Learning Object-Oriented Programming

Learning Object-Oriented Programming is an easy-to-follow guide full of hands-on examples of solutions to common problems with object-oriented code in Python, JavaScript, and C#. It starts by helping you to recognize objects from real-life scenarios and demonstrates that working with them makes it simpler to write code that is easy to understand and reuse. You will learn to protect and hide data with the data encapsulation features of Python, JavaScript, and C#.

You will explore how to maximize code reuse by writing code capable of working with objects of different types and discover the advantage of duck typing in both Python and JavaScript, while you work with interfaces and generics in C#. With a fair understanding of interfaces, multiple inheritance, and composition, you will move on to refactor existing code and to organize your source for easy maintenance and extension.

Learning Object-Oriented Programming will help you to make better, stronger, and reusable code.

Who this book is written for

If you’re a Python, JavaScript, or C# developer and want to learn the basics of object-oriented programming with real-world examples, then this book is for you.

What you will learn from this book

- Generate instances in three programming languages: Python, JavaScript, and C#
- Customize constructors and destructors
- Work with a combination of access modifiers, prefixes, properties, fields, attributes, and local variables to encapsulate and hide data
- Take advantage of specialization and the possibility to overload or override members
- Create reusable and easier to maintain code
- Use interfaces, generics, and multiple inheritance when available

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'Classes and Instances'
- A synopsis of the book’s content
- More information on Learning Object-Oriented Programming
About the Author

**Gastón C. Hillar** is an Italian and has been working with computers since he was 8 years old. In the early 80s, he began programming with the legendary Texas TI-99/4A and Commodore 64 home computers. Gaston has a bachelor's degree in computer science and graduated with honors. He also holds an MBA, in which he graduated with an outstanding thesis. At present, Gaston is an independent IT consultant and a freelance author who is always looking for new adventures around the world.

He has been a senior contributing editor at Dr. Dobb's and has written more than a hundred articles on software development topics. Gatson was also a former Microsoft MVP in technical computing. He has received the prestigious Intel® Black Belt Software Developer award seven times.

He is a guest blogger at Intel® Software Network ([http://software.intel.com](http://software.intel.com)). You can reach him at gastonhillar@hotmail.com and follow him on Twitter at [http://twitter.com/gastonhillar](http://twitter.com/gastonhillar). Gastón's blog is [http://csharpmulticore.blogspot.com](http://csharpmulticore.blogspot.com).

He lives with his wife, Vanesa, and his two sons, Kevin and Brandon.
Object-oriented programming, also known as OOP, is a required skill in absolutely any modern software developer job. It makes a lot of sense because object-oriented programming allows you to maximize code reuse and minimize the maintenance costs. However, learning object-oriented programming is challenging because it includes too many abstract concepts that require real-life examples to make it easy to understand. In addition, object-oriented code that doesn't follow best practices can easily become a maintenance nightmare.

Nowadays, you need to work with more than one programming language at the same time to develop applications. For example, a modern Internet of Things project may require the Python code running on a board and a combination of C#, JavaScript, and HTML code to develop both the web and mobile apps that allow users to control the Internet of Things device. Thus, learning object-oriented programming for a single programming language is usually not enough.

This book allows you to develop high-quality reusable object-oriented code in Python, JavaScript, and C#. You will learn the object-oriented programming principles and how they are or will be used in each of the three covered programming languages. You will also learn how to capture objects from real-world elements and create object-oriented code that represents them. This book will help you understand the different approaches of Python, JavaScript, and C# toward object-oriented code. You will maximize code reuse in the three programming languages and reduce maintenance costs. Your code will become easy to understand and it will work with representations of real-life elements.
What this book covers

Chapter 1, *Objects Everywhere*, covers the principles of object-oriented paradigms and some of the differences in the approaches toward object-oriented code in each of the three covered programming languages: Python, JavaScript, and C#. You will understand how real-world objects can become part of fundamental elements in the code.

Chapter 2, *Classes and Instances*, tells you how to generate blueprints in order to create objects. You will understand the difference between classes, prototypes, and instances in object-oriented programming.

Chapter 3, *Encapsulation of Data*, teaches you how to organize data in the blueprints that generate objects. You will understand the different members of a class, learn the difference between mutability and immutability, and customize methods and fields to protect them against undesired access.

Chapter 4, *Inheritance and Specialization*, explores how to create a hierarchy of blueprints that generate objects. We will take advantage of inheritance and many related features to specialize behavior.

Chapter 5, *Interfaces, Multiple Inheritance, and Composition*, works with more complex scenarios in which we have to use instances that belong to more than one blueprint. We will use the different features included in each of the three covered programming languages to code an application that requires the combination of multiple blueprints in a single instance.

Chapter 6, *Duck Typing and Generics*, covers how to maximize code reuse by writing code capable of working with objects of different types. In this chapter, you will learn parametric polymorphism, generics, and duck typing.

Chapter 7, *Organization of Object-Oriented Code*, provides information on how to write code for a complex application that requires dozens of classes, interfaces, and constructor functions according to the programming language that you use. It will help you understand the importance of organizing object-oriented code and think about the best solution to organize object-oriented code.

Chapter 8, *Taking Full Advantage of Object-Oriented Programming*, talks about how to refactor existing code to take advantage of all the object-oriented programming techniques that you learned so far. The difference between writing object-oriented code from scratch and refactoring existing code is explained in this chapter. It will also help you prepare object-oriented code for future requirements.
In this chapter, we will start generating blueprints to create objects in each of the three programming languages: Python, JavaScript, and C#. We will:

- Understand the differences between classes, prototypes, and instances in object-oriented programming
- Learn an object’s lifecycle and how object constructors and destructors work
- Declare classes in Python and C# and use workarounds to have a similar feature in JavaScript
- Customize the process that takes place when you create instances in Python, C#, and JavaScript
- Customize the process that takes place when you destroy instances in Python, C#, and JavaScript
- Create different types of objects in Python, C#, and JavaScript

Understanding classes and instances

In the previous chapter, you learned some of the basics of the object-oriented paradigm, including classes and objects, also known as instances. Now, when you dive deep into the programming languages, the class is always going to be the type and the blueprint. The object is the working instance of the class, and one or more variables can hold a reference to an instance.
Let’s move to the world of our best friends, the dogs. If we want to model an object-oriented application that has to work with dogs and about a dozen dog breeds, we will definitely have a Dog abstract class. Each dog breed required in our application will be a subclass of the Dog superclass. For example, let’s assume that we have the following subclasses of Dog:

- TibetanSpaniel: This is a blueprint for the dogs that belong to the Tibetan Spaniel breed
- SmoothFoxTerrier: This is a blueprint for the dogs that belong to the Smooth Fox Terrier breed

So, each dog breed will become a subclass of Dog and a type in the programming language. Each dog breed is a blueprint that we will be able to use to create instances. Brian and Merlin are two dogs. Brian belongs to the Tibetan Spaniel breed, and Merlin belongs to the Smooth Fox Terrier breed. In our application, Brian will be an instance of the TibetanSpaniel subclass, and Merlin will be an instance of the SmoothFoxTerrier subclass.

As both Brian and Merlin are dogs, they will share many attributes. Some of these attributes will be initialized by the class, because the dog breed they belong to determines some features, for example, the area of origin, the average size, and the watchdog ability. However, other attributes will be specific to the instance, such as the name, weight, age, and hair color.

Understanding constructors and destructors

When you ask the programming language to create an instance of a specific class, something happens under the hood. The programming language runtime creates a new instance of the specified type, allocates the necessary memory, and then executes the code specified in the constructor. When the runtime executes the code within the constructor, there is already a live instance of the class. Thus, you have access to the attributes and methods defined in the class. However, as you might have guessed, you must be careful with the code you put within the constructor, because you might end up generating large delays when you create instances of the class.

 Constructors are extremely useful to execute setup code and properly initialize a new instance.
So, for example, before you can call the `CalculateArea` method, you want the `Width` and `Height` attributes for each new `Rectangle` instance to have a value initialized to 0. Constructors are extremely useful when we want to define the attributes of the instances of a class right after their creation.

Sometimes, we need specific arguments to be available when we are creating an instance. We can design different constructors with the necessary arguments and use them to create instances of a class. This way, we can make sure that there is no way of creating specific classes without using the authorized constructors that ask for the necessary arguments.

At some point, your application won't need to work with an instance anymore. For example, once you calculate the perimeter of an ellipse and display the results to the user, you don't need the specific `Ellipse` instance anymore. Some programming languages require you to be careful about leaving live instances alive. You have to explicitly destroy them and de-allocate the memory that it was consuming.

The runtimes of Python, C#, and JavaScript use a garbage-collection mechanism that automatically de-allocates memory used by instances that aren't referenced anymore. The garbage-collection process is a bit more complicated, and each programming language and runtime has specific considerations that should be taken into account to avoid unnecessary memory pressure. However, let's keep our focus on the object's life cycle. In these programming languages, when the runtime detects that you aren't referencing an instance anymore and when a garbage collection occurs, the runtime executes the code specified within the instance's destructor.

You can use the destructor to perform any necessary cleanup before the object is destroyed and removed from memory. However, take into account that JavaScript doesn't provide you with the possibility to customize a destructor.

For example, think about the following situation. You need to count the number of instances of a specific class that are being kept alive. You can have a variable shared by all the classes. Then, you customize the class constructor to atomically increase the value for the counter, that is, increase the value of the variable shared by all the classes of the same time. Finally, you customize the class destructor to automatically decrease the value for the counter. This way, you can check the value of this variable to know the objects that are being referenced in your application.
Declaring classes in Python

Throughout this book, we will work with Python 3.4.3. However, all the explanations and code samples are compatible with Python 3.x.x. Therefore, you can work with previous Python versions as long as the major version number is 3. We will use JetBrains PyCharm Community Edition 4 as the main Python IDE and the supplier of an interactive Python console. However, you can use your favorite Python IDE or just the Python console.

Everything is a class in Python, that is, all the elements that can be named in Python are classes. Guido van Rossum designed Python according to the first-class everything principle. Thus, all the types are classes, from the simplest types to the most complex ones: integers, strings, lists, dictionaries, and so on. This way, there is no difference between an integer (int), a string, and a list. Everything is treated in the same way. Even functions, methods, and modules are classes.

For example, when we enter the following lines in a Python console, we create a new instance of the int class. The console will display <class 'int'> as a result of the second line. This way, we know that area is an instance of the int class:

```python
area = 250
type(area)
```

When we type the following lines in a Python console, we create a new instance of the function class. The console will display <class 'function'> as a result of the second line. Thus, `calculateArea` is an instance of the function class:

```python
def calculateArea(width, height):
    return width * height
type(calculateArea)
```

Let’s analyze the simple `calculateArea` function. This function receives two arguments: width and height. It returns the width value multiplied by the height value. If we call the function with two int values, that is, two int instances, the function will return a new instance of int with the result of width multiplied by height. The following lines call the `calculateArea` function and save the returned int instance in the `rectangleArea` variable. The console will display <class 'int'> as a result of the third line. Thus, `rectangleArea` is an instance of the int class:

```python
rectangleArea = calculateArea(300, 200)
print(rectangleArea)
type(rectangleArea)
```
The following lines create a new minimal `Rectangle` class in Python:

```python
class Rectangle:
    pass
```

The `class` keyword followed by the class name (`Rectangle`) and a colon (`:`) composes the header of the class definition. In this case, the class doesn't have a parent class or a superclass. Therefore, there aren't superclasses enclosed in parentheses after the class name and before the colon (`:`). The indented block of statements that follows the class definition composes the body of the class. In this case, there is just a single statement, `pass`, and the class doesn't define either attributes or methods. The `Rectangle` class is the simplest possible class we can declare in Python.

Any new class you create that doesn't specify a superclass will be a subclass of the `builtins.object` class. Thus, the `Rectangle` class is a subclass of `builtins.object`.

The following line prints `True` as a result in a Python console, because the `Rectangle` class is a subclass of `object`:

```python
issubclass(Rectangle, object)
```

The following lines represent an equivalent way of creating the `Rectangle` class in Python. However, we don't need to specify that the class inherits from an object because it adds unnecessary boilerplate code:

```python
class Rectangle(object):
    pass
```

### Customizing constructors in Python

We want to initialize instances of the `Rectangle` class with the values of both `width` and `height`. After we create an instance of a class, Python automatically calls the `__init__` method. Thus, we can use this method to receive both the `width` and `height` arguments. We can then use these arguments to initialize attributes with the same names. We can think of the `__init__` method as the equivalent of a constructor in other object-oriented programming languages.
The following lines create a `Rectangle` class and declare an `__init__` method within the body of the class:

```python
class Rectangle:
    def __init__(self, width, height):
        print("I'm initializing a new Rectangle instance.")
        self.width = width
        self.height = height
```

This method receives three arguments: `self`, `width`, and `height`. The first argument is a reference to the instance that called the method. We used the name `self` for this argument. It is important to notice that `self` is not a Python keyword. It is just the name for the first argument, and it is usually called `self`. The code within the method prints a message indicating that the code is initializing a new `Rectangle` instance. This way, we will understand when the code within the `__init__` method is executed.

Then, the following two lines create the `width` and `height` attributes for the instance and assign them the values received as arguments with the same names. We use `self.width` and `self.height` to create the attributes for the instance. We create two attributes for the `Rectangle` instance right after its creation.

The following lines create two instances of the `Rectangle` class named `rectangle1` and `rectangle2`. The Python console will display "I'm initializing a new Rectangle instance." after we enter each line in the Python console:

```python
rectangle1 = Rectangle(293, 117)
rectangle2 = Rectangle(293, 137)
```
Chapter 2

The preceding screenshot shows the Python console. Each line that creates an instance specifies the class name followed by the desired values for both the width and the height as arguments enclosed in parentheses. If we take a look at the declaration of the `__init__` method within the `Rectangle` class, we will notice that we just need to specify the second and third arguments (width and height). Also, we just need to skip the required first parameter (`self`). Python resolves many things under the hood. We just need to make sure that we specify the values for the required arguments after `self` to successfully create and initialize an instance of `Rectangle`.

After we execute the previous lines, we can check the values for `rectangle1.width`, `rectangle1.height`, `rectangle2.width`, and `rectangle2.height`.

The following line will generate a `TypeError` error and won't create an instance of `Rectangle` because we missed the two required arguments: width and height. The specific error message is `TypeError: __init__() missing 2 required positional arguments: 'width' and 'height'`. The error message is shown in the following screenshot:

```
rectangleError = Rectangle()
```

Customizing destructors in Python

We want to know when the instances of the `Rectangle` class are removed from memory, that is, when the objects become inaccessible and get deleted by the garbage-collection mechanism. However, it is very important to notice that the ways in which garbage collection works depends on the implementation of Python. Remember that, Python runs on a wide variety of platforms.
Before Python removes an instance from memory, it calls the \_\_del\_\_ method. Thus, we can use this method to add any code we want to run before the instance is destroyed. We can think of the \_\_del\_\_ method as the equivalent of a destructor in other object-oriented programming languages.

The following lines declare a \_\_del\_\_ method within the body of the Rectangle class. Remember that Python always receives self as the first argument for any instance method:

```python
def \_\_del\_\_ (self):
    print('A Rectangle instance is being destroyed.')
```

The following lines create two instances of the Rectangle class: rectangleToDelete1 and rectangleToDelete2. Then, the next lines assign None to both variables. Therefore, the reference count for both objects reaches 0, and the garbage-collection mechanism deletes them. The Python console will display I'm initializing a new Rectangle instance. and then A Rectangle instance is being destroyed. twice in the Python console. Python executes the code within the \_\_del\_\_ method after we assign None to each variable that had the only reference to an instance of the Rectangle class:

```python
rectangleToDelete1 = Rectangle(293, 117)
rectangleToDelete2 = Rectangle(293, 137)
rectangleToDelete1 = None
rectangleToDelete2 = None
```
You can add some cleanup code within the \_\_del\_\_ method. However, take into account that most of the time, you can follow best practices to release resources without having to add code to the \_\_del\_\_ method. Remember that you don’t know exactly when the \_\_del\_\_ method is going to be executed. Even when the reference count reaches 0, the Python implementation might keep the resources until the appropriate garbage collection destroys the instances.

The following lines create a rectangle3 instance of the Rectangle class and then assign a referenceToRectangle3 reference to this object. Thus, the reference count to the object increases to 2. The next line assigns None to rectangle3, and therefore, the reference count for the object goes down from 2 to 1. As the referenceToRectangle3 variable stills holds a reference to the Rectangle instance, Python doesn’t destroy the instance, and we don’t see the results of the execution of the \_\_del\_\_ method:

```python
rectangle3 = Rectangle(364, 257)
referenceToRectangle3 = rectangle3
rectangle3 = None
```

Python destroys the instance if we add a line that assigns None to referenceToRectangle3:

```python
referenceToRectangle3 = None
```

However, it is very important to know that you don’t need to assign None to a reference to force Python to destroy objects. In the previous examples, we wanted to understand how the \_\_del\_\_ method worked. Python will automatically destroy the objects when they aren’t referenced anymore.

## Creating instances of classes in Python

We already created instances of the simple Rectangle class. We just needed to use the class name, specify the required arguments enclosed in parentheses, and assign the result to a variable.

The following lines declare a calculate_area method within the body of the Rectangle class:

```python
def calculate_area(self):
    return self.width * self.height
```
Classes and Instances

The method doesn't require arguments to be called from an instance because it just declares the previously explained `self` parameter. The following lines create an instance of the `Rectangle` class named `rectangle4` and then print the results of the call to the `calculate_area` method for this object:

```python
rectangle4 = Rectangle(143, 187)
print(rectangle4.calculate_area())
```

Now, imagine that we want to have a function that receives the width and height values of a rectangle and returns the calculated area. We can take advantage of the `Rectangle` class to code this new function. We just need to create an instance of the `Rectangle` class with the `width` and `height` received as parameters and return the result of the call to the `calculate_area` method. Remember that we don't have to worry about releasing the resources required by the `Rectangle` instance, because the reference count for this object will become 0 after the function returns the result. The following lines show the code for the `calculateArea` independent function, which isn't part of the `Rectangle` class body:

```python
def calculateArea(width, height):
    return Rectangle(width, height).calculate_area()

print(calculateArea(143, 187))
```

Notice that the Python console displays the following messages. Thus, we can see that the instance is destroyed and the code within the `__del__` method is executed. The messages are shown in the following screenshot:

```
I'm initializing a new Rectangle instance.
A Rectangle instance is being destroyed.
26741
```
Declaring classes in C#

Throughout this book, we will work with C# 6.0 (introduced in Microsoft Visual Studio 2015). However, most of the explanations and code samples are also compatible with C# 5.0 (introduced in Visual Studio 2013). If a specific example uses C# 6.0 syntax and isn't compatible with C# 5.0, the code will be properly labeled with the compatibility warning. We will use Visual Studio Community 2015 as the main IDE. However, you can also run the examples using Mono or Xamarin.

The following lines declare a new minimal Circle class in C#:

```csharp
class Circle
{
}
```

The `class` keyword followed by the class name (Circle) composes the header of the class definition. In this case, the class doesn't have a parent class or a superclass. Therefore, there aren't any superclasses listed after the class name and a colon (:). A pair of curly braces ({}) encloses the class body after the class header. In this case, the class body is empty. The Circle class is the simplest possible class we can declare in C#.

Any new class you create that doesn't specify a superclass will be a subclass of the System.Object class in C#. Thus, the Circle class is a subclass of System.Object.

The following lines represent an equivalent way of creating the Circle class in C#. However, we don't need to specify that the class inherits from System.Object, because it adds unnecessary boilerplate code:

```csharp
class Circle : System.Object
{
}
```

Customizing constructors in C#

We want to initialize instances of the Circle class with the radius value. In order to do so, we can take advantage of the constructors in C#. Constructors are special class methods that are automatically executed when we create an instance of a given type. The runtime executes the code within the constructor before any other code within a class.
We can define a constructor that receives the radius value as an argument and use it to initialize an attribute with the same name. We can define as many constructors as we want. Therefore, we can provide many different ways of initializing a class. In this case, we just need one constructor.

The following lines create a `Circle` class and define a constructor within the class body.

```csharp
class Circle
{
    private double radius;

    public Circle(double radius)
    {
        Console.WriteLine(String.Format("I'm initializing a new Circle instance with a radius value of {{0}}.", radius));
        this.radius = radius;
    }
}
```

The constructor is a public class method that uses the same name as the class: `Circle`. The method receives a single argument: `radius`. The code within the method prints a message on the console, indicating that the code is initializing a new `Circle` instance with a specific radius value. This way, we will understand when the code within the constructor is executed. As the constructor has an argument, it is known as a parameterized constructor.

Then, the following line assigns the radius double value received as an argument to the private radius double field. We use `this.radius` to access the private radius attribute for the instance and `radius` to reference the argument. In C#, the `this` keyword provides access to the instance that has been created and the one we want to initialize. The line before the constructor declares the private `radius` double field. At this time, we won't pay attention to the difference between the `private` and `public` keywords. We will dive deep into the proper usage of these keywords in Chapter 3, Encapsulation of Data.

The following lines create two instances of the `Circle` class: `circle1` and `circle2`. The Windows Console application will display "I'm initializing a new Circle instance with a radius value of", followed by the radius value specified in the call to the constructor of each instance:

```csharp
class Chapter01
{
    public static void Main(string[] args)
    {
```
var circle1 = new Circle(25);
var circle2 = new Circle(50);
Console.ReadLine();

Each line that creates an instance uses the `new` keyword, followed by the desired value for the radius as an argument enclosed in parentheses. We used the `var` keyword to let C# automatically assign the `Circle` type for each of the variables. After we execute the two lines that create the instances of `Circle`, we can use an inspector, such as the `Autos Window`, the `Watch Window`, or the `Immediate Window`, to check the values for `circle1.radius` and `circle2.radius`.

The following line prints "System.Object" as a result in the `Immediate Window` in the IDE. This is because the `Circle` class is a subclass of `System.Object`:

```csharp
circle1.GetType().BaseType.ToString()
```
The following line won't allow the console application to compile and will display a build error. This is because the compiler cannot find a parameterless constructor declared in the Circle class. The specific error message is ConsoleApplication does not contain a constructor that takes 0 arguments. The following screenshot displays the var circleError = new Circle(); error:

![Screenshot of console application error]

**Customizing destructors in C#**

We want to know when the instances of the Circle class are removed from memory, that is, when the objects go out of scope and the garbage-collection mechanism removes them from memory. Destructors are the special class methods that are automatically executed when the run time destroys an instance of a given type. Thus, we can use them to add any code we want to run before the instance is destroyed.

The destructor is a special class method that uses the same name as the class, but prefixed with a tilde (-): ~Circle. The destructor must be parameterless, and it cannot return a value.
The following lines declare a destructor (a \texttt{-Circle} method) within the body of the \texttt{Circle} class:

\begin{verbatim}
-Circle()
{
    Console.WriteLine(String.Format("I'm destroying the Circle instance with a radius value of \{0\}.", radius));
}
\end{verbatim}

The code within the destructor prints a message on the console indicating that the runtime is destroying a \texttt{Circle} instance with a specific radius value. This way, we will understand when the code within the destructor is executed.

If we execute the console application after adding the code for the destructor to the \texttt{Circle} class, we will see the following lines in the console output. The first two lines will appear before we press a key. After we press a key, we will see the two lines indicating that the code within the destructor has been executed. This is because the two variables \texttt{circle1} and \texttt{circle2} have run out of scope and the garbage collector has destroyed the objects:

\begin{verbatim}
I'm initializing a new Circle instance with a radius value of 25.
I'm initializing a new Circle instance with a radius value of 50.
I'm destroying the Circle instance with a radius value of 50.
I'm destroying the Circle instance with a radius value of 25.
\end{verbatim}

\section*{Creating instances of classes in C#}

We already created instances of the simple \texttt{Circle} class. We just needed to use the \texttt{new} keyword followed by the class name, specify the required arguments enclosed in parentheses, and assign the result to a variable.

The following lines declare a public \texttt{CalculateArea} method within the body of the \texttt{Circle} class:

\begin{verbatim}
public double CalculateArea()
{
    return Math.PI * Math.Pow(this.radius, 2);
}
\end{verbatim}
The method doesn't require arguments to be called. It returns the result of the multiplication of $\pi$ by the square of the radius field value (this.radius). The following lines show a new version of the Main method. These lines create two instances of the Circle class: circle1 and circle2. The lines then display the results of calling the CalculateArea method for the two objects. The new lines are highlighted, as follows:

```csharp
class Chapter01
{
    public static void Main(string[] args)
    {
        var circle1 = new Circle(25f);
        var circle2 = new Circle(50f);
        Console.WriteLine(String.Format("The area for circle #1 is {0}",
                                        circle1.CalculateArea()));
        Console.WriteLine(String.Format("The area for circle #2 is {0}",
                                        circle2.CalculateArea()));
        Console.ReadLine();
    }
}
```

Now, imagine that we want to have a function that receives the radius value of a circle and has to return the calculated area. We can take advantage of the Circle class to code this new function. We just need to create an instance of the Circle class with the radius received as a parameter and return the result of the call to the CalculateArea method. Remember that, we don't have to worry about releasing the resources required by the Circle instance, because the object will go out of scope after the function returns the result. The following lines show the code for the new CalculateCircleArea function that isn't part of the Circle class body. The function is a method of the Chapter 1, Objects Everywhere class body, which also has the Main method:

```csharp
class Chapter01
{
    private static double CalculateCircleArea(double radius)
    {
        return new Circle(radius).CalculateArea();
    }

    static void Main(string[] args)
    {
        double radius = 50;
        Console.WriteLine(String.Format("The area for a circle with a
                                      radius of {0} is {1} ", radius,
                                      CalculateCircleArea(radius)));
        Console.ReadLine();
    }
}
```
The Windows command line displays the following messages. Thus, we can see that the instance is destroyed and the code within the destructor is executed:

I'm initializing a new Circle instance with a radius value of 50.
The area for a circle with a radius of 50 is 7853.98163397448
I'm destroying the Circle instance with a radius value of 50.

Understanding that functions are objects in JavaScript

We will use Chrome Developer Tools (CDT), as the main JavaScript console. However, you can run the examples in any other browser that provides a JavaScript console.

Functions are first-class citizens in JavaScript. In fact, functions are objects in JavaScript. When we type the following lines in a JavaScript console, we create a new function object. Thus, calculateArea is an object, and its type is function. Notice the results of writing the following lines in a JavaScript console. The displayed type for calculateArea is a function, as follows:

```javascript
function calculateArea(width, height) { return width * height; }
typeof(calculateArea)
```

The calculateArea function receives two arguments: width and height. It returns the width value multiplied by the height value. The following line calls the calculateArea function and saves the returned number in the rectangleArea variable:

```javascript
var rectangleArea = calculateArea(300, 200);
console.log(rectangleArea);
```

Functions are special objects in JavaScript that contain code and that you can invoke. They contain properties and methods. For example, if we type the following line, the JavaScript console will display the value for the name property of the function object, that is, the calculateArea function:

```javascript
console.log(calculateArea.name);
```

Downloading the example code

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
Working with constructor functions in JavaScript

The following lines of code create an object named `myObject` without any specific properties or methods. This line checks the type of the variable (`myObject`) and then prints the key-value pairs that define the object on the JavaScript console:

```javascript
var myObject = {};
typeof(myObject);
myObject
```

The preceding lines created an empty object. Therefore, the result of the last line shows `Object {}` on the console. There are no properties or methods defined in the object. However, if we enter `myObject` (myObject followed by a dot) in a JavaScript console with autocomplete features, we will see many properties and methods listed, as shown in the following screenshot. The object includes many built-in properties and methods:
The following lines of code create an object named `myRectangle` with two key-value pairs enclosed within a pair of curly braces ({}). A colon (:) separates the key from the value and a comma (,) separates the key-value pairs. The next line checks the type of the variable (`object`) and prints the key-value pairs that define the object on the JavaScript console:

```javascript
var myRectangle = { width: 300, height: 200 };
typeof(myRectangle);
myRectangle
```

The preceding lines created an object with two properties: `width` and `height`. The result of the last line shows `Object {width: 300, height: 200}` on the console. Thus, the `width` property has an initial value of 300, and the `height` property has an initial value of 200. If we enter `myRectangle. (myRectangle followed by a dot)` in a JavaScript console with autocomplete features, we will see the `width` and `height` properties listed with the built-in properties and methods, as shown in the following screenshot:

So far, we have been creating independent objects. The first example was an empty object, and the second case was an object with two properties with their initial values initialized. However, if we want to create another object with the same properties but different values, we would have to specify the property names again. For example, the following line creates another object named `myRectangle2`, with the same two keys, but different values:

```javascript
var myRectangle2 = { width: 500, height: 150 };`
However, we are repeating code and can have typos when we enter the key names, that is, the names for the future properties of the instance. Imagine that we had written the following line instead of the preceding line (notice that the code contains typos):

```javascript
var myRectangle2 = { widh: 500, hight: 150 };
```

The previous line will generate `widh` and `hight` properties instead of `width` and `height`. Thus, if we want to retrieve the value from `myRectangle2.width`, we would receive `undefined` as a response. This is because `myRectangle2` created the `widh` property instead of `width`.

We want to initialize new rectangle objects with the values of both the width and the height. However, we don't want a typo to generate problems in our code. Thus, we need a blueprint that generates and initializes properties with the same names. In addition, we want to log a message to the console whenever we have a new rectangle object. In order to do so, we can take advantage of the constructor function. The following lines declare a `Rectangle` constructor function in JavaScript:

```javascript
function Rectangle(width, height) {
  console.log("I'm creating a new Rectangle");
  this.width = width;
  this.height = height;
}
```

Notice the capitalized name of the function, `Rectangle` instead of `rectangle`. It is a good practice to capitalize constructor functions to distinguish them from the other functions.

The constructor function receives two arguments: `width` and `height`. The code within the function is able to access the new instance of the current object that has been created with the `this` keyword. Thus, you can translate `this` to the current object. The code within the function prints a message on the JavaScript console, indicating that it is creating a new rectangle. The code then uses the received `width` and `height` arguments to initialize properties with the same names. We use `this.width` and `this.height` to create the properties for the instance. We create two properties for the instance right after its creation. We can think of the constructor function as the equivalent of a constructor in other object-oriented programming languages.
The following lines create two `Rectangle` objects named `rectangle1` and `rectangle2`. Notice the usage of the `new` keyword to call the constructor function, with the `width` and `height` values enclosed in parentheses. The Python console will display `I'm initializing a new Rectangle instance. after we enter each line in the Python console:

```javascript
var rectangle1 = new Rectangle(293, 117);
var rectangle2 = new Rectangle(293, 137);
```

Each line that creates an instance uses the `new` keyword followed by the constructor function and the desired values for both the width and the height as arguments enclosed in parentheses. After we execute the previous lines, we can check the values for `rectangle1.width`, `rectangle1.height`, `rectangle2.width`, and `rectangle2.height`.

Enter the following two lines in the console:

```javascript
rectangle1;
rectangle2;
```

The console will display the following two lines:

```
Rectangle {width: 293, height: 117}
Rectangle {width: 293, height: 137}
```

It is very clear that we have created two `Rectangle` objects and not just two `Object` objects. We can see that the constructor function name appears before the key-value pairs.
Enter the following line in the console:

```javascript
rectangle1 instanceof Rectangle
```

The console will display `true` as a result of the evaluation of the previous expression, because `rectangle1` is an instance of `Rectangle`. This way, we can determine whether an object is a `Rectangle` object, that is, an instance created using the `Rectangle` constructor function.

## Creating instances in JavaScript

We already created instances with the simple `Rectangle` constructor function. We just needed to use the `new` keyword and the constructor function name. We then need to specify the required arguments enclosed in parentheses and assign the result to a variable.

The following lines declare a new version of the `Rectangle` constructor function that adds a `calculateArea` function to the blueprint:

```javascript
function Rectangle(width, height) {
    console.log("I'm creating a new Rectangle");
    this.width = width;
    this.height = height;

    this.calculateArea = function() {
        return this.width * this.height;
    }
}
```

The new constructor function adds a parameterless `calculateArea` method to the instance. The following lines of code create a new `Rectangle` object named `rectangle3` and then print the results of the call to the `calculateArea` method for this object:

```javascript
var rectangle3 = new Rectangle(143, 187);
rectangle3.calculateArea();
```

If we enter the following line, the JavaScript console will display the same code we entered to create the new version of the `Rectangle` constructor function. Thus, we might create a new `Rectangle` object by calling the `rectangle3.constructor` function in the next line. Remember that the `constructor` property is automatically generated by JavaScript, and this property in is a function:

```javascript
var rectangle4 = new rectangle3.constructor(300, 200);
```
Now, imagine that we want to have a function that receives the width and height values of a rectangle and returns the calculated area. We can take advantage of a `Rectangle` object to code this new function. We just need to create a `Rectangle` object using the `Rectangle` constructor function with `width` and `height` received as parameters. We then need to return the result of the call to the `calculateArea` method. Remember that we don't have to worry about releasing the resources required by the `Rectangle` object, because the variable will go out of scope after the function returns the result. The following lines show the code for the `calculateArea` independent function, which isn't a part of the `Rectangle` constructor function:

```javascript
function calculateArea(width, height) {
    return new Rectangle(width, height).calculateArea();
}

calculateArea(143, 187);
```

**Summary**

In this chapter, you learned about an object's life cycle. You also learned how object constructors and destructors work. We declared classes in Python and C# and used constructor functions in JavaScript to generate blueprints for objects. We customized constructors and destructors, and tested their personalized behavior in action. We understood different ways of generating instances in the three programming languages.

Now that you have learned to start creating classes and instances, we are ready to share, protect, and hide data with the data-encapsulation features of Python, JavaScript, and C#, which is the topic of the next chapter.
Where to buy this book
You can buy Learning Object-Oriented Programming from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.