returns the new Point. If it hits a boundary, it returns the original Point.

If the end location is outside the boundaries, the new location stays the same as the starting point.

Finally, this new location is returned (which might still be the same as the starting location!).

The Player class keeps track of the player

Here's a start on the `Player` class. Start with this code in the IDE, and then get ready to add to it.

The Player and Enemy objects need to stay inside the dungeon, which means they need to know the boundaries of the playing area. Use the Contains() method of the boundaries Rectangle to make sure they don't move out of bounds.

All of the properties of Player are hidden from direct access.

```
class Player : Mover {
  private Weapon equippedWeapon;
  private int hitPoints;
  public int HitPoints { get { return hitPoints; } }

  private List<Weapon> inventory = new List<Weapon>();
  public List<string> Weapons {
    get {
      List<string> names = new List<string>();
      foreach (Weapon weapon in inventory)
        names.Add(weapon.Name);
      return names;
    }
  }

  public Player(Game game, Point location);
         : base(game, location) {
    hitPoints = 10;
  }

  public void Hit(int maxDamage, Random random) {
    hitPoints -= random.Next(1, maxDamage);
  }

  public void IncreaseHealth(int health, Random random) {
    hitPoints += random.Next(1, health);
  }

  public void Equip(string weaponName) {
    foreach (Weapon weapon in inventory) {
      if (weapon.Name == weaponName)
        equippedWeapon = weapon;
    }
  }
}
```

A Player can hold multiple weapons in inventory, but can only equip one at a time.

Player inherits from Mover, so this passes in the Game and location to that base class.

The player's constructor sets its hitPoints to 10 and then calls the base class constructor.

When an enemy hits the player, it causes a random amount of damage. And when a potion increases the player's health, it increases it by a random amount.

The Equip() method tells the player to equip one of his weapons. The Game object calls this method when one of the inventory icons is clicked.

A Player object can only have one Weapon object equipped at a time.

Even though potions help the player rather than hurt the enemy, they're still **considered weapons** by the game. That way the inventory can be a List<Weapon>, and the game can point to one with its WeaponInRoom reference.

Write the Move() method for the Player

`Game` calls the `Player`'s `Move()` method to tell a player to move in a certain direction. `Move()` takes the direction to move as an argument (using the `Direction` enum you should have already added). Here's the start of that method:

This happens when one of the movement buttons on the form is clicked.

```
public void Move(Direction direction) {
  base.location = Move(direction, game.Boundaries);
  if (!game.WeaponInRoom.PickedUp) {
    // see if the weapon is nearby, and possibly pick it up
  }
}
```

Move is in the Mover base class.

When the player picks up a weapon, it needs to disappear from the dungeon and appear in the inventory.

You've got to fill in the rest of this method. Check and see if the weapon is near the player (within a single unit of distance). If so, pick up the weapon and add it to the player's inventory.

The Weapon and form will handle making the weapon's PictureBox invisible when the player picks it up... that's not the job of the Player class.

If the weapon is the only weapon the player has, go ahead and equip it immediately. That way, the player can use it right away, on the next turn.

Add an Attack() method, too

The weapons all have an Attack() method that takes a Direction enum and a Random object. The player's Attack() will figure out which weapon is equipped and call its Attack().

Next up is the `Attack()` method. This is called when one of the form's attack buttons is clicked, and carries with it a direction (again, from the `Direction` enum). Here's the method signature:

If the weapon is a potion, then Attack() removes it from the inventory after the player drinks it.

```
public void Attack(Direction direction, Random random) {
  // Your code goes here
}
```

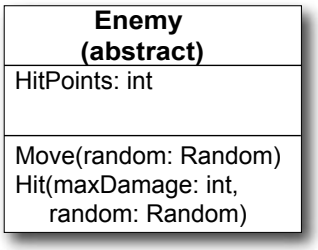
If the player doesn't have an equipped weapon, this method won't do anything. If the player does have an equipped weapon, this should call the weapon's `Attack()` method.

But potions are a special case. If a potion is used, remove it from the player's inventory, since it's not available anymore.

Potions will implement an IPotion interface (more on that in a minute), so you can use the "is" keyword to see if a Weapon is an implementation of IPotion.

Bats, ghosts, and ghouls inherit from the Enemy class

We'll give you another useful abstract class: `Enemy`. Each different sort of enemy has its own class that inherits from the `Enemy` class. The different kinds of enemies move in different ways, so the `Enemy` abstract class leaves the `Move` method as an abstract method—the three enemy classes will need to implement it differently, depending on how they move.




```
abstract class Enemy : Mover {
  private const int NearPlayerDistance = 25;
  private int hitPoints;
  public int HitPoints { get { return hitPoints; } }
  public bool Dead { get {
       if (hitPoints <= 0) return true;
       else return false;
     }
  }
  public Enemy(Game game, Point location, int hitPoints)
       : base(game, location) { this.hitPoints = hitPoints; }

  public abstract void Move(Random random);

  public void Hit(int maxDamage, Random random) {
     hitPoints -= random.Next(1, maxDamage);
  }

  protected bool NearPlayer() {
    return (Nearby(game.PlayerLocation,
                   NearPlayerDistance));
  }
  protected Direction FindPlayerDirection(Point playerLocation) {
    Direction directionToMove;
       if (playerLocation.X > location.X + 10)
          directionToMove = Direction.Right;
    else if (playerLocation.X < location.X - 10)
          directionToMove = Direction.Left;
    else if (playerLocation.Y < location.Y - 10)
          directionToMove = Direction.Up;
    else
          directionToMove = Direction.Down;
     return directionToMove;
  }
}
```

The form can use this read-only property to see if the enemy should be visible in the game dungeon.

Each subclass of Enemy implements this.

When the player attacks an enemy, it calls the enemy's Hit() method, which subtracts a random number from the hit points.

The Enemy class inherited the Nearby() method from Mover, which it can use to figure out whether it's near the player.

If you feed FindPlayerDirection() the player's location, it'll use the base class's location field to figure out where the player is in relation to the enemy and return a Direction enum that tells you in which direction the enemy needs to move in order to move toward the player.

Write the different Enemy subclasses

The three `Enemy` subclasses are pretty straightforward. Each enemy has a different number of starting hit points, moves differently, and does a different amount of damage when it attacks. You'll need to have each one pass a different `startingHitPoints` parameter to the `Enemy` base constructor, and you'll have to write different `Move()` methods for each subclass.

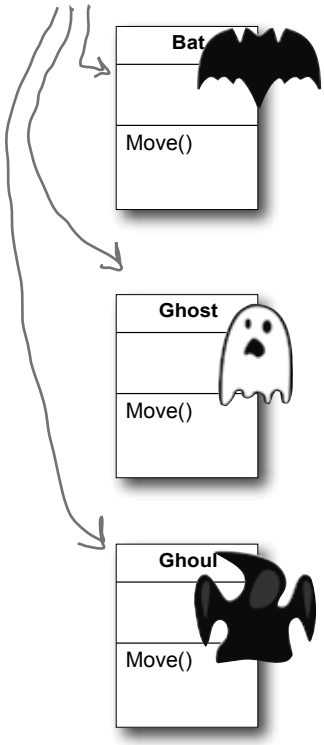
Here's an example of how one of those classes might look:

```
class Bat : Enemy {
    public Bat(Game game, Point location)
        : base(game, location, 6)
    { }
    public override void Move(Random random) {
        // Your code will go here
    }
}
```

The bat starts with 6 hit points, so it passes 6 to the base class constructor.

You probably won't need any constructor for these; the base class handles everything.

Each of these subclasses the Enemy base class, which in turn subclasses Mover.



The bat flies around somewhat randomly, so it uses Random to fly in a random direction half the time.

Once an enemy has no more hit points, the form will no longer display it. But it'll still be in the game's Enemies list until the player finishes the level.

The bat starts with 6 hit points. It'll keep moving toward the player and attacking **as long as it has one or more hit points.** When it moves, there's a 50% chance that it'll move toward the player, and a 50% chance that it'll move in a random direction. After the bat moves, it checks if it's near the player—if it is, then it attacks the player with up to 2 hit points of damage.

We'll have to make sure the form sees if an enemy should be visible at every turn.

The ghost is harder to defeat than the bat, but like the bat, it will only move and attack if its hit points are greater than zero. It starts with 8 hit points. When it moves, there's a 1 in 3 chance that it'll move toward the player, and a 2 in 3 chance that it'll stand still. If it's near the player, it attacks the player with up to 3 hit points of damage.

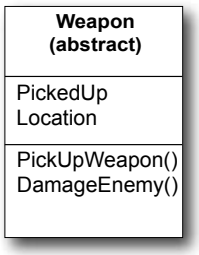
The ghost and ghoul use Random to make them move more slowly than the player.

The ghoul is the toughest enemy. It starts with 10 hit points, and only moves and attacks if its hit points are greater than zero. When it moves, there's a 2 in 3 chance that it'll move toward the player, and a 1 in 3 chance that it'll stand still. If it's near the player, it attacks the player with up to 4 hit points of damage.

Weapon inherits from Mover, each weapon inherits from Weapon

We need a base `Weapon` class, just like we had a base `Enemy` class. And each weapon has a location, as well as a property indicating whether or not it's been picked up. Here's the base `Weapon` class:

Weapon inherits from Mover because it uses its Nearby() and Move() methods in DamageEnemy().




```
abstract class Weapon : Mover {

  protected Game game;
  private bool pickedUp;
  public bool PickedUp { get { return pickedUp; } }
  private Point location;
  public Point Location { get { return location; } }

  public Weapon(Game game, Point location) {
    this.game = game;
    this.location = location;
    pickedUp = false;
  }

  public void PickUpWeapon() { pickedUp = true; }

  public abstract string Name { get; }

  public abstract void Attack(Direction direction, Random random);

  protected bool DamageEnemy(Direction direction, int radius,
                             int damage, Random random) {
    Point target = game.PlayerLocation;
    for (int distance = 0; distance < radius; distance++) {
      foreach (Enemy enemy in game.Enemies) {
        if (Nearby(enemy.Location, target, radius)) {
          enemy.Hit(damage, random);
          return true;
        }
      }
      target = Move(direction, target, game.Boundaries);
    }
    return false;
  }
}
```

A pickedUp weapon shouldn't be displayed anymore...the form can use this get accessor to figure that out.

Every weapon has a location in the game dungeon.

The constructor sets the game and location fields, and sets pickedUp to false (because it hasn't been picked up yet).

Each weapon class needs to implement a Name property and an Attack() method that determines how that weapon attacks.

Each weapon's Name property returns its name ("Sword", "Mace", "Bow").

Each weapon has a different range and pattern of attack, so the weapons implement the Attack() method differently.

The Nearby() method in the Mover class only takes two parameters, a Point and an int, and it compares the Point to the Mover field location. You'll need to add an overloaded Nearby() that's almost identical, except that it takes three parameters, two Points and a distance, which compares the first Point to the second Point (instead of location).

The DamageEnemy() method is called by Attack(). It attempts to find an enemy in a certain direction and radius. If it does, it calls the enemy's Hit() method and returns true. If no enemy's found, it returns false.

Different weapons attack in different ways

Each subclass of `Weapon` has its own name and attack logistic. Your job is to implement these classes. Here's the basic skeleton for a `Weapon` subclass:

Each subclass represents one of the three weapons: a sword, bow, or mace.

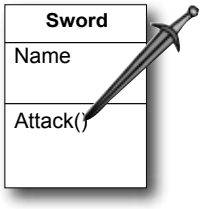
```
class Sword : Weapon {

  public Sword(Game game, Point location)
      : base(game, location) { }

  public override string Name { get { return "Sword"; } }

  public override void Attack(Direction direction, Random random) {
    // Your code goes here
  }
}
```

Each subclass relies on the base class to do the initialization work.

You're basically hardcoding in the name of each weapon.

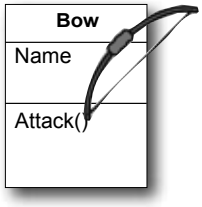
The Game object will pass on the direction to attack in.

The player can use the weapons over and over—they never get dropped or used up.

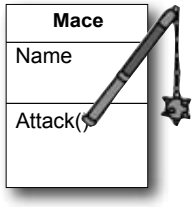


The sword is the first weapon the player picks up. It's got a wide angle of attack: if he attacks up, then it first tries to attack an enemy that's in that direction—if there's no enemy there, it looks in the direction that's clockwise from the original attack and attacks any enemy there, and if it still fails to hit then it attempts to attack an enemy counterclockwise from the original direction of attack. It's got a radius of 10, and causes 3 points of damage.

Think carefully about this...what is to the right of the direction left? What is to the left of up?



The bow has a very narrow angle of attack, but it's got a very long range—it's got an attack radius of 30, but only causes 1 point of damage. Unlike the sword, which attacks in three directions (because the player swings it in a wide arc), when the player shoots the bow in a direction, it only shoots in that one direction.



The mace is the most powerful weapon in the dungeon. It doesn't matter in which direction the player attacks with it—since he swings it in a full circle, it'll attack any enemy within a radius of 20 and cause up to 6 points of damage.

The different weapons will call DamageEnemy() in various ways. The Mace attacks in all directions, so if the player's attacking to the right, it'll call DamageEnemy(Direction.Right, 20, 6, random). If that didn't hit an enemy, it'll attack Up. If there's no enemy there, it'll try Left, then Down—that makes it swing in a full circle.

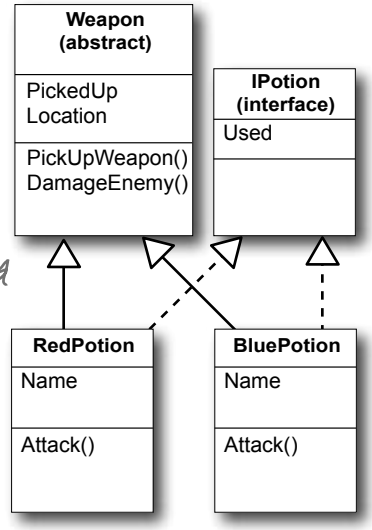
Potions implement the IPotion interface

There are two potions, a blue potion and a red potion, which increase the player's health. They act just like weapons—the player picks them up in the dungeon, equips them by clicking on the inventory, and **uses them by clicking one of the attack buttons**. So it makes sense for them to inherit from the abstract `Weapon` class.

But potions act a little differently, too, so you'll need to add an `IPotion` interface so they can have extra behavior: increasing the player's health. The `IPotion` interface is really simple. Potions only need to add one read-only property called `Used` that returns `false` if the player hasn't used the potion, and `true` if he has. The form will use it to determine whether or not to display the potion in the inventory.

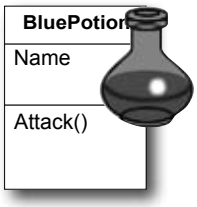
```
interface IPotion {
  bool Used { get; }
}
```

IPotion makes potions usable only once. It's also possible to find out if a Weapon is a potion with "if (weapon is IPotion)" because of this interface.



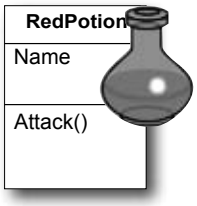
The potions inherit from the Weapon class because they're used just like weapons—the player clicks on the potion in the inventory scroll to equip it, and then clicks any of the attack buttons to use it.

You should be able to write these classes using this class diagram and the information below.



The `BluePotion` class's `Name` property should return the string "Blue Potion". Its `Attack()` method will be called when the player uses the blue potion—it should increase the player's health by up to 5 hit points by calling the `IncreasePlayerHealth()` method. After the player uses the potion, the potion's `Used()` method should return `true`.

If the player picks up a blue potion on level 2, uses it, and then picks up another one on level 4, the game will end up creating two different BluePotion instances.



The `RedPotion` class is very similar to `BluePotion`, except that its `Name` property returns the string "Red Potion", and its `Attack()` method increases the player's health by up to 10 hit points.

The form brings it all together

There's one instance of the `Game` object, and it lives as a private field of your form. It's created in the form's `Load` event, and the various event handlers in the form use the fields and methods on the `Game` object to keep the game play going.

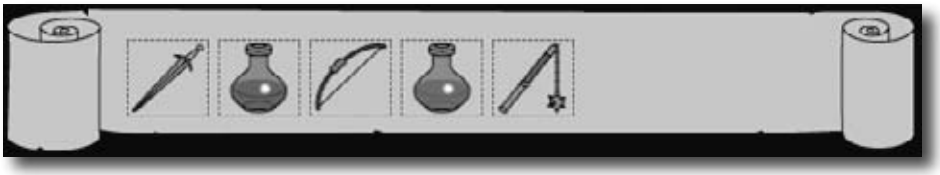
Everything begins with the form's `Load` event handler, which passes the `Game` a `Rectangle` that defines the boundaries of the dungeon play area. Here's some form code to get you going:

```
private Game game;
private Random random = new Random();
private void Form1_Load(object sender,
                        EventArgs e) {
    game = new Game(new Rectangle(78, 57, 420, 155));
    game.NewLevel(random);
    UpdateCharacters();
}
```

These are the boundaries of the dungeon in the background image you'll download and add to the form.

Using a Rectangle

You'll find a lot of Rectangles any time you work with forms. You can create one by passing it X, Y, Width, and Height values, or two Points (for opposite corners). Once you've got a rectangle instance, you can also access its Left, Right, Top, and Bottom, as well as its X, Y, Width, and Height values.



Remember to double-click on **each** PictureBox so the IDE adds a separate event handler method for each of them.

The form has a separate event handler for each of these `PictureBox`'s `Click` events. When the player clicks on the sword, it first checks to make sure the sword is in the player's inventory using the `Game` object's `CheckPlayerInventory()` method. If the player's holding the sword, the form calls `game.Equip()` to equip it. It then sets each `PictureBox`'s `BorderStyle` property to draw a box around the sword, and make sure none of the other icons has a box around it.

There's an event handler for each of the four movement buttons. They're pretty simple. First the button calls `game.Move()` with the appropriate `Direction` value, and then it calls the form's `UpdateCharacters()` method.

Make sure you change the buttons back when the player equips the sword, bow, or mace.

The four attack button event handlers are also really simple. Each button calls `game.Attack()`, and then calls the form's `UpdateCharacters()` method. If the player equips a potion, it's still used the same way—by calling `game.Attack()`—but potions have no direction. So make the Left, Right, and Down buttons invisible when the player equips a potion, and change the text on the Up button to say "Drink".

The form's UpdateCharacters() method moves the PictureBoxes into position

The last piece of the puzzle is the form's `UpdateCharacters()` method. Once all the objects have moved and acted on each other, the form updates everything…so weapons that been dropped have their `PictureBoxes`' `Visible` properties set to false, enemies and players are drawn in their new locations (and dead ones are made invisible), and inventory is updated.

Here's what you need to do:

1 **Update the player's position and stats**

The first thing you'll do is update the player's `PictureBox` location and the label that shows his hit points. Then you'll need a few variables to determine whether you've shown each of the various enemies.

```
public void UpdateCharacters() {
  Player.Location = game.PlayerLocation;
  playerHitPoints.Text =
    game.PlayerHitPoints.ToString();

  bool showBat = false;
  bool showGhost = false;
  bool showGhoul = false;
  int enemiesShown = 0;
  // more code to go here...
```

The showBat variable will be set to true if we made the bat's PictureBox visible. Same goes for showGhost and showGhoul.

2 **Update each enemy's location and hit points**

Each enemy could be in a new location and have a different set of hit points. You need to update each enemy after you've updated the player's location:

```
foreach (Enemy enemy in game.Enemies) {
  if (enemy is Bat) {
    bat.Location = enemy.Location;
    batHitPoints.Text = enemy.HitPoints.ToString();
    if (enemy.HitPoints > 0) {
      showBat = true;
      enemiesShown++;
    }
  }
  // etc...
```

This goes right after the code from above.

This will affect the visibility of the enemy PictureBox controls in just a bit.

You'll need two more if statements like this in your foreach loop—one for the ghost and one for the ghoul.

Once you've looped through all the enemies on the level, check the `showBat` variable. If the bat was killed, then `showBat` will still be false, so make its `PictureBox` invisible and clear its hit points label. Then do the same for `showGhost` and `showGhoul`.

3 Update the weapon PictureBoxes

Declare a `weaponControl` variable and use a big `switch` statement to set it equal to the `PictureBox` that corresponds to the weapon in the room.

```
sword.Visible = false;
bow.Visible = false;
redPotion.Visible = false;
bluePotion.Visible = false;
mace.Visible = false;
Control weaponControl = null;
switch (game.WeaponInRoom.Name) {
   case "Sword":
      weaponControl = sword; break;
```

Make sure your controls' names match these names. It's easy to end up with bugs that are difficult to track down if they don't match.

You'll have more cases for each weapon type.

The rest of the cases should set the variable `weaponControl` to the correct control on the form. After the `switch`, set `weaponControl.Visible` to `true` to display it.

4 Set the Visible property on each inventory icon PictureBox

Check the `Game` object's `CheckPlayerInventory()` method to figure out whether or not to display the various inventory icons.

5 Here's the rest of the method

The rest of the method does three things. First it checks to see if the player's already picked up the weapon in the room, so it knows whether or not to display it. Then it checks to see if the player died. And finally, it checks to see if the player's defeated all of the enemies. If he has, then the player advances to the next level.

```
weaponControl.Location = game.WeaponInRoom.Location;
if (game.WeaponInRoom.PickedUp) {
  weaponControl.Visible = false;
} else {
  weaponControl.Visible = true;
}
if (game.PlayerHitPoints <= 0) {
  MessageBox.Show("You died");
  Application.Exit();
}
if (enemiesShown < 1) {
  MessageBox.Show("You have defeated the enemies on this level");
  game.NewLevel(random);
  UpdateCharacters();
}
```

Every level has one weapon. If it's been picked up, we need to make its icon invisible.

Application.Exit() immediately quits the program. It's part of System.Windows.Forms, so you'll need the appropriate using statement if you want to use it outside of a form.

If there are no more enemies on the level, then the player's defeated them all and it's time to go to the next level.

The fun's just beginning!

Seven levels, three enemies…that's a pretty decent game. But you can make it even better. Here are a few ideas to get you started.…

Make the enemies smarter

Can you figure out how to change the enemies' Move() methods so that they're harder to defeat? Then see if you can change their constants to properties, and add a way to change them in the game.

Add more levels

The game doesn't have to end after seven levels. See if you can add more...can you figure out how to make the game go on indefinitely? If the player does win, make a cool ending animation with dancing ghosts and bats! And the game ends pretty abruptly if the player dies. Can you think of a more user-friendly ending? Maybe you can let the user restart the game or retry his last level.

Add different kinds of enemies

You don't need to limit the dangers to ghouls, ghosts, and bats. See if you can add more enemies to the game.

Add more weapons

The player will definitely need more help defeating any new enemies you've added. Think of new ways that the weapons can attack, or different things that potions can do. Take advantage of the fact that Weapon is a subclass of Mover—make magic weapons the player has to chase around!

Add more graphics

You can go to **www.headfirstlabs.com/books/hfcsharp/** to find more graphics files for additional enemies, weapons, and other images to help spark your imagination.

Make it an action game

Here's an interesting challenge. Can you figure out how to use the KeyDown event and Timer you used in the Key Game in Chapter 4 to change this from a turn-based game into an action game?

This is your chance to show off! Did you come up with a cool new version of the game? Join the Head First C# forum and claim your bragging rights: www.headfirstlabs.com/books/hfcsharp/