Contents

Combining Classes and Decision Structures

Now that we have learned about decision structures, we can revisit classes and methods. Decision structures can make our methods more flexible, useful, and functional.

Using `if` Statements with Methods

There are several ways we can use **`if`**-**`else`** and **`if`**-**`else`**-**`if`** statements with methods:

- For input validation in setters and properties
- For input validation in constructors
- With Boolean parameters to change a method's behavior
- Inside a method to evaluate instance variables

Setters with Input Validation

- Recall that getters and setters are used to implement **encapsulation**: an object's attributes (instance variables) can only be changed by code in that object's class
- For example, this Item class (which represents an item for sale in a store) has two attributes, a price and a description. Code outside the Item class (e.g. in the `Main` method) can only change these attributes by calling `SetPrice` and `SetDescription`

```
class Item
{
  private string description;
  private decimal price;

  public Item(string initDesc, decimal initPrice)
  {
    description = initDesc;
    price = initPrice;
  }

  public decimal GetPrice()
  {
    return price;
  }

  public void SetPrice(decimal p)
  {
    price = p;
  }

  public string GetDescription()
  {
    return description;
  }

  public void SetDescription(string desc)
  {
    description = desc;
  }
}
```

- Right now, it is possible to set the price to any value, including a negative number, but a negative price does not make sense. If we add an **if** statement to SetPrice, we can check that the new value is a valid price before changing the instance variable:

```
public void SetPrice(decimal p)
{
    if(p >= 0)
    {
        price = p;
    }
    else
    {
        price = 0;
    }
```

```
}
```

  - If the parameter `p` is less than 0, we do not assign it to `price`; instead we set `price` to the nearest valid value, which is 0.
  - Since code outside the Item class cannot access `price` directly, this means it is now impossible to give an item a negative price: If your code calls `myItem.SetPrice(-90m)`, `myItem`'s price will be 0, not -90.

- Alternatively, we could write a setter that simply ignores invalid values, instead of changing the instance variable to the "nearest valid" value
- For example, in the `Rectangle` class, the length and width attributes must also be non-negative. We could write a setter for width like this:

```
public void SetWidth(int newWidth)
{
    if(newWidth >= 0)
    {
        width = newWidth
    }
}
```

  - This means if `myRectangle` has a width of 6, and your code calls `myRectangle.SetWidth(-18)`, then `myRectangle` will still have a width of 6.

- A setter with input validation is a good example of where a conditional operator can be useful. We can write the `SetPrice` method with one line of code using a conditional operator:

```
public void SetPrice(decimal p)
{
    price = (p >= 0) ? p : 0;
}
```

  The instance variable `price` is assigned to the result of the conditional operator, which is either `p`, if `p` is a valid price, or 0, if `p` is not a valid price.

- If you have a class that uses properties instead of getters and setters, the same kind of validation can be added to the `set` component of a property
  - For example, the "price" attribute could be implemented with a property like this:

```
public decimal Price
{
```

```
        get
        {
            return price;
        }
        set
        {
            price = value;
        }
    }
```

- We can add an **if** statement or a conditional operator to the `set` accessor to ensure the price is not set to a negative number:

```
public decimal Price
{
    get
    {
        return price;
    }
    set
    {
        price = (value >= 0) ? value : 0;
    }
}
```

- If a class's attributes have a more limited range of valid values, we might need to write a more complex condition in the setter. For example, consider the Time class:

```
class Time
{
    private int hours;
    private int minutes;
    private int seconds;
}
```

- In a Time object, `hours` can be any non-negative number, but `minutes` and `seconds` must be between 0 and 59 for it to represent a valid time interval
- The `SetMinutes` method can be written as follows:

```
public void SetMinutes(int newMinutes)
{
    if(newMinutes >= 0 && newMinutes < 60)
    {
        minutes = newMinutes;
    }
```

```
    else if(newMinutes >= 60)
    {
        minutes = 59;
    }
    else
    {
        minutes = 0;
    }
}
```

- If the parameter `newMinutes` is between 0 and 59 (both greater than or equal to 0 and less than 60), it is valid and can be assigned to `minutes`
- If `newMinutes` is 60 or greater, we set `minutes` to the largest possible value, which is 59
- If `newMinutes` is less than 0, we set `minutes` to the smallest possible value, which is 0
- Note that we need an if-else-if statement because there are two different ways that `newMinutes` can be invalid (too large or too small) and we need to distinguish between them. When the condition `newMinutes >= 0 && newMinutes < 60` is false, it could either be because `newMinutes` is less than 0 or because `newMinutes` is greater than 59. The **`else if`** clause tests which of these possibilities is true.

Constructors with Input Validation

- A constructor's job is to initialize the object's instance variables, so it is very similar to a "setter" for all the instance variables at once
- If the constructor uses parameters to initialize the instance variables, it can use **`if`** statements to ensure the instance variables are not initialized to "bad" values
- Returning to the `Item` class, this is how we could write a 2-argument constructor that initializes the price to 0 if the parameter `initPrice` is not a valid price:

```
public Item(string initDesc, decimal initPrice)
{
    description = initDesc;
    price = (initPrice >= 0) ? initPrice : 0;
}
```

With both this constructor and the `SetPrice` method we wrote earlier, we can now guarantee that it is impossible for an Item object to have a negative price. This will make it easier to write a large program that uses many Item objects without introducing bugs: the

program cannot accidentally reduce an item's price below 0, and it can add up the prices of all the items and be sure to get the correct answer.

- Recall the `ClassRoom` class from an earlier lecture, which has a room number as one of its attributes. If we know that no classroom building has more than 3 floors, then the room number must be between 100 and 399. The constructor for `ClassRoom` could check that the room number is valid using an if-else-if statement, as follows:

```
public ClassRoom(string buildingParam, int
 ↪  numberParam)
{
    building = buildingParam;
    if(numberParam >= 400)
    {
        number = 399;
    }
    else if(numberParam < 100)
    {
        number = 100;
    }
    else
    {
        number = numberParam;
    }
}
```

  - Here, we have used similar logic to the `SetMinutes` method of the Time class, but with the conditions tested in the opposite order
  - First, we check if `numberParam` is too large (greater than 399), and if so, initialize `number` to 399
  - Then we check if `numberParam` is too small (less than 100), and if so, initialize `number` to 100
  - If both of these conditions are false, it means `numberParam` is a valid room number, so we can initialize `number` to `numberParam`

- The `Time` class also needs a constructor that checks if its parameters are within a valid range, since both minutes and seconds must be between 0 and 59

- However, with this class we can be "smarter" about the way we handle values that are too large. If a user attempts to construct a Time object with a value of 0 hours and 75 minutes, the constructor could "correct" this to 1 hour and 15 minutes and initialize the Time

object with these equivalent values. In other words, this code:

```
Time classTime = new Time(0, 75, 0);
Console.WriteLine($"{classTime.GetHours()} hours,
 ↪  {classTime.GetMinutes()} minutes");
```

should produce the output "1 hours, 15 minutes", not "0 hours, 59 minutes"

- Here's a first attempt at writing the Time constructor:

```
public Time(int hourParam, int minuteParam, int
 ↪  secondParam)
{
    hours = (hourParam >= 0) ? hourParam : 0;
    if(minuteParam >= 60)
    {
        minutes = minuteParam % 60;
        hours += minuteParam / 60;
    }
    else if(minuteParam < 0)
    {
        minutes = 0;
    }
    else
    {
        minutes = minuteParam;
    }
    if(secondParam >= 60)
    {
        seconds = secondParam % 60;
        minutes += secondParam / 60;
    }
    else if(secondParam < 0)
    {
        seconds = 0;
    }
    else
    {
        seconds = secondParam;
    }
}
```

  - First, we initialize `hours` using `hourParam`, unless `hourParam` is negative. There is no upper limit on the value of `hours`
  - If `minuteParam` is 60 or greater, we perform an integer division by 60 and add the result to `hours`, while using the remainder after dividing by 60 to initialize `minutes`. This separates the value

into a whole number of hours and a remaining, valid, number of minutes. Since `hours` has already been initialized, it is important to use `+=` (to add to the existing value).
  - Similarly, if `secondParam` is 60 or greater, we divide it into a whole number of minutes and a remaining number of seconds, and add the number of minutes to `minutes`
  - With all three parameters, any negative value is replaced with 0

- This constructor has an error, however: If `minuteParam` is 59 and `secondParam` is 60 or greater, `minutes` will be initialized to 59, but then the second if-else-if statement will increase `minutes` to 60. There are two ways we can fix this problem.
  - One is to add a nested `if` statement that checks if `minutes` has been increased past 59 by `secondParam`:

```
public Time(int hourParam, int minuteParam, int
 ↪  secondParam)
{
    hours = (hourParam >= 0) ? hourParam : 0;
    if(minuteParam >= 60)
    {
        minutes = minuteParam % 60;
        hours += minuteParam / 60;
    }
    else if(minuteParam < 0)
    {
        minutes = 0;
    }
    else
    {
        minutes = minuteParam;
    }
    if(secondParam >= 60)
    {
        seconds = secondParam % 60;
        minutes += secondParam / 60;
        if(minutes >= 60)
        {
            hours += minutes / 60;
            minutes = minutes % 60;
        }
    }
    else if(secondParam < 0)
    {
        seconds = 0;
```

```
    }
    else
    {
        seconds = secondParam;
    }
}
```

- Another is to use the `AddMinutes` method we have already written to increase `minutes`, rather than the += operator, because this method ensures that `minutes` stays between 0 and 59 and increments `hours` if necessary:

```
public Time(int hourParam, int minuteParam, int
 ↪  secondParam)
{
    hours = (hourParam >= 0) ? hourParam : 0;
    if(minuteParam >= 60)
    {
        AddMinutes(minuteParam);
    }
    else if(minuteParam < 0)
    {
        minutes = 0;
    }
    else
    {
        minutes = minuteParam;
    }
    if(secondParam >= 60)
    {
        seconds = secondParam % 60;
        AddMinutes(secondParam / 60);
    }
    else if(secondParam < 0)
    {
        seconds = 0;
    }
    else
    {
        seconds = secondParam;
    }
}
```

Note that we can also use `AddMinutes` in the first **`if`** statement, since it will perform the same integer division and remainder operations that we originally wrote for `minuteParam`.

Boolean Parameters

- When writing a method, we might want a single method to take one of two different actions depending on some condition, instead of doing the same thing every time. In this case we can declare the method with a `bool` parameter, whose value represents whether the method should (true) or should not (false) have a certain behavior.
- For example, in the `Room` class we wrote in lab, we wrote two separate methods to compute the area of the room: `ComputeArea()` would compute and return the area in meters, while `ComputeAreaFeet()` would compute and return the area in feet. Instead, we could write a single method that computes the area in either feet or meters depending on a parameter:

```
public double ComputeArea(bool useMeters)
{
    if(useMeters)
        return length * width;
    else
        return GetLengthFeet() * GetWidthFeet();
}
```

  - If the `useMeters` parameter is **`true`**, this method acts like the original ComputeArea method and returns the area in meters
  - If the `useMeters` parameter is **`false`**, this method acts like ComputeAreaFeet and returns the area in feet
  - We can use the method like this:

```
Console.WriteLine("Compute area in feet (f) or
 ↪  meters (m)?");
char userChoice = char.Parse(Console.ReadLine());
if(userChoice == 'f')
{
    Console.WriteLine($"Area:
 ↪  {myRoom.ComputeArea(false)}");
}
else if(userChoice == 'm')
{
    Console.WriteLine($"Area:
 ↪  {myRoom.ComputeArea(true)}");
}
else
{
    Console.WriteLine("Invalid choice");
}
```

Regardless of whether the user requests feet or meters, we can call the same method. Instead of calling `ComputeAreaFeet()` when the user requests the area in feet, we call `ComputeArea(false)`

- Note that the `bool` argument to `ComputeArea` can be any expression that results in a Boolean value, not just true or false. This means that we can actually eliminate the `if` statement from the previous example:

  ```
  Console.WriteLine("Compute area in feet (f) or
  ↪  meters (m)?");
  char userChoice = char.Parse(Console.ReadLine());
  bool wantsMeters = userChoice == 'm';
  Console.WriteLine($"Area:
  ↪  {myRoom.ComputeArea(wantsMeters)}");
  ```

  The expression `userChoice == 'm'` is true if the user has requested to see the area in meters. Instead of testing this expression in an `if` statement, we can simply use it as the argument to `ComputeArea` – if the expression is true, we should call `ComputeArea(true)` to get the area in meters.

- Constructors are also methods, and we can add Boolean parameters to constructors as well, if we want to change their behavior. Remember that the parameters of a constructor do not need to correspond directly to instance variables that the constructor will initialize.

- For example, in the lab we wrote two different constructors for the `Room` class: one that would interpret its parameters as meters, and one that would interpret its parameters as feet. Since parameter names ("meters" or "feet") are not part of a method's signature, we ensured the two constructors had different signatures by omitting the "name" parameter from the feet constructor.

  - Meters constructor:

    ```
    public Room(double lengthMeters, double
    ↪  widthMeters, string initName)
    ```

  - Feet constructor:

    ```
    public Room(double lengthFeet, double widthFeet)
    ```

  - The problem with this approach is that the feet constructor cannot initialize the name of the room; if we gave it a `string` parameter for the room name, it would have the same signature as the meters constructor.

  - Using a Boolean parameter, we can write a single constructor that accepts either meters or feet, and is equally capable of

initializing the name attribute in both cases:

```
public Room(double lengthP, double widthP, string
 ↪  nameP, bool meters)
{
    if(meters)
    {
        length = lengthP;
        width = widthP;
    }
    else
    {
        length = lengthP * 0.3048;
        width = widthP * 0.3048;
    }
    name = nameP;
}
```

- If the parameter `meters` is true, this constructor interprets the length and width parameters as meters (acting like the previous "meters constructor"), but if `meters` is false, this constructor interprets the length and width parameters as feet (acting like the previous "feet constructor").

Ordinary Methods Using `if`

- Besides enhancing our "setter" methods, we can also use **`if`** statements to write other methods that change their behavior based on conditions
- For example, we could add a `GetFloor` method to `ClassRoom` that returns a string describing which floor the classroom is on. This looks very similar to the example **`if-else-if`** statement we wrote in a previous lecture, but inside the `ClassRoom` class rather than in a `Main` method:

```
public string GetFloor()
{
    if(number >= 300)
    {
        return "Third floor";
    }
    else if(number >= 200)
    {
        return "Second floor";
    }
    else if(number >= 100)
```

```
    {
        return "First floor";
    }
    else
    {
        return "Invalid room";
    }
}
```

  - Now we can replace the **if-else-if** statement in the `Main` method with a single statement: `Console.WriteLine(myRoom.GetFloor());`

- We can add a `MakeCube` method to the `Prism` class that transforms the prism into a cube by "shrinking" two of its three dimensions, so that all three are equal to the smallest dimension. For example, if `myPrism` is a prism with length 4, width 3, and depth 6, `myPrism.MakeCube()` should change its length and depth to 3.

```
public void MakeCube()
{
    if(length <= width && length <= depth)
    {
        width = length;
        depth = length;
    }
    else if(width <= length && width <= depth)
    {
        length = width;
        depth = width;
    }
    else
    {
        length = depth;
        width = depth;
    }
}
```

  - This **if-else-if** statement first checks to see if `length` is the smallest dimension, and if so, sets the other two dimensions to be equal to `length`
  - Similarly, if `width` is the smallest dimension, it sets both other dimensions to `width`
  - No condition is necessary in the **else** clause, because one of the three dimensions must be the smallest. If the first two conditions are false, `depth` must be the smallest dimension.
  - Note that we need to use <= in both comparisons, not <: if `length` is equal to `width`, but smaller than `depth`, we should still set all dimensions to be equal to `length`

Boolean Instance Variables

- A class might need a `bool` instance variable if it has an attribute that can only be in one of two states, e.g. on/off, feet/meters, on sale/not on sale
- For example, we can add an instance variable called "taxable" to the Item class to indicate whether or not the item should have sales tax added to its price at checkout. The UML diagram for Item with this instance variable would look like this:

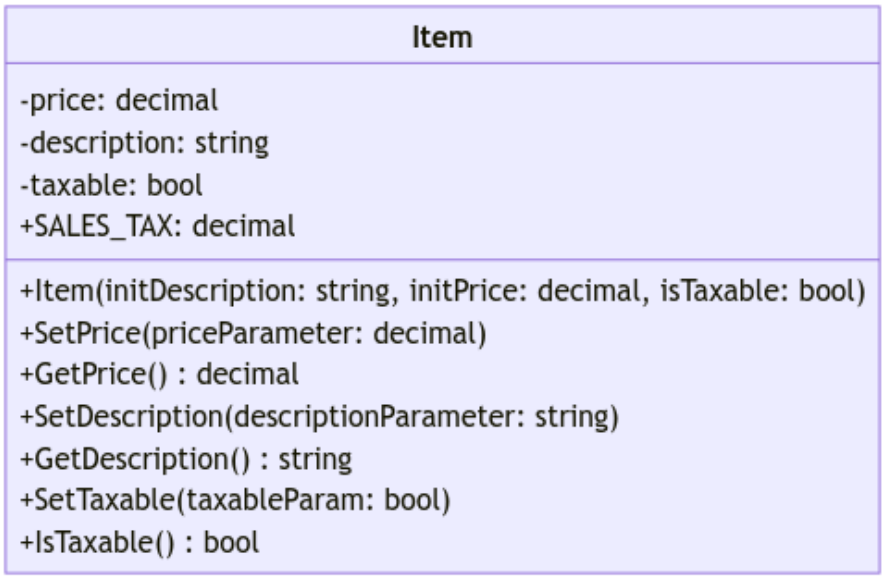

Figure 1: A UML diagram for the Item class (text version[1])

  - Note that the "getter" for a Boolean variable is conventionally named with a word like "Is" or "Has" rather than "Get"
  - We will add a constant named SALES_TAX to the Item class to store the sales tax rate that should be applied if the item is taxable. The sales tax rate is not likely to change during the program's execution, but it is better to store it in a named variable instead of writing the same literal value (e.g. `0.08m`) every time we want to compute a total price with tax.
- The instance variables and constructor for `Item` now look like this:

```
class Item
{
    private string description;
    private decimal price;
```

```
    private bool taxable
    public const decimal SALES_TAX = 0.08m;

    public Item(string initDesc, decimal initPrice,
     ↪  bool isTaxable)
    {
        description = initDesc;
        price = (initPrice >= 0) ? initPrice : 0;
        taxable = isTaxable;
    }
...
}
```

- We can use this instance variable in a `Main` method to compute the final price of an Item based on whether or not it is taxable:

```
Item myItem = new Item("Blue Polo Shirt", 19.99m,
 ↪  true);
decimal totalPrice = myItem.GetPrice();
if(myItem.isTaxable())
{
    totalPrice = totalPrice + (totalPrice *
 ↪  Item.SALES_TAX);
}
Console.WriteLine($"Final price: {totalPrice:C}");
```

- However, if we were writing a program that handled large numbers of items, we might find it tedious to write this **`if`** statement every time. To make it easier to compute the "real" (with tax) price of an item, we could instead modify the `GetPrice()` method to automatically include sales tax if applicable:

```
public decimal GetPrice()
{
    if(taxable)
        return price + (price * SALES_TAX);
    else
        return price;
}
```

  Now, `myItem.GetPrice()` will return the price with tax if the item is taxable, so our `Main` method can simply use `myItem.GetPrice()` as the total price without needing to check `myItem.isTaxable()`.

Using `while` Loops with Classes

There are several ways that **`while`** loops are useful when working with classes and methods:

- To validate input before calling a method
- Inside a method, to interact with the user
- Inside a method, to take repeated action based on the object's attributes
- To control program behavior based on the return value of a method

Input Validation with Objects

- As we have seen in a previous section (Loops and Input Validation), **`while`** loops can be used with the `TryParse` method to repeatedly prompt the user for input until he/she enters a valid value
- This is a useful technique to use before initializing an object's attributes with user-provided data
- For example, the length and width of a `Rectangle` object should be non-negative integers. If we want to create a `Rectangle` with a length and width provided by the user, we can use a **`while`** loop for each attribute to ensure the user enters valid values before constructing the `Rectangle`.

```
int length, width;
bool isInt;
do
{
    Console.WriteLine("Enter a positive length");
    isInt = int.TryParse(Console.ReadLine(), out
 ↪  length);
} while(!isInt || length < 0);
do
{
    Console.WriteLine("Enter a positive width");
    isInt = int.TryParse(Console.ReadLine(), out
 ↪  width);
} while(!isInt || width < 0);
Rectangle myRectangle = new Rectangle(length, width);
```

  - Each loop asks the user to enter a number, and repeats if the user enters a non-integer (`TryParse` returns **`false`**) or enters a negative number (`length` or `width` is less than 0).
  - Note that we can re-use the `bool` variable `isInt` to contain the return value of `TryParse` in the second loop, since it would otherwise have no purpose or meaning after the first loop ends.
  - After both loops have ended, we know that `length` and `width` are sensible values to use to construct a `Rectangle`

- Similarly, we can use **`while`** loops to validate user input before calling a non-constructor method that takes arguments, such as

`Rectangle`'s `Multiply` method or `Item`'s `SetPrice` method

- For example, if a program has an already-initialized `Item` object named `myItem` and wants to use `SetPrice` to change its price to a user-provided value, we can use a **`while`** loop to keep prompting the user for input until he/she enters a valid price.

```
bool isNumber;
decimal newPrice;
do
{
    Console.WriteLine($"Enter a new price for
 ↪  {myItem.GetDescription()}");
    isNumber = decimal.TryParse(Console.ReadLine(),
 ↪  out newPrice);
} while(!isNumber || newPrice < 0);
myItem.SetPrice(newPrice);
```

  - Like with our previous example, the **`while`** loop's condition will be **`true`** if the user enters a non-numeric string, or a negative value. Thus the loop will only stop when `newPrice` contains a valid price provided by the user.
  - Although it is "safe" to pass a negative value as the argument to `SetPrice`, now that we added an **`if`** statement to `SetPrice`, it can still be useful to write this **`while`** loop
  - The `SetPrice` method will use a default value of 0 if its argument is negative, but it will not alert the user that the price they provided is invalid or give them an opportunity to provide a new price

- The `ComputeArea` method that we wrote earlier for the `Room` class demonstrates another situation where it is useful to write a **`while`** loop before calling a method

  - Note that in the version of the code that passes the user's input directly to the `ComputeArea` method, instead of using an **`if-else-if`** statement, there is nothing to ensure the user enters one of the choices "f" or "m":

```
Console.WriteLine("Compute area in feet (f) or
 ↪  meters (m)?");
char userChoice = char.Parse(Console.ReadLine());
Console.WriteLine($"Area:
 ↪  {myRoom.ComputeArea(userChoice == 'm')}");
```

  - This means that if the user enters a multiple-letter string the program will crash (`char.Parse` throws an exception if its input string is larger than one character), and if the user enters a letter other than "m" the program will act as if he/she entered

"f"

- Instead, we can use `TryParse` and a **`while`** loop to ensure that `userChoice` is either "f" or "m" and nothing else

```
bool validChar;
char userChoice;
do
{
    Console.WriteLine("Compute area in feet (f) or
 ↪  meters (m)?");
    validChar = char.TryParse(Console.ReadLine(),
 ↪  out userChoice);
} while(!validChar || !(userChoice == 'f' ||
 ↪  userChoice == 'm'));
Console.WriteLine($"Area:
 ↪  {myRoom.ComputeArea(userChoice == 'm')}");
```

- This loop will prompt the user for input again if `TryParse` returns **`false`**, meaning he/she did not enter a single letter. It will also prompt again if the user's input was not equal to `'f'` or `'m'`.
- Note that we needed to use parentheses around the expression `!(userChoice == 'f' || userChoice == 'm')` in order to apply the `!` operator to the entire "OR" condition. This represents the statement "it is not true that userChoice is equal to 'f' or 'm'." We could also write this expression as `(userChoice != 'f' && userChoice != 'm')`, which represents the equivalent statement "userChoice is not equal to 'f' and also not equal to 'm'."

Using Loops Inside Methods

- A class's methods can contain **`while`** loops if they need to execute some code repeatedly. This means that when you call such a method, control will not return to the `Main` program until the loop has stopped.
- Reading input from the user, validating it, and using it to set the attributes of an object is a common task in the programs we have been writing. If we want to do this for several objects, we might end up writing many very similar **`while`** loops in the `Main` method. Instead, we could write a method that will read and validate user input for an object's attribute every time it is called.
  - For example, we could add a method `SetLengthFromUser` to the `Rectangle` class:

```
public void SetLengthFromUser()
{
    bool isInt;
    do
    {
        Console.WriteLine("Enter a positive
↪   length");
        isInt = int.TryParse(Console.ReadLine(),
↪   out length);
    } while(!isInt || length < 0);
}
```

- This method is similar to a setter, but it has no parameters because its only input comes from the user
- The **while** loop is just like the one we wrote before constructing a `Rectangle` in a previous example, except the **out** parameter of `TryParse` is the instance variable `length` instead of a local variable in the `Main` method
- `TryParse` will assign the user's input to the `length` instance variable when it succeeds, so by the time the loop ends, the Rectangle's length has been set to the user-provided value
- Assuming we wrote a similar method `SetWidthFromUser()` (substituting `width` for `length` in the code), we would use these methods in the `Main` method like this:

  ```
  Rectangle rect1 = new Rectangle();
  Rectangle rect2 = new Rectangle();
  rect1.SetLengthFromUser();
  rect1.SetWidthFromUser();
  rect2.SetLengthFromUser();
  rect2.SetWidthFromUser();
  ```

  After executing this code, both `rect1` and `rect2` have been initialized with length and width values the user entered.

- Methods can also contain **while** loops that are not related to validating input. A method might use a **while** loop to repeat an action several times based on the object's instance variables.
  - For example, we could add a method to the `Rectangle` class that will display the Rectangle object as a rectangle of asterisks on the screen:

    ```
    public void DrawInConsole()
    {
        int counter = 1;
        while(counter <= width * length)
    ```

```
        {
            Console.Write(" * ");
            if(counter % width == 0)
            {
                Console.WriteLine();
            }
            counter++;
        }
    }
```

- This **while** loop prints a number of asterisks equal to the area of the rectangle. Each time it prints `width` of them on the same line, it adds a line break with `WriteLine()`.

Using Methods to Control Loops

- Methods can return Boolean values, as we showed previously in the section on Boolean instance variables
- Other code can use the return value of an object's method in the loop condition of a **while** loop, so the loop is controlled (in part) by the state of the object
- For example, recall the `Time` class, which stores hours, minutes, and seconds in instance variables.
  - In a previous example we wrote a `GetTotalSeconds()` method to convert these three instance variables into a single value:

    ```
    public int GetTotalSeconds()
    {
        return hours * 60 * 60 + minutes * 60 +
        ↪  seconds;
    }
    ```

  - We can now write a method `ComesBefore` that compares two Time objects:

    ```
    public bool ComesBefore(Time otherTime)
    {
        return GetTotalSeconds() <
        ↪  otherTime.GetTotalSeconds();
    }
    ```

    This method will return **true** if the calling object (i.e. **this** object) represents a smaller amount of time than the other Time object passed as an argument

- Since it returns a Boolean value, we can use the `ComesBefore` method to control a loop. Specifically, we can write a program that asks the user to enter a Time value that is smaller than a specified maximum, and use `ComesBefore` to validate the user's input.

```
Time maximumTime = new Time(2, 45, 0);
Time userTime;
do
{
    Console.WriteLine($"Enter a time less than
 ↪  {maximumTime}");
    int hours, minutes, seconds;
    do
    {
        Console.Write("Enter the hours: ");
    } while(!int.TryParse(Console.ReadLine(), out
     ↪  hours));
    do
    {
        Console.Write("Enter the minutes: ");
    } while(!int.TryParse(Console.ReadLine(), out
     ↪  minutes));
    do
    {
        Console.Write("Enter the seconds: ");
    } while(!int.TryParse(Console.ReadLine(), out
     ↪  seconds));
    userTime = new Time(hours, minutes, seconds);
} while(!userTime.ComesBefore(maximumTime));
//At this point, userTime is valid Time object
```

- Note that there are **`while`** loops to validate each number the user inputs for hours, minutes, and seconds, as well as an outer **`while`** loop that validates the Time object as a whole.
- The outer loop will continue until the user enters values that make `userTime.ComesBefore(maximumTime)` return **`true`**.

Examples

The Room Class

The class and its associated `Main` method presented in this archive[2] show how you can use classes, methods, constructors and decision structures

[2] https:/princomp.github.io/code/projects/Room.zip

all in the same program. It also exemplifies how a method can take *an object* as a parameter with `InSameBuilding`.

The corresponding UML diagram is:

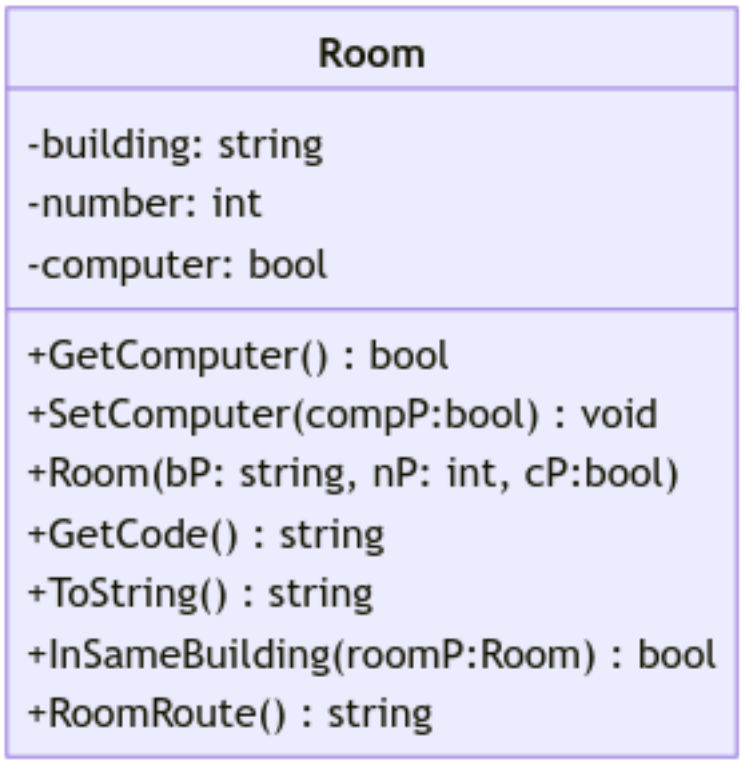

Figure 2: A UML diagram for the Room class (text version[3])

The Loan Class

Similarly, this class and its associated `Main` method show how you can use classes, methods, constructors, decision structures, and user input validation all in the same program. This lab[4] asks you to add the user input validation code, and you can download the following code in this archive[5].

```
/*
 * Application program for the "Loan" class.
```

[4] https:/princomp.github.io/labs/ValidatingInput
[5] https:/princomp.github.io/code/projects/LoanCalculator.zip

```
 * This program gathers from the user all the information
 ↪  needed
 * to create a "proper" Loan object.
*/

using System;

class Program
{
  static void Main()
  {
    Console.WriteLine("What is your name?");
    string name = Console.ReadLine();

    Console.WriteLine(
      "Do you want a loan for an Auto (A, a), a House (H,
       ↪  h), or for some Other (O, o) reason?"
    );
    char type = Console.ReadKey().KeyChar; // This part
     ↪  of the code reads *a char* from the user.
    // We haven't studied it, but it's pretty
     ↪  straightforward.
    Console.WriteLine();

    /*
     * The part of the code that follows
     * does the convertion from the character
     * to the corresponding string.
     * We could have a method in the Loan
     * class that does it for us, but
     * we'll just do it "by hand" here
     * for simplicity.
     */
    string typeOfLoan;

    if (type == 'A' || type == 'a')
    {
      type = 'a';
      typeOfLoan = "an auto";
    }
    else if (type == 'H' || type == 'h')
    {
      type = 'h';
      typeOfLoan = "a house";
    }
    else
```

```
    {
      type = 'o';
      typeOfLoan = "some other reason";
    }

    // We display the information back to the user, and
     ↪  ask the next question:
    Console.WriteLine(
      $"{name}, you need money for {typeOfLoan},
↪   great.\nWhat is your current credit score?"
    );
    int cscore = int.Parse(Console.ReadLine());

    Console.WriteLine("How much do you need, total?");
    decimal need = decimal.Parse(Console.ReadLine());

    Console.WriteLine("What is your down payment?");
    decimal down = decimal.Parse(Console.ReadLine());

    Loan myLoan = new Loan(name, type, cscore, need,
↪   down);
    Console.WriteLine(myLoan);
  }
}

/*
 * "Loan" class.
 * This class helps primarily in computing
 * an APR based on information provided from the user.
 * A ToString method is provided.
 */
using System;

class Loan
{
  private string name; // For the name of the loan holder.
  private char type; // For the type ('a'uto, 'h'ouse or
   ↪  'o'ther) of the loan
  private int cscore; // For the credit score.
  private decimal amount; // For the amount of money
   ↪  loaned.
  private decimal rate; // For the A.P.R., the interest
   ↪  rate.

  /*
   * Our constuctor will compute the amount and the rate
```

```
   * based on the information given as arguments.
   * The name, type and credit score will simply be given
↪   as arguments.
   */
  public Loan(
    string nameP,
    char typeP,
    int cscoreP,
    decimal needP,
    decimal downP
  )
  {
    name = nameP;
    type = typeP;
    cscore = cscoreP;
    if (cscore < 421)
    {
      Console.WriteLine(
        "Sorry, we can't accept your application."
      );
      amount = -1;
      rate = -1;
    }
    else
    {
      amount = needP - downP;

      switch (type)
      {
        case ('a'):
          rate = .05M;
          break;

        case ('h'):
          if (cscore > 600 && amount < 1000000M)
            rate = .03M;
          else
            rate = .04M;
          break;

        case ('o'):
          if (cscore > 650 || amount < 10000M)
            rate = .07M;
          else
            rate = .09M;
          break;
```

```
      }
    }
  }

  public override string ToString()
  {
    string typeName = "";
    switch (type)
    {
      case ('a'):
        typeName = "an auto";
        break;

      case ('h'):
        typeName = "a house";
        break;
      case ('o'):
        typeName = "another reason";
        break;
    }
    return "Dear "
      + name
      + $", you borrowed {amount:C} at {rate:P} for "
      + typeName
      + ".";
  }
}
```