

Chapter 6

Reacting to Player Input

6.1 Introduction

In this chapter, we will show you how your game program can react to mouse clicks and button presses. In order to do this, we need an instruction called **if** that executes an instruction (or a group of instructions) if a condition is met. We will also introduce enumerated types as another kind of primitive type.

6.2 Reacting to a Mouse Click

6.2.1 *ButtonState: An Enumerated Type*

In the previous examples, we have used the current mouse state to retrieve the mouse position. However, a `MouseState` object contains a lot of other information as well. For example, it can be used to find out whether a mouse button is pressed or not. For this, we can use the properties `LeftButton`, `MiddleButton` and `RightButton`. What these properties give is a value of type `ButtonState`. So we could save this value as follows:

```
ButtonState left = currentMouseState.LeftButton;
```

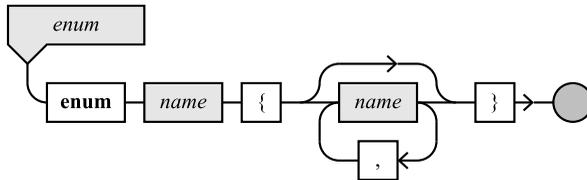
Primitive types—C# makes a distinction between the more complicated types representing a class and the very basic types representing things like integers or enumerations. The latter types are also called *primitive types*, because they form the building blocks of the more complicated *class types*.

You might guess that `ButtonState` is a class, however, it is actually an *enumerated type*. An enumerated type is very similar to the integer type, with the difference that

instead of numeric values, the type contains words describing different states. In the case of the `ButtonState` type, these states are `Pressed` and `Released`, because a button is either pressed or released. Enumerated types are quite handy for representing a variable that can contain a few different meaningful states. For example, we might want to store the type of character that a player represents by using an enumerated type. We can decide ourselves what kind of different states there are in our type, so before we can use the enumerated type, we first have to define it:

```
enum CharacterClan { Warrior, Wizard, Elf, Spy };
```

The `enum` keyword indicates that we are going to define an enumerated type. After that follows the name of this type and, between braces, the different states that can be stored inside a variable of this type. This is the syntax diagram describing the `enum type definition`:



The type definition can be placed inside a method, but you may also define it at the class body level, so that all the methods in the class can use the type. You may even define it as a top-level declaration (outside of the class body). Here is an example of using the `CharacterClan` enumerated type:

```
CharacterClan myClan = CharacterClan.Warrior;
```

In this case, we have created a variable of type `CharacterClan`, which may contain one of four values: `CharacterClan.Warrior`, `CharacterClan.Wizard`, `CharacterClan.Elf`, or `CharacterClan.Spy`. In a very similar way, the `ButtonState` type is defined somewhere in a library probably looking something like

```
enum ButtonState { Pressed, Released };
```

Another example of using enumerated types would be to define a type for indicating the days in the week or the months in a year:

```
enum MonthType { January, February, March, April, May, June, July, August,
                September, October, November, December };
enum DayType { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
MonthType currentMonth = MonthType.February;
DayType today = DayType.Tuesday;
```

6.2.2 *The if-Instruction: Executing an Instruction Depending on a Condition*

As a simple example of how we can use the mouse button state to do something, let us make a simple extension of the Painter1 program, where we only calculate the new angle of the cannon barrel if the left mouse button is down. This means that we have to change the instructions in the Update method, because that is where we calculate the barrel angle.

Until now, all the instructions we have written had to be executed all the time. For example, drawing the background sprite and the cannon barrel sprite always needs to happen. But calculating the barrel angle only has to happen *sometimes*, namely when the player presses the left mouse button. In broader terms, we want to execute an instruction only if some condition holds true. This kind of instruction is called a *conditional instruction*, and it uses a new keyword: **if**.

With the **if**-instruction, we can provide a condition, and execute a block of instructions if this condition holds (in total, this is sometimes also referred to as a *branch*). Examples of conditions are:

1. the number of seconds passed since the start of the game is larger than 1000, or
2. the balloon sprite is exactly in the middle of the screen, or
3. the monster has eaten my character.

These conditions can either be **true** or **false**. A condition is an *expression*, because it has a value (it is either **true** or **false**). This value is also called a *boolean* value. It is associated with a type called **bool**, which we will talk about more later on. With an **if**-instruction, we can execute a block of instructions if a condition is **true**. Take a look at this example:

```
if (mouse.X > 200)
{
    spriteBatch.Draw(background, Vector2.Zero, Color.White);
}
```

This is an example of an **if**-instruction. The condition is always placed in parentheses. After that, a block of instructions follows, enclosed by braces. In this example, the background is only drawn if the mouse *x*-position is larger than 200. This means that if you move the mouse too far to the left of the screen, the background is not drawn anymore. We can place multiple instructions between the braces if we want:

```
if (mouse.X > 200)
{
    spriteBatch.Draw(background, Vector2.Zero, Color.White);
    spriteBatch.Draw(cannonBarrel, Vector2.Zero, Color.White);
}
```

If there is only one instruction, you may omit the braces to shorten the code a bit:

```
if (mouse.X > 200)
    spriteBatch.Draw(background, Vector2.Zero, Color.White);
```

In our example, we want to update the cannon barrel angle only when the player presses the left mouse button. This means that we have to check if the state of the left mouse button currently is pressed. This condition is given as follows:

```
mouse.LeftButton == ButtonState.Pressed
```

The `==` operator compares two values and returns **true** if they are the same, and **false** otherwise. On the left hand side of this comparison operator, we find the left mouse button state. On the right hand side, we find the state `ButtonState.Pressed`. So, this condition checks whether the left button is currently pressed. We can now use it in an `if`-instruction as follows in the `Update` method:

```
if (mouse.LeftButton == ButtonState.Pressed)
{
    double opposite = mouse.Y - barrelPosition.Y;
    double adjacent = mouse.X - barrelPosition.X;
    angle = (float)Math.Atan2(opposite, adjacent);
}
```

So, only if the left mouse button is pressed, we calculate the angle and store its value in the `angle` member variable. In order to see this program working, run the `Painter1a` example in the solution belonging to this chapter.

6.3 Boolean Values

6.3.1 Comparison Operators

The condition in the header of an `if`-instruction is an expression that returns a truth value: ‘yes’ or ‘no’. When the outcome of the expression is ‘yes’, the body of the `if` instruction is executed. In these conditions you are allowed to use comparison operators. The following operators are available:

- `<` smaller than
- `<=` smaller than or equal to
- `>` larger than
- `>=` larger than or equal to
- `==` equal to
- `!=` not equal to

These operators may be used between two numbers. On the left hand side and the right hand side of these operators you may put constant values, variables, or complete expressions with additions, multiplications and whatever you want, provided

that they are of the same type. Again, watch out that testing the equality of two values is done using a double equals sign (`==`). This is needed, because the single equals sign is already used for assignments. The difference between these two operators is very important:

- `x=5;` this instruction means: **assign** the value 5 to `x`!
- `x==5` this expression means: **is** `x` equal to 5?

6.3.2 Logic Operators

In logical terms, a condition is also called a *predicate*. The operators that are used in logic to connect predicates ('and', 'or', and 'not') can also be used in C#. These operators have a special notation in C#:

- `&&` is the logical 'and' operator
- `||` is the logical 'or' operator
- `!` is the logical 'not' operator

We can use these operators to check for complicated logical statements, so that we can execute instructions only in very particular cases. For example, we can draw a 'You win!' overlay only if the player has more than 10,000 points, the enemy has a life force of 0, and the player life force is larger than 0:

```
if (playerPoints > 10000 && enemyLifeForce == 0 && playerLifeForce > 0)
    spriteBatch.Draw(winningOverlay, Vector2.Zero, Color.White);
```

6.3.3 The Boolean Type

Expressions that use comparison operators, or that connect other expressions with logical operators also have a type, just like expressions that use arithmetic operators. After all, the result of such an expression is a value: one of the two truth values: 'yes' or 'no'. In logic, these values are called 'true' and 'false'. In C#, these truth values are represented by the **true** and **false** keywords.

Next to being used for expression a condition in an **if**-instruction, logical expressions can be applied for a lot of different things. A logical expression is similar to an arithmetic expression, except that it has a different type. For example, you can store the result of a logical expression in a variable, pass it as a parameter, or use that result again in another expression.

The type of logical values is called **bool**. This is one of the primitive types of C#. The type is named after the English mathematician and philosopher George Boole (1815–1864). Here is an example of a declaration and an assignment of a boolean variable:

```
bool test;
test = x>3 && y<5;
```

In this case, if `x` contains, for example, the value 6 and `y` contains the value 3, then the boolean expression `x>3 && y<5` will evaluate to **true** and this value will be stored in the variable `test`. We can also store the boolean values **true** and **false** directly in a variable:

```
bool isAlive = false;
```

Boolean variables are extremely handy to store the status of different objects in the game. For example, you could use a boolean variable to store whether or not the player is still alive, or if the player is currently jumping, or if a level is finished, and so on. We can use boolean variables as an expression in an **if**-instruction:

```
if (isAlive)
    do something
```

In this case, if the expression `isAlive` evaluates to **true**, the body of the **if**-instruction is executed. You might think that this code generates a compiler error, and that we need to do a comparison of the boolean variable, like this:

```
if (isAlive == true)
    do something
```

However, this extra comparison is not necessary. A conditional expression like in the **if**-instruction *has to evaluate to true or false*. Since a boolean variable already represents either one of these two values, we do not need to perform the comparison anymore. In fact, if the previous comparison would be needed, then we also would need to compare that outcome again with a boolean value:

```
if ((isAlive == true) == true)
    do something
```

And this gets worse:

```
if (((((((isAlive == true) == true) == true) == true) == true) == true) == true)
    do something
```

In summary: do not make things more complicated than they are. If the outcome is already a boolean value, we do not have to compare it to anything anymore.

We can use the **bool** type to store complex expressions that are either **true** or **false**. Let us look at a few additional examples:

```
bool a = 12 > 5;
bool b = a && 3+4==8;
bool c = a || b;
if (!c)
    a = false;
```

Before you read on, try and find out what the value is of the variables `a`, `b`, and `c` after these instructions have been executed. In the first line, we declare and initialize a boolean `a`. The truth value that is stored in this boolean is evaluated from the expression `12 > 5`, which evaluates to **true**. So, variable `a` has the value **true**. In the second line, we declare and initialize a new variable `b`, in which we store the result of a more complex expression. The first part of this expression is the variable `a`, which contains the value **true**. The second part of the expression is a comparison `3+4==8`. This comparison is not true (`3 + 4` does not equal `8`), so this evaluates to **false**, and therefore the logical ‘and’ also results in **false**. Therefore, the variable `b` contains the value **false**.

The third line stores the result of the logical ‘or’ operation on variables `a` and `b` in variable `c`. Since `a` contains the value **true**, the outcome of this operation is also **true**, and it is assigned to `c`. Finally, there is an `if`-instruction, which assigns the value **false** to variable `a`, but only if `!c` evaluates to **true**, that is, `c` evaluates to **false**. In this particular case, `c` is **true**, so the body of the `if` instruction is not executed. Therefore, after all the instructions are executed, `a` and `c` contain the value **true**, and `b` contains the value **false**.

6.4 More on `if`-Instructions

6.4.1 *An `if`-Instruction with an Alternative*

We can use the `if`-instruction to check if the left mouse button is down. If so, we update the angle of the cannon barrel:

```
if (mouse.LeftButton == ButtonState.Pressed)
{
    double opposite = mouse.Y - barrelPosition.Y;
    double adjacent = mouse.X - barrelPosition.X;
    angle = (float)Math.Atan2(opposite, adjacent);
}
```

In the `Painter1a` example, the angle stays the same when the left mouse button is not pressed. But suppose that we want to have the angle set to zero again once the player releases the left mouse button. We could add another `if`-instruction, like this:

```
if (mouse.LeftButton != ButtonState.Pressed)
    angle = 0f;
```

However, there is a nicer way of dealing with this: by using an `if`-instruction with an alternative. The alternative instruction is executed when the condition in the `if`-

instruction is *not* true, and we use the **else** keyword for that:

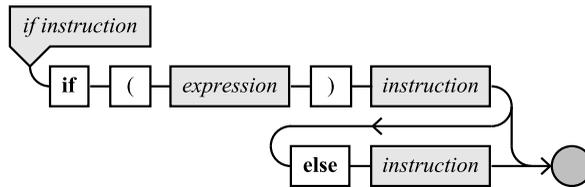
```

if (mouse.LeftButton == ButtonState.Pressed)
{
    double opposite = mouse.Y - barrelPosition.Y;
    double adjacent = mouse.X - barrelPosition.X;
    angle = (float)Math.Atan2(opposite, adjacent);
}
else
    angle = 0f;

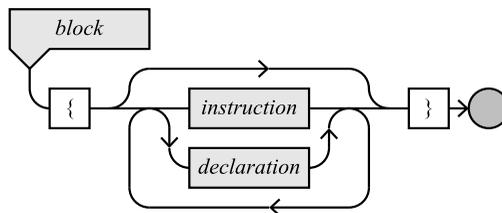
```

This instruction does exactly the same thing as the two **if**-instructions before, but we only have to write down the condition once. Execute the Painter1b program and see what it does. You will note that the angle of the cannon barrel is zero as soon as you release the left mouse button.

The syntax of the **if**-instruction with an alternative is represented by the following syntax diagram:



The reason that the body of an **if**-instruction can consist of multiple instructions between braces is that an instruction can also be a *block* of instructions, which is defined in the following syntax diagram:



6.4.2 A Number of Different Alternatives

When there are multiple categories of values, you can find out with **if**-instructions which case we are dealing with. The second test is placed behind the **else** of the first **if**-instruction, so that the second test is only executed when the first test failed. A possible third test would be placed behind the **else** of the second **if**-instruction.

The following fragment determines within which age segment a player falls, so that we can draw different player sprites:

```

if (age<4)
    spriteBatch.Draw(babyPlayer, playerPosition, Color.White);
else if (age<12)
    spriteBatch.Draw(youngPlayer, playerPosition, Color.White);
    else if (age<65)
        spriteBatch.Draw(adultPlayer, playerPosition, Color.White);
    else
        spriteBatch.Draw(oldPlayer, playerPosition, Color.White);

```

Behind every **else** (except the last one), there is another **if**-instruction. For babies, the `babyPlayer` sprite is drawn, and the rest of the instructions are ignored (they are behind the **else** after all). Old players on the other hand, go through all the tests (younger than 4? younger than 12? younger than 65?) before we conclude that we have to draw the `oldPlayer` sprite.

We used indentation in this program to indicate which **else** belongs to which **if**. When there are many different categories, the text of the program becomes less and less readable. Therefore, as an exception to the usual rule that instructions after the **else** should be indented, we allow for a simpler layout with such complicated **if**-instructions.

```

if (age<4)
    spriteBatch.Draw(babyPlayer, playerPosition, Color.White);
else if (age<12)
    spriteBatch.Draw(youngPlayer, playerPosition, Color.White);
else if (age<65)
    spriteBatch.Draw(adultPlayer, playerPosition, Color.White);
else
    spriteBatch.Draw(oldPlayer, playerPosition, Color.White);

```

The additional advantage here is that using this layout, it is a lot easier to see which cases are handled.

6.4.3 *Toggling the Cannon Barrel Behavior*

As a final example of using the **if**-instruction to handle mouse button presses, let us try to handle a mouse button *click* instead of a mouse button press. We know how to check with an **if** instruction if the mouse button is currently pressed, but how do we find out if the player has clicked (meaning pressing and then releasing the mouse button)? Have a look at the program `Painter1c`. In this program, the cannon barrel rotation follows the mouse pointer after you click the left button. When you click again, the cannon stops following the mouse pointer.

The issue with this kind of ‘toggle’ behavior is that we only know the current status of the mouse in the `Update` method. This is not enough information for determining when a ‘click’ happens, because a click is partly defined by what happened the previous time we were in the `Update` method. We can say that a player has clicked the mouse button if these two things happen:

- the player did not press the mouse button during the last Update method;
- in the current Update method, the player presses the mouse button.

In order to solve this, we need to store the *previous mouse state*, so that we can compare it with the current mouse state the next time we are in the Update method. Let us therefore store the previous and the current mouse state in two member variables:

```
MouseState currentMouseState, previousMouseState;
```

Getting the current mouse state is easy, we have done it before:

```
currentMouseState = Mouse.GetState();
```

Now, how do we get the ‘previous mouse state’? There is no way of doing this directly by calling a method from the Mouse class. However, we do know that what is now the current mouse state will be the previous mouse state in the next Update call. So, we can solve this problem by assigning the value of the current mouse state to the previous mouse state before we get the new mouse state. Therefore, our Update method will look something like:

```
protected override void Update(GameTime gameTime)
{
    previousMouseState = currentMouseState;
    currentMouseState = Mouse.GetState();
    Here we do something with the mouse state
}
```

Now, we can write the code needed to toggle the cannon barrel behavior by looking at the previous and current mouse state. We check this by using an **if**-instruction:

```
if (currentMouseState.LeftButton == ButtonState.Pressed
    && previousMouseState.LeftButton == ButtonState.Released)
    calculateAngle = !calculateAngle;
```

The conditional expression in this case checks that the state of the left mouse button currently is ‘pressed’, while its state was ‘released’ the previous time we retrieved the mouse state. If this condition evaluates to **true**, we toggle the calculateAngle variable. This is a member variable of type boolean (so it is either true or false). In order to get the toggling behavior, we make use of the logical ‘not’ operator. The result of the ‘not’ operation on the variable calculateAngle is stored again in the variable calculateAngle. So, if that variable contained the value **true**, we will store in the same variable the value **false** and vice versa. The result of this is that the value of the calculateAngle variable toggles every time we execute that instruction.

We can now use that variable in another **if**-instruction to determine whether we should update the angle or not:

```

if (calculateAngle)
{
    double opposite = currentMouseState.Y - barrelPosition.Y;
    double adjacent = currentMouseState.X - barrelPosition.X;
    angle = (float)Math.Atan2(opposite, adjacent);
}
else
    angle = 0.0f;

```

6.5 Handling Keyboard and Gamepad Input

6.5.1 Basics of Handling Keyboard Input

Handling keyboard and gamepad input is dealt with in a very similar way to dealing with mouse input. Instead of getting the mouse state, we have to get the *keyboard* or *gamepad* state. This can be done as follows:

```

KeyboardState currentKBState = Keyboard.GetState();
GamePadState currentGPState = GamePad.GetState();

```

Also, just like the mouse state, these variables have several methods for checking if the player presses a key on the keyboard, or a button on the gamepad. For example, we can check if the player presses the 'A' button on the gamepad by calling `currentGPState.IsButtonDown(Buttons.A)`. Similarly, we check if the 'A' key on the keyboard is pressed by calling `currentKBState.IsKeyDown(Keys.A)`. The classes `Buttons` and `Keys` provide a number of properties for defining the available keys and buttons.

6.5.2 A Multicolored Cannon

The program `Painter2` is an extension of the `Painter1` program. It also features a rotating cannon, but now the player can also select the color of the cannon by pressing different keys (R, G, or B). The current color of the cannon is displayed on the screen by drawing a colored ball in the center of the rotating barrel. The cannon can be either red, green, or blue. So, we will need three images of a ball, one for each of the three colors:

```

Texture2D colorRed, colorGreen, colorBlue;

```

To make things a bit more convenient, we will declare an extra `Texture2` variable called `currentColor`, in which we keep track of what the current color of the cannon is:

```
Texture2D currentColor;
```

Finally, we need a variable for storing the origin of the ball:

```
Vector2 colorOrigin;
```

Now that we have our member variables in place, let us have a look at the `LoadContent` method. In this method, we need to load a few extra sprites, and store them in the member variables that we just declared:

```
colorRed = Content.Load<Texture2D>("spr_cannon_red");
colorGreen = Content.Load<Texture2D>("spr_cannon_green");
colorBlue = Content.Load<Texture2D>("spr_cannon_blue");
```

At the start of the program, we assume that the cannon is blue, so we assign the blue sprite to the `currentColor` variable:

```
currentColor = colorBlue;
```

Finally, we calculate the origin of the ball sprite which we choose as its center:

```
colorOrigin = new Vector2(currentColor.Width, currentColor.Height) / 2;
```

6.5.3 Handling the Keyboard Input

For handling the keyboard input, we will need the keyboard state. For completeness, we will store the previous and current state for both the keyboard and the mouse. so, we need the following member variables:

```
MouseState currentMouseState, previousMouseState;
KeyboardState currentKeyboardState, previousKeyboardState;
```

When we enter the `Update` method, we have to first update all these member variables so that they contain the right values when we are actually going to handle the player input. We follow the same procedure for the keyboard states as for the mouse states:

```
previousMouseState = currentMouseState;
previousKeyboardState = currentKeyboardState;
currentMouseState = Mouse.GetState();
currentKeyboardState = Keyboard.GetState();
```

If we want to know if the player has pressed the 'R' key, we use both the current and previous keyboard states, like we did with the mouse state:

```

if (currentKeyboardState.IsKeyDown(Keys.R)
      && previousKeyboardState.IsKeyUp(Keys.R))
    currentColor = colorRed;

```

If the player did press the ‘R’ key, we assign the red colored ball sprite to the `currentColor` variable. We follow the same procedure for the ‘G’ and ‘B’ keys, which gives us the following **if**-instruction:

```

if (currentKeyboardState.IsKeyDown(Keys.R)
      && previousKeyboardState.IsKeyUp(Keys.R))
    currentColor = colorRed;
else if (currentKeyboardState.IsKeyDown(Keys.G)
          && previousKeyboardState.IsKeyUp(Keys.G))
    currentColor = colorGreen;
else if (currentKeyboardState.IsKeyDown(Keys.B)
          && previousKeyboardState.IsKeyUp(Keys.B))
    currentColor = colorBlue;

```

Finally, we only need to add an additional drawing call to the `Draw` method so that the current colored ball is drawn on top of the cannon:

```

spriteBatch.Draw(currentColor, barrelPosition, null, Color.White, 0f,
                 colorOrigin, 1.0f, SpriteEffects.None, 0);

```

Try to run the `Painter2` program now, and see how the program responds to moving the mouse and pressing the ‘R’, ‘G’ or ‘B’ keys.

6.6 What You Have Learned

In this chapter, you have learned:

- what an enumerated type is;
- how to react to mouse clicks and button presses using the **if**-instruction;
- how to formulate conditions for these instructions using boolean values;
- how to use **if**-instructions with different alternatives;
- how to deal with keyboard and gamepad input.



<http://www.springer.com/978-3-642-36579-9>

Learning C# by Programming Games
Egges, A.; Fokker, J.D.; Overmars, M.H.
2013, XXII, 443 p., Hardcover
ISBN: 978-3-642-36579-9