}

Bonus exercise - Tile-based movement

In this exercise, you will create a simple tile-based layout of walls and floor, along with a ball that can be controlled with the arrow-keys and cannot pass through walls. Movement will be constrained to the tile-grid, so when the ball is stopped it will always be exactly aligned with a tile (not, for example, half way between two tiles). See the attached video for the desired result.

1. First we'll create a class called GameObject to act as the base class for all our other objects:

```
public class GameObject
{
	protected Vector2 position;
	protected Texture2D texture;

	public GameObject(Texture2D texture, Vector2 position) {
		this.texture = texture;
		this.position = position;
	}

	public virtual void Draw(SpriteBatch spriteBatch) {
		spriteBatch.Draw(texture, position, Color.White);
	}
}
```

2. Next, create a class Tile, which inherits from GameObject.

```
public class Tile : GameObject
{
	protected bool wall;
	public bool Wall {
		get { return wall; }
	}

	public Tile(Texture2D texture, Vector2 position, bool wall)
		: base(texture, position)
	{
		this.wall = wall;
	}
}
```

Notice the highlighted lines in the code above. *Wall* is a **property**. It is a convenient way of giving public read access to the value of our protected bool *wall*, while preventing the value of wall from being altered from outside the Tile class (we can read the value, but not write to it).

If we instead wrote:

```
public bool Wall {
      get { return wall; }
      set { wall = value; }
}
```

We could then also set the value of wall from outside the Tile class. The highlighted code above is almost equivalent to using a method to get the value of wall - for example:

```
public bool GetWall(){
      return wall;
}
```

... which is exactly how we would get access to the value of the variable in many other programming languages (such as Java and C++)!

3. Next we'll make a Ball class - this is the object that will be moved around the tile-grid by keyboard input.

```
class Ball : GameObject
{
      Game1 game;

      public Ball(Texture2D texture, Vector2 position, Game1 game)
            : base(texture, position)
      {
            this.game = game;
      }

      public void Update(GameTime gameTime)
      {

      }
}
```

4. Add all of the included images to the project (walltile, floortile, ball).
5. Now we'll load in a map-layout from a file and create a 2-dimensional array of tiles. Create a file called map.txt with the following content and save it in the project's debug directory (as in the previous exercises)

```
wwwww
w---w
w-b-w
w---w
wwwww
```

Create variables Tile[,] tiles; and Ball ball; in Game1 and then add the following to Game1's LoadContent method (I won't go over this since it should be repetition from earlier exercises):

```
Texture2D wallTileTex = Content.Load<Texture2D>("walltile");
Texture2D floorTileTex = Content.Load<Texture2D>("floortile");
Texture2D ballTex = Content.Load<Texture2D>("ball");

List<string> strings = new List<string>();
StreamReader sr = new StreamReader("map.txt");
while (!sr.EndOfStream)
{
      strings.Add(sr.ReadLine());
}
sr.Close();

tiles = new Tile[strings[0].Length, strings.Count];
for (int i = 0; i < tiles.GetLength(0); i++)
{
      for (int j = 0; j < tiles.GetLength(1); j++)
      {
            if (strings[j][i] == 'w')
            {
                  tiles[i, j] = new Tile(wallTileTex, new
                  Vector2(wallTileTex.Width * i, wallTileTex.Height
                  * j), true);
            }
            else if (strings[j][i] == '-')
            {
                  tiles[i, j] = new Tile(floorTileTex, new
                  Vector2(floorTileTex.Width * i,
                  floorTileTex.Height * j), false);
            }
            else if (strings[j][i] == 'b')
            {
                  tiles[i, j] = new Tile(floorTileTex, new
                  Vector2(floorTileTex.Width * i,
                  floorTileTex.Height * j), false);
```

```
                    ball = new Ball(ballTex, new
                    Vector2(floorTileTex.Width * i,
                    floorTileTex.Height * j), this);
               }
          }
     }
```

6. Add code to Game1's Draw method to draw the tiles and the ball.

7. Add a new method to Game1 - we will call this from Ball in order to check which directions the ball can move in (note, we're hard-coding the tile width as 50 - really a variable should be used here so we can change the tile width easily without breaking the game)

   ```
   public Tile GetTileAtPosition(Vector2 vec) {
         return tiles[(int)vec.X / 50, (int)vec.Y / 50];
   }
   ```

   This method is the reason we pass a reference to our Game1 object to Ball's constructor - we need to be able to call it from inside Ball (and preferably without making the method static!)

8. Now it's time to get the ball moving. Add four new variables to Ball:

   ```
   Vector2 destination;
   Vector2 direction;
   float speed = 100.0f;
   bool moving = false;
   ```

   Speed should be fairly self-explanatory.
   Direction will contain a vector with the direction the ball is moving in (so (1,0), (-1,0), (0,1) or (0,-1)).
   Destination will hold the position we want to move the ball to. Each time the player presses an arrow-key, we will check if the tile in that direction is traversable (that is, not a wall) and, if so, set moving to true and destination to the location of that next tile. While moving is true, we won't access further keyboard input. When the destination is reached, moving will be set to false again and the cycle starts over.

   To implement the above idea, add a new method to Ball:

   ```
   private void ChangeDirection(Vector2 dir)
   {
         direction = dir;
         Vector2 newDestination = position + direction * 50.0f;
   ```

```
        //Check if we can move in the desired direction, if not, do
        nothing
        if (!game.GetTileAtPosition(newDestination).Wall)
        {
                destination = newDestination;
                moving = true;
        }
}
```

and then change Ball’s Update method to the following:

```
public void Update(GameTime gameTime)
{
        //If we're not already moving, pick a new direction and check
        if we can move in that direction
        //Otherwise, move towards the destination
        if (!moving)
        {
                if (Keyboard.GetState().IsKeyDown(Keys.Left))
                {
                        ChangeDirection(new Vector2(-1,0));
                }
                else if (Keyboard.GetState().IsKeyDown(Keys.Right))
                {
                        ChangeDirection(new Vector2(1, 0));
                }
                else if (Keyboard.GetState().IsKeyDown(Keys.Up))
                {
                        ChangeDirection(new Vector2(0, -1));
                }
                else if (Keyboard.GetState().IsKeyDown(Keys.Down))
                {
                        ChangeDirection(new Vector2(0, 1));
                }
        }
        else
        {
                position += direction * speed *
                (float)gameTime.ElapsedGameTime.TotalSeconds;
                //Check if we are near enough to the destination
                if(Vector2.Distance(position,destination) < 1){
                        position = destination;
                        moving = false;
                }
        }
}
```

And now we should have functional tile-based movement! Note that we have to keep an arrow-key held down if we want the ball to keep on moving over multiple tiles. I’ll leave it as an exercise to let the ball carry on moving in the same direction automatically though (if that’s the kind of movement you want)!